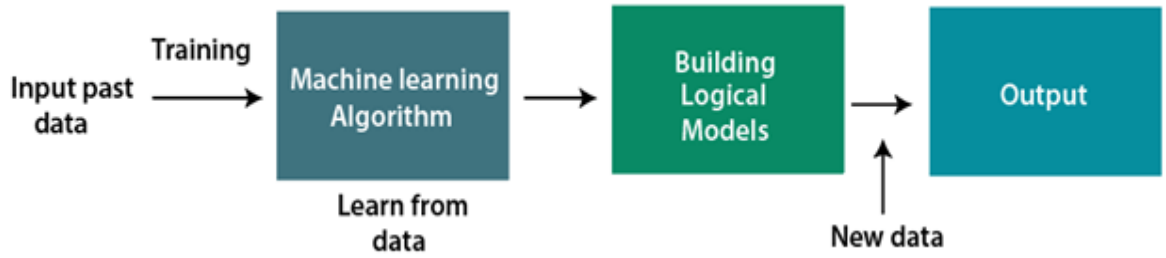
Introduction to machine Learning

Machine Learning enables a machine to automatically learn from data, improve performance from experiences, and predict things without being explicitly programmed. The term machine learning was first introduced by **Arthur Samuel** in **1959**.
Machine learning is one of the most exciting technologies that one would have ever come across. As it is evident from the name, it gives the computer that makes it more similar to humans: The ability to learn. Machine learning is actively being used today, perhaps in many more places than one would expect.

How machine learning work:

A Machine Learning system learns from historical data, builds the prediction models, and whenever it receives new data, predicts the output for it. The accuracy of predicted output depends upon the amount of data, as the huge amount of data helps to build a better model which predicts the output more accurately.
Suppose we have a complex problem, where we need to perform some predictions, so instead of writing a code for it, we just need to feed the data to generic algorithms, and with the help of these algorithms, machine builds the logic as per the data and predict the output. Machine learning has changed our way of thinking about the problem. The below block diagram explains the working of Machine Learning algorithm:



Need for machine learning:

The need for machine learning is increasing day by day. The reason behind the need for machine learning is that it is capable of doing tasks that are too complex for a person to implement directly. As a human, we have some limitations as we cannot access the

huge amount of data manually, so for this, we need some computer systems and here comes machine learning to make things easy for us.

We can train machine learning algorithms by providing them with a huge amount of data and let them explore the data, construct the models, and predict the required output automatically. The performance of the machine learning algorithm depends on the amount of data, and it can be determined by the cost function. With the help of machine learning, we can save both time and money.

Some key points which shows the importance of machine learning:

- Solving complex problems, which are difficult for a human.
- Rapid increment in the production of data.
- Decision making in various sectors including finance.
- Finding hidden patterns and extracting useful information from data.

Some common applications of machine learning:

- **E-commerce product recommendation:**

  One of the prominent elements of typically any e-commerce website is product recommendation, which involves the sophisticated use of machine learning algorithms. Websites track customer behavior based on past purchases, browsing habits, and cart history and then recommend products using machine learning and AI.

- **Catching email spam:**

  One of the most popular applications of machine learning that everyone is familiar with is in detecting email spam. Email service providers build applications with spam filters that use an ML algorithm to classify an incoming email as spam and direct it to the spam folder.

- **Self-Driving Cars:**

  Self-driving cars use an unsupervised learning algorithm that heavily relies on machine learning techniques. This algorithm enables the vehicle to collect information from cameras and sensors about its surroundings, understand it, and choose what actions to perform.

- **Online fraud Detection:**

  One of the most essential applications of machine learning is fraud detection. Every time a customer completes a transaction, the machine learning model carefully examines their profile in search of any unusual patterns to detect online fraud.

- **Stock market trading:**

  When it comes to the stock market and day trading, machine learning employs algorithmic trading to extract important data to automate or support crucial investment activities. Successful portfolio management, and choosing when to buy and sell stocks are some tasks accomplished using ML.

- **Catching malware:**

  The process of using machine learning (ML) to detect malware consists of two basic stages. First, analyzing suspicious activities in an Android environment to generate a suitable collection of features; second, training the system to use the machine and deep learning (DL) techniques on the generated features to detect future cyberattacks in such environments.

- **Speech Recognition:**

  ML software can make measurements of words spoken using a collection of numbers that represent the speech signal. Popular applications that employ speech recognition include Amazon's Alexa, Apple's Siri, and Google Maps.

- **Image Recognition:**

  One of the most notable machine learning applications is image recognition, which is a method for cataloging and detecting an object or feature in a digital image. In addition, this technique is used for further analysis, such as pattern recognition, face detection, and face recognition.

- **Cancer prognosis and prediction:**

  As ML algorithms can identify critical traits in complicated datasets, it is applied in cancer research. It is used to construct prediction models using techniques like Artificial Neural Networks (ANNs), Bayesian Networks (BNs), and Decision

Trees (DTs). This helps in precise decision-making and modeling of the evolution and therapy of malignant diseases.

Classification of machine learning

It is classified into:

1. Supervised machine learning.
2. Unsupervised machine learning.

Supervised machine

Supervised learning is a type of machine learning method in which we provide sample labeled data to the machine learning system in order to train it, and on that basis, it predicts the output.
Supervised learning further divided into two categories:

- Regression.
- Classification.

Unsupervised machine learning:

The training is provided to the machine with the set of data that has not been labeled, classified, or categorized, and the algorithm needs to act on that data without any supervision. The goal of unsupervised learning is to restructure the input data into new features or a group of objects with similar patterns.
Supervised learning further divided into two categories:

- Clustering.
- Association.

Scikit-Learn:

scikit-learn, also known as sklearn, is a popular open-source machine learning library for Python. It provides a wide range of tools and algorithms for various tasks in machine learning, including classification, regression, clustering, dimensionality reduction, and more. Here are some key features and components of scikit-learn:

1. Consistent API: scikit-learn provides a consistent and intuitive API for building and using machine learning models. It follows the same pattern for all algorithms, making it easy to switch between different models without major code modifications.

2. Comprehensive Algorithms: The library offers a wide range of machine learning algorithms, including linear regression, logistic regression, support vector machines (SVM), random forests, gradient boosting, k-means clustering, and more. These algorithms are implemented efficiently and optimized for large-scale datasets.

3. Data Preprocessing: scikit-learn provides various tools for data preprocessing, including handling missing values, feature scaling, encoding categorical variables, and feature extraction. These preprocessing techniques help to prepare the data before training the models.

4. Model Evaluation: The library offers metrics and scoring functions for evaluating the performance of machine learning models. It includes metrics for classification (e.g., accuracy, precision, recall, F1-score), regression (e.g., mean squared error, R-squared), and clustering (e.g., silhouette coefficient). It also supports techniques like cross-validation and hyperparameter tuning for robust model evaluation.

5. Feature Selection and Dimensionality Reduction: scikit-learn provides methods for feature selection, such as selecting the most important features or removing irrelevant features. It also offers dimensionality reduction techniques like principal component analysis (PCA) and manifold learning methods to reduce the number of features while preserving important information.

6. Integration with NumPy and pandas: scikit-learn seamlessly integrates with other popular Python libraries like NumPy and pandas. It accepts NumPy arrays and pandas DataFrames as input, making it convenient to work with data in those formats.

7. Community and Documentation: scikit-learn has a vibrant community, with active development and ongoing support. It provides comprehensive documentation, including user guides, API references, tutorials, and examples, to help users get started and understand the library's capabilities.

Overall, scikit-learn is widely used for its simplicity, versatility, and extensive collection of machine learning tools. It is suitable for both beginners and experienced practitioners, enabling them to build and deploy machine learning models effectively.

Installation using pip:

```
Pip install scikit-learn
```

Installation using conda:

```
Conda install scikit-learn
```

After the installation is complete, you can import scikit-learn in your python script or interactive environment.

```
Import scikit-learn
```

Train-test split:

The train-test split is a technique used in machine learning to divide a dataset into two separate subsets: the training set and the testing set. The training set is used to train the machine learning model, while the testing set is used to evaluate the model's performance on unseen data.

Syntax for train-test split using `train_test_split` function:

```
   from sklearn.model_selection import train_test_split
  # Split the data into training and testing sets
   X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Explanation:

1. Import the `train_test_split` function from the `sklearn.model_selection` module:
  ```python
   from sklearn.model_selection import train_test_split
  ```
2. Call the `train_test_split` function and pass the input features (`X`) and the target variable (`y`) as arguments, along with additional parameters:
  ```python
   X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)  ```
  `X`: The input features of the dataset.
  ```

`y`: The corresponding target variable.

`test_size`: This parameter specifies the proportion of the dataset that should be allocated to the testing set. It can be a float value (0.0 to 1.0) representing the percentage of the data, or an integer representing the absolute number of samples.

`random_state`: This parameter sets the random seed, which ensures reproducibility of the split. It allows you to obtain the same split every time you run the code with the same seed value. It's optional but recommended for reproducibility purposes.

3. The `train_test_split` function returns four subsets of the data:

`X_train`: The training set's input features.

`X_test`: The testing set's input features.

`y_train`: The training set's target variable.

`y_test`: The testing set's target variable.

After the split, you can use `X_train` and `y_train` to train your machine learning model, and then use `X_test` to make predictions or evaluate the model's performance by comparing the predicted values to `y_test`.

Remember to adapt the code to your specific dataset and task, ensuring that `X` and `y` contain the appropriate data.

Hyperparameter tuning:

A Machine Learning model is defined as a mathematical model with a number of parameters that need to be learned from the data. By training a model with existing data, we are able to fit the model parameters.

However, there is another kind of parameter, known as Hyperparameters, that cannot be directly learned from the regular training process. They are usually fixed before the actual training process begins. These parameters express important properties of the model such as its complexity or how fast it should learn.

Some examples of model hyperparameters include:

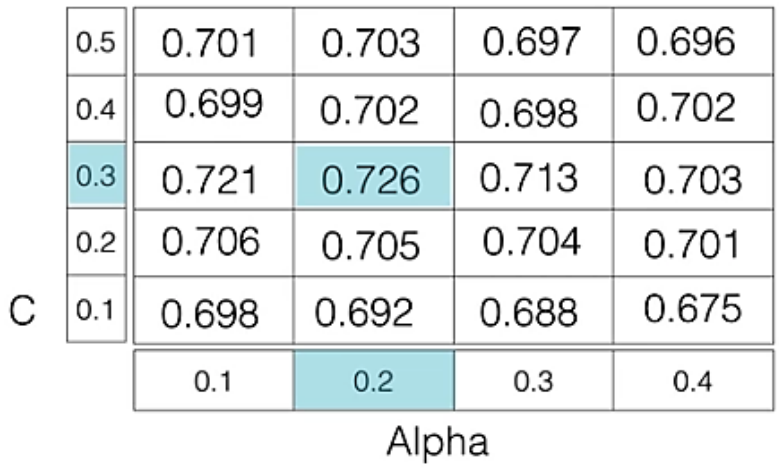
1. The penalty in Logistic Regression Classifier i.e. L1 or L2 regularization
2. The learning rate for training a neural network.
3. The C and sigma hyperparameters for support vector machines.
4. The k in k-nearest neighbors.

The aim of this article is to explore various strategies to tune hyperparameters for Machine learning models.
Models can have many hyperparameters and finding the best combination of parameters can be treated as a search problem. The two best strategies for Hyperparameter tuning are:

- GridSearchCV
- RandomizedSearchCV

GridSearchCV

In GridSearchCV approach, the machine learning model is evaluated for a range of hyperparameter values. This approach is called GridSearchCV, because it searches for the best set of hyperparameters from a grid of hyperparameters values.
For example, if we want to set two hyperparameters C and Alpha of the Logistic Regression Classifier model, with different sets of values. The grid search technique will construct many versions of the model with all possible combinations of hyperparameters and will return the best one.
As in the image, for C = [0.1, 0.2, 0.3, 0.4, 0.5] and Alpha = [0.1, 0.2, 0.3, 0.4]. For a combination of **C=0.3 and Alpha=0.2**, the performance score comes out to be **0.726(Highest)**, therefore it is selected.



The following code illustrates how to use GridSearchCV

```
# Necessary imports
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

# Creating the hyperparameter grid
c_space = np.logspace(-5, 8, 15)
param_grid = {'C': c_space}
```

```
# Instantiating logistic regression classifier
logreg = LogisticRegression()

# Instantiating the GridSearchCV object
logreg_cv = GridSearchCV(logreg, param_grid, cv = 5)

logreg_cv.fit(X, y)

# Print the tuned parameters and score
print("Tuned Logistic Regression Parameters:
{}".format(logreg_cv.best_params_))
print("Best score is {}".format(logreg_cv.best_score_))

Output: Tuned Logistic Regression
Parameters:{'C':3.7275937203149381} Best score is 0.770833333
```

Drawback: GridSearchCV will go through all the intermediate combinations of hyperparameters which makes grid search computationally very expensive.

RandomizedSearchCV

RandomizedSearchCV solves the drawbacks of GridSearchCV, as it goes through only a fixed number of hyperparameter settings. It moves within the grid in a random fashion to find the best set of hyperparameters. This approach reduces unnecessary computation.

The following code illustrates how to use RandomizedSearchCV

```
# Necessary imports
from scipy.stats import randint
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import RandomizedSearchCV

# Creating the hyperparameter grid
param_dist = {"max_depth": [3, None],
              "max_features": randint(1, 9),
              "min_samples_leaf": randint(1, 9),
              "criterion": ["gini", "entropy"]}

# Instantiating Decision Tree classifier
tree = DecisionTreeClassifier()

# Instantiating RandomizedSearchCV object
tree_cv = RandomizedSearchCV(tree, param_dist, cv = 5)

tree_cv.fit(X, y)

# Print the tuned parameters and score
```

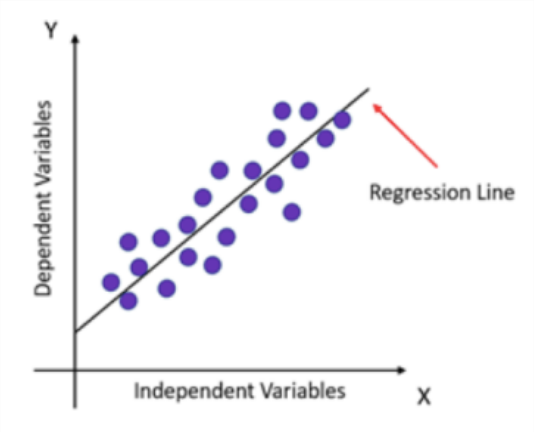
```
print("Tuned Decision Tree Parameters:
{}".format(tree_cv.best_params_))
print("Best score is {}".format(tree_cv.best_score_))
```

```
Output: Tuned Decision Tree Parameters: {'min_samples_leaf':
5,'max_depth': 3,'max_features': 5,'criterion':'gini'} Best
score is 0.7265625
```

Supervised machine learning

Linear Regression:

Regression is a statistical technique that relates a dependent variable to one or more independent variables. A regression model is able to show whether changes observed in the dependent variable are associated with changes in one or more of the independent variables.



Linear regression line

Linear regression can be expressed mathematically as:

$y = b_0 + b_1 x + \varepsilon$

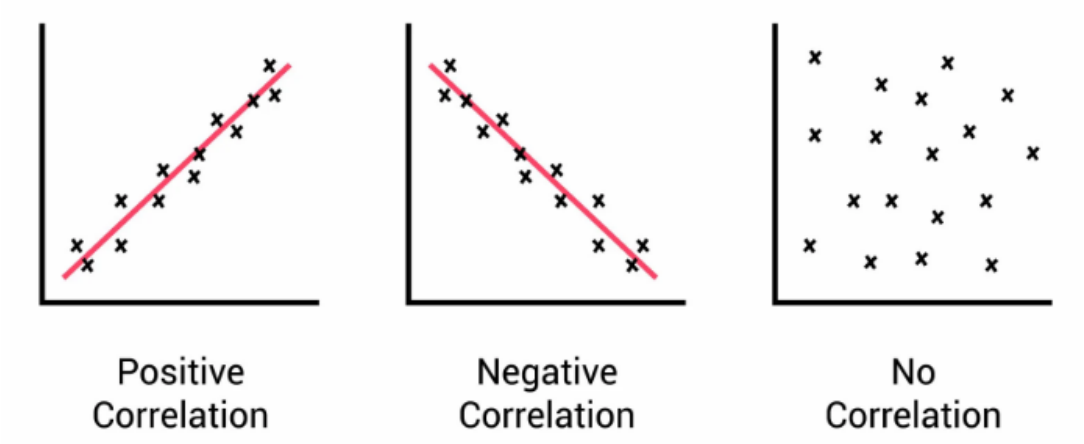
Here,

- y= Dependent variable
- x= Independent variable
- b_0= Intercept of the line
- b_1= Linear regression coefficient (slope of the line)
- ε= Random error

The last parameter, random error ε, is required as the best fit line also doesn't include the data points perfectly.

Linear regression line:

A linear line showing the relationship between the dependent and independent variables is called a regression line. A regression line can show two types of relationship:



Positive linear relationship:

If the dependent variable increases on the Y-axis and independent variable increases on X-axis, then such a relationship is termed as a Positive linear relationship. Equation: $y = b_0+b_1x$

Negative linear relationship:

If the dependent variable decreases on the Y-axis and independent variable increases on the X-axis, then such a relationship is called a negative linear relationship. Equation: $y = -b_0+b_1x$

Zero correlation:

A zero correlation exists when there is no relationship between two variables.

Simple linear regression:

It is a type of regression algorithm that models the relationship between a dependent variable and a single independent variable.

The key point in simple linear regression is that the dependent variable must be a continuous value. However, the independent variable can be measured on continuous and categorical values. Simple linear regression algorithm has mainly two objectives:

- Model the relationship between the two variables: Such as the relationship between Income and expenditure, experience and salary, etc.
- Forecasting new observations: Such as Weather forecasting according to temperature, Revenue of a company according to the investments in a year, etc.

A simple straight-line equation involving slope (dy/dx) and intercept (an integer/continuous value) is utilized in simple Linear Regression.

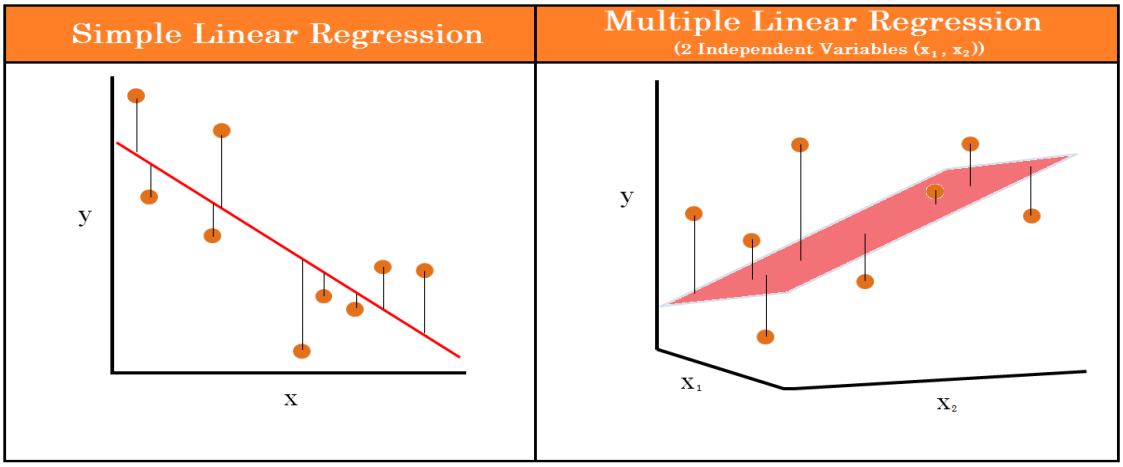
Here a simple form is:

y=mx+c where y denotes the output x is the independent variable, and c is the intercept when x=0. With this equation, the algorithm trains the model of machine learning and gives the most accurate output.

Multiple linear regression:

When a number of independent variables more than one, the governing linear equation applicable to regression takes a different form like:

y= c+m1x1+m2x2... mnxn which represents the coefficient responsible for impact of different independent variables x1, x2 etc. This machine learning algorithm, when applied, finds the values of coefficients m1, m2, etc., and gives the best fitting line.



Simple linear vs multiple linear regression

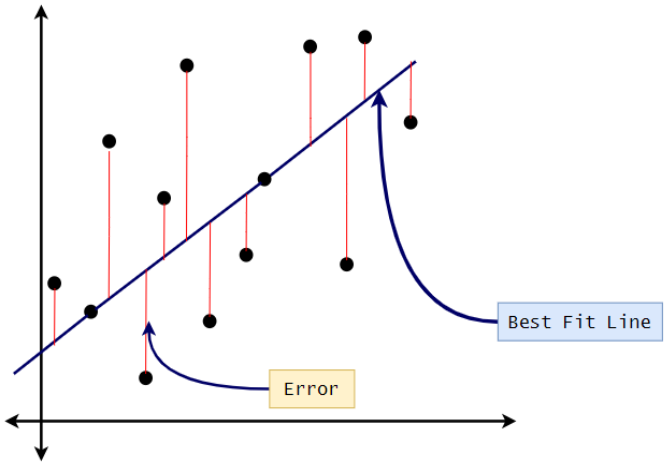
Advantages:

- Simple to implement.
- Perform well on the data with linear relationship.

Disadvantages:

- Not suitable for data having non-linear relationship.
- Underfitting issue.
- Sensitive to outliers.

Find the best fit line:



- In linear regression our purpose is to find the best fit line(model). So, here" loss function" comes into picture.

Loss function:

- Loss function measures how far an estimated value is from its true value, it is helpful to determine which model performs better and which parameters are better.

Loss function/MSE $= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

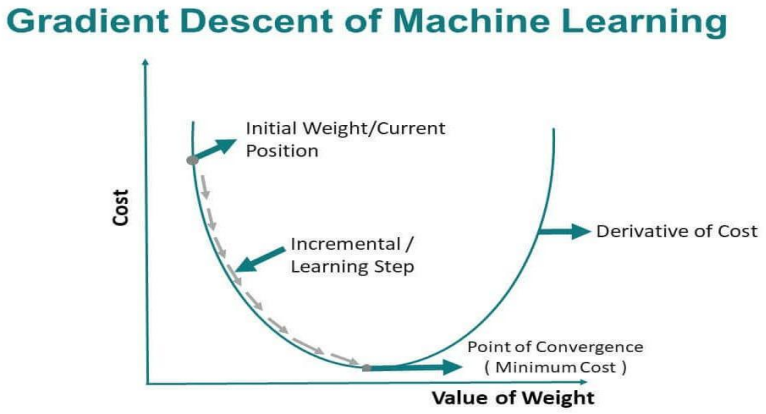
- From the above formula, the error difference is squared for each value and then the average of the sum of squares of error gives us the cost function. It is also referred to as Mean Square Error (MSE).
- Low loss value → High accuracy.
- High loss value → Low accuracy.
- We can improve the model by some optimization technique called "gradient descent".

Residuals:

The distance between the actual value and predicted values is called residual. If the observed points are far from the regression line, then the residual will be high, and so cost function will be high. If the scatter points are close to the regression line, then the residual will be small and hence the cost function.

Gradient descent:

- Gradient descent is used to minimize the MSE by calculating the gradient of the cost function.
- A regression model uses gradient descent to update the coefficients of the line by reducing the cost function.
- It is done by a random selection of values of coefficient and then iteratively updating the values to reach the minimum cost function.



Model performance:

The Goodness of fit determines how the line of regression fits the set of observations. The process of finding the best model out of various models is called optimization. It can be achieved by the R-squared method.

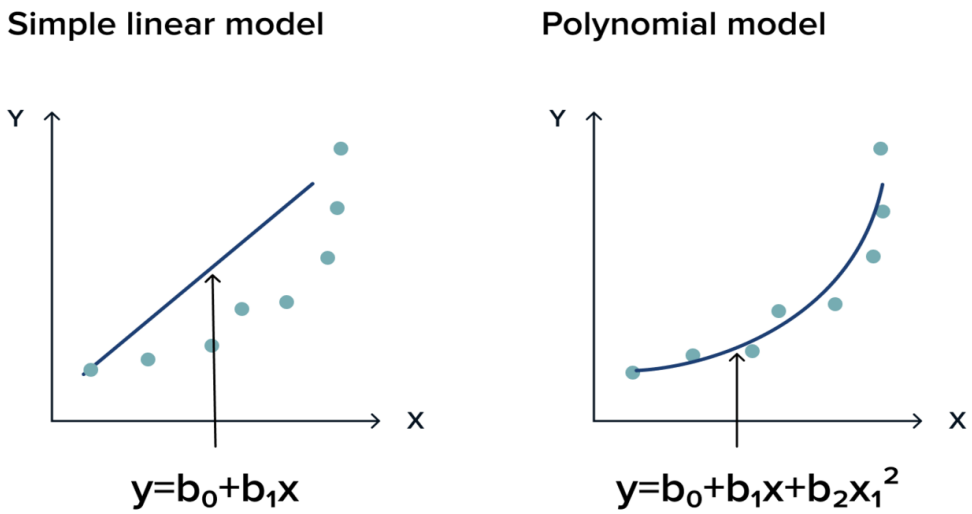
Polynomial regression:

- It is a regression algorithm that model the relationship between a dependent variable(y) and independent variable(x) as nth degree polynomial.

$$y= c+m_1x_1 + m_2x_1^2 + m_3x_1^3 + m_4x_1^4 + \ldots\ldots\ldots m_nx_1^n$$

- It is also called the special case of Multiple Linear Regression in ML. Because we add some polynomial terms to the Multiple Linear regression equation to convert it into Polynomial Regression.
- It is a linear model with some modification in order to increase the accuracy.I
- Hence, "In Polynomial regression, the original features are converted into Polynomial features of required degree (2,3,4,..,n) and then modeled using a linear model.

- Polynomial regression is required because if we apply a linear model on a linear dataset, then it provides us a good result as we have seen in Simple Linear Regression, but if we apply the same model without any modification on a non-linear dataset, then it will produce a drastic output. Due to which loss function will increase, the error rate will be high, and accuracy will be decreased.
- So for such cases, where data points are arranged in a non-linear fashion, we need the Polynomial Regression model.



- In the above image, we have taken a dataset which is arranged non-linearly. So if we try to cover it with a linear model, then we can clearly see that it hardly covers any data point. On the other hand, a curve is suitable to cover most of the data points, which is of the Polynomial model.
- Hence, if the datasets are arranged in a non-linear fashion, then we should use the Polynomial Regression model instead of Simple Linear Regression.

Assumptions of Linear Regression:

Normally most statistical tests and results rely upon some specific assumptions regarding the variables involved. Naturally, if these assumptions are not considered, the results will not be reliable. Linear Regression also comes under the same consideration. There are some common assumptions to be considered while using Linear Regression:

1. **Linearity:** The models of Linear Regression models must be linear in the sense that the output must have a linear association with the input values, and it only suits data that has a linear relationship between the two entities.
2. **Homoscedasticity:** Homoscedasticity means the standard deviation and the variance of the residuals difference of $(y - \hat{y})^2$ must be the same for any value of x. Multiple Linear Regression assumes that the amount of error in the residuals is similar at each point of the linear model. We can check the Homoscedasticity using Scatter plots.
3. **Non-multicollinearity:** The data should not have multicollinearity, which means the independent variables should not be highly correlated with each other. If this occurs, it will be difficult to identify those specific variables which actually contribute to the variance in the dependent variable. We can check the data for this using a correlation matrix.
4. **No Autocorrelation:** When data are obtained across time, we assume that successive values of the disturbance component are momentarily independent in the conventional Linear Regression model. When this assumption is not followed, the situation is referred to as autocorrelation.
5. **Not applicable to Outliers:** The value of the dependent variable cannot be estimated for a value of an independent variable which lies outside the range of values in the sample data.

All the above assumptions are critical because if they are not followed, they can lead to drawing conclusions that may become invalid and unreliable.

Evaluation metrics for regression

Mean Absolute Error(MAE):

- MAE is a simple metric which calculates the absolute difference between actual and predicted values.
- To better understand, let's take an example where you have input data and output data and use Linear Regression, which draws a best-fit line.
- Now you have to find the MAE of your model which is basically a mistake made by the model known as an error. Now find the difference between the actual value and predicted value that is an absolute error but we have to find the mean absolute of the complete dataset.

- so, sum all the errors and divide them by a total number of observations, and this is MAE. And we aim to get a minimum MAE because this is a loss.

MAE equation, $MAE=\frac{1}{N}\sum\left|y - \hat{y}\right|$

N= Total number of data points, y=Actual output, $\hat{y}$= Predicted output.

$\sum\left|y - \hat{y}\right|$= Sum of actual values of residuals.

Advantages of MAE

- The MAE you get is in the same unit as the output variable.
- It is most Robust to outliers.

Disadvantages of MAE

- The graph of MAE is not differentiable so we have to apply various optimizers like Gradient descent which can be differentiable.

Mean Squared Error(MSE):

- MSE represents the squared distance between actual and predicted values. We perform squared to avoid the cancellation of negative terms and it is the benefit of MSE.

  MSE equation: $MSE=\frac{1}{N}(y - \hat{y})^2$

Advantages of MSE

- The graph of MSE is differentiable, so you can easily use it as a loss function.

Disadvantages of MSE

- The value you get after calculating MSE is a squared unit of output. for example, the output variable is in meter(m) then after calculating MSE the output we get is in meter squared.
- If you have outliers in the dataset then it penalizes the outliers most and the calculated MSE is bigger. So, in short, It is not Robust to outliers which were an advantage in MAE.

Root Mean Squared Error(RMSE):

- As RMSE is clear by the name itself, that it is a simple square root of mean squared error.

  RMSE Equation: $RMSE=\sqrt{\frac{1}{N}(y - \hat{y})^2}$

Advantages of RMSE

- The output value you get is in the same unit as the required output variable which makes interpretation of loss easy.

Disadvantages of RMSE

- It is not that robust to outliers as compared to MAE.

R-Squared(R2):

- R squared is a popular metric for identifying model accuracy. It tells how close the data points to the fitted line generated by a regression algorithm.

  $$\text{R-Squared(R2)} = 1 - \frac{\sum(y-\hat{y})^2}{\sum(y-\bar{y})^2} = 1 - \frac{Sum\ of\ squared\ residuals(SSR)}{Total\ sum\ of\ squares(SST)}$$

  Here,

  - 'y ' is the actual target value.
  - '$\hat{y}$' is the predicted target value.
  - '$\bar{y}$' is the mean of the actual target value.
- R^2 score ranges from 0 to 1. The closer to 1 the R^2, the better the regression model is. If R^2 is equal to 0, the model is not performing better than a random model. If R^2 is negative, the regression model is erroneous.
- But the problem is when we add an irrelevant feature in the dataset then at that time R2 sometimes starts increasing which is incorrect.

Adjusted R-Squared(R2):

- Adjusted R^2 is the same as standard R^2 except that it penalizes models when additional features are added.
- To counter the problem which is faced by R-square, Adjusted r-square penalizes adding more independent variables which don't increase the explanatory power of the regression model.
- The value of adjusted r-square is always less than or equal to the value of r-square.
- It ranges from 0 to 1, the closer the value is to 1, the better it is.
- It measures the variation explained by only the independent variables that actually affect the dependent variable.

  Adjusted R-Squared Equation: $$R^2_{Adjusted} = \left[\frac{(1-R^2)(n-1)}{n-k-1}\right]$$

  Where,

  - n is the number of data points.

- K is the number of independent variables.

Example
Model Building

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

In [2]:
```
#importing dataset
df = pd.read_csv("med-insurance.csv")
```

In [3]:
```
df.head()
```

Out[3]:

	age	sex	bmi	children	smoker	region	expenses
0	19	female	27.9	0	yes	southwest	16884.92
1	18	male	33.8	1	no	southeast	1725.55
2	28	male	33.0	3	no	southeast	4449.46
3	33	male	22.7	0	no	northwest	21984.47
4	32	male	28.9	0	no	northwest	3866.86

In [5]:
```
1 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       1338 non-null   int64
 1   sex       1338 non-null   object
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64
 4   smoker    1338 non-null   object
 5   region    1338 non-null   object
 6   expenses  1338 non-null   float64
dtypes: float64(2), int64(2), object(3)
memory usage: 73.3+ KB
```

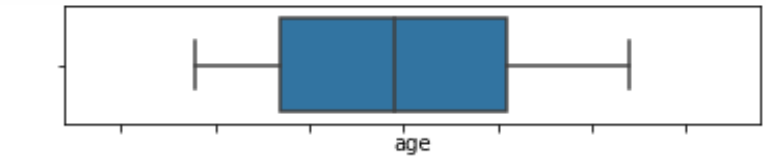
In [6]:
```
1 df.describe()
```

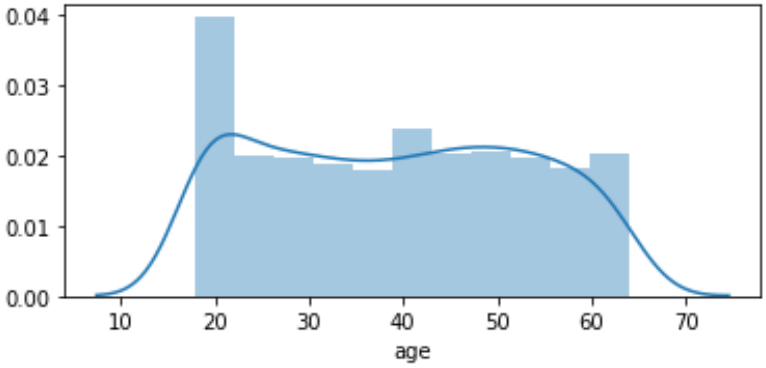
Out[6]:

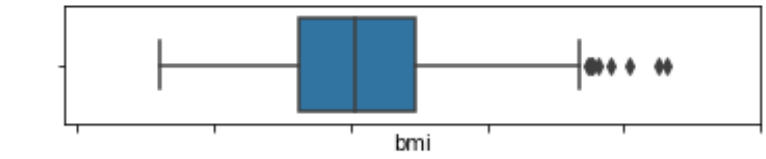
	age	bmi	children	expenses
count	1338.000000	1338.000000	1338.000000	1338.000000
mean	39.207025	30.665471	1.094918	13270.422414
std	14.049960	6.098382	1.205493	12110.011240
min	18.000000	16.000000	0.000000	1121.870000
25%	27.000000	26.300000	0.000000	4740.287500
50%	39.000000	30.400000	1.000000	9382.030000
75%	51.000000	34.700000	2.000000	16639.915000
max	64.000000	53.100000	5.000000	63770.430000

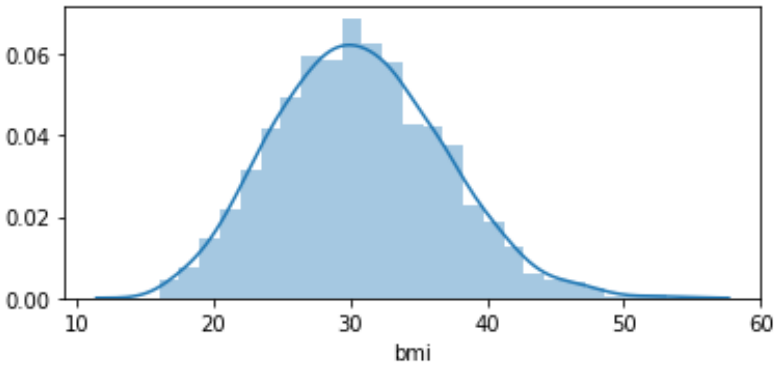
Performing EDA

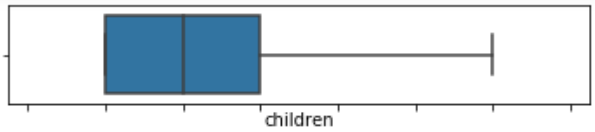
In [7]:
```
1  #for continuous variables
2  f,(ax_box, ax_hist) = plt.subplots(2, sharex=True, gridspec_kw={"height_ratios": (0.2 , 0.5)})
3  sns.boxplot(df['age'], ax=ax_box)
4  sns.distplot(df['age'], ax=ax_hist)
5  plt.show()
6  f,(ax_box, ax_hist) = plt.subplots(2, sharex=True, gridspec_kw={"height_ratios": (0.2 , 0.5)})
7  sns.boxplot(df['bmi'], ax=ax_box)
8  sns.distplot(df['bmi'], ax=ax_hist)
9  plt.show()
10 f,(ax_box, ax_hist) = plt.subplots(2, sharex=True, gridspec_kw={"height_ratios": (0.2 , 0.5)})
11 sns.boxplot(df['children'], ax=ax_box)
12 sns.distplot(df['children'], ax=ax_hist)
13 plt.show()
14 f,(ax_box, ax_hist) = plt.subplots(2, sharex=True, gridspec_kw={"height_ratios": (0.2 , 0.5)})
15 sns.boxplot(df['expenses'], ax=ax_box)
16 sns.distplot(df['expenses'], ax=ax_hist)
17 plt.show()
```

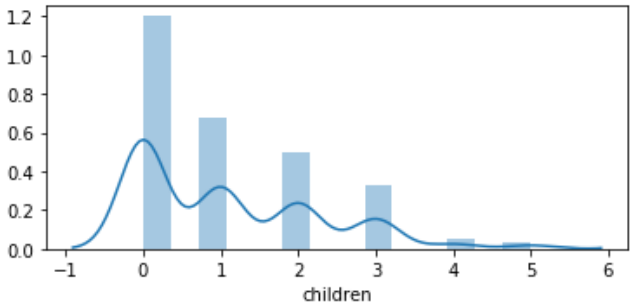

age

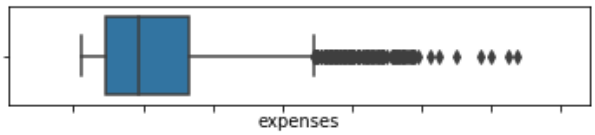

0.04
0.03
0.02
0.01
0.00
10
20
30
40
50
60
70
age

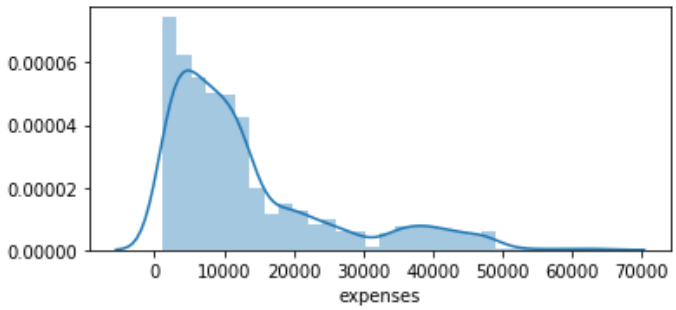

bmi


0.06
0.04
0.02
0.00
10
20
30
40
50
60
bmi





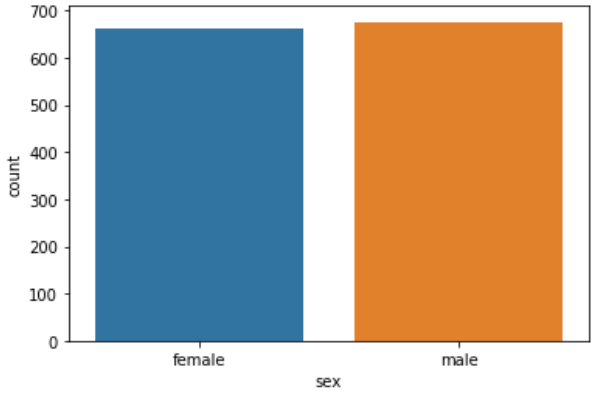


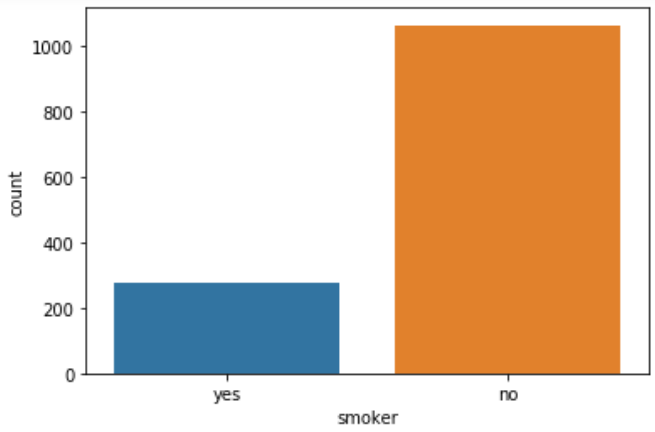


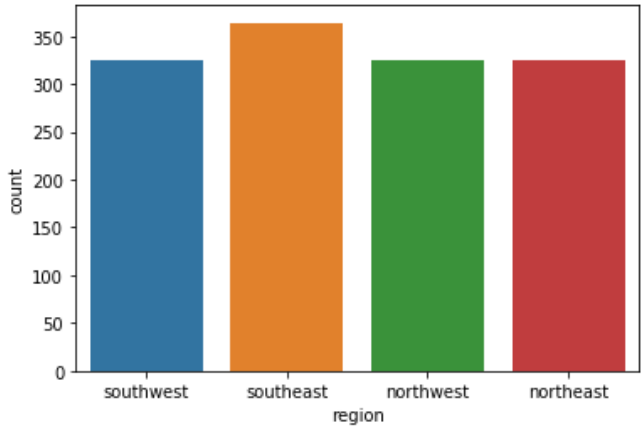

```
Observations
#Age data has a uniform distribution.
#bmi has a normal distribution with few outliers.
#expenses has an unequal distribution with more outliers.
```

In [9]:

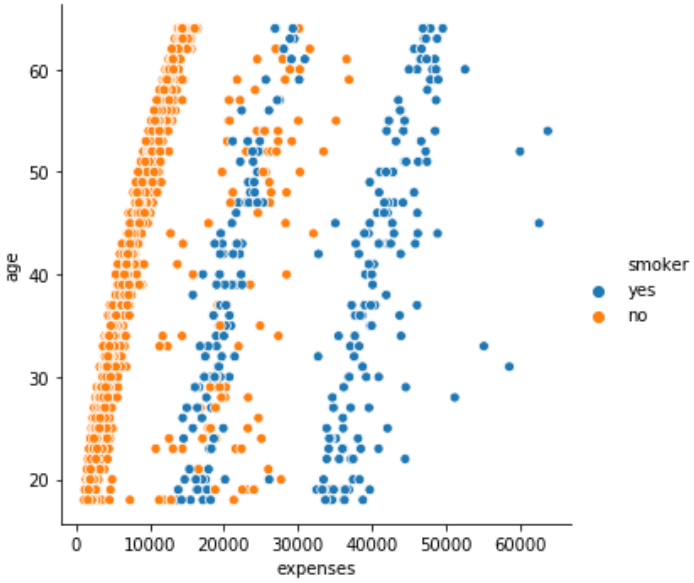
```
#for discrete variables
sns.countplot('sex', data=df)
plt.show()
sns.countplot('smoker',data=df)
plt.show()
sns.countplot('region',data=df)
plt.show()
```





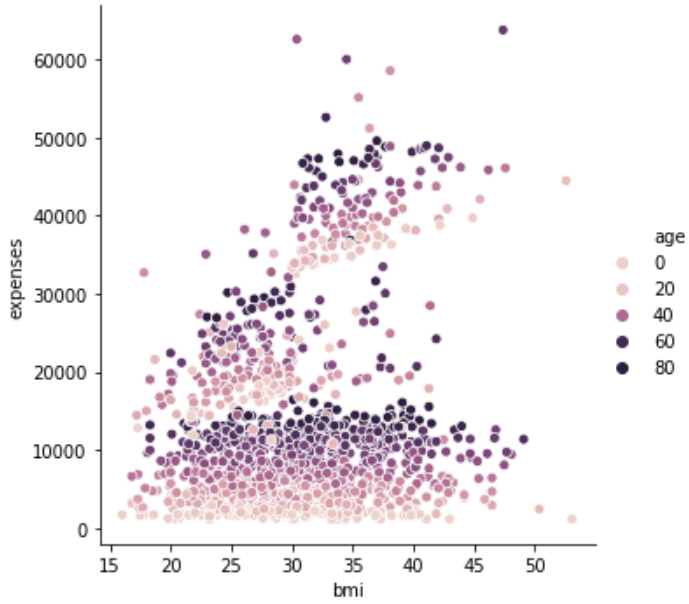


In [10]:
```
1 sns.relplot(x='expenses', y='age', data=df, hue='smoker')
2 plt.show()
```

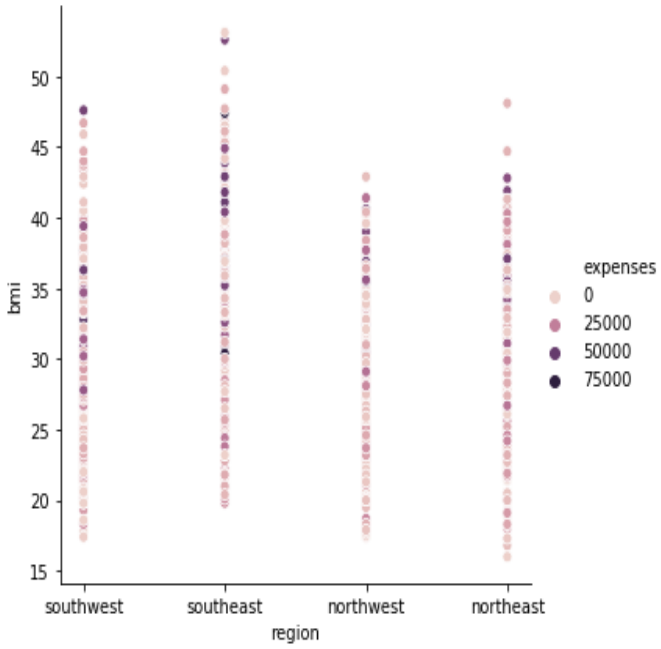



```
1 #here we observed among all age groups smokers are paying more medical expenses
```

In [12]:
```
sns.relplot(x='bmi', y='expenses', data=df, hue='age')
plt.show()
```



In [13]:
```
sns.relplot(x='region', y='bmi', data=df, hue='expenses')
plt.show()
```




```
#Here we have observed that southeast region people are having more BMI among all regions.
#And northwest people are having less BMI compared to all regions.
```

In [18]:
```
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
#Using LabelEncoder for 2 unique values
for col in ['sex','smoker']:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col])
```

In [19]:
```
df.head()
```

Out[19]:

	age	sex	bmi	children	smoker	region	expenses
0	19	0	27.9	0	1	southwest	16884.92
1	18	1	33.8	1	0	southeast	1725.55
2	28	1	33.0	3	0	southeast	4449.46
3	33	1	22.7	0	0	northwest	21984.47
4	32	1	28.9	0	0	northwest	3866.86

In [21]:
```
dum = pd.get_dummies(df['region'], drop_first=True)
dum.head()
```

Out[21]:

	northwest	southeast	southwest
0	0	0	1
1	0	1	0
2	0	1	0
3	1	0	0
4	1	0	0

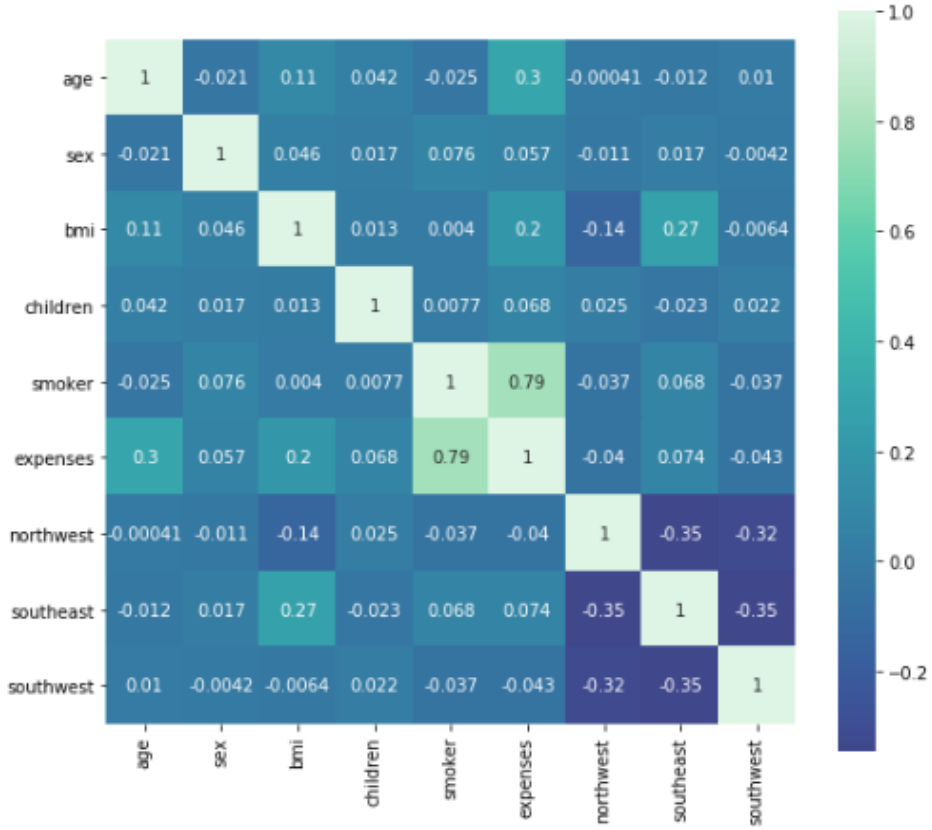
In [22]:
```
df = pd.concat([df,dum], axis=1)
df.head()
```

Out[22]:

	age	sex	bmi	children	smoker	region	expenses	northwest	southeast	southwest
0	19	0	27.9	0	1	southwest	16884.92	0	0	1
1	18	1	33.8	1	0	southeast	1725.55	0	1	0
2	28	1	33.0	3	0	southeast	4449.46	0	1	0
3	33	1	22.7	0	0	northwest	21984.47	1	0	0
4	32	1	28.9	0	0	northwest	3866.86	1	0	0

In [24]:
```
plt.figure(figsize=(9,8))
sns.heatmap(df.corr(), square=True, annot=True, cmap='mako', center = 0)
```

Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x1fc87b4ae88>




```
#Here we can observe there is direct correlation between smoker and expenses i.e; almost 0.8.
#so, we can tell that smoker has to pay more medical expenses.
```

In [30]:
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=42)
```

Linear Regression

In [31]:
```
#Modelling
#LinearRegression
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(x_train, y_train)
```

Out[31]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

In [32]:
```
model.intercept_
```

Out[32]: -12376.785237284665

In [33]:
```
model.coef_
```

Out[33]:
```
array([  261.28251281,   104.99524716,   348.96600937,   424.41067944,
       23627.8945956 ,  -486.46995207,  -970.61815579,  -925.06307896])
```

In [34]:
```
train_predictions = model.predict(x_train)
```

In [35]:
```
test_predictions = model.predict(x_test)
```

In [36]:
```
#evaluation metrics
from sklearn.metrics import mean_squared_error
test_RMSE = np.sqrt(mean_squared_error(y_test, test_predictions))
train_RMSE = np.sqrt(mean_squared_error(y_train, train_predictions))
print(train_RMSE, test_RMSE)
```

```
6142.373139201786 5811.806354382952
```

In [37]:
```
model.score(x_train, y_train) #train R2
```

Out[37]:
```
0.7424103110139746
```

In [38]:
```
model.score(x_test, y_test)  #test r2
```

Out[38]:
```
0.7696351080608885
```

In [39]:
```
#checking cross vadidation score
```

In [40]:
```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model,x,y,cv=5)
print(scores)
scores.mean()
```

```
[0.76148215 0.70650918 0.7780752  0.73273236 0.75559751]
```

Out[40]:
```
0.746879280539993
```

```
#here test accuracy and cross vadidation are nearly same.
```

Polynomial Regression

In [38]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
polynomial_converter = PolynomialFeatures(degree=2, include_bias=False)
```

In [39]:
```
X_poly = polynomial_converter.fit_transform(X)
```

In [40]:
```
X_poly.shape
```

Out[40]:
```
(1338, 44)
```

In [41]:
```
#train_test_split
X_Train, X_Test, Y_Train, Y_Test = train_test_split(X_poly, Y, test_size=0.3, random_state=30)
```

In [42]:
```
model = LinearRegression()
model.fit(X_Train, Y_Train)
```

Out[42]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

In [43]:
```
train_predictions = model.predict(X_Train)
```

In [44]:
```
test_predictions = model.predict(X_Test)
```

In [45]:
```
train_res = Y_Train - train_predictions
test_res = Y_Test - test_predictions
```

In [58]:
```
#train R2
model.score(X_train, y_train)
```

Out[58]:
```
0.8546283366451466
```

In [59]:
```
#test R2
model.score(X_test, y_test)
```

Out[59]:
```
0.8255318580804683
```

In [60]:
```
Scores = cross_val_score(model,x_poly,y,cv=5)
Scores
#average of MSE scores
abs(Scores.mean())
```

Out[60]:
```
0.8354827706855652
```

In [61]:
```
from sklearn.metrics import mean_absolute_error
```

In [62]:
```
MAE = mean_absolute_error(y_test, test_predictions)
MAE
```

Out[62]:
```
2964.6497128464352
```

In [63]:
```
MSE = mean_squared_error(y_test, test_predictions)
MSE
```

Out[63]:
```
25285383.23334269
```

In [64]:
```
RMSE = np.sqrt(MSE)
RMSE
```

Out[64]:
```
5028.457341306844
```

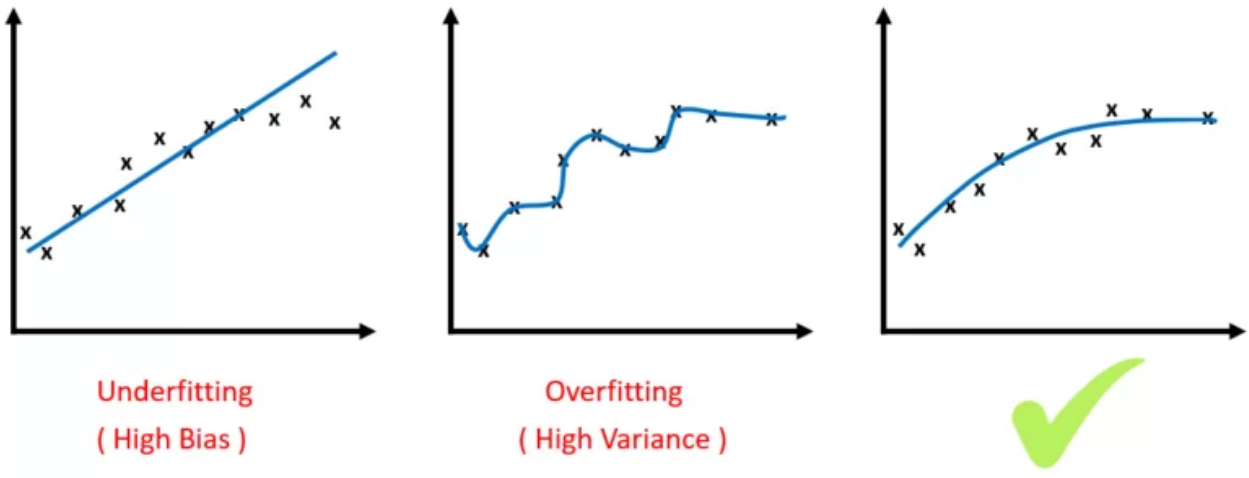
References for above topics:

- https://www.javatpoint.com/machine-learning
- https://www.knowledgehut.com/blog/data-science/linear-regression-for-machine-learning

- https://www.analyticsvidhya.com/blog/2021/10/understanding-polynomial-regression-model/
- https://www.analyticsvidhya.com/blog/2021/05/know-the-best-evaluation-metrics-for-your-regression-model/

Understanding overfitting and underfitting:

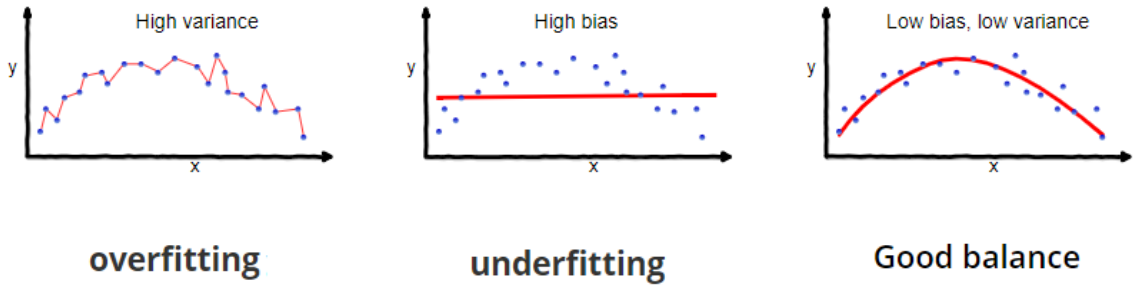
- To train our machine learning model, we provide it with data to learn from. The process of plotting a series of data points and drawing a line of best fit to understand the relationship between variables is called Data Fitting. Our model is best suited when it can find all the necessary patterns in our data and avoid random data points, and unnecessary patterns called noise.
- If we allow our machine learning model to look at the data too many times, it will find many patterns in our data, including some that are unnecessary. It will learn well on the test dataset and fits very well. It will learn important patterns, but it will also learn from the noise in our data and will not be able to make predictions on other data sets.
- A scenario where a machine learning model tries to learn from the details along with the noise in the data and tries to fit each data point to a curve is called Overfitting.
- In the figure below, we can see that the model is fit for every point in our data. If new data is provided, the model curves may not match the patterns in the new data, and the model may not predict very well.



- Conversely, in the scenario where the model has not been allowed to look at our data enough times, the model will not be able to find patterns in our test data set. It won't fit our test data set properly and won't work on new data either.
- A scenario where a machine learning model can neither learn the relationship between variables in the test data nor predict or classify a new data point is called Underfitting.
- The image above shows an under equipped model. We can see that it doesn't fit the data given correctly. He did not find patterns in the data and ignored much of the data set. It cannot work with both known and unknown data.

Bias and variance:

- Bias refers to the errors which occur when we try to fit a statistical model on real-world data which does not fit perfectly well on some mathematical model. If we use a way too simplistic a model to fit the data then we are more likely to face the situation of High Bias which refers to the case when the model is unable to learn the patterns in the data at hand and hence performs poorly.
- Variance implies the error value that occurs when we try to make predictions by using data that is not previously seen by the model. There is a situation known as high variance that occurs when the model learns noise that is present in the data.



Regularization:

When training a machine learning model, the model can be easily overfitted or under fitted. To avoid this, we use regularization in machine learning to properly fit the model to our test set. Regularization techniques help reduce the possibility of overfitting and

help us obtain an optimal model. It refers to techniques used to calibrate machine learning models to minimize the adjusted loss function and avoid overfitting or underfitting.

Ridge regression:

A regression model that uses the L2 regularization technique is called Ridge regression. Ridge regression adds the “squared magnitude” of the coefficient as a penalty term to the loss function(L).

Equation: $\text{loss}=\frac{1}{N}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2+\lambda\sum_{i=1}^{m}w_i^2$

λ is the penalization factor. We can control the values of λ. When w_i^2 value is high the mean square value also increases. So by adding the penalty it controls the w value and doesn’t allow going too high.

Lasso regression:

A regression model which uses the L1 Regularization technique is called LASSO(Least Absolute Shrinkage and Selection Operator) regression. Lasso Regression adds the “absolute value of magnitude” of the coefficient as a penalty term to the loss function(L). Lasso regression also helps us achieve feature selection by penalizing the weights to approximately equal to zero if that feature does not serve any purpose in the model.

Equation: $\text{loss}=\frac{1}{N}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2+\lambda\sum_{i=1}^{m}|w_i|$

The difference between Ridge and Lasso regularization is in Ridge w_i^2 is used and in Lasso $|w_i|$ absolute value is used.

Elastic Net regression:

This model is a combination of L1 as well as L2 regularization. That implies that we add the absolute norm of the weights as well as the squared measure of the weights.

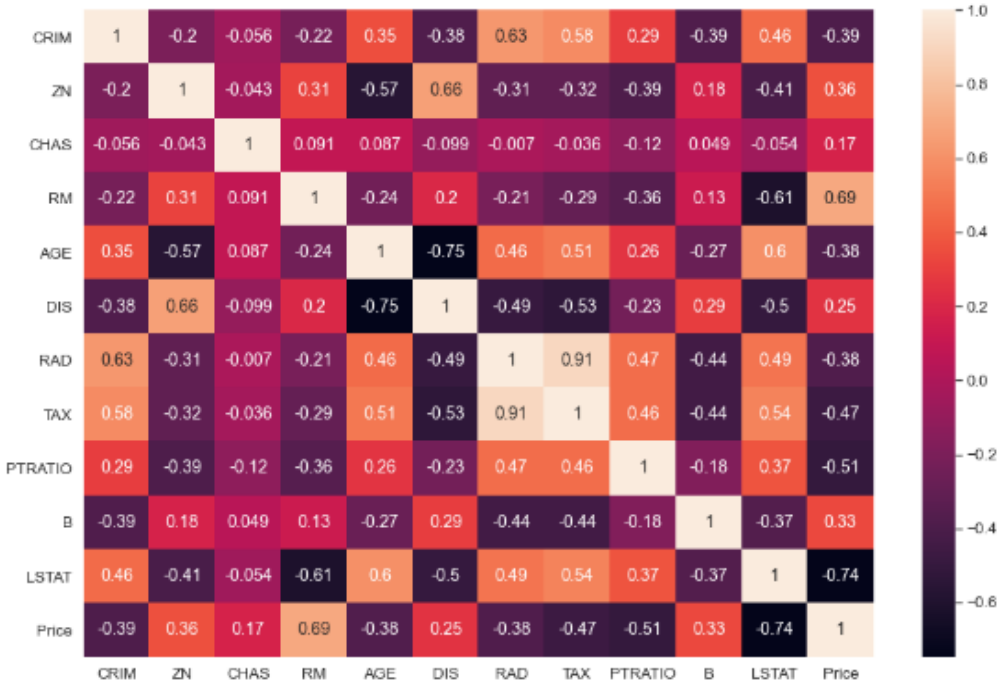
Equation: $\text{loss}=\frac{1}{N}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2+\lambda((1-\alpha)\sum_{i=1}^{m}|w_i|+\alpha\sum_{i=1}^{m}w_i^2)$

Example

Model Building

For this implementation, we will use the Boston housing dataset found in Sklearn. What we intend to see is:

```
#libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge, RidgeCV, Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_boston
#data
boston = load_boston()
boston_df=pd.DataFrame(boston.data,columns=boston.feature_names)
#target variable
boston_df['Price']=boston.target
#Exploration
plt.figure(figsize = (10, 10))
sns.heatmap(boston_df.corr(), annot = True)
```



```
#There are cases of multicolinearity, we will drop a few columns
boston_df.drop(columns = ["INDUS", "NOX"], inplace = True)
#we will log the LSTAT Column
boston_df.LSTAT = np.log(boston_df.LSTAT)
```

Note that we logged the LSTAT column as it doesn't have a linear relationship with the price column. Linear models assume a linear relationship between x and y variables.

Data Splitting and Scaling

```
#preview
features = boston_df.columns[0:11]
target = boston_df.columns[-1]
#X and y values
X = boston_df[features].values
y = boston_df[target].values
#splot
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=17)
print("The dimension of X_train is {}".format(X_train.shape))
print("The dimension of X_test is {}".format(X_test.shape))
#Scale features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Output:
```
The dimension of x_train is(354,11)
The dimension of x_test is(152,11)
```

Lasso Regression

Selecting Optimal Alpha Values Using Cross-Validation in Sklearn

We may need to try out different alpha values to find the optimal constraint value. For this case, we can use the cross-validation model in the sklearn package. This will try out different combinations of alpha values and then choose the best model.

```
#Using the linear CV model
from sklearn.linear_model import LassoCV
#Lasso Cross validation
lasso_cv = LassoCV(alphas = [0.0001, 0.001,0.01, 0.1, 1, 10], random_state=0).fit(X_train, y_train)
```

```
#score
print(lasso_cv.score(X_train, y_train))
print(lasso_cv.score(X_test, y_test))
```

Output: The train score for lasso model is 0.78591
The test score for lasso model is 0.76723

Ridge Regression

```
#Ridge Regression Model
ridgeReg = Ridge(alpha=10)
ridgeReg.fit(X_train,y_train)
#train and test score for ridge regression
train_score_ridge = ridgeReg.score(X_train, y_train)
test_score_ridge = ridgeReg.score(X_test, y_test)
print("The train score for ridge model is {}".format(train_score_ridge))
print("The test score for ridge model is {}".format(test_score_ridge))
```

Output: The train score ridge model is 0.78442
The test score for ridge model is 0.76967

Using an alpha value of 10, the evaluation of the model, the train, and test data indicate better performance on the ridge model.

References for above topics:

- https://www.geeksforgeeks.org/regularization-in-machine-learning/
- https://www.javatpoint.com/regularization-in-machine-learning

Classification:

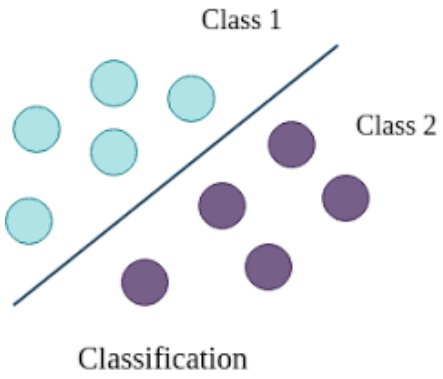
As we know, the Supervised Machine Learning algorithm can be broadly classified into Regression and Classification Algorithms. In Regression algorithms, we have predicted the output for continuous values, but to predict the categorical values, we need Classification algorithms.

The Classification algorithm is a Supervised Learning technique that is used to identify the category of new observations on the basis of training data. In Classification, a program learns from the given dataset or observations and then classifies new observations into a number of classes or groups. Such as, **Yes or No, 0 or 1, Spam or Not Spam, cat or dog,** etc. Classes can be called as targets/labels or categories.

Unlike regression, the output variable of Classification is a category, not a value, such as "Green or Blue", "fruit or animal", etc. Since the Classification algorithm is a Supervised learning technique, hence it takes labeled input data, which means it contains input with the corresponding output.

In the classification algorithm, a discrete output function(y) is mapped to the input variable(x).

y=f(x), where y = categorical output



Types of Classification tasks in machine learning:

Binary classification:

Binary is a type of problem in classification in machine learning that has only two possible outcomes. For example, yes or no, true or false, spam or not spam, etc. Some common binary classification algorithms are logistic regression, decision trees, simple bayes, and support vector machines.

Multi-Class Classification:

Multi-class is a type of classification problem with more than two outcomes and does not have the concept of normal and abnormal outcomes. Here each outcome is assigned to only one label. For example, classifying images, classifying species, and categorizing faces, among others. Some common multi-class algorithms are choice trees, progressive boosting, nearest k neighbors, and rough forest.

Multi-Label Classification:

Multi-label is a type of classification problem that may have more than one class label assigned to the data. Here the model will have multiple outcomes. For example, a book

or a movie can be categorized into multiple genres, or an image can have multiple objects. Some common multi-label algorithms are multi-label decision trees, multi-label gradient boosting, and multi-label random forests.

Imbalance Classification:

Most machine learning algorithms assume equal data distribution. When the data distribution is not equal, it leads to imbalance. An imbalanced classification problem is a classification problem where the distribution of the dataset is skewed or biased. This method employs specialized techniques to change the composition of data samples. Some examples of imbalanced classification are spam filtering, disease screening, and fraud detection.

Learners in classification problems:

1. **Lazy Learners:** Lazy Learner firstly stores the training dataset and waits until it receives the test dataset. In the Lazy learner case, classification is done on the basis of the most related data stored in the training dataset. It takes less time in training but more time for predictions.
2. **Eager Learners:**Eager Learners develop a classification model based on a training dataset before receiving a test dataset. Opposite to Lazy learners, Eager Learner takes more time in learning, and less time in prediction.

Types of ML Classification Algorithms:

Classification Algorithms can be further divided into the Mainly two category:
Linear Models:

- Logistic Regression
- Support Vector Machines

Non-linear Models:

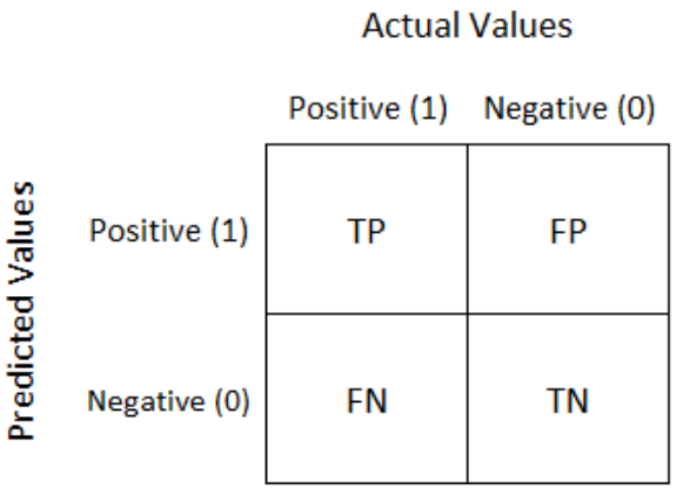
- K-Nearest Neighbours
- Kernel SVM
- Naive Bayes
- Decision Tree Classification

- Random Forest Classification

Evaluation metrics for classification:

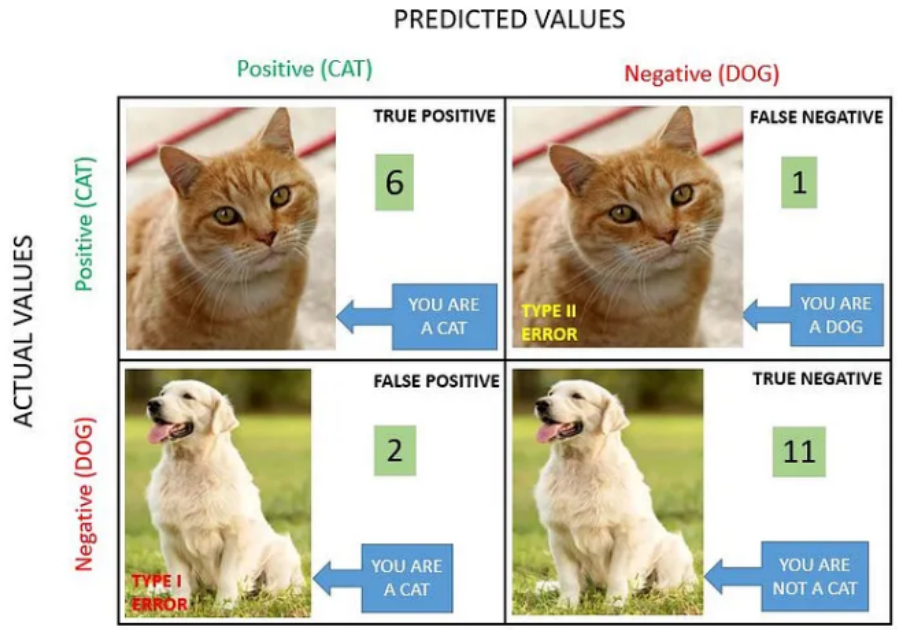
Confusion matrix:

Confusion Matrix is a performance measurement for the machine learning classification problems where the output can be two or more classes. It is a table with combinations of predicted and actual values.



It is extremely useful for measuring the Recall, Precision, Accuracy, and AUC-ROC curves.

Let's try to understand TP, FP, FN, TN with an example of pregnancy analogy.



True Positive: We predicted positive and it's true. In the image, we predicted that cat as a cat and it was actually correct.

True Negative: We predicted negative and it's true. In the image, we predicted that the dog was not a cat and it was actually correct.

False Positive (Type 1 Error)- We predicted positive and it's false. In the image, we predicted the dog as a cat but it was actually incorrect.

False Negative (Type 2 Error)- We predicted negative and it's false. In the image, we predicted the cat as a dog but it was actually incorrect.

Accuracy:

Accuracy is the most basic evaluation metric and represents the ratio of correctly classified instances to the total number of instances in the dataset.

It is defined as:

$$\text{Accuracy}=\frac{TP+TN}{TP+TN+FP+FN}$$

Here,

TP: True Positive (Correctly predicted positive instances)

TN: True Negative (Correctly predicted negative instances)
FP: False Positive (Incorrectly predicted positive instances)
FN: False Negative (Incorrectly predicted negative instances)

Precision:
Precision explains how many of the correctly predicted cases actually turned out to be positive. Precision is useful in the cases where False Positive is a higher concern than False Negatives.

$$\text{Precision} = \frac{TP}{TP+FP}$$

Recall(Sensitivity):
Recall explains how many of the actual positive cases we were able to predict correctly with our model. It is a useful metric in cases where False Negative is of higher concern than False Positive.

$$\text{Recall} = \frac{TP}{TP+FN}$$

F1-Score:
The F1-Score is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall.

$$\text{F1-Score} = 2 \times \frac{Precision \times Recall}{Precision+Recall}$$

The F1 score punishes extreme values more. F1 Score could be an effective evaluation metric in the following cases:

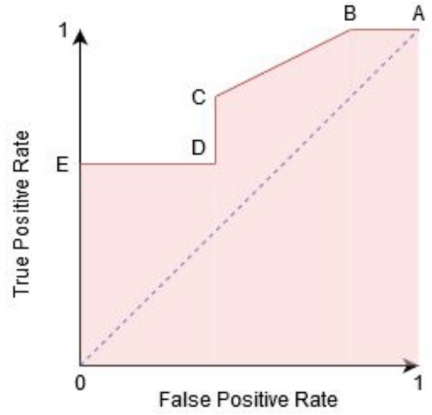
- When FP and FN are equally costly.
- Adding more data doesn't effectively change the outcome
- True Negative is high

AUC-ROC:

The Receiver Operator Characteristic (ROC) is a probability curve that plots the TPR(True Positive Rate) against the FPR(False Positive Rate) at various threshold values and separates the ‘signal’ from the ‘noise’.

The Area Under the Curve (AUC) is the measure of the ability of a classifier to distinguish between classes. From the graph, we simply say the area of the curve ABDE and the X and Y-axis.

From the graph shown below, the greater the AUC, the better is the performance of the model at different threshold points between positive and negative classes. This simply means that When AUC is equal to 1, the classifier is able to perfectly distinguish between all Positive and Negative class points. When AUC is equal to 0, the classifier would be predicting all Negatives as Positives and vice versa. When AUC is 0.5, the classifier is not able to distinguish between the Positive and Negative classes.



Working of AUC—In a ROC curve, the X-axis value shows False Positive Rate (FPR), and Y-axis shows True Positive Rate (TPR). Higher the value of X means higher the number of False Positives(FP) than True Negatives(TN), while a higher Y-axis value indicates a higher number of TP than FN. So, the choice of the threshold depends on the ability to balance between FP and FN.

Specitivity:

Specificity is the ratio of true negatives to the sum of true negatives and false positives. It measures the model’s ability to identify negative instances.

$$\text{Specificity}=\frac{TN}{TN+FP}$$

Log Loss or Cross-Entropy Loss:

- It is used for evaluating the performance of a classifier, whose output is a probability value between 0 and 1.
- For a good binary Classification model, the value of log loss should be near to 0.
- The value of log loss increases if the predicted value deviates from the actual value.
- The lower log loss represents the higher accuracy of the model.

$$-(y\log(p)+(1-y)\log(1-p))$$

References for above topics:

- https://www.analyticsvidhya.com/blog/2021/07/metrics-to-evaluate-your-classification-model-to-take-the-right-decisions/
- https://www.javatpoint.com/classification-algorithm-in-machine-learning

Logistic regression:

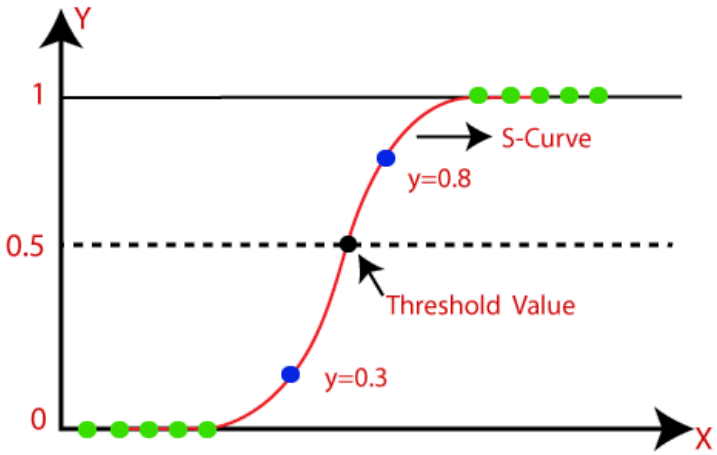
Logistic Regression is much similar to Linear Regression except that how they are used. Linear Regression is used for solving Regression problems, whereas Logistic regression is used for solving the classification problems.

In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function, which predicts two maximum values (0 or 1).

Logistic regression predicts the output of a categorical dependent variable. Therefore the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, True or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1.

Logistic Regression can be used to classify the observations using different types of data and can easily determine the most effective variables used for the classification.

The below image is showing the logistic function:



Note:Logistic regression uses the concept of predictive modelling as regression; therefore, it is called logistic regression, but is used to classify samples; Therefore, it falls under the classification algorithm.

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.

The value of the logistic regression must be between 0 and 1, which cannot go beyond this limit, so it forms a curve like the "S" form. The S-form curve is called the Sigmoid function or the logistic function.

In logistic regression, we use the concept of the threshold value, which defines the probability of either 0 or 1. Such as values above the threshold value tends to 1, and a value below the threshold values tends to 0.

Assumptions of Logistic Regression:

- The dependent variable must be categorical in nature.
- The independent variable should not have multi-collinearity.

Logistic regression equation:

The Logistic regression equation can be obtained from the Linear Regression equation. The mathematical steps to get Logistic Regression equations are given below:

- We know the equation of the straight line can be written as:

  $$y = b_0 + b_1 x_1 + b_2 x_2 + \ldots + b_n x_n$$

- In Logistic Regression y can be between 0 and 1 only, so for this let's divide the above equation by (1-y):

  $\frac{y}{1-y}$; 0 for y=0, and infinity for y=1.

- But we need range between -[infinity] to +[infinity], then take logarithm of the equation it will become:

  $$log[\frac{y}{1-y}] = b_0 + b_1x_1 + b_2x_2 + ... + b_nx_n$$

The above equation is the final equation for Logistic Regression.

Type of logistic regression:

On the basis of the categories, Logistic Regression can be classified into three types:

- **Binomial:** In binomial Logistic regression, there can be only two possible types of the dependent variables, such as 0 or 1, Pass or Fail, etc.
- **Multinomial:** In multinomial Logistic regression, there can be 3 or more possible unordered types of the dependent variable, such as "cat", "dogs", or "sheep"
- **Ordinal:** In ordinal Logistic regression, there can be 3 or more possible ordered types of dependent variables, such as "low", "Medium", or "High".

Example:

Model building using scikit-learn

Let's build a diabetes prediction model,

```
#import pandas
import pandas as pd

col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age', 'label']

# load dataset
pima = pd.read_csv("data/diabetes.csv", header=1, names=col_names)
```

```
pima.head()
```

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	1	85	66	29	0	26.6	0.351	31	0
1	8	183	64	0	0	23.3	0.672	32	1
2	1	89	66	23	94	28.1	0.167	21	0
3	0	137	40	35	168	43.1	2.288	33	1
4	5	116	74	0	0	25.6	0.201	30	0

5 rows

Selecting Feature

Here, you need to divide the given columns into two types of variables dependent(or target variable) and independent variable(or feature variables).

```
#split dataset in features and target variable
feature_cols = ['pregnant', 'insulin', 'bmi', 'age','glucose','bp','pedigree']
X = pima[feature_cols] # Features
y = pima.label # Target variable
```

Splitting Data

```
# split X and y into training and testing sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=16)
```

Model Development and Prediction

```
X.head()
```

	pregnant	insulin	bmi	age	glucose	bp	pedigree
0	1	0	26.6	31	85	66	0.351
1	8	0	23.3	32	183	64	0.672
2	1	94	28.1	21	89	66	0.167
3	0	168	43.1	33	137	40	2.288
4	5	0	25.6	30	116	74	0.201

5 rows

```
# import the class
from sklearn.linear_model import LogisticRegression

# instantiate the model (using the default parameters)
logreg = LogisticRegression(random_state=16)

# fit the model with data
logreg.fit(X_train, y_train)

y_pred = logreg.predict(X_test)
```

Model evaluation using confusion matrix

```
# import the metrics class
from sklearn import metrics

cnf_matrix = metrics.confusion_matrix(y_test, y_pred)
cnf_matrix
```

```
array([[115,   8],
       [ 30,  39]])
```

The dimension of this matrix is 2*2 because this model is binary classification. You have two classes 0 and 1. Diagonal values represent accurate predictions, while non-diagonal elements are inaccurate predictions. In the output, 115 and 39 are actual predictions, and 30 and 8 are incorrect predictions.

Confusion matrix evaluation metrics

```
from sklearn.metrics import classification_report
target_names = ['without diabetes', 'with diabetes']
print(classification_report(y_test, y_pred, target_names=target_names))
```

```
                precision    recall  f1-score   support

without diabetes       0.79      0.93      0.86       123
   with diabetes       0.83      0.57      0.67        69

        accuracy                           0.80       192
       macro avg       0.81      0.75      0.77       192
    weighted avg       0.81      0.80      0.79       192
```

Well, you got a classification rate of 80%, considered as good accuracy.

Precision: Precision is about being precise, i.e., how accurate your model is. In other words, you can say, when a model makes a prediction, how often it is correct. In your prediction case, when your Logistic Regression model predicted patients are going to suffer from diabetes, that patients have 83% of the time.

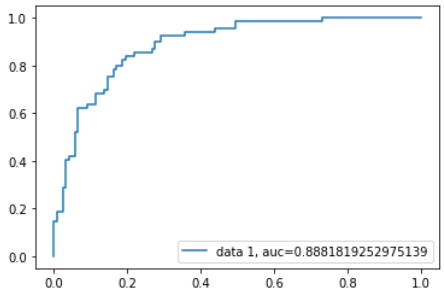
Recall: If there are patients who have diabetes in the test set and your Logistic Regression model can identify it 57% of the time.

ROC Curve

Receiver Operating Characteristic(ROC) curve is a plot of the true positive rate against the false positive rate. It shows the tradeoff between sensitivity and specificity.

```
y_pred_proba = logreg.predict_proba(X_test)[::,1]
fpr, tpr, _ = metrics.roc_curve(y_test,  y_pred_proba)
auc = metrics.roc_auc_score(y_test, y_pred_proba)

plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.show()
```



AUC score for the case is 0.89. AUC score 1 represents a perfect classifier, and 0.5 represents a worthless classifier.

References for above topic:

- https://www.javatpoint.com/logistic-regression-in-machine-learning
- https://www.analyticsvidhya.com/blog/2021/10/building-an-end-to-end-logistic-regression-model/

K-Nearest Neighbor(KNN):

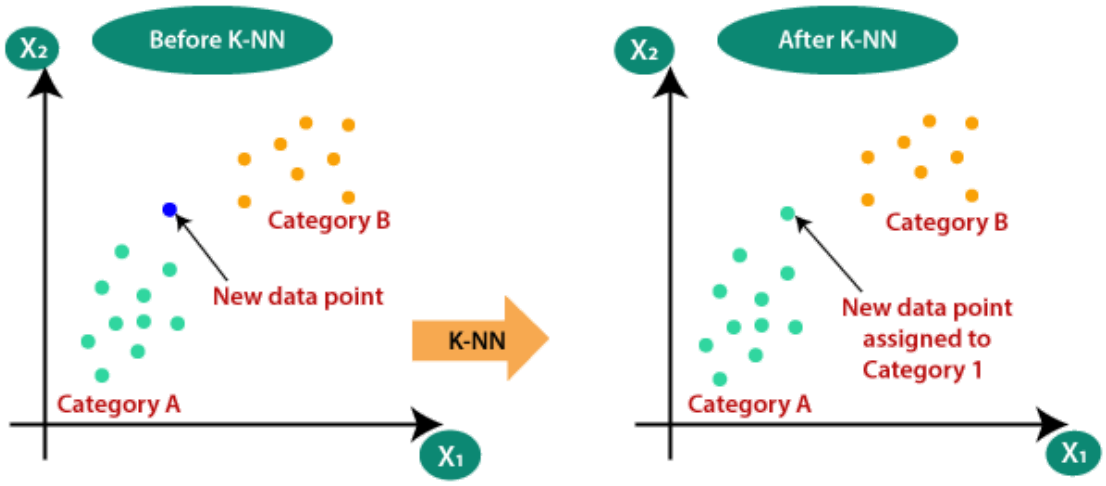
- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- K-NN algorithm assumes the similarity between the new case/data and available cases and puts the new case into the category that is most similar to the available categories.
- K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suited category by using K- NN algorithm.
- K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
- K-NN is a **non-parametric algorithm**, which means it does not make any assumption on underlying data.
- It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
- K-NN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.

Why do we need a K-NN algorithm?:

Suppose there are two categories, i.e., Category A and Category B, and we have a new data point x1, so this data point will lie in which of these categories. To solve this type

of problem, we need a K-NN algorithm. With the help of K-NN, we can easily identify the category or class of a particular dataset.

Consider the below diagram:

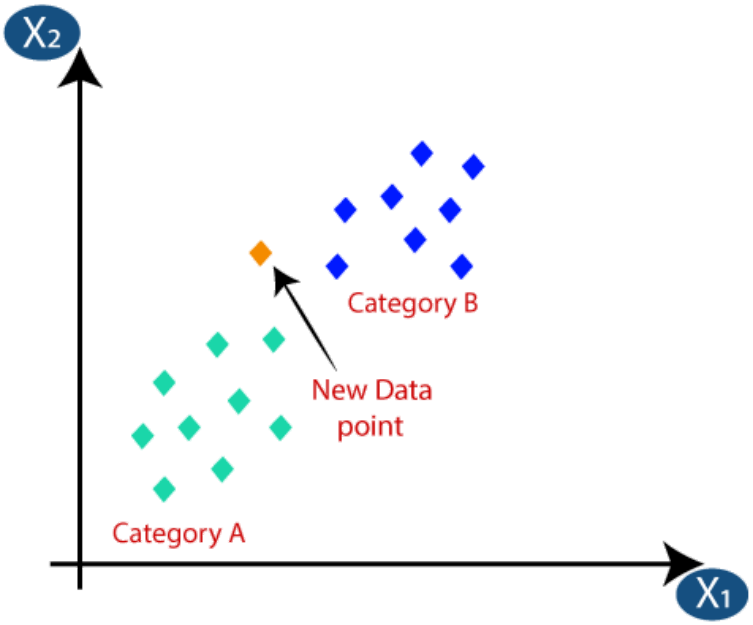


Working of K-NN:

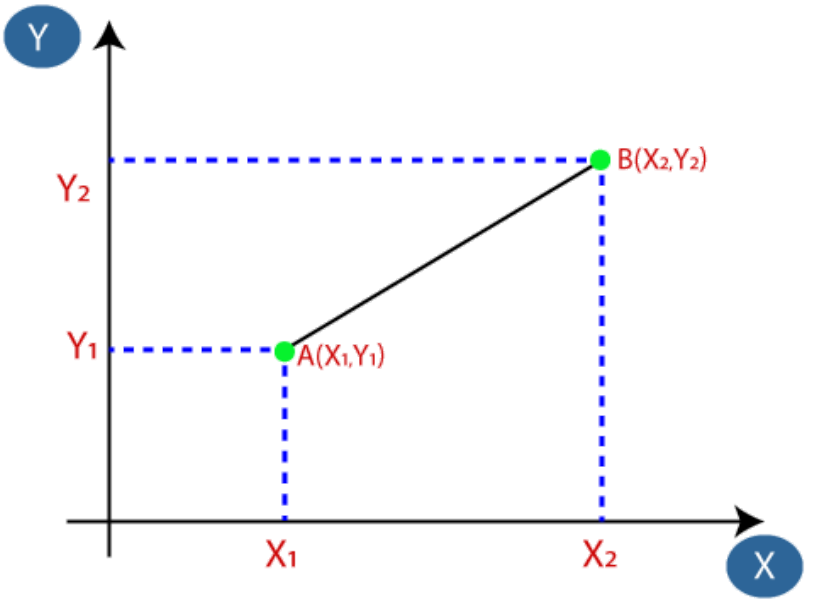
The K-NN working can be explained on the basis of the below algorithm:

- **Step-1:** Select the number K of the neighbors
- **Step-2:** Calculate the Euclidean distance of **K number of neighbors**
- **Step-3:** Take the K nearest neighbors as per the calculated Euclidean distance.
- **Step-4:** Among these k neighbors, count the number of the data points in each category.
- **Step-5:** Assign the new data points to that category for which the number of the neighbor is maximum.
- **Step-6:** Our model is ready.

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

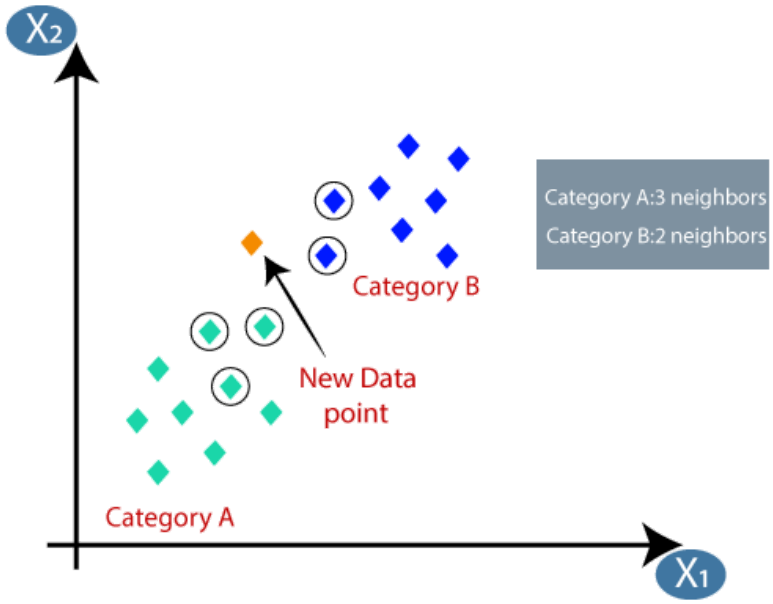


- Firstly, we will choose the number of neighbors, so we will choose the k=5.
- Next, we will calculate the **Euclidean distance** between the data points. The Euclidean distance is the distance between two points, which we have already studied in geometry. It can be calculated as:



Euclidean Distance between A_1 and $B_2 = \sqrt{(X_2-X_1)^2+(Y_2-Y_1)^2}$

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



As we can see the 3 nearest neighbors are from category A, hence this new data point must belong to category A.

How to select the K value in K-NN algorithm?:

- There is no particular way to determine the best value for "K", so we need to try some values to find the best out of them. The most preferred value for K is 5.
- A very low value for K such as K=1 or K=2, can be noisy and lead to the effects of outliers in the model.
- Large values for K are good, but it may have some difficulties.

Advantages of K-NN:

- It is simple to implement.
- It is robust to the noisy training data
- It can be more effective if the training data is large.

Disadvantages of K-NN:

- Always needs to determine the value of K which may be complex some time.
- The computation cost is high because of calculating the distance between the data points for all the training samples.

Example:

Model Building

Here is the simplistic code for fitting the K-NN model using the Sklearn IRIS dataset. Pay attention to some of the following in the code given below:

- Sklearn.model_selection **train_test_split** is used for creating training and test split data
- Sklearn.preprocessing **StandardScaler (fit & transform** method) is used for feature scaling.
- Sklearn.neighbors KNeighborsClassifier is used as implementation for the K-nearest neighbors algorithm for fitting the model. The following are some important parameters for K-NN algorithm:
    - **n_neighbors**: Number of neighbors to use
    - **weights**: Weight is used to associate the weight assigned to points in the neighborhood. If the value of weights is “uniform”, it means that all points in each neighborhood are weighted equally. If the value of weights is “distance”, it means that closer neighbors of a query point will have a greater influence than neighbors which are further away.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV
#
# Load the Sklearn IRIS dataset
#
iris = datasets.load_iris()
X = iris.data
y = iris.target
#
# Create train and test split
#
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)
#
# Feature Scaling using StandardScaler
#
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
#
# Fit the model
#
knn = KNeighborsClassifier(n_neighbors=5, p=2, weights='uniform',
algorithm='auto')
knn.fit(X_train_std, y_train)
#
# Evaluate the training and test score
#
print('Training accuracy score: %.3f' % knn.score(X_train_std, y_train))
print('Test accuracy score: %.3f' % knn.score(X_test_std, y_test))
```

The model score for the training data set comes out to be 0.981 and the test data set is 0.911.

References for above topic :

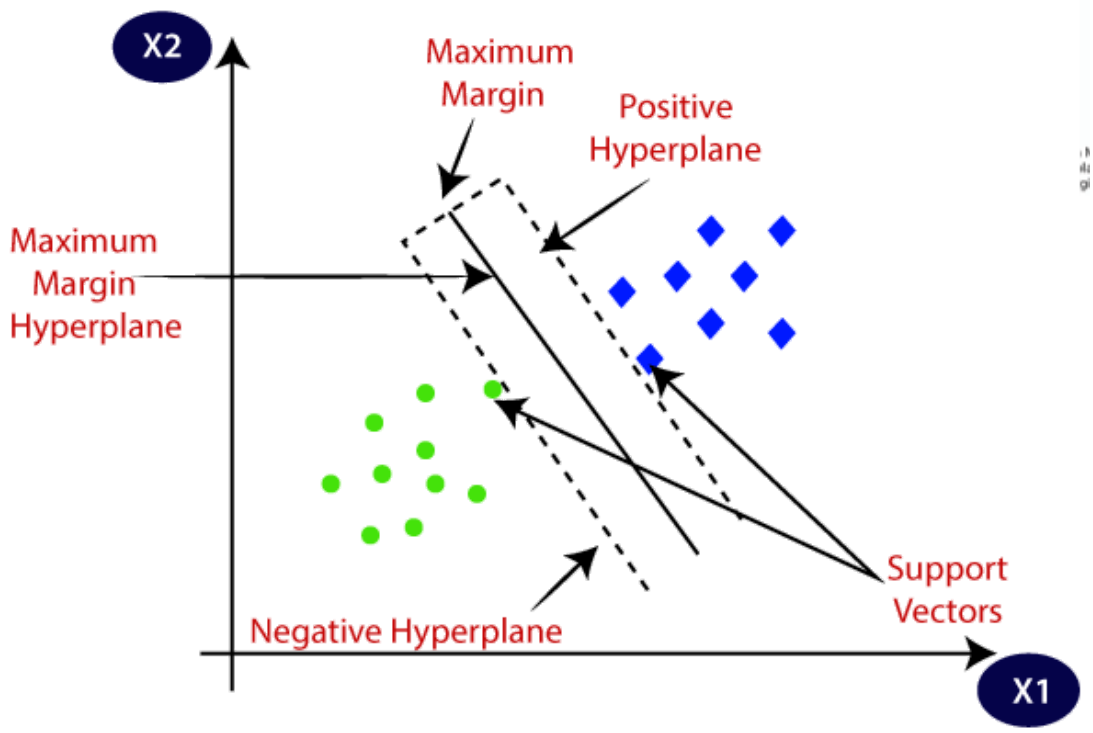
https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning

Support vector machine(SVM):

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called support vectors, and hence the algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:



Types of SVM:

Linear SVM:

When the data is perfectly linearly separable only then we can use Linear SVM. Perfectly linearly separable means that the data points can be classified into 2 classes by using a single straight line(if 2D).

Non-Linear SVM:

When the data is not linearly separable then we can use Non-Linear SVM, which means when the data points cannot be separated into 2 classes by using a straight line (if 2D) then we use some advanced techniques like kernel tricks to classify them. In most real-world applications we do not find linearly separable data points hence we use kernel tricks to solve them.

Now let's define two main terms which will be repeated again and again in this article:

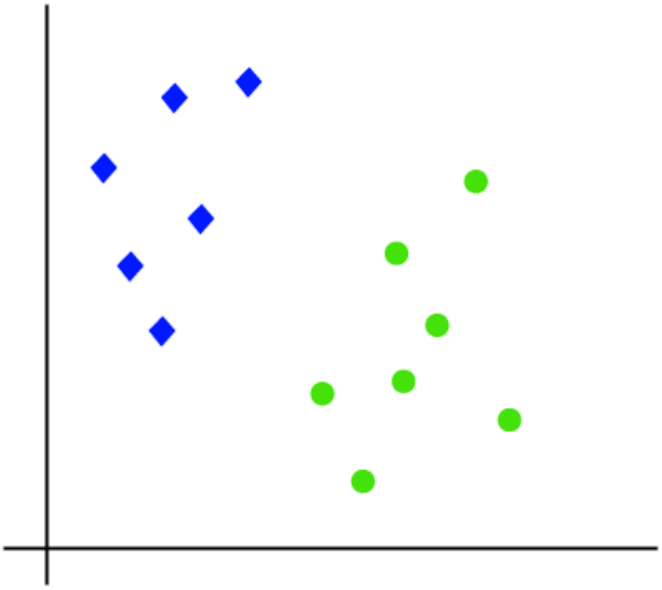
Support Vectors: These are the points that are closest to the hyperplane. A separating line will be defined with the help of these data points.

Margin: it is the distance between the hyperplane and the observations closest to the hyperplane (support vectors). In SVM large margin is considered a good margin. There are two types of margins: hard margin and soft margin.

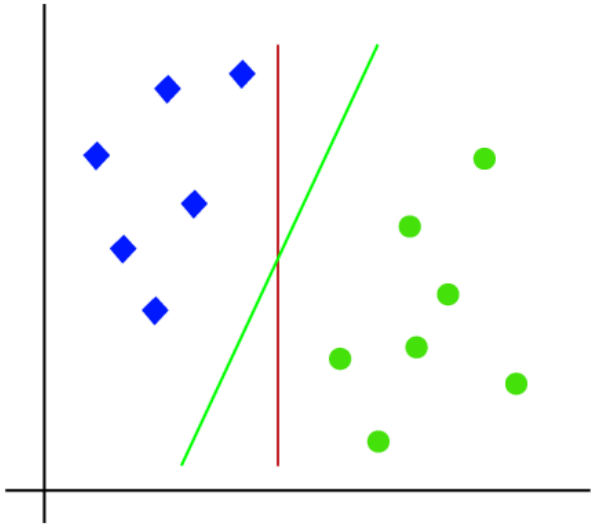
Working on SVM:

Linear SVM:

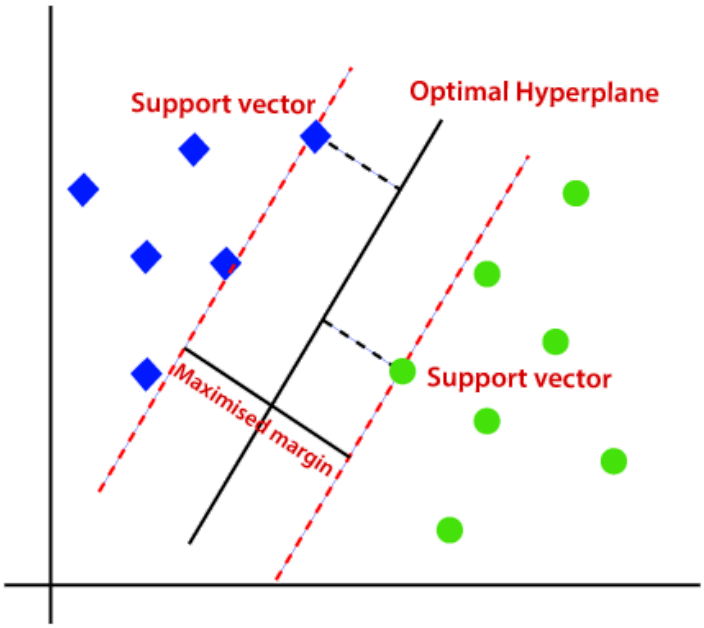
The working of the SVM algorithm can be understood by using an example. Suppose we have a dataset that has two tags (green and blue), and the dataset has two features x1 and x2. We want a classifier that can classify the pair(x1, x2) of coordinates in either green or blue. Consider the below image:



So as it is 2-d space so by just using a straight line, we can easily separate these two classes. But there can be multiple lines that can separate these classes. Consider the below image:

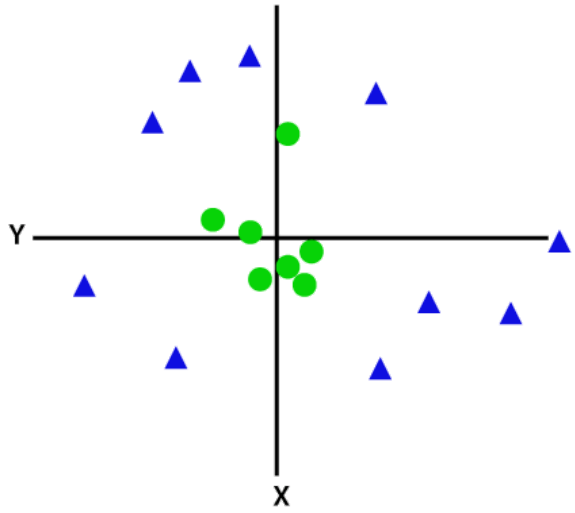


Hence, the SVM algorithm helps to find the best line or decision boundary; this best boundary or region is called a **hyperplane**. SVM algorithm finds the closest point of the lines from both the classes. These points are called support vectors. The distance between the vectors and the hyperplane is called the margin. And the goal of SVM is to maximize this margin. The **hyperplane** with maximum margin is called the **optimal hyperplane**.



Non-Linear SVM:

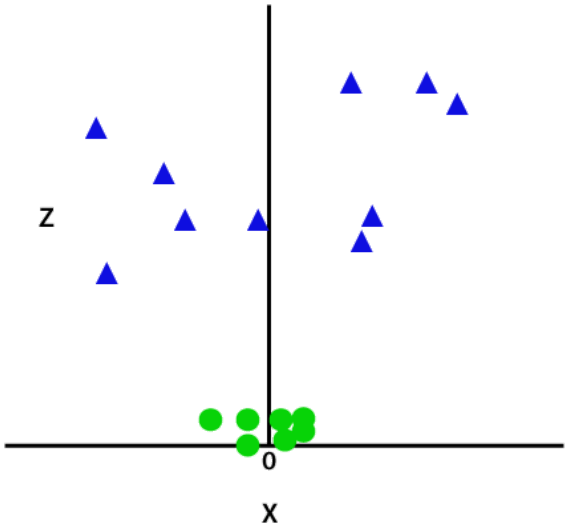
If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we cannot draw a single straight line. Consider the below image:



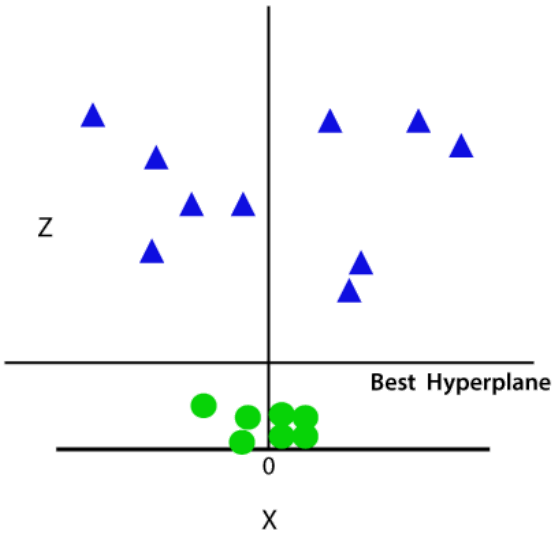
So to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third dimension z. It can be calculated as:

$$z=x^2+y^2$$

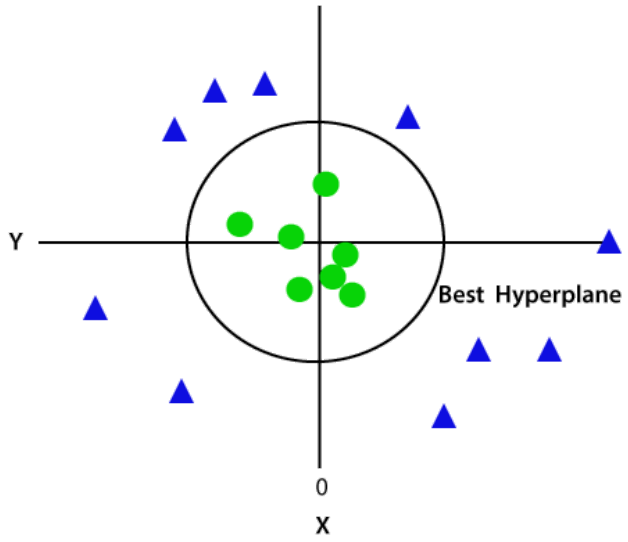
By adding the third dimension, the sample space will become as below image:



So now, SVM will divide the datasets into classes in the following way. Consider the below image:



Since we are in 3-d Space, hence it is looking like a plane parallel to the x-axis. If we convert it in 2d space with z=1, then it will become as:



Hence we get a circumference of radius 1 in case of non-linear data.

Mathematical Intuition of SVM:

Consider a binary classification problem where we have a dataset with two classes, labeled as -1 and +1. We want to find a hyperplane in the feature space that separates the two classes as effectively as possible.

1. Hyperplane Equation:

In a d-dimensional feature space, the hyperplane is defined by the equation:

$w_0 + w_1x_1 + w_2x_2 + ... + w\_dxd = 0$,

where (x_1, x_2, ..., x_d) are the coordinates of a data point, w_0 is the bias term, and (w_1, w_2, ..., w_d) is the normal vector to the hyperplane.

2. Margin and Support Vectors:

The goal of SVM is to find the hyperplane with the maximum margin, which is the perpendicular distance between the hyperplane and the nearest data points of each class. These data points are called support vectors. The margin is denoted as 2d, where d is the distance between the hyperplane and the support vectors.

3. Optimization Problem:

To find the optimal hyperplane, SVM formulates an optimization problem. The objective is to maximize the margin while maintaining the correct classification of training data. Mathematically, we want to maximize 2d, which is equivalent to maximizing d.

4. Constraints:

SVM introduces constraints to ensure that the data points are classified correctly. The following constraints are enforced:

- For data points belonging to class -1, the hyperplane equation should satisfy $w_0 + w_1x_1 + w_2x_2 + ... + w\_dxd \leq -1$.

- For data points belonging to class +1, the hyperplane equation should satisfy $w_0 + w_1x_1 + w_2x_2 + ... + w\_dxd \geq +1$.

These constraints ensure that the data points are classified on the correct side of the hyperplane, with a margin of at least 1.

5. Optimization Objective:

Combining the constraints, the optimization objective becomes:

Minimize: $\frac{1}{2}||w||^2$,

subject to: $y_i(w_0 + w_1x_{1i} + w_2x_{2i} + ... + w\_dxd_i) \geq 1$ for all training examples (x_{1i}, x_{2i}, ..., x_d_i),

where y_i represents the class label (-1 or +1) of the i-th training example.

6. Lagrange Multipliers:

To solve this optimization problem, SVM introduces Lagrange multipliers (α_i) for each training example. The Lagrangian function is defined as:

$L(w, w_0, \alpha) = \frac{1}{2}||w||^2 - \sum\alpha_i[y_i(w_0 + w_1x_{1i} + w_2x_{2i} + ... + w\_dxd_i) - 1]$,

where α is the vector of Lagrange multipliers.

7. Dual Optimization Problem:

By applying optimization techniques, the original problem can be transformed into its dual form. The objective becomes maximizing the dual function:

$W(\alpha) = \sum\alpha_i - \frac{1}{2}\sum\sum\alpha_i\alpha\square y_iy\square(x_{1i}x_{1}\square + x_{2i}x_{2}\square + ... + x\_d_ix\_d\square)$,

subject to: $\sum\alpha_iy_i = 0$, and $\alpha_i \geq 0$ for all training examples.

8. Support Vectors:

Solving the dual problem yields

the values of the Lagrange multipliers (α_i). The non-zero α_i values correspond to the support vectors that lie on or within the margin.

9. Classification:

Once we have the α_i values, we can compute the normal vector w as the weighted sum of the support vectors:

$w = \sum \alpha_i y_i (x_{1i}, x_{2i}, ..., x\_d_i)$.

10. Prediction:

To classify a new data point $(x_1, x_2, ..., x\_d)$, we compute its distance to the hyperplane:

$f(x_1, x_2, ..., x\_d) = w_0 + w_1x_1 + w_2x_2 + ... + w\_dxd$.

If $f(x_1, x_2, ..., x\_d) > 0$, the point belongs to class +1; otherwise, it belongs to class -1.

That's the mathematical intuition behind the Support Vector Machine algorithm. By solving the dual optimization problem, SVM finds the optimal hyperplane that maximizes the margin and effectively separates the classes in the feature space.

Example:

Model building

Python3

```
# Import necessary libraries
from sklearn import svm
from sklearn import datasets

# Load the Iris dataset
iris = datasets.load_iris()
# We only take the first two
# features for simplicity
X = iris.data[:, :2]
y = iris.target
```

Now we will use SupportVectorClassifier as currently we are dealing with a classification problem.

Python3

```
# Fit the SVM model
model = svm.SVC(kernel='linear')
model.fit(X, y)

# Predict using the SVM model
predictions = model.predict(X)

# Evaluate the predictions
accuracy = model.score(X, y)
print("Accuracy of SVM:", accuracy)
```

Output :

```
Accuracy of SVM: 0.82
```

The code first imports the necessary modules and libraries, including the SVM module from Scikit-learn and the Iris dataset from Scikit-learn's datasets module. Then, it loads the Iris dataset and extracts the first two features from each example (sepal length and width), as well as the target labels (the species of the flower).

Next, the code creates an SVM model using the SVC class from Scikit-learn, and specifies that it should use a linear kernel. Then, it trains the model on the data using the fit() method, and makes predictions on the same data using the predict() method. Finally, it evaluates the predictions by computing the accuracy of the model (the fraction of examples that were predicted correctly) using the score() method. The accuracy is then printed to the console.

Advantages: SVM Classifiers offer good accuracy and perform faster prediction compared to Naïve Bayes algorithm. They also use less memory because they use a subset of training points in the decision phase. SVM works well with a clear margin of separation and with high dimensional space.

Disadvantages: SVM is not suitable for large datasets because of its high training time and it also takes more time in training compared to Naïve Bayes. It works poorly with overlapping classes and is also sensitive to the type of kernel used.

References for above topic:

https://www.javatpoint.com/machine-learning-support-vector-machine-algorithm

Naïve Bayes Classifier Algorithm:

- Naive Bayes classifiers are a collection of classification algorithms based on **Bayes' Theorem**. It is not a single algorithm but a family of algorithms where all of them share a common principle, i.e. every pair of features being classified is independent of each other.
- It is mainly used in *text classification* that includes a high-dimensional training dataset.
- Naïve Bayes Classifier is one of the simplest and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.
- It is a probabilistic classifier, which means it predicts on the basis of the probability of an object.
- Some popular examples of Naïve Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.

Why is it called Naïve Bayes?:

The Naïve Bayes algorithm is comprised of two words Naïve and Bayes, Which can be described as:

- **Naïve**: It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the basis of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other.
- **Bayes**: It is called Bayes because it depends on the principle of Bayes' Theorem.

Bayes' Theorem:

Bayes' Theorem finds the probability of an event occurring given the probability of another event that has already occurred. Bayes' theorem is stated mathematically as the following equation:

$$P\left(\frac{A}{B}\right) = \frac{P\left(\frac{B}{A}\right)P(A)}{P(B)}$$

where A and B are events and P(B) ≠ 0.

- Basically, we are trying to find the probability of event A, given that event B is true. Event B is also termed as **evidence**.
- P(A) is the **priori** of A (the prior probability, i.e. Probability of event before evidence is seen). The evidence is an attribute value of an unknown instance(here, it is event B).
- P(A|B) is a posteriori probability of B, i.e. probability of event after evidence is seen.

Working of Naïve Bayes' Classifier:

Working of Naïve Bayes' Classifier can be understood with the help of the below example:

Suppose we have a dataset of **weather conditions** and corresponding target variable "**Play**". So using this dataset we need to decide whether we should play or not on a particular day according to the weather conditions. So to solve this problem, we need to follow the below steps:

1. Convert the given dataset into frequency tables.
2. Generate Likelihood table by finding the probabilities of given features.
3. Now, use Bayes theorem to calculate the posterior probability.

Problem: If the weather is sunny, then the Player should play or not?

Solution: To solve this, first consider the below dataset:

Index	Outlook	Play
0	Rainy	Yes
1	Sunny	Yes
2	Overcast	Yes
3	Overcast	Yes
4	Sunny	No

5	Rainy	Yes
6	Sunny	Yes
7	Overcast	Yes
8	Rainy	No
9	Sunny	No
10	Sunny	Yes
11	Rainy	No
12	Overcast	Yes
13	Overcast	Yes

Frequency table for the Weather Conditions:

Weather	Yes	No
Overcast	5	0
Rainy	2	2
Sunny	3	2
Total	10	5

Likelihood table weather condition:

Weather	No	Yes	
Overcast	0	5	5/14= 0.35

Rainy	2	2	4/14=0.29
Sunny	2	3	5/14=0.35
All	4/14=0.29	10/14=0.71	

Applying Bayes'theorem:

P(Yes|Sunny)= P(Sunny|Yes)*P(Yes)/P(Sunny)

P(Sunny|Yes)= 3/10= 0.3

P(Sunny)= 0.35

P(Yes)=0.71

So P(Yes|Sunny) = 0.3*0.71/0.35= **0.60**

P(No|Sunny)= P(Sunny|No)*P(No)/P(Sunny)

P(Sunny|NO)= 2/4=0.5

P(No)= 0.29

P(Sunny)= 0.35

So P(No|Sunny)= 0.5*0.29/0.35 = **0.41**

So as we can see from the above calculation that **P(Yes|Sunny)>P(No|Sunny)**

Hence on a Sunny day, Players can play the game.

Types of Naïve Bayes Model:

There are three types of Naive Bayes Model, which are given below:

- **Gaussian:** The Gaussian model assumes that features follow a normal distribution. This means if predictors take continuous values instead of discrete, then the model assumes that these values are sampled from the Gaussian distribution.
- **Multinomial:** The Multinomial Naïve Bayes classifier is used when the data is multinomial distributed. It is primarily used for document classification problems, it means a particular document belongs to which category such as

Sports, Politics, education, etc.
The classifier uses the frequency of words for the predictors.

- **Bernoulli:** The Bernoulli classifier works similar to the Multinomial classifier, but the predictor variables are the independent Booleans variables. Such as if a particular word is present or not in a document. This model is also famous for document classification tasks.

Example:

Model building

In this example, we will be loading **Loan Data** using pandas '**read_csv**` function.

```
import pandas as pd
df = pd.read_csv('loan_data.csv')
df.head()
```

Data Exploration

To understand more about the dataset we will use `.info()`.

- The dataset consists of 14 columns and 9578 rows.
- Apart from “purpose”, columns are either floats or integers.
- Our target column is “not.fully.paid”.

```
df.info()
RangeIndex: 9578 entries, 0 to 9577
Data columns (total 14 columns):
#   Column             Non-Null Count  Dtype
---  ------             --------------  -----
0   credit.policy      9578 non-null   int64
1   purpose            9578 non-null   object
2   int.rate           9578 non-null   float64
3   installment        9578 non-null   float64
4   log.annual.inc     9578 non-null   float64
5   dti                9578 non-null   float64
6   fico               9578 non-null   int64
7   days.with.cr.line  9578 non-null   float64
8   revol.bal          9578 non-null   int64
9   revol.util         9578 non-null   float64
```

```
10  inq.last.6mths     9578 non-null   int64
11  delinq.2yrs        9578 non-null   int64
12  pub.rec            9578 non-null   int64
13  not.fully.paid     9578 non-null   int64
dtypes: float64(6), int64(7), object(1)
memory usage: 1.0+ MB
```

After that, we will define feature (X) and target (y) variables, and split the dataset into training and testing sets.

```
from sklearn.model_selection import train_test_split

X = pre_df.drop('not.fully.paid', axis=1)
y = pre_df['not.fully.paid']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=125)
```

Model Building and Training

Model building and training is quite simple. We will be training a model on a training dataset using default hyperparameters.

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X_train, y_train);
```

Model Evaluation

We will use accuracy and f1 score to determine model performance, and it looks like the Gaussian Naive Bayes algorithm has performed quite well.

```
from sklearn.metrics import (
    accuracy_score,
    confusion_matrix,
    ConfusionMatrixDisplay,
    f1_score,
    classification_report,)

y_pred = model.predict(X_test)
accuray = accuracy_score(y_pred, y_test)
f1 = f1_score(y_pred, y_test, average="weighted")
```

```
print("Accuracy:", accuray)
print("F1 Score:", f1)

Accuracy: 0.8206263840556786
F1 Score: 0.8686606980013266
```

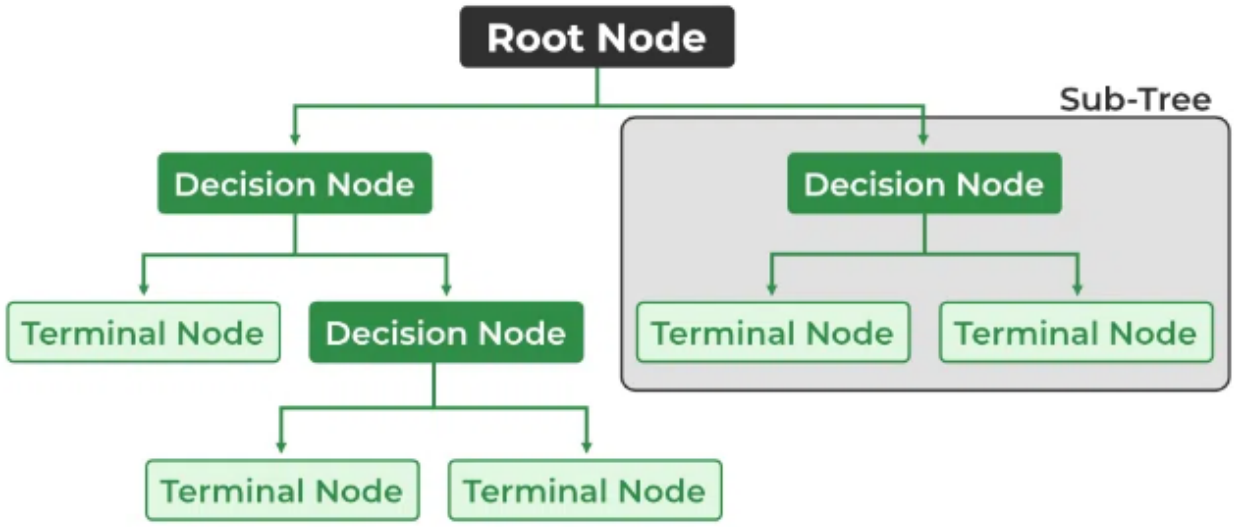
Reference for above topic:

- https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/
- https://www.javatpoint.com/machine-learning-naive-bayes-classifier
- https://www.datacamp.com/tutorial/naive-bayes-scikit-learn

Decision Tree Classifier Algorithm:

- Decision Tree is a **Supervised learning technique** that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where **internal nodes represent the features of a dataset, branches represent the decision rules** and **each leaf node represents the outcome.**
- In a Decision tree, there are two nodes, which are the **Decision Node** and **Leaf Node.** Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.
- The decisions or the test are performed on the basis of features of the given dataset.
- **It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.**
- It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.
- In order to build a tree, we use the **CART algorithm,** which stands for **Classification and Regression Tree algorithm.**
- A decision tree simply asks a question, and based on the answer (Yes/No), it further splits the tree into subtrees.

- Below diagram explains the general structure of a decision tree:



Why use decision trees?:

There are various algorithms in Machine learning, so choosing the best algorithm for the given dataset and problem is the main point to remember while creating a machine learning model. Below are the two reasons for using the Decision tree:

- Decision Trees usually mimic human thinking ability while making a decision, so it is easy to understand.
- The logic behind the decision tree can be easily understood because it shows a tree-like structure.

Decision Tree Terminologies:

Root Node: Root node is from where the decision tree starts. It represents the entire dataset, which further gets divided into two or more homogeneous sets.

Leaf Node: Leaf nodes are the final output node, and the tree cannot be segregated further after getting a leaf node.

Splitting: Splitting is the process of dividing the decision node/root node into sub-nodes according to the given conditions.

Branch/Sub Tree: A tree formed by splitting the tree.

Pruning: Pruning is the process of removing the unwanted branches from the tree.

Parent/Child node: The root node of the tree is called the parent node, and other nodes are called the child nodes.

Working of Decision Tree:

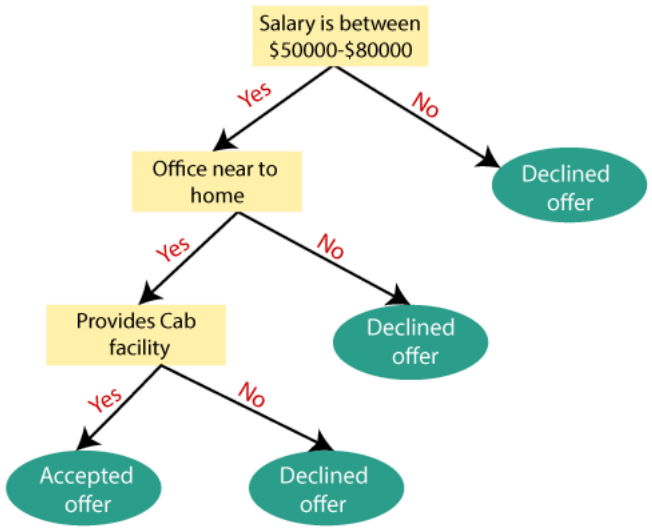
In a decision tree, for predicting the class of the given dataset, the algorithm starts from the root node of the tree. This algorithm compares the values of the root attribute with the record (real dataset) attribute and, based on the comparison, follows the branch and jumps to the next node.

For the next node, the algorithm again compares the attribute value with the other sub-nodes and moves further. It continues the process until it reaches the leaf node of the tree. The complete process can be better understood using the below algorithm:

- **Step-1:** Begin the tree with the root node, says S, which contains the complete dataset.
- **Step-2:** Find the best attribute in the dataset using **Attribute Selection Measure (ASM).**
- **Step-3:** Divide the S into subsets that contain possible values for the best attributes.
- **Step-4:** Generate the decision tree node, which contains the best attribute.
- **Step-5:** Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and call the final node as a leaf node.

Example:

Suppose there is a candidate who has a job offer and wants to decide whether he should accept the offer or Not. So, to solve this problem, the decision tree starts with the root node (Salary attribute by ASM). The root node splits further into the next decision node (distance from the office) and one leaf node based on the corresponding labels. The next decision node further gets split into one decision node (Cab facility) and one leaf node. Finally, the decision node splits into two leaf nodes (Accepted offers and Declined offer). Consider the below diagram:



Attribute Selection Measures:

While implementing a Decision tree, the main issue arises that how to select the best attribute for the root node and for sub-nodes. So, to solve such problems there is a technique which is called as **Attribute selection measure or ASM.** By this measurement, we can easily select the best attribute for the nodes of the tree.

There are two popular techniques for ASM, which are:

- **Information Gain**
- **Gini Index**

Information Gain:

- Information gain is the measurement of changes in entropy after the segmentation of a dataset based on an attribute.
- It calculates how much information a feature provides us about a class.
- According to the value of information gain, we split the node and build the decision tree.
- A decision tree algorithm always tries to maximize the value of information gain, and a node/attribute having the highest information gain is split first. It can be calculated using the below formula:

  Information Gain= Entropy(S)- [(Weighted Avg) *Entropy(each feature)

Entropy:

Entropy is a metric to measure the impurity in a given attribute. It specifies randomness in data. Entropy can be calculated as:

Entropy(s)= -P(yes)log2 P(yes)- P(no) log2 P(no)

Where,

- S= Total number of samples
- P(yes)= probability of yes
- P(no)= probability of no

Gini Index:

- Gini index is a measure of impurity or purity used while creating a decision tree in the CART(Classification and Regression Tree) algorithm.
- An attribute with the low Gini index should be preferred as compared to the high Gini index.
- It only creates binary splits, and the CART algorithm uses the Gini index to create binary splits.

- Gini index can be calculated using the below formula:

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

  Where Pi denotes the probability of an element being classified for a distinct class.

Pruning:

It is another method that can help us avoid overfitting. It helps in improving the performance of the tree by cutting the nodes or sub-nodes which are not significant. It removes the branches which have very low importance.

There are mainly 2 ways for pruning:

- **Pre-pruning** – we can stop growing the tree earlier, which means we can prune/remove/cut a node if it has low importance while growing the tree.

- **Post-pruning** – once our tree is built to its depth, we can start pruning the nodes based on their significance.

Advantages of the Decision Tree:

- It is simple to understand as it follows the same process which a human follows while making any decision in real-life.
- It can be very useful for solving decision-related problems.
- It helps to think about all the possible outcomes for a problem.
- There is less requirement of data cleaning compared to other algorithms.

Disadvantages of the Decision Tree:

- The decision tree contains lots of layers, which makes it complex.
- It may have an overfitting issue, which can be resolved using the **Random Forest algorithm.**
- For more class labels, the computational complexity of the decision tree may increase.

Example:

Problem Statement:

Predict the loan eligibility process from given data.

The above problem statement is taken from Analytics Vidhya Hackathon.

You can find the dataset and more information about the variables in the dataset on Analytics Vidhya.

Step1: Load the data and finish the cleaning process

```
Loan_ID               0
Gender               13
Married               3
Dependents           15
Education             0
Self_Employed        32
ApplicantIncome       0
CoapplicantIncome     0
LoanAmount           22
Loan_Amount_Term     14
Credit_History       50
Property_Area         0
Loan_Status           0
dtype: int64
```

There are two possible ways to either fill the null values with some value or drop all the missing values(I dropped all the missing values).

If you look at the original dataset's shape, it is (614,13), and the new data-set after dropping the null values is (480,13).

Step 2: Take a Look at the data-set

We found there are many categorical values in the dataset.

NOTE: The decision tree does not support categorical data as features.

So the optimal step to take at this point is you can use feature engineering techniques like label encoding and one hot label encoding.

Step 3: Split the data-set into train and test sets

Step 4: Build the model and fit the train set.

Before we visualize the tree, let us do some calculations and find out the root node by using Entropy.

Calculation 1: Find the Entropy of the total dataset

Entropy=$\frac{-p}{p+n}log_2\left(\frac{p}{p+n}\right)-\frac{n}{p+n}log_2\left(\frac{n}{p+n}\right)$

p = no of positive cases(Loan_Status accepted)

n = number of negative cases(Loan_Status not accepted)

In the data set, we have

p = 332 , n=148, p+n=480

E(s)=$\left(\frac{-332}{480}\right)log\left(\frac{332}{480}\right)-\left(\frac{148}{480}\right)log\left(\frac{148}{480}\right)$

=0.8912402012

here log is with base 2

Entropy = E(s) = 0.89

Calculation 2: Now find the Entropy and gain for every column

1.Gender Column

There are two types in this male(1) and female(0)

Condition 1: Male

data-set with all male's in it and then,

p = 278, n=116 , p+n=489

Entropy(G=Male) = 0.87

Condition 2: Female

data-set with all female's in it and then,

p = 54 , n = 32 , p+n = 86

Entropy(G=Female) = 0.95

Average Information in Gender column is

$$I(Attribute) = \sum \frac{p_i+n_i}{p+n} Entropy(A)$$

I(Gender)=(Entropy(G=male)*p+n/480)+(Entropy(G=female)*p+n/480)

=[0.87*(276+116)/480]+[0.95*(54+32)/480]

=0.88

Gain=Entropy(s)-I(Attribute)

Gain(Gender) = E(s)—I(Gender)

Gain = 0.89–0.88

Gain = 0.001

2.Married Column

In this column we have yes and no values

Condition 1: Married = Yes(1)

In this split the whole data-set with Married status yes

p = 227 , n = 84 , p+n = 311

E(Married = Yes) = 0.84

Condition 2: Married = No(0)

In this split the whole data-set with Married status no

p = 105 , n = 64 , p+n = 169

E(Married = No) = 0.957

Average Information in Married column is

I(married)=((227+84)/480)*((105+64)/480)0.957

=0.88

Gain = 0.89–0.88 = 0.001

3.Education Column

In this column, we have graduate and not graduate values.

Condition 1: Education = Graduate(1)

p = 271 , n = 112 , p+n = 383

E(Education = Graduate) = 0.87

Condition 2: Education = Not Graduate(0)

p = 61 , n = 36 , p+n = 97

E(Education = Not Graduate) = 0.95

Average Information of Education column= 0.886

Gain = 0.01

4.Self-Employed column

In this column we have Yes or No values

Condition 1: Self-Employed = Yes(1)

p = 43 , n = 23 , p+n = 66

E(Self-Employed=Yes) = 0.93

Condition 2: Self-Employed = No(0)

p = 289 , n = 125 , p+n = 414

E(Self-Employed=No) = 0.88

Average Information in Self-Employed in Education Column = 0.886

Gain = 0.01

5.Credit Score Column

In this column we have 1 and 0 values

Condition 1: Credit Score = 1

p = 325 , n = 85 , p+n = 410

E(Credit Score = 1) = 0.73

Condition 2: Credit Score = 0

p = 63 , n = 7 , p+n = 70

E(Credit Score = 0) = 0.46

Average Information in Credit Score column = 0.69

Gain = 0.2

Compare all the gain values

Gain(Gender)=0.01

Gain(Married)=0.01

Gain(Education)=0.01

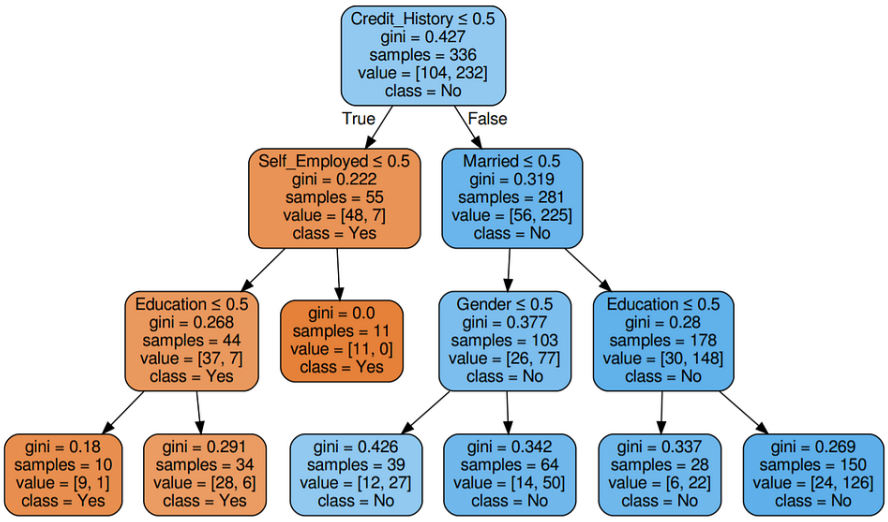
Gain(Self-Employed)=0.01

Gain(Credit Score)=0.2

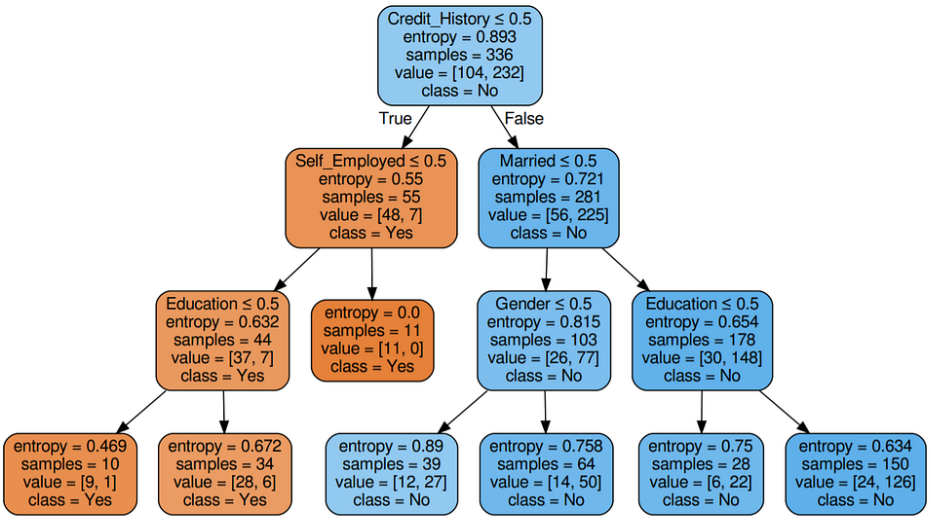
Credit Score has the highest gain so that will be used in the root node

Now let's verify with the decision tree of the model.

Step 5: Visualize the Decision Tree



Decision Tree with criterion=gini



Decision Tree with criterion=entropy

So the same procedure repeats until there is no possibility for further splitting.

Step 6: Check the score of the model

We almost got 80% percent accuracy. Which is a decent score for this type of problem statement.

Let's get back with the theoretical information about the decision tree.

Assumptions made while creating the decision tree:

- While starting the training, the whole data-set is considered as the root.
- The input values are preferred to be categorical.
- Records are distributed based on attribute values.
- The attributes are placed as the root node of the tree is based on statistical results.

Optimizing the performance of the trees

- **max_depth:** The maximum depth of the tree is defined here.
- **criterion:** This parameter takes the criterion method as the value. The default value is Gini.
- **splitter**: This parameter allows us to choose the split strategy. Best and random are available types of the split. The default value is best.

Python Example:

Business Problem

Create a model that can help predict a species of a penguin based on physical attributes, then we can use that model to help researchers classify penguins in the field, instead of needing an experienced biologist

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
df = pd.read_csv("penguins_size.csv")
df.head()
```

Out[2]:

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0
3	Adelie	Torgersen	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0

penguins_size.csv: Simplified data from original penguin data sets.

- species: penguin species (Chinstrap, Adélie, or Gentoo)
- culmen_length_mm: culmen length (mm)
- culmen_depth_mm: culmen depth (mm)
- flipper_length_mm: flipper length (mm)
- body_mass_g: body mass (g)
- island: island name (Dream, Torgersen, or Biscoe) in the Palmer Archipelago (Antarctica)
- sex: penguin sex

After completion of EDA and Feature engineering let's split the data.

```
In [15]: X = pd.get_dummies(df.drop('species',axis=1),drop_first=True)
         y = df['species']
```

Train/Test Split

```
In [16]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rando
         m_state=101)
```

Modelling

Decision Tree Classifier - with default Hyperparameters

```
In [17]: from sklearn.tree import DecisionTreeClassifier

In [18]: model = DecisionTreeClassifier()

In [19]: model.fit(X_train,y_train)
Out[19]: DecisionTreeClassifier()
```

Prediction

```
In [20]: base_pred = model.predict(X_test)

In [21]: pred_train = model.predict(X_train)
```

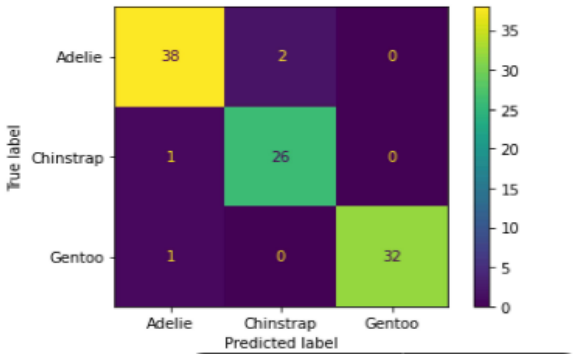
Evaluation

```
In [22]: from sklearn.metrics import classification_report,plot_confusion_matrix,accura
         cy_score

In [23]: accuracy_score(y_train,pred_train)
Out[23]: 1.0

In [24]: accuracy_score(y_test,base_pred)
Out[24]: 0.96

In [25]: plot_confusion_matrix(model,X_test,y_test)
         plt.show()
```



In [26]:
```
print(classification_report(y_test,base_pred))
```

```
              precision    recall  f1-score   support

      Adelie       0.95      0.95      0.95        40
   Chinstrap       0.93      0.96      0.95        27
      Gentoo       1.00      0.97      0.98        33

    accuracy                           0.96       100
   macro avg       0.96      0.96      0.96       100
weighted avg       0.96      0.96      0.96       100
```

In [27]:
```
model.feature_importances_
```

Out[27]:
```
array([0.33350103, 0.02010577, 0.57575804, 0.        , 0.03806069,
       0.00685778, 0.02571668])
```

In [28]:
```
pd.DataFrame(index=X.columns,data=model.feature_importances_,columns=['Feature Importance'])
```

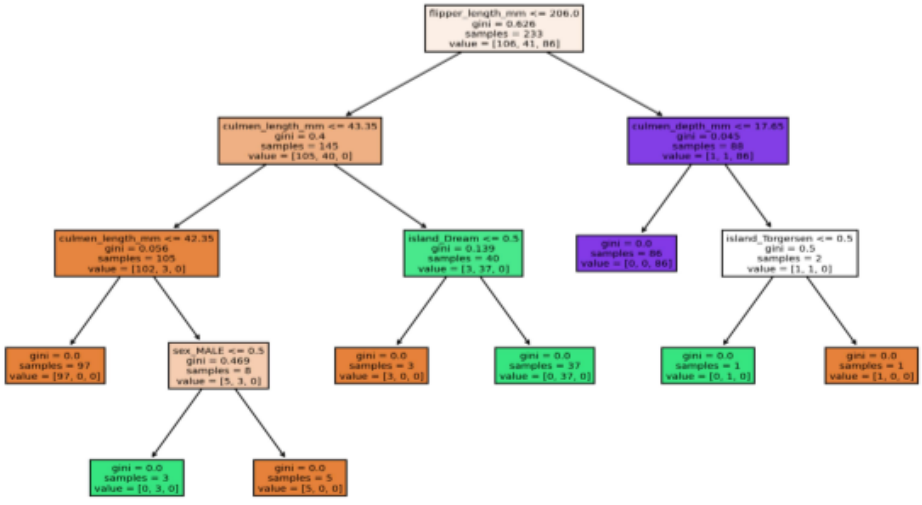
Out[28]:

	Feature Importance
culmen_length_mm	0.333501
culmen_depth_mm	0.020106
flipper_length_mm	0.575758
body_mass_g	0.000000
island_Dream	0.038061
island_Torgersen	0.006858
sex_MALE	0.025717

Visualize the tree

In [29]:
```
from sklearn.tree import plot_tree
```

In [30]:
```
plt.figure(figsize=(12,8),dpi=150)
plot_tree(model,filled=True,feature_names=X.columns)
plt.show()
```



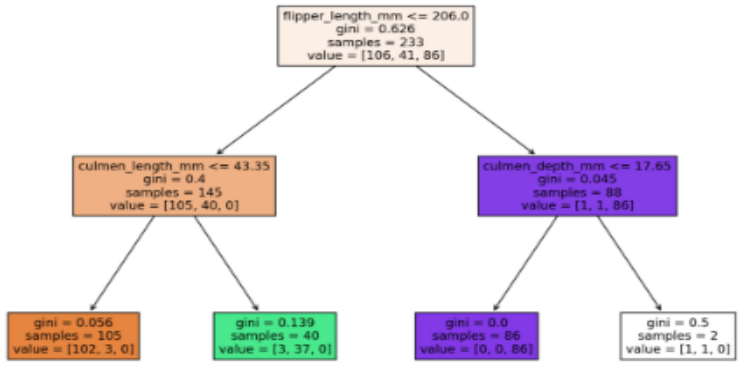
Optimization-Pruning of decision tree
Max depth

```
In [31]: pruned_tree = DecisionTreeClassifier(max_depth=2)
```

```
In [32]: def report_model(model):
             model.fit(X_train,y_train)
             model_preds = model.predict(X_test)
             pred_train = model.predict(X_train)
             print("Train Accuracy:",accuracy_score(y_train,pred_train))
             print("Test Accuracy:",accuracy_score(y_test,model_preds))
             plt.figure(figsize=(12,8),dpi=150)
             plot_tree(model,filled=True,feature_names=X.columns)
```

```
In [33]: report_model(pruned_tree)
```

```
Train Accuracy: 0.9699570815450643
Test Accuracy: 0.92
```

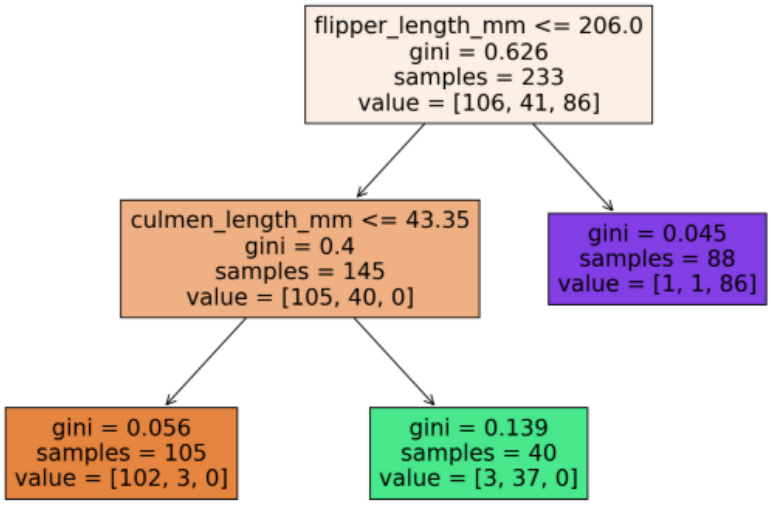


Max Leaf Nodes

```
In [34]: pruned_tree = DecisionTreeClassifier(max_leaf_nodes=3)
```

```
In [35]: report_model(pruned_tree)
```

```
Train Accuracy: 0.9656652360515021
Test Accuracy: 0.91
```



Reference for above topic:

https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm

https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/

https://towardsai.net/p/programming/decision-trees-explained-with-a-practical-example-fe47872d3b53

Random Forest Algorithm:

As the name suggests, "Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset." Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.

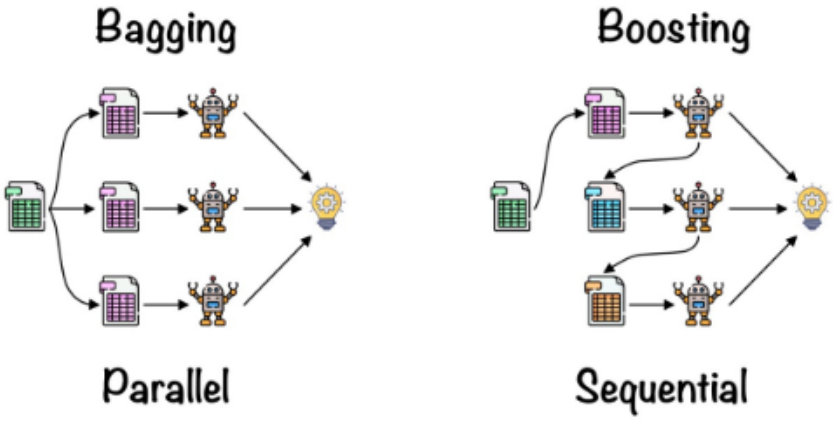
The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.

Before understanding the working of the random forest algorithm in machine learning, we must look into the ensemble learning technique. **Ensemble** simply means combining multiple models. Thus a collection of models is used to make predictions rather than an individual model.

Ensemble uses two types of methods:

1. **Bagging**– It creates a different training subset from sample training data with replacement & the final output is based on majority voting. For example, Random Forest.

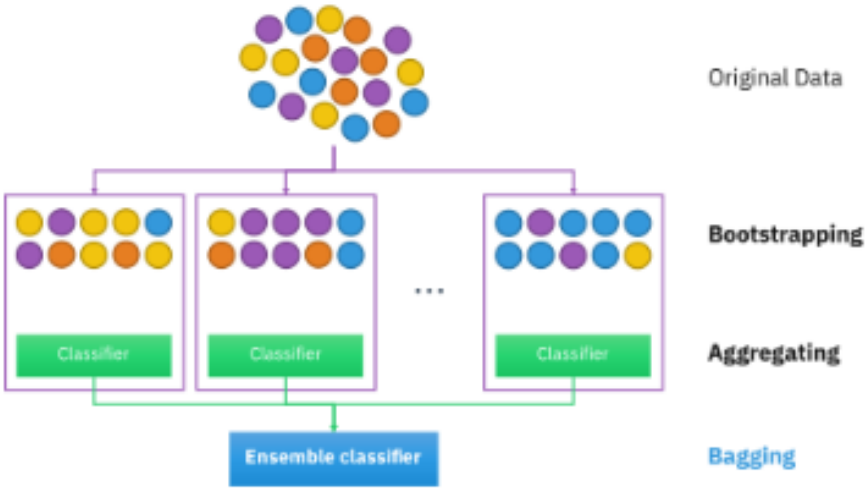
2. **Boosting**– It combines weak learners into strong learners by creating sequential models such that the final model has the highest accuracy. For example, ADA BOOST, XGBOOST.



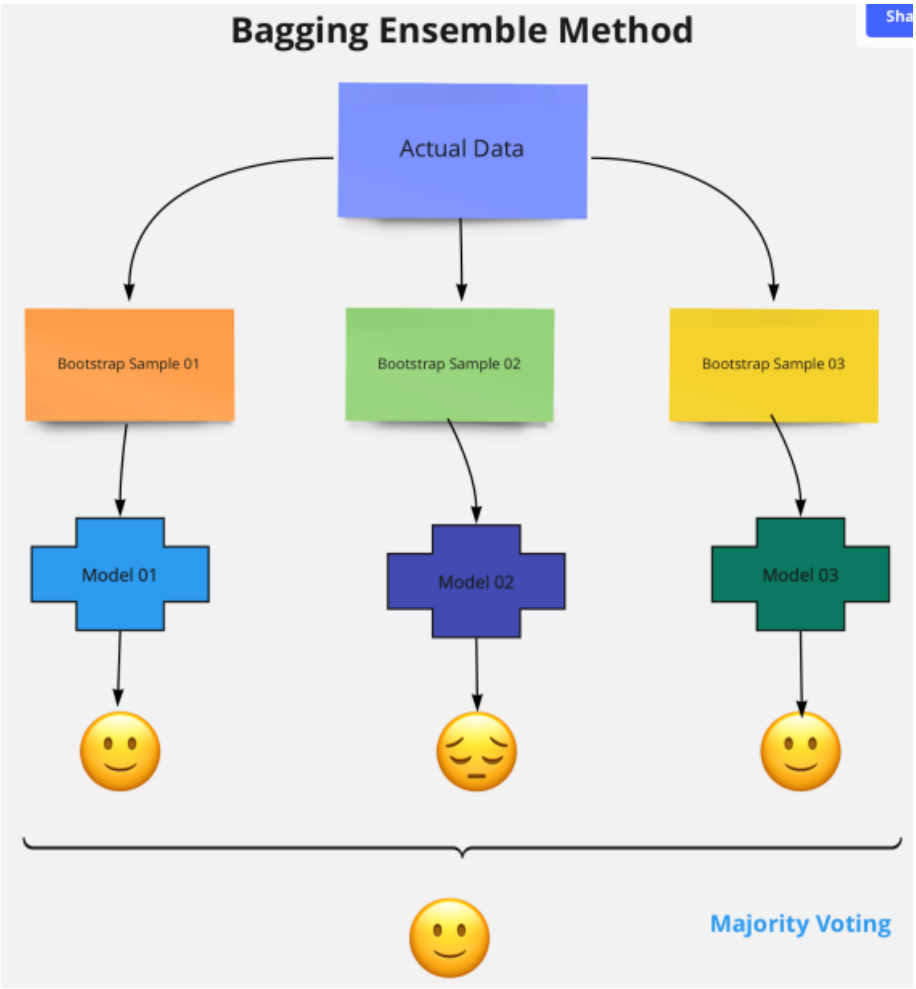
As mentioned earlier, Random forest works on the Bagging principle. Now let's dive in and understand bagging in detail.

Bagging:

Bagging, also known as Bootstrap Aggregation, is the ensemble technique used by random forest.Bagging chooses a random sample/random subset from the entire data set. Hence each model is generated from the samples (Bootstrap Samples) provided by the Original Data with replacement known as row sampling. This step of row sampling with replacement is called bootstrap. Now each model is trained independently, which generates results. The final output is based on majority voting after combining the results of all models. This step which involves combining all the results and generating output based on majority voting, is known as aggregation.



Now let's look at an example by breaking it down with the help of the following figure. Here the bootstrap sample is taken from actual data (Bootstrap sample 01, Bootstrap sample 02, and Bootstrap sample 03) with a replacement which means there is a high possibility that each sample won't contain unique data. The model (Model 01, Model 02, and Model 03) obtained from this bootstrap sample is trained independently. Each model generates results as shown. Now the Happy emoji has a majority when compared to the Sad emoji. Thus based on majority voting final output is obtained as Happy emoji.



Assumptions of random forest:

Since the random forest combines multiple trees to predict the class of the dataset, it is possible that some decision trees may predict the correct output, while others may not. But together, all the trees predict the correct output. Therefore, below are two assumptions for a better Random forest classifier:

- There should be some actual values in the feature variable of the dataset so that the classifier can predict accurate results rather than a guessed result.
- The predictions from each tree must have very low correlations.

Why use Random Forest?:

Below are some points that explain why we should use the Random Forest algorithm:

- It takes less training time as compared to other algorithms.

- It predicts output with high accuracy, even for the large dataset it runs efficiently.
- It can also maintain accuracy when a large proportion of data is missing.

Working of Random Forest Algorithm:

Random Forest works in two-phase first is to create the random forest by combining N decision tree, and second is to make predictions for each tree created in the first phase.

The Working process can be explained in the below steps and diagram:

Step-1: Select random K data points from the training set.

Step-2: Build the decision trees associated with the selected data points (Subsets).

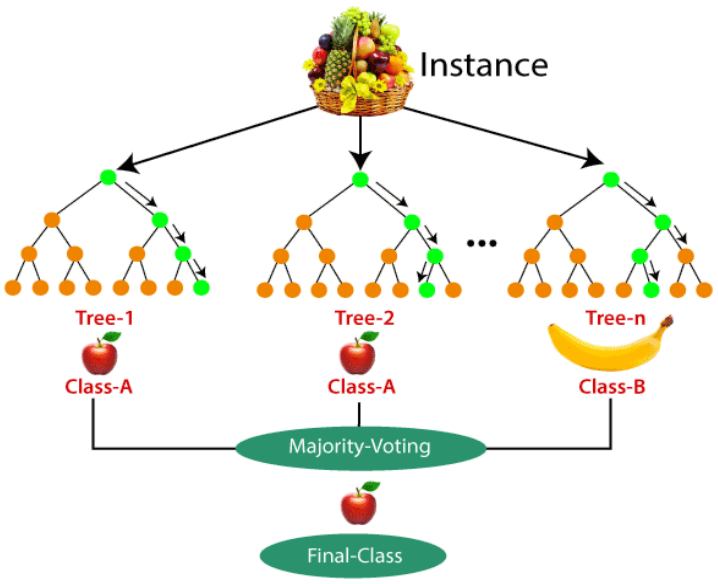
Step-3: Choose the number N for decision trees that you want to build.

Step-4: Repeat Step 1 & 2.

Step-5: For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes.

The working of the algorithm can be better understood by the below example:

Example: Suppose there is a dataset that contains multiple fruit images. So, this dataset is given to the Random forest classifier. The dataset is divided into subsets and given to each decision tree. During the training phase, each decision tree produces a prediction result, and when a new data point occurs, then based on the majority of results, the Random Forest classifier predicts the final decision. Consider the below image:



Applications of Random Forest:

There are mainly four sectors where Random forest mostly used:

1. **Banking:** Banking sector mostly uses this algorithm for the identification of loan risk.
2. **Medicine:** With the help of this algorithm, disease trends and risks of the disease can be identified.
3. **Land Use:** We can identify the areas of similar land use by this algorithm.
4. **Marketing:** Marketing trends can be identified using this algorithm.

Advantages of Random Forest:

- Random Forest is capable of performing both Classification and Regression tasks.
- It is capable of handling large datasets with high dimensionality.
- It enhances the accuracy of the model and prevents the overfitting issue.

Disadvantages of Random Forest:

- Although random forest can be used for both classification and regression tasks, it is not more suitable for Regression tasks.

Example

Model building

We will build a Random Forest Classifier using the Scikit-Learn library of Python programming language and in order to do this, we use the IRIS dataset which is quite a common and famous dataset. The Random forest or Random Decision Forest is a supervised Machine learning algorithm used for classification, regression, and other tasks using decision trees.

The Random forest classifier creates a set of decision trees from a randomly selected subset of the training set. It is basically a set of decision trees (DT) from a randomly selected subset of the training set and then It collects the votes from different decision trees to decide the final prediction.

In this classification algorithm, we will use IRIS flower datasets to train and test the model. We will build a model to classify the type of flower.

```
# importing required libraries
# importing Scikit-learn library and datasets package
from sklearn import datasets

# Loading the iris plants dataset (classification)
iris = datasets.load_iris()
print(iris.target_names)
output:['setosa' 'versicolor' 'virginica']

print(iris.feature_names)

output:['sepal length (cm)', 'sepal width (cm)', 'petal length
(cm)', 'petal width (cm)']

# dividing the datasets into two parts i.e. training datasets
and test datasets
X, y = datasets.load_iris( return_X_y = True)

# Splitting arrays or matrices into random train and test
subsets
from sklearn.model_selection import train_test_split
# i.e. 70 % training dataset and 30 % test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.30)
# importing random forest classifier from assemble module
from sklearn.ensemble import RandomForestClassifier
import pandas as pd
# creating dataframe of IRIS dataset
data = pd.DataFrame({'sepallength': iris.data[:, 0],
'sepalwidth': iris.data[:, 1],
                        'petallength': iris.data[:, 2],
'petalwidth': iris.data[:, 3],
                        'species': iris.target})
# printing the top 5 datasets in iris dataset
print(data.head())
```

Output:

	sepallength	sepalwidth	petallength	petalwidth	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

```
# creating a RF classifier
clf = RandomForestClassifier(n_estimators = 100)

# Training the model on the training dataset
# fit function is used to train the model using the training
sets as parameters
clf.fit(X_train, y_train)
# performing predictions on the test dataset
y_pred = clf.predict(X_test)
# metrics are used to find accuracy or error
from sklearn import metrics
print()
# using metrics module for accuracy calculation
print("ACCURACY OF THE MODEL: ", metrics.accuracy_score(y_test,
y_pred))
```

output:ACCURACY OF THE MODEL: 0.9238095238095239

```
# predicting which type of flower it is.
clf.predict([[3, 3, 2, 2]])
output:array([0])
```

This implies it is **setosa** flower type as we got the three species or classes in our data set: **Setosa, Versicolor, and Virginia.** Now we will also find out the important features or selecting features in the IRIS dataset by using the following lines of code.

Calculating feature importance

```
# importing random forest classifier from assemble module
from sklearn.ensemble import RandomForestClassifier
# Create a Random forest Classifier
clf = RandomForestClassifier(n_estimators = 100)

# Train the model using the training sets
clf.fit(X_train, y_train)
# using the feature importance variable
import pandas as pd
feature_imp = pd.Series(clf.feature_importances_, index =
iris.feature_names).sort_values(ascending = False)
feature_imp
```

Output:

```
petal width (cm)     0.458607
petal length (cm)    0.413859
sepal length (cm)    0.103600
sepal width (cm)     0.023933
dtype: float64
```

Reference for above topic:

https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

https://www.javatpoint.com/machine-learning-random-forest-algorithm

Boosting:

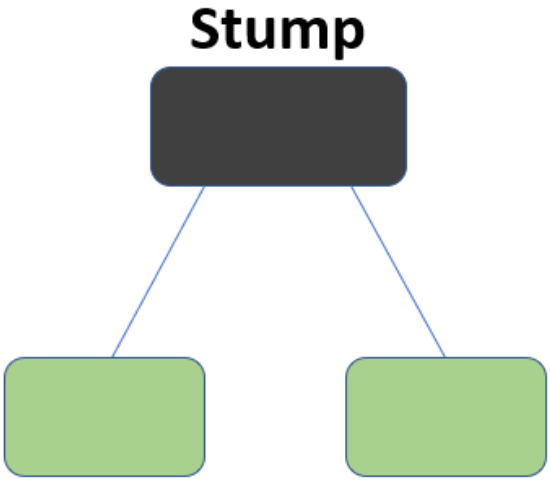
The principle behind boosting algorithms is that we first build a model on the training dataset and then build a second model to rectify the errors present in the first model. This procedure is continued until and unless the errors are minimized and the dataset is predicted correctly. Boosting algorithms work in a similar way, it combines multiple models (weak learners) to reach the final output (strong learners).

There are mainly 3 types of boosting algorithms:

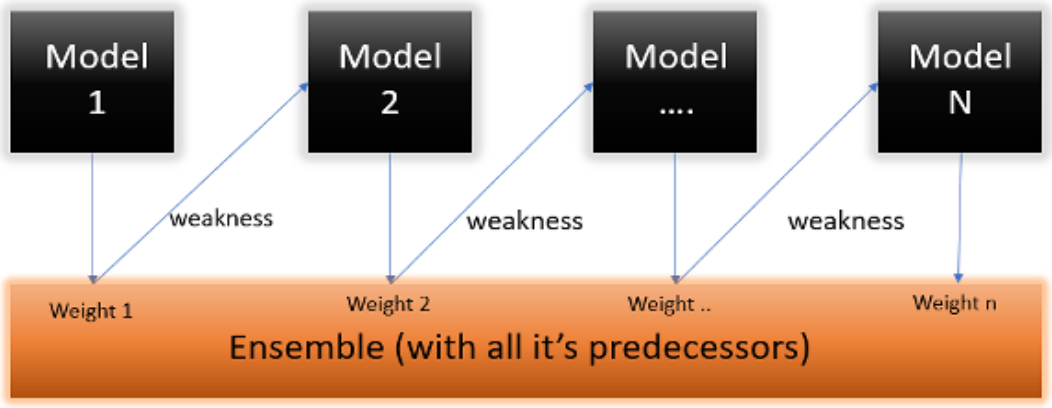
- AdaBoost algorithm in Machine Learning
- Gradient descent algorithm
- Xtreme gradient descent algorithm

AdaBoost Algorithm:

AdaBoost, also called Adaptive Boosting, is a technique in Machine Learning used as an Ensemble Method. The most common estimator used with AdaBoost is decision trees with one level which means Decision trees with only 1 split. These trees are also called Decision Stumps.



What this algorithm does is that it builds a model and gives equal weights to all the data points. It then assigns higher weights to points that are wrongly classified. Now all the points with higher weights are given more importance in the next model. It will keep training models until and unless a lower error is received.



Let's take an example to understand this, suppose you built a decision tree algorithm on the Titanic dataset, and from there, you get an accuracy of 80%. After this, you apply a different algorithm and check the accuracy, and it comes out to be 75% for KNN and 70% for Linear Regression.

We see the accuracy differs when we build a different model on the same dataset. But what if we use combinations of all these algorithms to make the final prediction? We'll get more accurate results by taking the average of the results from these models. We can increase the prediction power in this way.

Working of the AdaBoost Algorithm:

Let's understand what and how this algorithm works under the hood with the following tutorial.

Step 1 – The Image shown below is the actual representation of our dataset. Since the target column is binary, it is a classification problem. First of all, these data points will be assigned some weights. Initially, all the weights will be equal.

Row No.	Gender	Age	Income	Illness	Sample Weights
1	Male	41	40000	Yes	1/5
2	Male	54	30000	No	1/5
3	Female	42	25000	No	1/5
4	Female	40	60000	Yes	1/5
5	Male	46	50000	Yes	1/5

The formula to calculate the sample weights is:

$$w(x_i, y_i) = \frac{1}{N}, \; i = 1, 2, \ldots . n$$

Where N is the total number of data points

Here since we have 5 data points, the sample weights assigned will be 1/5.

Step 2 – We start by seeing how well "*Gender*" classifies the samples and will see how the variables (Age, Income) classify the samples.

We'll create a decision stump for each of the features and then calculate the *Gini Index* of each tree. The tree with the lowest Gini Index will be our first stump.

Here in our dataset, let's say *Gender* has the lowest gini index, so it will be our first stump.

Step 3 – We'll now calculate the "Amount of Say" or "Importance" or "Influence" for this classifier in classifying the data points using this formula:

$$\frac{1}{2}\log\frac{1 - Total\ Error}{Total\ Error}$$

The total error is nothing but the summation of all the sample weights of misclassified data points.

Here in our dataset, let's assume there is 1 wrong output, so our total error will be 1/5, and the alpha (performance of the stump) will be:

$$Performance\ of\ the\ stump = \frac{1}{2}\log_e(\frac{1 - Total\ Error}{Total\ Error})$$

$$\alpha = \frac{1}{2}\log_e\left(\frac{1 - \frac{1}{5}}{\frac{1}{5}}\right)$$

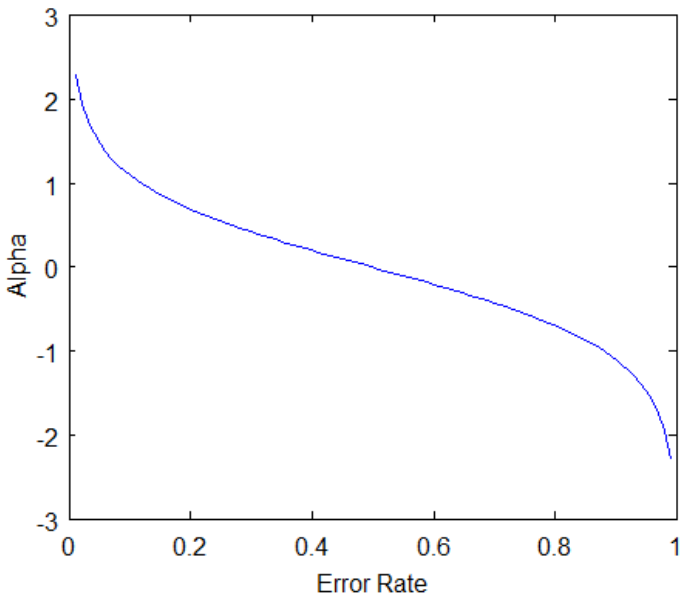
$$\alpha = \frac{1}{2}\log_e\left(\frac{0.8}{0.2}\right)$$

$$\alpha = \frac{1}{2}\log_e(4) = \frac{1}{2} * (1.38)$$

$$\alpha = 0.69$$

Note: Total error will always be between 0 and 1.

0 Indicates perfect stump, and 1 indicates horrible stump.



From the graph above, we can see that when there is no misclassification, then we have no error (Total Error = 0), so the "amount of say (alpha)" will be a large number.
When the classifier predicts half right and half wrong, then the Total Error = 0.5, and the importance (amount of say) of the classifier will be 0.
If all the samples have been incorrectly classified, then the error will be very high (approx. to 1), and hence our alpha value will be a negative integer.
Step 4 – You must be wondering why it is necessary to calculate the TE and performance of a stump. Well, the answer is very simple, we need to update the weights because if the same weights are applied to the next model, then the output received will be the same as what was received in the first model.
The wrong predictions will be given more weight, whereas the correct predictions weights will be decreased. Now when we build our next model after updating the weights, more preference will be given to the points with higher weights.
After finding the importance of the classifier and total error, we need to finally update the weights, and for this, we use the following formula:

$$New\ sample\ weight\ =\ old\ weight\ *\ e^{\pm Amount\ of\ say\ (\alpha)}$$

The amount of, say (alpha) will be negative when the sample is correctly classified.
The amount of, say (alpha) will be positive when the sample is miss-classified.
There are four correctly classified samples and 1 wrong. Here, the sample weight of that datapoint is 1/5, and the amount of say/performance of the stump of Gender is 0.69.

New weights for correctly classified samples are:

$$New\ sample\ weight\ =\ \frac{1}{5}\ *\ \exp(-0.69)$$

$$New\ sample\ weight\ =\ 0.2\ *\ 0.502\ =\ 0.1004$$

For wrongly classified samples, the updated weights will be:

$$New\ sample\ weight\ =\ \frac{1}{5}\ *\ \exp(0.69)$$

$$New\ sample\ weight\ =\ 0.2\ *\ 1.994\ =\ 0.3988$$

Note: See the sign of alpha when I am putting the values, the alpha is negative when the data point is correctly classified, and this *decreases the sample weight* from 0.2 to 0.1004. It is positive when there is misclassification, and this will increase the sample weight from 0.2 to 0.3988

Row No.	Gender	Age	Income	Illness	Sample Weights	New Sample Weights
1	Male	41	40000	Yes	1/5	0.1004
2	Male	54	30000	No	1/5	0.1004
3	Female	42	25000	No	1/5	0.1004
4	Female	40	60000	Yes	1/5	0.3988
5	Male	46	50000	Yes	1/5	0.1004

We know that the total sum of the sample weights must be equal to 1, but here if we sum up all the new sample weights, we will get 0.8004. To bring this sum equal to 1, we will normalize these weights by dividing all the weights by the total sum of updated weights, which is 0.8004. So, after normalizing the sample weights, we get this dataset, and now the sum is equal to 1.

Row No.	Gender	Age	Income	Illness	Sample Weights	New Sample Weights
1	Male	41	40000	Yes	1/5	0.1004/0.8004 =0.1254
2	Male	54	30000	No	1/5	0.1004/0.8004 =0.1254
3	Female	42	25000	No	1/5	0.1004/0.8004 =0.1254
4	Female	40	60000	Yes	1/5	0.3988/0.8004 =0.4982
5	Male	46	50000	Yes	1/5	0.1004/0.8004 =0.1254

Step 5 – Now, we need to make a new dataset to see if the errors decreased or not. For this, we will remove the "sample weights" and "new sample weights" columns and then, based on the "new sample weights," divide our data points into buckets.

Row No.	Gender	Age	Income	Illness	New Sample Weights	Buckets
1	Male	41	40000	Yes	0.1004/0.8004= 0.1254	0 to 0.1254
2	Male	54	30000	No	0.1004/0.8004= 0.1254	0.1254 to 0.2508
3	Female	42	25000	No	0.1004/0.8004= 0.1254	0.2508 to 0.3762
4	Female	40	60000	Yes	0.3988/0.8004= 0.4982	0.3762 to 0.8744
5	Male	46	50000	Yes	0.1004/0.8004= 0.1254	0.8744 to 0.9998

Step 6 – We are almost done. Now, what the algorithm does is selects random numbers from 0-1. Since incorrectly classified records have higher sample weights, the probability of selecting those records is very high.

Suppose the 5 random numbers our algorithm takes are 0.38,0.26,0.98,0.40,0.55.

Now we will see where these random numbers fall in the bucket, and according to it, we'll make our new dataset shown below.

Row No.	Gender	Age	Income	Illness
1	Female	40	60000	Yes
2	Male	54	30000	No
3	Female	42	25000	No
4	Female	40	60000	Yes
5	Female	40	60000	Yes

This comes out to be our new dataset, and we see the data point, which was wrongly classified, has been selected 3 times because it has a higher weight.

Step 7 – Now this act as our new dataset, and we need to repeat all the above steps i.e.

- Assign equal weights to all the data points.
- Find the stump that does the best job classifying the new collection of samples by finding their Gini Index and selecting the one with the lowest Gini index.
- Calculate the “Amount of Say” and “Total error” to update the previous sample weights.
- Normalize the new sample weights.

Iterate through these steps until and unless a low training error is achieved.

Suppose, with respect to our dataset, we have constructed 3 decision trees (DT1, DT2, DT3) in a sequential manner. If we send our test data now, it will pass through all the decision trees, and finally, we will see which class has the majority, and based on that, we will do predictions for our test dataset.

Example

Model building

Let's first load the required libraries.

```
# Load libraries
from sklearn.ensemble import AdaBoostClassifier
from sklearn import datasets
# Import train_test_split function
from sklearn.model_selection import train_test_split
#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics
```

Loading Dataset

In the model of the building part, you can use the IRIS dataset, which is a very famous multi-class classification problem. This dataset comprises 4 features (sepal length, sepal width, petal length, petal width) and a target (the type of flower). This data has three types of flower classes: Setosa, Versicolour, and Virginica. The dataset is available in the scikit-learn library, or you can also download it from the UCI Machine Learning Library.

```
# Load data
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

Split dataset

To understand model performance, dividing the dataset into a training set and a test set is a good strategy.

Let's split the dataset by using the function train_test_split(). you need to pass 3 parameters: features, target, and test_set size.

```
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3) # 70% training and 30% test
```

Building the AdaBoost Model

Let's create the AdaBoost Model using Scikit-learn. AdaBoost uses Decision Tree Classifier as default Classifier.

```
# Create adaboost classifier object
abc = AdaBoostClassifier(n_estimators=50,
                         learning_rate=1)
# Train Adaboost Classifier
model = abc.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = model.predict(X_test)
```

"The most important parameters are base_estimator, n_estimators, and learning_rate."

- **base_estimator:** It is a weak learner used to train the model. It uses DecisionTreeClassifier as default weak learner for training purpose. You can also specify different machine learning algorithms.
- **n_estimators:** Number of weak learners to train iteratively.

- **learning_rate:** It contributes to the weights of weak learners. It uses 1 as a default value.

Evaluate Model

Let's estimate how accurately the classifier or model can predict the type of cultivars. Accuracy can be computed by comparing actual test set values and predicted values.

```
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
output:Accuracy: 0.8888888888888888
```

Well, you got an accuracy of 88.88%, considered as good accuracy.

Reference for above topic:
https://www.analyticsvidhya.com/blog/2021/09/adaboost-algorithm-a-complete-guide-for-beginners/?utm_source=reading_list&utm_medium=https://www.analyticsvidhya.com/blog/2022/01/boosting-in-machine-learning-definition-functions-types-and-features/

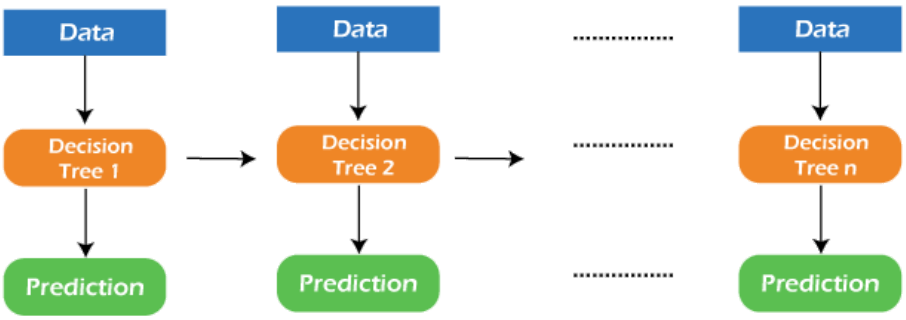
Gradient Boosting:

Gradient Boosting Machine (GBM) is one of the most popular forward learning ensemble methods in machine learning. It is a powerful technique for building predictive models for regression and classification tasks.

GBM helps us to get a predictive model in the form of an ensemble of weak prediction models such as decision trees. Whenever a decision tree performs as a weak learner then the resulting algorithm is called gradient-boosted trees.

It enables us to combine the predictions from various learner models and build a final predictive model having the correct prediction.

But here one question may arise if we are applying the same algorithm then how multiple decision trees can give better predictions than a single decision tree? Moreover, how does each decision tree capture different information from the same data?



So, the answer to these questions is that a different subset of features is taken by the nodes of each decision tree to select the best split. It means that each tree behaves differently, and hence captures different signals from the same data.

Working of Gradient Boosting:

Generally, most supervised learning algorithms are based on a single predictive model such as linear regression, penalized regression model, decision trees, etc. But there are some supervised algorithms in ML that depend on a combination of various models together through the ensemble. In other words, when multiple base models contribute their predictions, an average of all predictions is adapted by boosting algorithms.

Gradient boosting machines consist 3 elements as follows:

- Loss function
- Weak learners
- Additive model

Let's understand these three elements in detail.

1. Loss function:

Although, there is a big family of Loss functions in machine learning that can be used depending on the type of tasks being solved. The use of the loss function is estimated by the demand of specific characteristics of the conditional distribution such as robustness. While using a loss function in our task, we must specify the loss function and the function to calculate the corresponding negative gradient. Once we get these two functions, they can be implemented into gradient boosting machines easily. However, several loss functions have already been proposed for GBM algorithms.

Classification of Loss function:

Based on the type of response variable y, loss function can be classified into different types as follows:

1. **Continuous response, $y \in R$:**
    - Gaussian L2 loss function
    - Laplace L1 loss function
    - Huber loss function, δ specified
    - Quantile loss function, α specified
2. **Categorical response, $y \in \{0, 1\}$:**
    - Binomial loss function
    - Adaboost loss function
3. **Other families of response variables:**
    - Loss functions for survival models
    - Loss functions count data
    - Custom loss functions

2. Weak Learner:

Weak learners are the base learner models that learn from past errors and help in building a strong predictive model design for boosting algorithms in machine learning. Generally, decision trees work as weak learners in boosting algorithms.

Boosting is defined as the framework that continuously works to improve the output from base models. Many gradient boosting applications allow you to "plugin" various classes of weak learners at your disposal. Hence, decision trees are most often used for weak (base) learners.

How to train weak learners:

Machine learning uses training datasets to train base learners and based on the prediction from the previous learner, it improves the performance by focusing on the rows of the training data where the previous tree had the largest errors or residuals.

E.g. shallow trees are considered weak learners to decision trees as it contains a few splits. Generally, in boosting algorithms, trees having up to 6 splits are most common.

Below is a sequence of training the weak learner to improve their performance where each tree is in the sequence with the previous tree's residuals. Further, we are introducing each new tree so that it can learn from the previous tree's errors. These are as follows:

1. Consider a data set and fit a decision tree into it.
   F1(x)=y
2. Fit the next decision tree with the largest errors of the previous tree.
   h1(x)=y?F1(x)
3. Add this new tree to the algorithm by adding both in steps 1 and 2.
   F2(x)=F1(x)+h1(x)
4. Again fit the next decision tree with the residuals of the previous tree.
   h2(x)=y?F2(x)
5. Repeat the same which we have done in step 3.
   F3(x)=F2(x)+h2(x)

Continue this process until some mechanism (i.e. cross-validation) tells us to stop. The final model here is a stagewise additive model of b individual trees:

f(x)=B∑b=1fb(x)

Hence, trees are constructed greedily, choosing the best split points based on purity scores like Gini or minimizing the loss.

3. Additive Model:

The additive model is defined as adding trees to the model. Although we should not add multiple trees at a time, only a single tree must be added so that existing trees in the model are not changed. Further, we can also prefer the gradient descent method by adding trees to reduce the loss.

In the past few years, the gradient descent method was used to minimize the set of parameters such as the coefficient of the regression equation and weight in a neural

network. After calculating error or loss, the weight parameter is used to minimize the error. But recently, most ML experts prefer weak learner sub-models or decision trees as a substitute for these parameters. In which, we have to add a tree in the model to reduce the error and improve the performance of that model. In this way, the prediction from the newly added tree is combined with the prediction from the existing series of trees to get a final prediction. This process continues until the loss reaches an acceptable level or is no longer improvement required.

This method is also known as functional gradient descent or gradient descent with functions.

Extreme Gradient Boosting Machine (XGB):

XGBM is the latest version of gradient boosting machines which also works very similar to GBM. In XGBM, trees are added sequentially (one at a time) that learn from the errors of previous trees and improve them. Although, XGBM and GBM algorithms are similar in look and feel but still there are a few differences between them as follows:

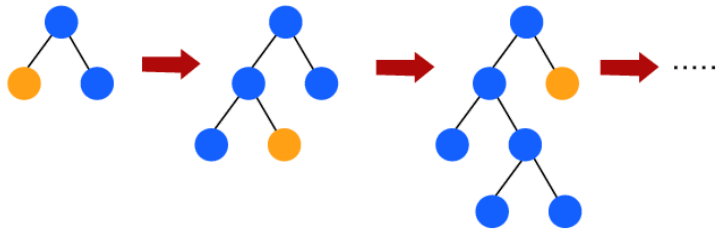
- XGBM uses various regularization techniques to reduce under-fitting or over-fitting of the model which also increases model performance more than gradient boosting machines.
- XGBM follows parallel processing of each node, while GBM does not, which makes it more rapid than gradient boosting machines.
- XGBM helps us to get rid of the imputation of missing values because by default the model takes care of it. It learns on its own whether these values should be in the right or left node.

Light Gradient Boosting Machines (Light GBM):

Light GBM is a more upgraded version of the Gradient boosting machine due to its efficiency and fast speed. Unlike GBM and XGBM, it can handle a huge amount of data without any complexity. On the other hand, it is not suitable for those data points that are lesser in number.

Instead of level-wise growth, Light GBM prefers leaf-wise growth of the nodes of the tree. Further, in light GBM, the primary node is split into two secondary nodes and

later it chooses one secondary node to be split. This split of a secondary node depends upon which between two nodes has a higher loss.



Hence, due to leaf-wise split, the Light Gradient Boosting Machine (LGBM) algorithm is always preferred over others where a large amount of data is given.

CATBOOST:

The catboost algorithm is primarily used to handle the categorical features in a dataset. Although GBM, XGBM, and Light GBM algorithms are suitable for numeric data sets, Catboost is designed to handle categorical variables into numeric data. Hence, the catboost algorithm consists of an essential preprocessing step to convert categorical features into numerical variables which are not present in any other algorithm.

Advantages of Boosting Algorithms:

- Boosting algorithms follow ensemble learning which enables a model to give a more accurate prediction that cannot be trumped.
- Boosting algorithms are much more flexible than other algorithms as they can optimize different loss functions and provide several hyperparameter tuning options.
- It does not require data pre-processing because it is suitable for both numeric as well as categorical variables.
- It does not require imputation of missing values in the dataset, it handles missing data automatically.

Disadvantages of Boosting Algorithms:

- Boosting algorithms may cause overfitting as well as overemphasizing the outliers.
- Gradient boosting algorithm continuously focuses to minimize the errors and requires multiple trees hence, it is computationally expensive.
- It is a time-consuming and memory exhaustive algorithm.

- Less interpretative in nature, although this is easily addressed with various tools.

Example:

Classification Dataset

We will use the make_classification() function to create a test binary classification dataset.
The dataset will have 1,000 examples, with 10 input features, five of which will be informative and the remaining five that will be redundant. We will fix the random number seed to ensure we get the same examples each time the code is run.
An example of creating and summarizing the dataset is listed below.

```
# test classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=10,
n_informative=5, n_redundant=5, random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

Running the example creates the dataset and confirms the expected number of samples and features.

```
output:(1000, 10) (1000,)
```

Gradient Boosting Machine for Classification
The example below first evaluates a GradientBoostingClassifier on the test problem using repeated k-fold cross-validation and reports the mean accuracy. Then a single model is fit on all available data and a single prediction is made.

```
# gradient boosting for classification in scikit-learn
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from matplotlib import pyplot
# define dataset
X, y = make_classification(n_samples=1000, n_features=10,
n_informative=5, n_redundant=5, random_state=1)
```

```
# evaluate the model
model = GradientBoostingClassifier()
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3,
random_state=1)
n_scores = cross_val_score(model, X, y, scoring='accuracy',
cv=cv, n_jobs=-1, error_score='raise')
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
# fit the model on the whole dataset
model = GradientBoostingClassifier()
model.fit(X, y)
# make a single prediction
row = [[2.56999479, -0.13019997, 3.16075093, -4.35936352,
-1.61271951, -1.39352057, -2.48924933, -1.93094078, 3.26130366,
2.05692145]]
yhat = model.predict(row)
print('Prediction: %d' % yhat[0])
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.
Running the example first reports the evaluation of the model using repeated k-fold cross-validation, then the result of making a single prediction with a model fit on the entire dataset.

```
Accuracy: 0.915 (0.025)
Prediction: 1
```

Gradient Boosting With XGBoost
XGBoost, which is short for "*Extreme Gradient Boosting*," is a library that provides an efficient implementation of the gradient boosting algorithm.
The main benefit of the XGBoost implementation is computational efficiency and often better model performance.
Library Installation:

```
pip install xgboost
```

Next, let's confirm that the library is installed and you are using a modern version.

```
# check xgboost version
import xgboost
print(xgboost.__version__)
```

Running the example, you should see the following version number or higher.

```
output:1.0.1
```

The XGBoost library provides wrapper classes so that the efficient algorithm implementation can be used with the scikit-learn library, specifically via the XGBClassifier and XGBregressor classes.
Let's take a closer look at XGBClassifier.

```
# xgboost for classification
from numpy import asarray
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from xgboost import XGBClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from matplotlib import pyplot
# define dataset
X, y = make_classification(n_samples=1000, n_features=10,
n_informative=5, n_redundant=5, random_state=1)
# evaluate the model
model = XGBClassifier()
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3,
random_state=1)
n_scores = cross_val_score(model, X, y, scoring='accuracy',
cv=cv, n_jobs=-1, error_score='raise')
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
# fit the model on the whole dataset
model = XGBClassifier()
model.fit(X, y)
# make a single prediction
row = [2.56999479, -0.13019997, 3.16075093, -4.35936352,
-1.61271951, -1.39352057, -2.48924933, -1.93094078, 3.26130366,
2.05692145]
row = asarray(row).reshape((1, len(row)))
yhat = model.predict(row)
print('Prediction: %d' % yhat[0])
```

Running the example first reports the evaluation of the model using repeated k-fold cross-validation, then the result of making a single prediction with a model fit on the entire dataset.

```
Accuracy: 0.936 (0.019)
Prediction: 1
```

Reference for above topic:

- https://www.javatpoint.com/gbm-in-machine-learning
- https://www.analyticsvidhya.com/blog/2021/09/gradient-boosting-algorithm-a-complete-guide-for-beginners/

- https://machinelearningmastery.com/gradient-boosting-with-scikit-learn-xgboost-lightgbm-and-catboost/

Clustering in machine learning:

- Clustering or cluster analysis is a machine learning technique, which groups the unlabelled dataset. It can be defined as **"A way of grouping the data points into different clusters, consisting of similar data points. The objects with the possible similarities remain in a group that has less or no similarities with another group."**
- It does it by finding some similar patterns in the unlabelled dataset such as shape, size, color, behavior, etc., and divides them as per the presence and absence of those similar patterns.
- It is an unsupervised learning method, hence no supervision is provided to the algorithm, and it deals with the unlabeled dataset.
- After applying this clustering technique, each cluster or group is provided with a cluster-ID. ML systems can use this id to simplify the processing of large and complex datasets.
- The clustering technique is commonly used for **statistical data analysis.**

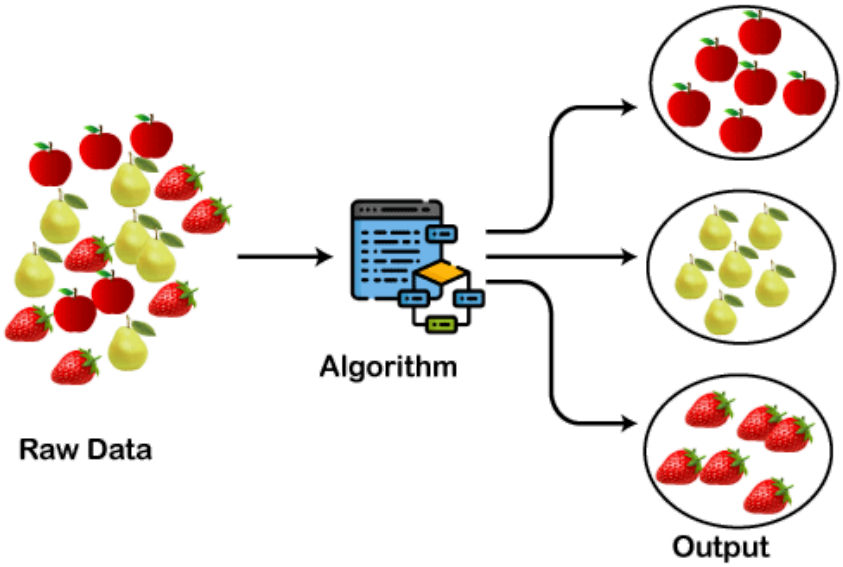
Example: Let's understand the clustering technique with the real-world example of Mall: When we visit any shopping mall, we can observe that the things with similar usage are grouped together. T-shirts are grouped in one section, and trousers are in other sections. Similarly, at vegetable sections, apples, bananas, Mangoes, etc., are grouped in separate sections, so that we can easily find out the things. The clustering technique also works in the same way. Other examples of clustering are grouping documents according to the topic.

The clustering technique can be widely used in various tasks. Some most common uses of this technique are:

- Market Segmentation
- Statistical data analysis
- Social network analysis
- Image segmentation
- Anomaly detection, etc.

Apart from these general usages, it is used by Amazon in its recommendation system to provide the recommendations as per the past search of products. **Netflix** also uses this technique to recommend the movies and web-series to its users as per the watch history.

The below diagram explains the working of the clustering algorithm. We can see the different fruits are divided into several groups with similar properties.



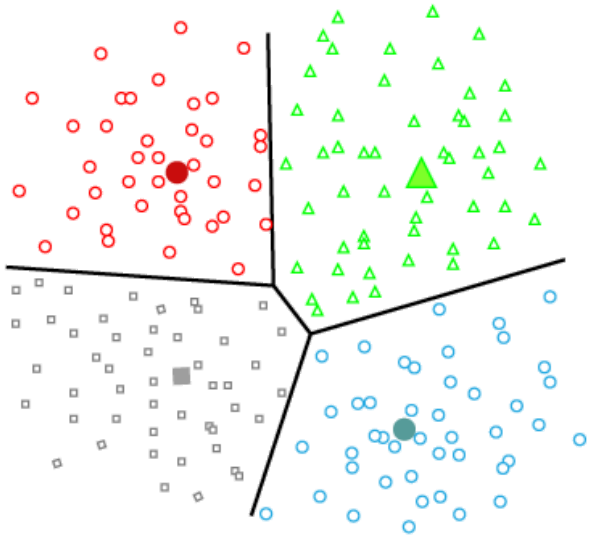
Types of Clustering Methods:

The clustering methods are broadly divided into **Hard clustering** (datapoint belongs to only one group) and **Soft Clustering** (data points can belong to another group also). But there are also other various approaches of Clustering exist. Below are the main clustering methods used in Machine learning:

1. Partitioning Clustering
2. Density-Based Clustering
3. Distribution Model-Based Clustering
4. Hierarchical Clustering
5. Fuzzy Clustering

Partitioning Clustering:

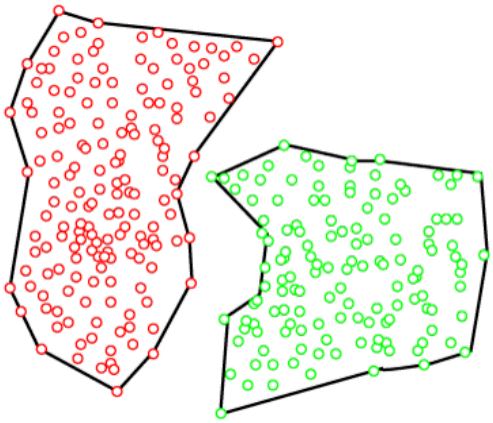
It is a type of clustering that divides the data into non-hierarchical groups. It is also known as the **centroid-based method**. The most common example of partitioning clustering is the **K-Means Clustering algorithm**.
In this type, the dataset is divided into a set of k groups, where K is used to define the number of pre-defined groups. The cluster center is created in such a way that the distance between the data points of one cluster is minimum as compared to another cluster centroid.



Density-Based Clustering:

The density-based clustering method connects the highly-dense areas into clusters, and the arbitrarily shaped distributions are formed as long as the dense region can be connected. This algorithm does it by identifying different clusters in the dataset and connects the areas of high densities into clusters. The dense areas in data space are divided from each other by sparser areas.

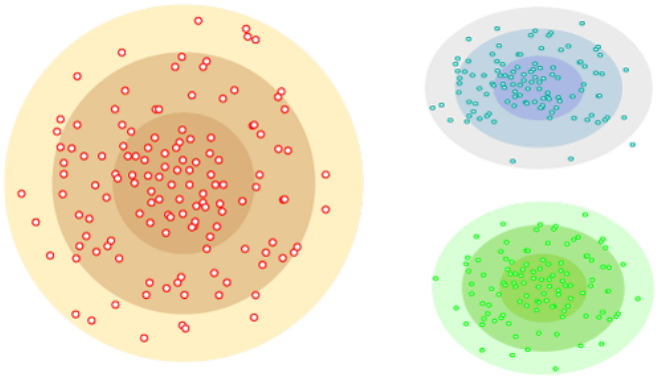
These algorithms can face difficulty in clustering the data points if the dataset has varying densities and high dimensions.



Distribution Model-Based Clustering:

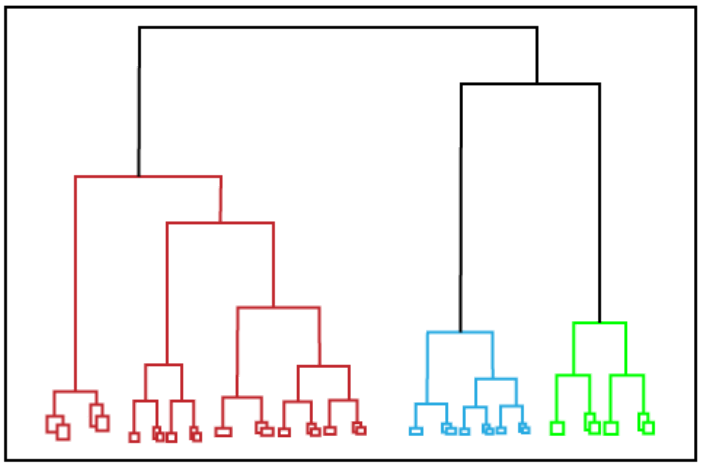
In the distribution model-based clustering method, the data is divided based on the probability of how a dataset belongs to a particular distribution. The grouping is done by assuming some distributions, commonly Gaussian Distribution.

The example of this type is the Expectation-Maximization Clustering algorithm that uses Gaussian Mixture Models (GMM).



Hierarchical Clustering

Hierarchical clustering can be used as an alternative for the partitioned clustering as there is no requirement of pre-specifying the number of clusters to be created. In this technique, the dataset is divided into clusters to create a tree-like structure, which is also called a **dendrogram**. The observations or any number of clusters can be selected by cutting the tree at the correct level. The most common example of this method is the **Agglomerative Hierarchical algorithm**.



Fuzzy Clustering:

Fuzzy clustering is a type of soft method in which a data object may belong to more than one group or cluster. Each dataset has a set of membership coefficients, which depend on the degree of membership to be in a cluster. The **Fuzzy C-means algorithm** is the example of this type of clustering; it is sometimes also known as the Fuzzy k-means algorithm.

Clustering algorithms

K-Means Clustering:

- K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabeled dataset into different clusters. Here K defines the number of predefined clusters that need to be created in the process, as if K=2, there will be two clusters, and for K=3, there will be three clusters, and so on.
- It is an iterative algorithm that divides the unlabeled dataset into K different clusters in such a way that each dataset belongs to only one group that has similar properties.
- It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabeled dataset on its own without the need for any training.
- It is a centroid-based algorithm, where each cluster is associated with a centroid. The main aim of this algorithm is to minimize the sum of distances between the data point and their corresponding clusters.

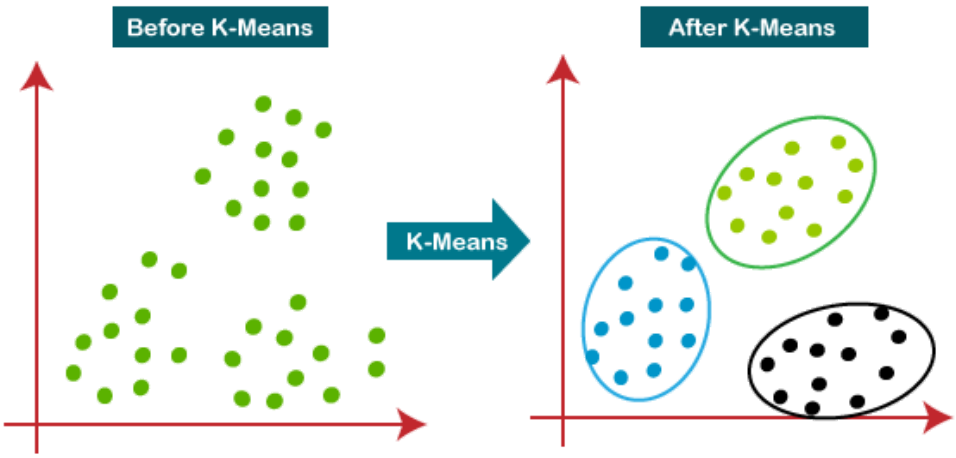
The algorithm takes the unlabeled dataset as input, divides the dataset into k-number of clusters, and repeats the process until it does not find the best clusters. The value of k should be predetermined in this algorithm.

The k-means clustering algorithm mainly performs two tasks:

- Determines the best value for K center points or centroids by an iterative process.
- Assigns each data point to its closest k-center. Those data points which are near to the particular k-center, create a cluster.

Hence each cluster has data points with some commonalities, and it is away from other clusters.

The below diagram explains the working of the K-means Clustering Algorithm:



Working of K-Means Clustering:

The working of the K-Means algorithm is explained in the below steps:

Step-1: Select the number K to decide the number of clusters.

Step-2: Select random K points or centroids. (It can be different from the input dataset).

Step-3: Assign each data point to their closest centroid, which will form the predefined K clusters.

Step-4: Calculate the variance and place a new centroid of each cluster.

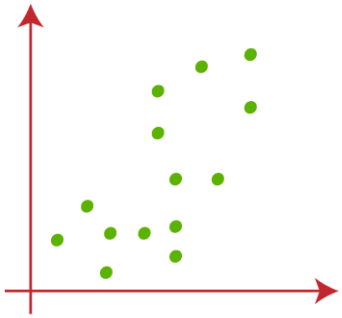
Step-5: Repeat the third steps, which means reassign each datapoint to the new closest centroid of each cluster.

Step-6: If any reassignment occurs, then go to step-4 else go to FINISH.

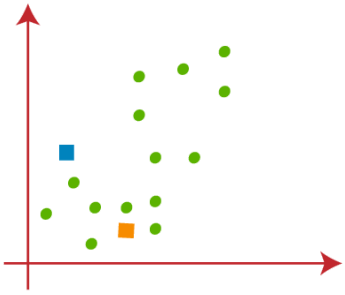
Step-7: The model is ready.

Let's understand the above steps by considering the visual plots:

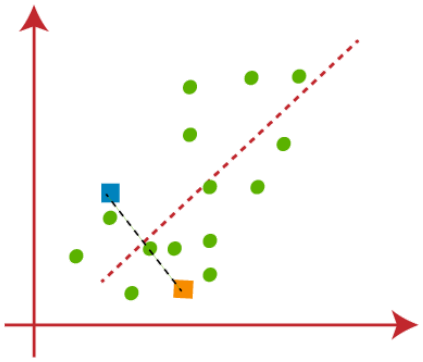
Suppose we have two variables M1 and M2. The x-y axis scatter plot of these two variables is given below:



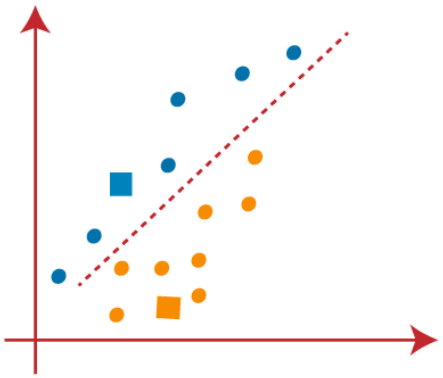
- Let's take the number k of clusters, i.e., K=2, to identify the dataset and to put them into different clusters. It means here we will try to group these datasets into two different clusters.
- We need to choose some random k points or centroid to form the cluster. These points can be either the points from the dataset or any other point. So, here we are selecting the below two points as k points, which are not the part of our dataset. Consider the below image:



Now we will assign each data point of the scatter plot to its closest K-point or centroid. We will compute it by applying some mathematics that we have studied to calculate the distance between two points. So, we will draw a median between both the centroids. Consider the below image:

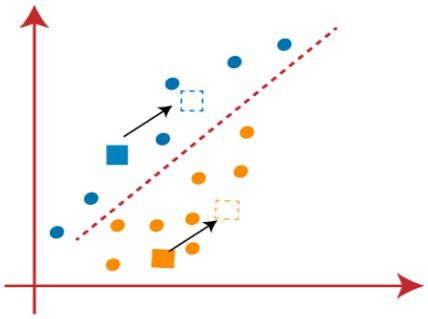


From the above image, it is clear that points on the left side of the line is near to the K1 or blue centroid, and points to the right of the line are close to the yellow centroid. Let's color them as blue and yellow for clear visualization

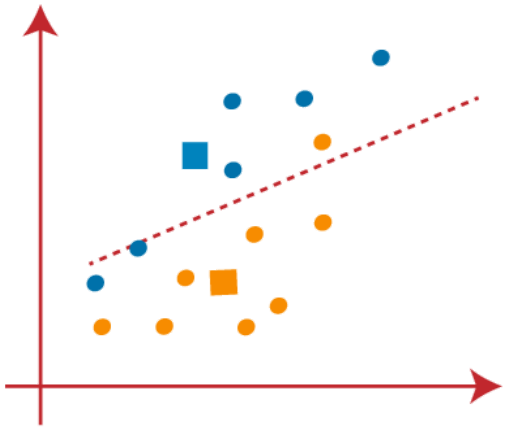


.

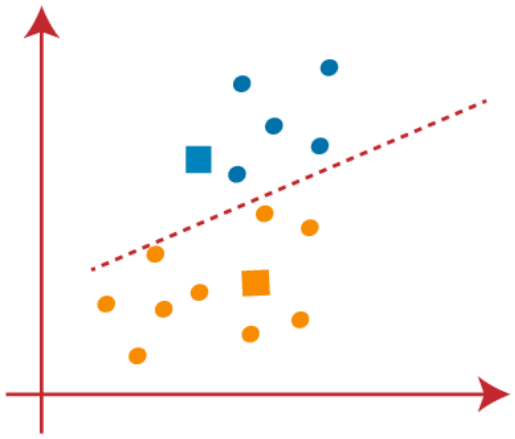
As we need to find the closest cluster, so we will repeat the process by choosing **a new centroid**. To choose the new centroids, we will compute the center of gravity of these centroids, and will find new centroids as below:



Next, we will reassign each datapoint to the new centroid. For this, we will repeat the same process of finding a median line. The median will be like below image:



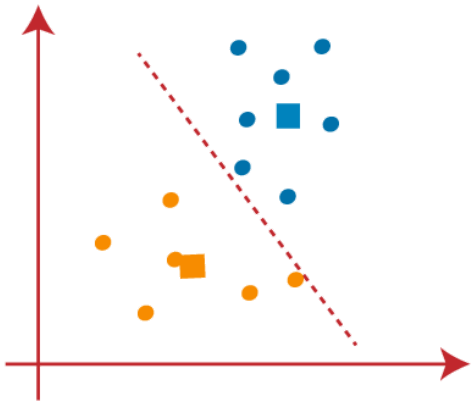
From the above image, we can see, one yellow point is on the left side of the line, and two blue points are right to the line. So, these three points will be assigned to new centroids.



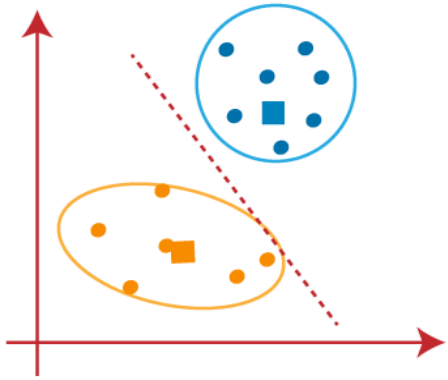
As reassignment has taken place, we will again go to step-4, which is finding new centroids or K-points.

- We will repeat the process by finding the center of gravity of centroids, so the new centroids will be as shown in the below image:

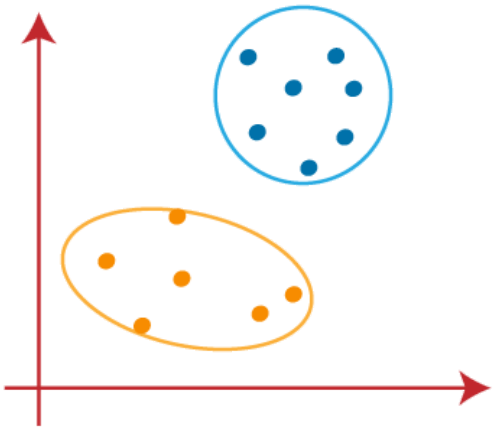
As we got the new centroids so again will draw the median line and reassign the data points. So, the image will be:



We can see in the above image; there are no dissimilar data points on either side of the line, which means our model is formed. Consider the below image:



As our model is ready, so we can now remove the assumed centroids, and the two final clusters will be as shown in the below image:



How to choose the value of "K number of clusters" in K-means Clustering?

The performance of the K-means clustering algorithm depends upon highly efficient clusters that it forms. But choosing the optimal number of clusters is a big task. There are some different ways to find the optimal number of clusters, but here we are discussing the most appropriate method to find the number of clusters or value of K. The method is given below:

Elbow Method:

The Elbow method is one of the most popular ways to find the optimal number of clusters. This method uses the concept of WCSS value. **WCSS** stands for **Within Cluster Sum of Squares**, which defines the total variations within a cluster. The formula to calculate the value of WCSS (for 3 clusters) is given below:

$$WCSS= \sum_{Pi \text{ in } Cluster1} distance(P_i\ C_1)^2 + \sum_{Pi \text{ in } Cluster2} distance(P_i\ C_2)^2 + \sum_{Pi \text{ in } CLuster3} distance(P_i\ C_3)^2$$

In the above formula of WCSS,

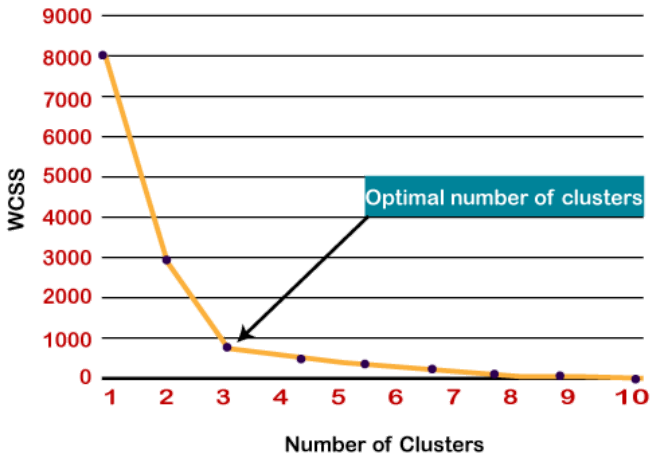
$\sum_{Pi \text{ in } Cluster1} distance(P_i\ C_1)^2$: It is the sum of the square of the distances between each data point and its centroid within a cluster1 and the same for the other two terms.

To measure the distance between data points and centroid, we can use any method such as Euclidean distance or Manhattan distance.

To find the optimal value of clusters, the elbow method follows the below steps:

- It executes the K-means clustering on a given dataset for different K values (ranges from 1-10).
- For each value of K, calculate the WCSS value.
- Plots a curve between calculated WCSS values and the number of clusters K.
- The sharp point of bend or a point of the plot looks like an arm, then that point is considered as the best value of K.

Since the graph shows the sharp bend, which looks like an elbow, hence it is known as the elbow method. The graph for the elbow method looks like the below image:



Example:

```
In [1]: # Importing the libraries
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
```

```
In [2]: # Importing the dataset
        dataset = pd.read_csv('Mall_Customers.csv')
        dataset.head()
```

Out[2]:

	CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

In [5]:
```
dataset.isnull().sum()
```

Out[5]:
```
CustomerID                0
Genre                     0
Age                       0
Annual Income (k$)        0
Spending Score (1-100)    0
dtype: int64
```

In [6]:
```
X = dataset.iloc[:, [3, 4]].values
```

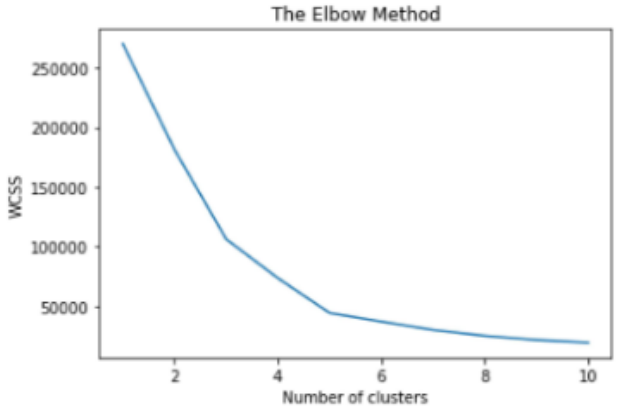
In [7]:
```
# Kmeans Model
from sklearn.cluster import KMeans
```

In [8]:
```
# Using the elbow method to find the optimal number of clusters
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
```

In [9]:
```
wcss
```

Out[9]:
```
[269981.28000000014,
 181363.59595959607,
 106348.37306211119,
 73679.78903948837,
 44448.45544793369,
 37265.86520484345,
 30241.34361793659,
 25336.94686147186,
 21850.16528258562,
 19634.554629349972]
```

In [10]:
```
#Elbow Curve
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



In [11]:
```
# fit the K-Means model on the data
kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)
```

In [12]:
```
# predict
y_kmeans = kmeans.fit_predict(X)
```

In [13]:
```
# Visualising the clusters
plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1],
            s = 100, c = 'red', label = 'Cluster 1')

plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1],
            s = 100, c = 'blue', label = 'Cluster 2')

plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1],
            s = 100, c = 'green', label = 'Cluster 3')

plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1],
            s = 100, c = 'cyan', label = 'Cluster 4')

plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1],
            s = 100, c = 'magenta', label = 'Cluster 5')

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            s = 300, c = 'yellow', label = 'Centroids')

plt.title('Clusters of customers')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.legend()
plt.show()
```

Reference for above topic:
https://www.javatpoint.com/k-means-clustering-algorithm-in-machine-learning

Hierarchical Clustering:

Hierarchical clustering is another unsupervised machine learning algorithm, which is used to group the unlabeled datasets into a cluster and also known as **hierarchical cluster analysis** or HCA.

In this algorithm, we develop the hierarchy of clusters in the form of a tree, and this tree-shaped structure is known as the **dendrogram**.
Sometimes the results of K-means clustering and hierarchical clustering may look similar, but they both differ depending on how they work. As there is no requirement to predetermine the number of clusters as we did in the K-Means algorithm.

The hierarchical clustering technique has two approaches:

- **Agglomerative:** Agglomerative is a **bottom-up** approach, in which the algorithm starts with taking all data points as single clusters and merging them until one cluster is left.
- **Divisive:** Divisive algorithm is the reverse of the agglomerative algorithm as it is a **top-down approach.**

Why Hierarchical Clustering?

As we already have other clustering algorithms such as **K-Means Clustering**, then why do we need hierarchical clustering? So, as we have seen in the K-means clustering, there are some challenges with this algorithm, which are a predetermined number of clusters, and it always tries to create the clusters of the same size. To solve these two challenges, we can opt for the hierarchical clustering algorithm because, in this algorithm, we don't need to have knowledge about the predefined number of clusters.

In this topic, we will discuss the Agglomerative Hierarchical clustering algorithm.

Agglomerative Hierarchical clustering:

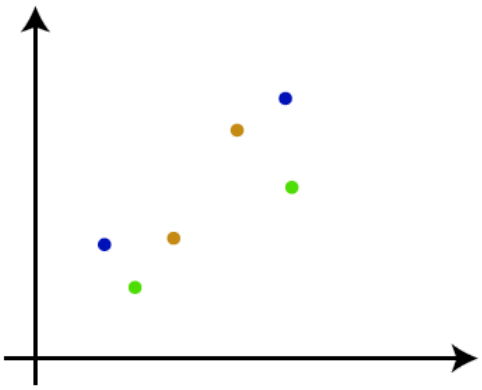
The agglomerative hierarchical clustering algorithm is a popular example of HCA. To group the datasets into clusters, it follows the **bottom-up approach**. It means, this algorithm considers each dataset as a single cluster at the beginning, and then starts combining the closest pair of clusters together. It does this until all the clusters are merged into a single cluster that contains all the datasets.

This hierarchy of clusters is represented in the form of the dendrogram.

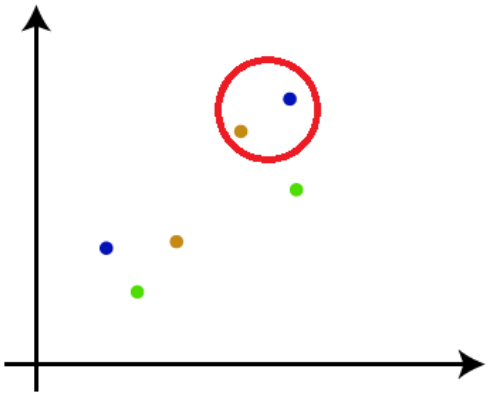
How does Agglomerative Hierarchical clustering Work?

The working of the AHC algorithm can be explained using the below steps:

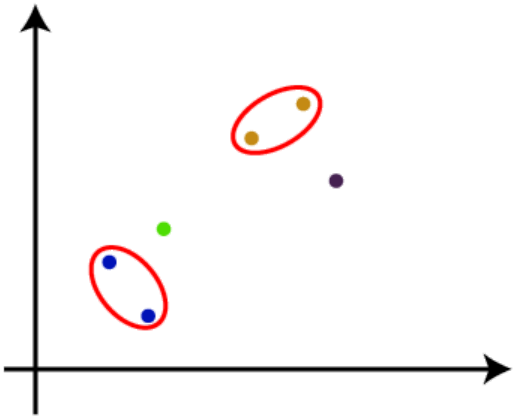
- **Step-1:** Create each data point as a single cluster. Let's say there are N data points, so the number of clusters will also be N.



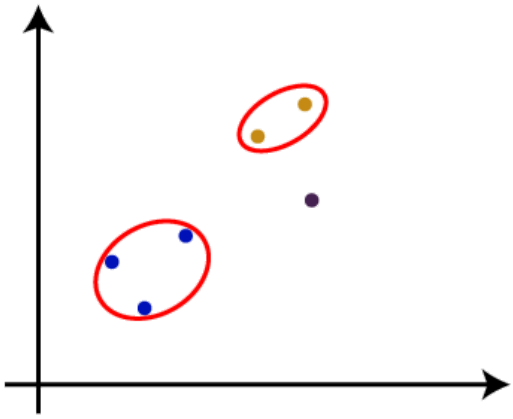
- **Step-2:** Take two closest data points or clusters and merge them to form one cluster. So, there will now be N-1 clusters.

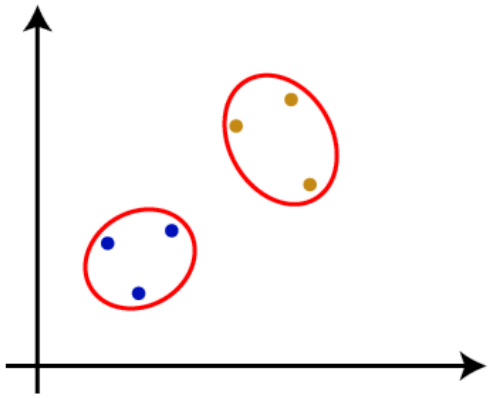


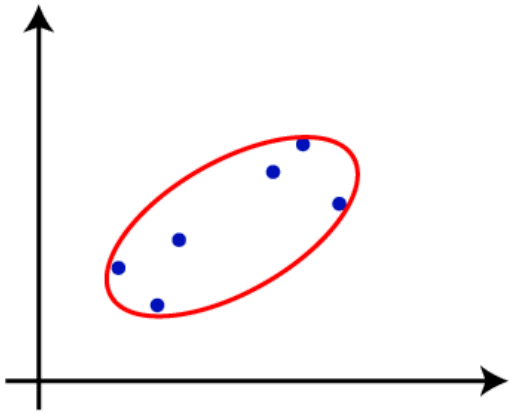
- **Step-3**: Again, take the two closest clusters and merge them together to form one cluster. There will be N-2 clusters.



- **Step-4:** Repeat Step 3 until only one cluster left. So, we will get the following clusters. Consider the below images:





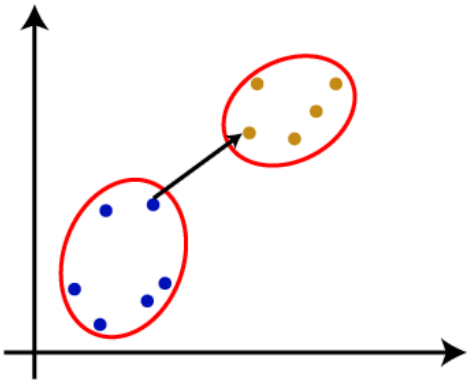


- **Step-5:** Once all the clusters are combined into one big cluster, develop the dendrogram to divide the clusters as per the problem.

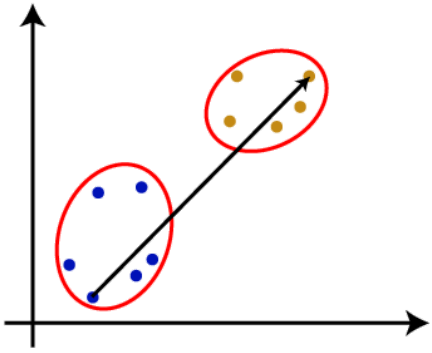
Measure for the distance between two clusters:

As we have seen, the **closest distance** between the two clusters is crucial for hierarchical clustering. There are various ways to calculate the distance between two clusters, and these ways decide the rule for clustering. These measures are called **Linkage methods**. Some of the popular linkage methods are given below:

Single Linkage: It is the Shortest Distance between the closest points of the clusters. Consider the below image:

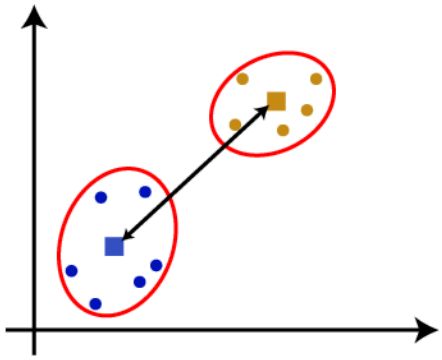


Complete Linkage: It is the farthest distance between the two points of two different clusters. It is one of the popular linkage methods as it forms tighter clusters than single-linkage.



Average Linkage: It is the linkage method in which the distance between each pair of datasets is added up and then divided by the total number of datasets to calculate the average distance between two clusters. It is also one of the most popular linkage methods.

Centroid Linkage: It is the linkage method in which the distance between the centroid of the clusters is calculated. Consider the below image:

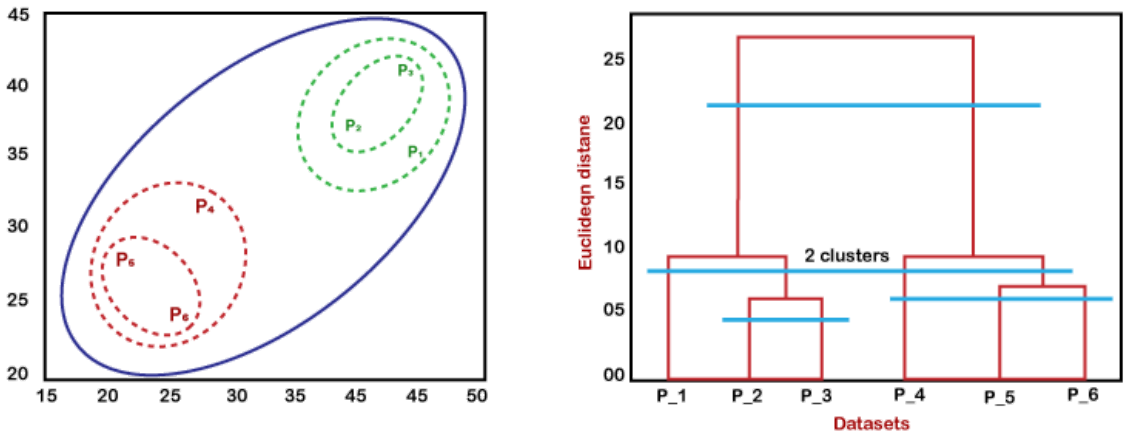


From the above-given approaches, we can apply any of them according to the type of problem or business requirement.

Woking of Dendrogram in Hierarchical clustering:

The dendrogram is a tree-like structure that is mainly used to store each step as a memory that the HC algorithm performs. In the dendrogram plot, the Y-axis shows the Euclidean distances between the data points, and the x-axis shows all the data points of the given dataset.

The working of the dendrogram can be explained using the below diagram:



In the above diagram, the left part is showing how clusters are created in agglomerative clustering, and the right part is showing the corresponding dendrogram.

- As we have discussed above, firstly, the data points P2 and P3 combine together and form a cluster, correspondingly a dendrogram is created, which connects P2 and P3 with a rectangular shape. The height is decided according to the Euclidean distance between the data points.

- In the next step, P5 and P6 form a cluster, and the corresponding dendrogram is created. It is higher than the previous, as the Euclidean distance between P5 and P6 is a little bit greater than the P2 and P3.
- Again, two new dendrograms are created that combine P1, P2, and P3 in one dendrogram, and P4, P5, and P6, in another dendrogram.
- At last, the final dendrogram is created that combines all the data points together.

We can cut the dendrogram tree structure at any level as per our requirement.

Syntax:

From sklearn.cluster import AgglomerativeClustering

model= AgglomerativeClustering(n_clusters=5, affinity='euclidean', linkage='ward')

model.fit(X)

Example:

In [1]:
```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

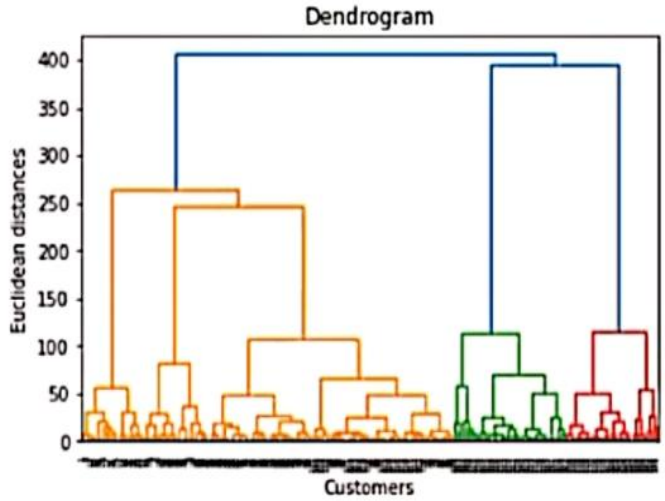
In [2]:
```
# Importing the dataset
dataset = pd.read_csv('Mall_Customers.csv')
X = dataset.iloc[:, [3, 4]].values
```

Using the dendrogram to find the optimal number of clusters

In [3]:
```
import scipy.cluster.hierarchy as sch

dendrogram = sch.dendrogram(sch.linkage(X, method = 'ward'))

plt.title('Dendrogram')
plt.xlabel('Customers')
plt.ylabel('Euclidean distances')
plt.show()
```



Hierarchical Clustering model

In [4]:
```
from sklearn.cluster import AgglomerativeClustering

hc = AgglomerativeClustering(n_clusters = 5, affinity = 'euclidean', linkage = 'v

y_hc = hc.fit_predict(X)
```

Visualizing the clusters:

```
In [5]: plt.scatter(X[y_hc == 0, 0], X[y_hc == 0, 1], s = 100, c = 'red', label = 'Cluste
        plt.scatter(X[y_hc == 1, 0], X[y_hc == 1, 1], s = 100, c = 'blue', label = 'Clust
        plt.scatter(X[y_hc == 2, 0], X[y_hc == 2, 1], s = 100, c = 'green', label = 'Clus
        plt.scatter(X[y_hc == 3, 0], X[y_hc == 3, 1], s = 100, c = 'cyan', label = 'Clust
        plt.scatter(X[y_hc == 4, 0], X[y_hc == 4, 1], s = 100, c = 'magenta', label = 'Cl
        plt.title('Clusters of customers')
        plt.xlabel('Annual Income (k$)')
        plt.ylabel('Spending Score (1-100)')
        plt.legend()
        plt.show()
```

Reference for above topic:

https://www.javatpoint.com/hierarchical-clustering-in-machine-learning

References:

- https://www.javatpoint.com/machine-learning
- https://www.knowledgehut.com/blog/data-science/linear-regression-for-machine-learning
- https://www.analyticsvidhya.com/blog/2021/10/understanding-polynomial-regression-model/
- https://www.analyticsvidhya.com/blog/2021/05/know-the-best-evaluation-metrics-for-your-regression-model/
- https://www.geeksforgeeks.org/regularization-in-machine-learning/
- https://www.javatpoint.com/regularization-in-machine-learning

- https://www.analyticsvidhya.com/blog/2021/07/metrics-to-evaluate-your-classification-model-to-take-the-right-decisions/
- https://www.javatpoint.com/classification-algorithm-in-machine-learning
- https://www.javatpoint.com/logistic-regression-in-machine-learning
- https://www.analyticsvidhya.com/blog/2021/10/building-an-end-to-end-logistic-regression-model/
- https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning
- https://www.javatpoint.com/machine-learning-support-vector-machine-algorithm
- https://www.geeksforgeeks.org/implementing-svm-and-kernel-svm-with-pythons-scikit-learn/
- https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/
- https://www.javatpoint.com/machine-learning-naive-bayes-classifier
- https://www.datacamp.com/tutorial/naive-bayes-scikit-learn
- https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm
- https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/
- https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/
- https://www.javatpoint.com/machine-learning-random-forest-algorithm
- https://www.analyticsvidhya.com/blog/2021/09/adaboost-algorithm-a-complete-guide-for-beginners/?utm_source=reading_list&utm_medium=https://www.analyticsvidhya.com/blog/2022/01/boosting-in-machine-learning-definition-functions-types-and-features/
- https://www.javatpoint.com/gbm-in-machine-learning
- https://www.analyticsvidhya.com/blog/2021/09/gradient-boosting-algorithm-a-complete-guide-for-beginners/
- https://www.javatpoint.com/k-means-clustering-algorithm-in-machine-learning

- https://www.javatpoint.com/hierarchical-clustering-in-machine-learning
-
-