**Machine Learning Notes**

**Machine Learning**

Machine Learning is the science of getting Computers to learn and act like humans do, and improve their learning overtime on autonomous fashion, by feeding those data and information in the form of observations and real-world interactions.

# How does Machine Learning Work?

A Machine Learning System learn from historical data, builds the prediction models, and whenever it receives new data, predicts the output for it.

# Classifications of Machine Learning

* Supervised Learning
* Unsupervised Learning
* Reinforcement Learning

# Advantages of Machine Learning

1. Easily Identifies trends and patterns
2. No human intervention needed (automation)
3. Continuous improvement
4. Handling multi-dimensional and multiversity data
5. Wide Applications

# Disadvantages of Machine Learning

1. Data acquisition
2. Time and Resources
3. Interpretation of Results
4. High error-susceptibility

# Use of Machine Learning



# Machine Learning Road Map

1. Programming Language – Python, R
2. Exploratory Data Analysis
3. Feature Engineering / Data Cleaning
   * Exploratory Data Analysis
   * Handling Missing Values
   * Handling Outliers
   * Categorical Encoding
   * Normalizing & Standardization
4. Feature Selection
   * Correlation
   * Forward Elimination
   * Backward Elimination
   * Univariate Selection
   * Random Forest Importance
   * Feature selection with Decision Trees
5. Machine Learning Algorithms – Regression and Classification, Clustering
6. Linear, Logistic Regression, Decision Tree, Random Forest, Kmeans
7. Hyper Parameter Tuning
8. Gridsearch, RandomisedSreach, Hypropot, Genetics Algorithms
9. Dockers and Kubernetes
10. Model Deployments
11. End to End ML Projects

# Learn Python Libraries for Machine Learning

# Numpy Pandas

# Matplotlib Seaborn

# Tensor Flow Scikit Learn

# SciPy

# Types of Variables in Machine Learning

## Data Types

Variables (columns)

* Numerical Data (int, float)
  + Discrete (that repeat again and again like [1,3,5,3,4,5])
  + Continuous (that given rang of decimal data [1.1,1.2,1.3 to 5] )
* Categorical Data (speaking a objects data)
  + Ordinal (e.g. male, female)
  + Nominal (e.ge python, java, c++, c#, html)

## Date and Time Data

Data time can contain data only, time only

## Mixed Data

Variables which contains numbers and categories data.

# Feature Engineering / Data Cleaning

* + Exploratory Data Analysis
  + Handling Missing Values
  + Handling Outliers
  + Categorical Encoding
  + Normalizing & Standardization cover all topics in data cleaning folder

# Feature Selection Techniques (column selection)

A feature is an attribute that has an impact on a problem or is useful for the problem, and choosing the important features for the model is known as feature selection.

## Forward Elimination (using MLXTEND)



from mlxtend.feature\_selection import SequentialFeatureSelector

x = dataset.iloc[:,:-1] # saperate x-axis and y-axis

y = dataset["species"]

from sklearn.linear\_model import LogisticRegression

lr = LogisticRegression()

fs = SequentialFeatureSelector(lr,k\_features=4, forward=True) /‘False’ for backward elimination

fs.fit(x,y)

fs.feature\_names

fs.k\_feature\_names\_

fs.k\_score\_ here save the score during checking different feature and select high score

## Backward Elimination (using MLXTEND)



fs = SequentialFeatureSelector(lr,k\_features=4, forward=False) change only this for backward elimination and otherwise above syntax is same for backward.

# Train Test Split in Data Set

input\_data = dataframe.iloc[:,:-1] separate data for input

output\_data = dataframe["Embarked"] separate data for output only select last column

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(input\_data, output\_data, test\_size=0.25)

use train data when we train machine learning model and use test data when we check accuracy

# REGRESSION ANALYSIS

## Supervise Learning

Regression Analysis Classification Analysis

# Regression analysis

**Linear Relationship** **Non Linear Relationship**

Simple Linear Regression Polynomial Regression

Multi Linear Regression Decision Tree Regression

Lasso Regression Random Forest Regression

Ridge Regression Support Vector Machine

K-Nearest Neighbor

## Regression Analysis

In a dataset, when determining the type of prediction, you base your decision on the output. If the output or outcome is continuous, you use regression analysis. Otherwise, you use classification analysis.

**Linear Regression**Linear regression is used when the relationship between the independent variable(s) (predictor) and the dependent variable (outcome) is linear. In other words, the data points can be approximately modeled by a straight line.

Linear regression, multi-linear regression, Lasso regression, Ridge regression

### **Nonlinear Regression**

Nonlinear regression is used when the relationship between the independent variable(s) and the dependent variable is nonlinear. This means the data cannot be modeled by a straight line and may follow a curve or more complex relationships.

Polynomial regression, Decision tree regression, Random forest regression, Support Vector Machine, k-nearest neighbor’s regression



## Linear Regression Algorithm (Simple Linear)

Simple Linear Regression Algorithm is a type of Regression Algorithms that models the relationship between a dependent variable and a single independent variable.



sl\_dataset = pd.read\_csv("simple\_linear\_regression\_dataset.csv")

sl\_dataset.head(3)

sl\_dataset.isnull().sum()

x=sl\_dataset[["Feature"]]

y=sl\_dataset["Target"]

plt.figure(figsize=(10, 6))

plt.scatter(x="Feature", y="Target", data=sl\_dataset,color='blue', alpha=0.6, label='Data Points')

plt.title('Scatter Plot of Feature vs Target', fontsize=16)

plt.xlabel('Feature (X)', fontsize=14)

plt.ylabel('Target (Y)', fontsize=14)

plt.axhline(y=0, color='black', linewidth=0.8, linestyle='--')

plt.axvline(x=0, color='black', linewidth=0.8, linestyle='--')

plt.legend()

plt.grid(True, linestyle='--', alpha=0.5)

plt.show()

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y, test\_size=0.2, random\_state=42) # 0 to 70,80

from sklearn.linear\_model import LinearRegression

lr = LinearRegression()

lr.fit(x\_train, y\_train)

lr.score(x\_test,y\_test)\*100

lr.predict([[37.454012]]) this is for user

# y = m\*x + c

lr.coef\_ # m

lr.intercept\_ # c

# y=2.52327729\*37.454012+8.772462946297566

y\_prd = lr.predict(x)

plt.figure(figsize=(10, 6))

plt.scatter(x="Feature", y="Target", data=sl\_dataset,color='blue', alpha=0.6, label='Data Points')

plt.plot(sl\_dataset["Feature"], y\_prd,color = "red", label="predict line")

plt.title('Scatter Plot of Feature vs Target', fontsize=16)

plt.xlabel('Feature (X)', fontsize=14)

plt.ylabel('Target (Y)', fontsize=14)

plt.axhline(y=0, color='black', linewidth=0.8, linestyle='--')

plt.axvline(x=0, color='black', linewidth=0.8, linestyle='--')

plt.legend()

plt.grid(True, linestyle='--', alpha=0.5)

plt.show()

## Multiple Linear Regression

Multiple linear regression is an extension of simple linear regression as it take more than one predictor variable to predict the response variable.

x = ml\_dataset.iloc[:,:-1]

y = ml\_dataset["Target"]

x.ndim

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size=0.2, random\_state=42)

from sklearn.linear\_model import LinearRegression

lr = LinearRegression()

lr.fit(x\_train,y\_train)

lr.score(x\_test,y\_test)

# y m1\*x1+m2\*x2+m3\*x3+c

ml\_lr.coef\_

ml\_lr.intercept\_

# y\_prd = 3.03737113\*Feature1+-1.44414216\*Feature2+2.0121138\*Feature3 + 15.89343046851019

## Polynomial Regression

Polynomial Regression is a regression algorithm that models the relationship between a dependent(y) and independent variable (x) as nth degree polynomial.

x = dataset[["Feature"]]

y = dataset["Target"]

from sklearn.preprocessing import PolynomialFeatures

pf = PolynomialFeatures(degree=2)

pf.fit(x)

x = pf.transform(x)

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2,random\_state=42)

from sklearn.linear\_model import LinearRegression

lr = LinearRegression()

lr.fit(pr\_x\_train,pr\_y\_train)

lr.score(pr\_x\_test,pr\_y\_test)

# y = m1\*x1+m2\*x2^2+c

# y = -1.05954074\*x1 + 0.46752086\*x2^2 - 0.8695309013342438

lr.coef\_ # m

lr.intercept\_ # c

prd = pr\_lr.predict(pr\_x)

plt.scatter(dataset["Feature"],dataset["Target"])

plt.plot(dataset["Feature"], prd, c="red")

plt.show()

**Model Deployment**

test = pf.transform([[4]]) 4 as a example

lr.predict(test)

# What is Cost Function

How to make best fit line.

Cost Function help us to make / predict a best fit line.

A cost function is an important parameter that determines how well a machine learning model performs for a given dataset.

Cost function is a measure of how wrong the model is in estimating the relationship between X (input) and Y (output) parameter.

## Types of Cost Function

* Regression cost function
* Classification cost function

## Regression cost function

Regression models are used to make a prediction for the continuous variables.

* **MSE** (Mean Square Error)
* **RMSE** (Root Mean Square Error)
* **MAE** (Mean Absolute Error)
* **R2** Accuracy

## Binary Classification Cost Function

Classification models are used to make predictions of categorical variables, such as predictions for 0 or 1, Cat or Dog, etc.

## Multi-class Classification Cost Function

A multi-class classification cost function is used in the classification problem for which instances are allocated to one of more than two classes.

* **Binary Cross Entropy Cost Function or Log Loss Function**

# Regression Cost Function

## Mean Square Error:

Mean Square Error (MSE) is the mean square difference between the actual and predicted values. MSE penalizes high errors caused by outliers by squaring the errors.

Mean Square Error is also known as L2 Loss.

Formula search on google MAE=*n*1​∑*i*=1*n*​(*yi*​−*y*^​*i*​)2

## Mean Absolute Error:

Mean Absolute Error (MAE) is the mean absolute difference between the actual values and the predicted values.

MAE is more robust to outliers. The insensitivity to outliers is because it does not penalize high error caused by outliers.

Formula search on google MAE=*n*1​∑*i*=1*n*​∣*yi*​−*y*^​*i*​∣

## Root Mean Square Error:

Root mean square error (RMSE) is the root squared mean of the difference between actual and predicted values.

RMSE can be used in situations where we want to penalize high errors but not as much as MSE does.

Formula search on google RMSE= √MSE

**Practical of (MSE, MAE, RMSE)**

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error, r2\_score

print(mean\_squared\_error(y\_test,lr.predict(x\_test)))

print(mean\_absolute\_error(y\_test,lr.predict(x\_test)))

print(np.sqrt(mean\_squared\_error(y\_test,lr.predict(x\_test))))

# How to Find Best Fit Line

Note in your notebook.

# L1 (Lasso Regularization), L2 (Ridge Regularization)

This is the form of regression, that constrains / regularizes or shrinks the coefficient estimates towards zero.

This techniques discourages learning a more complete or flexible model, so as to avoid the risk of overfitting.

**Regularization can achieve this motive with 2 techniques:**

* Ridge Regularization / L2
* Lasso Regularization / L1

## Lasso Regularization / L1

This is a Regularization Techniques used in Feature Selection using a shrinkage method also referred to as the penalized regression method.

Lasso Regularization magnitude of coefficient can be exactly zero.

Cost Function = Loss + **λ ∑ || W ||**

Loss = sum of squared residual

**λ** = penalty

W = slope of the curve

**Practical**

from sklearn.linear\_model import Lasso

ls = Lasso(alpha=0.5)

ls.fit(x\_train, y\_train)

ls.score(x\_test,y\_test)\*100

## Ridge Regularization / L2

Ridge Regularization, is also known as L2 Regularization, is an extension to linear regression that introduces a regularization term to reduce model complexity and help prevent overfitting.

Ridge Regularization is working value/ magnitude of coefficient is almost equal to zero.

Cost Function = loss + **λ** ∑||W||2

Loss = sum of squared residual

**λ** = penalty

W = slope of the curve

**Practical**

from sklearn.linear\_model import Ridge

ri = Ridge(alpha=10)

ri.fit(x\_train, y\_train)

ri.score(x\_test,y\_test)\*100

# Classification Analysis

* The classification algorithm is used to identify the category of new observations on the basis of training data.
* In Classification, a program learn 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.

## Types of Classifications:

There are two types of classifications

* **Binary Classifier:** If the Classification problem has only two possible outcomes, then it is called as Binary Classifier.

**Examples:** Spam or Not Spam, Cat or Dog etc.

* **Multi-Class Classifier:** If a Classification problem has more than two outcomes, then it is called as Multi Class Classifier.

**Examples:** Classifications of types of crops, Classifications of types of music.

# Types of ML Classifications Algorithms

## Non Linear Models Linear Models

K-Nearest Neighbors Logistic Regression

SVM (Kernel) Support Vector Machines

Naïve Bayes

Decision Tree Classification

Random Forest Classification

## Evaluating a Classification Model / Classification Cost Function

* Log Loss or Cross-Entropy Loss
* Confusion Matrix
* AUC-ROC curve

# Logistic Regression (Binary Classification)

* Logistic regression is one of the most popular machine learning algorithms, which comes under the supervised learning techniques.
* It is used for predicting the categorical dependent variable using a given set of independent variables.
* 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 given the exact values as 0 and 1, it gives the probabilistic which lie between 0 and 1.

## Types of Logistic Regression

On the basis of 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 and 1, Pass and Fail, etc.

**Multinomial:** In multinomial logistic regression, there can be 3 or more possible unordered types of the dependent variables, such as “cat”, “dog”, or ”sheep”.

**Ordinal:** In ordinal logistic regression, there can be 3 or more possible ordered types of the dependent variables, such as “Low”, “Medium”, or ”High”.

# Practice

## Logistic Regression (Binary Classification) (Binary input)

dataset = pd.read\_csv("logistic\_regression\_dataset.csv")

dataset.head()

plt.figure(figsize=(4,3))

sns.scatterplot(x="Feature",y="Target",data=dataset)

plt.show()

x = lo\_dataset[["Feature"]]

y = lo\_dataset["Target"]

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size=0.2, random\_state=42)

from sklearn.linear\_model import LogisticRegression

lo = LogisticRegression()

lo.fit(x\_train,y\_train)

lo.score(x\_test,y\_test)

lr.predict([[40]])

plt.figure(figsize=(4,3))

sns.scatterplot(x="Feature",y="Target",data=dataset)

sns.lineplot(x="Feature",y= lr.predict(x),data=dataset)

plt.show()

## Logistic Regression (Binary Classification) (Multiple input)

dataset = pd.read\_csv("logistic\_regression\_customer\_purchase\_dataset.csv")

dataset.head()

plt.figure(figsize=(5,4))

sns.scatterplot(x="Income",y="AdClicks",data=dataset, hue="Purchase")

plt.show()

x = lo\_dataset.iloc[:,:-1]

y = lo\_dataset["Purchase"]

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size=0.2, random\_state=42)

from sklearn.linear\_model import LogisticRegression

lo = LogisticRegression()

lo.fit(x\_train,y\_train)

lo.score(x\_test,y\_test)

lo.coef\_

lo.intercept\_

lo.predict([[23,100000, 25]])

from mlxtend.plotting import plot\_decision\_regions

plot\_decision\_regions(x.to\_numpy(),y.to\_numpy(), clf=lo)# this will work when have two inputs

plt.show()

## Logistic Regression (Binary Classification) (Polynomial input)

dataset = pd.read\_csv("logistic\_regression\_heart\_disease\_dataset.csv")

dataset.head()

plt.figure(figsize=(5,4))

sns.scatterplot(x="BMI",y="Cholesterol", data=dataset, hue="HeartDisease")

plt.show()

x = dataset[["BMI","Cholesterol"]]

y = dataset["HeartDisease"]

from sklearn.preprocessing import PolynomialFeatures

pf = PolynomialFeatures(degree=3)

pf.fit(x)

x = pd.DataFrame(pf.transform(x))

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2, random\_state=42)

from sklearn.linear\_model import LogisticRegression

lr = LogisticRegression()

lr.fit(x\_train, y\_train)

lr.score(x\_test, y\_test)

# Confusion Matrix

* A confusion matrix is a simple and useful tool for understanding the performance of a classification model, like one used in machine learning or statistics.
* It helps you evaluate how well your model is doing categorizing thinks correctly.
* It is also known as the error matrix.
* The matrix consists of predictions result in a summarized form, which has a total number of correct predictions and incorrect predictions.

## Formula

* Accuracy = TP + TN / N
* Error = FP +FN / N
* False Negative: The model has predicted no, but the actual value was yes, it is also called as **Type-II error**.
* False Positive: The model has predicted yes, but the actual value was no, it is also called as **Type-I error**.

## Confusion Matrix (sensitivity, precision, recall, F1-score)

Precision: TP / TP + FP

It helps us to measure the ability to classify positive samples in the model.

Recall: TP / TP + FN

It helps us to measure how many positive samples were correctly classified by the model.

F1 – Score:

It is the harmonic mean of precision and recall. It takes both false positive and false negative into account. Therefore, it performs well on an imbalanced dataset.

F1 – Score = 2 \* Precision \* Recall / Precision + Recall

## Practical

from sklearn.metrics import confusion\_matrix, precision\_score, recall\_score, f1\_score, accuracy\_score, classification\_report

cf = confusion\_matrix(y\_test, lr.predict(x\_test))

sns.heatmap(cf, annot=True)

plt.show()

precision\_score(y\_test, lr.predict(x\_test))\*100

recall\_score(y\_test, lr.predict(x\_test))\*100

f1\_score(y\_test, lr.predict(x\_test))\*100

# Imbalanced Data

## Techniques to Handle Imbalanced Data

## Random Under Sampling

We will reduce the majority of the class So that it will have same no of as the minority.

x = dataset[["BMI", "Cholesterol"]]

y = dataset["HeartDisease"]

y.value\_counts()

from imblearn.under\_sampling import RandomUnderSampler

ru = RandomUnderSampler()

ru\_x, ru\_y = ru.fit\_resample(x,y)

ru\_y.value\_counts()

## Random Over Sampling

We will increase the size of minority is inactive class to the size of majority class is active.

x = dataset[["BMI", "Cholesterol"]]

y = dataset["HeartDisease"]

y.value\_counts()

from imblearn.over\_sampling import RandomOverSampler

ro = RandomOverSampler()

ro\_x, ro\_y = ro.fit\_resample(x,y)

ro\_y.value\_counts()

# Naive Bayes

* Naïve Bayes is a classification algorithm based on Bayes’ theorem.
* Which is a probability theory that describes the probability of an event, based on prior knowledge of conditions that might be related to the event.
* Naive: It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features.
* Bayes: It is called Bayes because it depends on the principle of Bayes’ theorem.

## Bayes’ Theorem

Bayes’ Theorem is known as Bayes’ Rule or Bayes’ Law, which is used to determine the probability of hypothesis with prior knowledge. It depends on the conditional probability.

P(A | B) = P( B | A) P(A) / P(B)

## Types of Naïve Bayes Model:

There are three types of Naïve Bayes Model,

Which are given below:

* Gaussian
* Multinomial
* Bernoulli

## Gaussian Naïve Bayes

* Assumes that continuous feature follow a Gaussian (normal) distribution.
* Suitable for features that are continuous and have a normal distribution.

## Multinomial Naïve Bayes

* Assumes that features follow a multinomial distribution.
* Typically used for discrete data, such as text data, where each feature represents the frequency of a term.

## Bernoulli Naïve Bayes

* Assumes that feature are binary (Boolean) variables.
* Suitable for data that can be represented as binary features, such as document classification problems where each term is either present or absent.

# Practical

dataset = pd.read\_csv('naive\_bayes\_email\_spam\_dataset.csv')

dataset.head()

dataset.isnull().sum()

plt.figure(figsize=(4,3))

sns.kdeplot(data=dataset["NumWords"])

plt.show()

plt.figure(figsize=(4,3))

sns.scatterplot(x='NumWords', y='NumLinks', data =dataset, hue='IsSpam')

plt.show()

x = dataset.iloc[:, :-1]

y = dataset["IsSpam"]

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y, test\_size=0.2, random\_state=42)

from sklearn.naive\_bayes import GaussianNB, MultinomialNB, BernoulliNB

## GuassianNB

gnb = GaussianNB()

gnb.fit(x\_train, y\_train)

gnb.score(x\_test, y\_test)

## MultinomialNB

mnb = MultinomialNB()

mnb.fit(x\_train, y\_train)

mnb.score(x\_train, y\_train),mnb.score(x\_test, y\_test)

## BernoulliNB

bnb = BernoulliNB()

bnb.fit(x\_train, y\_train)

bnb.score(x\_train, y\_train),bnb.score(x\_test, y\_test)

from mlxtend.plotting import plot\_decision\_regions

plot\_decision\_regions(x.to\_numpy(),y.to\_numpy(),clf=gnb) use when input feature is 2

# Decision Tree

* Decision Tree is a supervised machine learning technique that can be used for both classification and regression problems, but mostly it is preferred for solving classification problems.
* In order to build a tree, we use the CART algorithm, which stands for Classification And Regression Tree algorithm.
* Root Node: It represents the entire population or sample and this further get divided into two or more homogeneous sets.
* Splitting: It is a process of dividing a node into two or more sub-nodes.
* Decision Node: When a sub-node splits further into sub-nodes, then it is called the decision node.
* Leaf / Terminal Node: Nodes do not split is called Leaf / Terminal Node.
* Pruning: When we removes sub-nodes of a decision node. This process is called pruning. You can say the opposite process of splitting.
* Branch / Sub-Tree: A subsection of the entire tree is called branch / sub-tree.
* Parent and Child Node: A node, which is divided into sub-nodes is called a parent node of sub-nodes whereas sub-nodes are the child of the parent node.

## Attribute Selection Measures

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
* Entropy / Gini Index

Entropy: Entropy is a matric to measure the impurity in a given attribute. It specifies randomness in data.

**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

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.

**Information Gain = Entropy(S) – [(Weighted Avg) \*Entropy(each feature)]**

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.

**Gini Index = 1 - ∑ jpj2**

## Practical (decision tree classification)

dataset = pd.read\_csv("decision\_tree\_loan\_approval\_dataset.csv")

dt\_dataset.head()

dt\_dataset.isnull().sum()

x = dt\_dataset.iloc[:,:-1]

y = dt\_dataset["LoanApproved"]

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

x = pd.DataFrame(sc.fit\_transform(x), columns=x.columns)

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2, random\_state=42)

from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier(criterion="entropy") gini index / criterion="entropy", max\_depth=5 for **pre pruning**

dt.fit(x\_train, y\_train)

dt.score(x\_test, y\_test)

dt.predict([[141958, 460, 25, 15616,3]])

from sklearn.tree import plot\_tree

plt.figure(figsize=(15,10))

plot\_tree(dt)

plt.show()

## PRE AND POST PRUNING IN A DECISION TREE

**Pre pruning**  
dt = DecisionTreeClassifier(max\_depth=5)

**Post pruning**

for i in range(1, 20):

dt = DecisionTreeClassifier(max\_depth=i)

dt.fit(x\_train, y\_train)

print(dt.score(x\_test,y\_test), dt.score(x\_train,y\_train), i)

## Practical (decision tree Regressor)

dataset = pd.read\_csv("decision\_tree\_house\_price\_dataset.csv")

dataset.head()

sns.pairplot(data=dataset)

plt.show()

x = dataset.iloc[:,:-1]

y = dataset["HousePrice"]

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2, random\_state=42)

from sklearn.tree import DecisionTreeRegressor, plot\_tree

dt = DecisionTreeRegressor(max\_depth=4)

dt.fit(x\_train,y\_train)

dt.score(x\_train,y\_train)\*100, dt.score(x\_test,y\_test)\*100

plt.figure(figsize=(15,10))

plot\_tree(dt)

plt.show()

# K-Nearest Neighbors

* K-NN algorithm can be used for regression as well as for classification but mostly it is used for 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.

**Manhattan Distance L1**= |x2 – x1| + |y2 – y1|

**Euclidean Distance L2** = √(x2 – x1)2 + (y2 – y1)2

## Practical - K-Nearest Neighbors (Classification)

knn\_dataset = pd.read\_csv("fruit\_knn\_classification\_dataset.csv")

knn\_dataset.head()

knn\_dataset.shape

knn\_dataset.isnull().sum()

x = knn\_dataset.iloc[:,:-1]

y = knn\_dataset["FruitType"]

sns.scatterplot(x="Weight", y="SugarContent", data=knn\_dataset, hue="FruitType")

plt.show()

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2, random\_state=42)

from sklearn.neighbors import KNeighborsClassifier

knc = KNeighborsClassifier(n\_neighbors=3,p=1) default k=5, other you choice it, default measure **Euclidean distance** represent by p=2, other use you **Manhattan** represent by p=1.

knc.fit(x\_train,y\_train)

knc.score(x\_test,y\_test)\*100, knc.score(x\_train,y\_train)\*100

knc.predict([[186.88, 9.93]])

from mlxtend.plotting import plot\_decision\_regions

plot\_decision\_regions(x.to\_numpy(),y.to\_numpy(),clf=knc)

plt.show()

## Practical - K-Nearest Neighbors (Regression)

knr\_dataset = pd.read\_csv("knn\_regression\_car\_dataset.csv")

knr\_dataset.head()

knr\_dataset.isnull().sum()

x = knr\_dataset.iloc[:,:-1]

y = knr\_dataset["FuelEfficiency"]

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()

x = pd.DataFrame(ss.fit\_transform(x))

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size=0.2,random\_state=42)

from sklearn.neighbors import KNeighborsRegressor

knr = KNeighborsRegressor(n\_neighbors=5, p=1) default k=5, other you choice it, default measure **Euclidean distance** represent by p=2, other use you **Manhattan** represent by p=1.

knr.fit(x\_train,y\_train)

knr.score(x\_test, y\_test)

# Support Vector Machine

## Support Vector Machine (Classification)

Hard Margin: The algorithm aims to find a hyperplane that perfectly separates the data into two classes without any misclassifications.

Soft Margin: The algorithm allows for some misclassifications to find a hyperplane that generalizes better to unseen data and is most robust to outliers.

**Types of Support Vector Machine**

* Linear SVM
* Non Linear SVM

Types of SVM: There are two different types of SVMs, each used for different things:

Simple SVM: Typically used for linear regression and classification problems.

Kernel SVM: Has more flexibility for non-linear data because you can add more features to fit a hyperplane instead of a two-dimensional space.

**Kernel Functions**

* Kernel function play a crucial role in transforming input data into a higher-dimensional space.
* The primary purpose of the kernel functions is to allow SVMs to handle non-linearly separable data by implicitly mapping the input data into a higher-dimensional feature space where linear separation may be more feasible.
* This transformation is done without explicitly calculating the coordination of the points

in that higher-dimensional space.

**Linear : (w, b) = wt x + b**

**Polynomial : (w, b) = (ᵧwt x + b)N**

**Gaussian RBF : (w, b) = exp (-ᵧ|| xi - xj||N**

**Sigmoid : K(xi, xj) = tanh (αxiT xj + b)**

## Practical - Support Vector Machine (Classification)

dataset = pd.read\_csv("svm\_classification\_dataset.csv")

dataset.head()

dataset.isnull().sum()

plt.figure(figsize=(4,3))

sns.scatterplot(x="BMI", y="GlucoseLevel", data=dataset, hue="Diabetes")

plt.show()

x = dataset.iloc[:,:-1]

y = dataset["Diabetes"]

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y, test\_size=0.2, random\_state=42)

from sklearn.svm import SVC

sv = SVC(kernel="poly") # linear, poly, rbf, sigmoid, precomputed

sv.fit(x\_train,y\_train)

sv.score(x\_test, y\_test), sv.score(x\_train, y\_train)

from mlxtend.plotting import plot\_decision\_regions

plot\_decision\_regions(x.to\_numpy(),y.to\_numpy(), clf=sv)

plt.show()

## Support Vector Machine (Classification)

Support Vector Regression (SVR) is a regression techniques that uses Support Vector Machine (SVM) for modeling and predicting continuous outcomes.

## Practical - Support Vector Machine (Regression)

dataset = pd.read\_csv("svm\_regression\_dataset.csv")

dataset.head(3)

dataset.isnull().sum()

plt.figure(figsize=(4,3))

sns.scatterplot(x="HouseSize", y="Price", data=dataset)

plt.show()

x = dataset.iloc[:,:-1]

y = dataset["Price"]

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size=0.2,random\_state=42)

from sklearn.svm import SVR

svr = SVR(kernel="linear") # linear, (poly/degree=7), rbf, sigmoid, precomputed

svr.fit(x\_train,y\_train)

svr.score(x\_test,y\_test)\*100, svr.score(x\_train,y\_train)\*100

sns.scatterplot(x="HouseSize", y="Price", data=dataset)

plt.scatter(dataset["HouseSize"], svr.predict(x), color="red")

plt.show()

# Hyper Parameter, Tuning, Model Parameter

Model Parameter are configuration variables that are internal to the model, and a model learn them on its own.

Hyper Parameter are those parameters that are explicitly defined by the user to control the learning process.

The best value can be determined either by the rule of thumb or by trail or error.

## Hyper Parameter Tuning

Models can have many parameters and finding the best combination of parameters can be treated as a search problem. The two best categories for hyper parameter tuning are:

* GridSearchCV
* RandomizedSearchCV

## GridSearchCV

GridSearchCV is a techniques to search through the best parameter values from the given set of the grid of parameters.

## RandomizedSearchCV

It goes though only a fixed number of hyper parameter setting.

It moves within the grid in a random fashion to find the best set of hyper parameters.

## Practical - GridSearchCV

gs\_dataset = pd.read\_csv("decision\_tree\_loan\_approval\_dataset.csv")

gs\_dataset.head()

X = gs\_dataset.iloc[:,:-1]

y = gs\_dataset["LoanApproved"]

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

from sklearn.tree import DecisionTreeClassifier

df = DecisionTreeClassifier(criterion = 'gini', max\_depth= 4, splitter= 'best')

df.fit(X\_train, y\_train)

df.score(X\_test, y\_test), df.score(X\_train,y\_train)

from sklearn.model\_selection import GridSearchCV, RandomizedSearchCV

df = {

"criterion": ["gini", "entropy", "log\_loss"], # valid for classification

"splitter": ["best", "random"],

"max\_depth": [i for i in range(2, 20)]

}

gd = GridSearchCV(DecisionTreeClassifier(), param\_grid=df, cv=5) default cv=1, cv=5 five time model train with different combination

gd.fit(X\_train,y\_train)

gd.best\_params\_

gd.best\_score\_

## Practical - GridSearchCV

Same you can used **RandomizedSearchCV** above.

# Cross-Validation in Machine Learning

Cross-Validation is a technique for validation the model efficiency by training it on the subset of input data and testing on previously unseen subset of the input data.

## Method used for Cross-Validation

* Leave p out cross-validation -
* Leave one out cross-validation -
* Holdout cross-validation
* Repeated random subsampling validation
* K-fold cross-validation -
* Stratified k-fold cross-validation -
* Time Series cross-validation
* Nested cross-validation

## K-fold cross-validation

The original dataset are equally partitioned into a k subparts or folds. Out of k-folds or groups, for each iteration, one group is selected as validation data, and the remaining (k-1) groups are selected as training data. Not suitable for an imbalanced data.

## Stratified k-fold cross-validation

The original dataset are equally partitioned into a k subparts or folds. Out of k-folds or groups, for each iteration, one group is selected as validation data, and the remaining (k-1) groups are selected as training data.

Stratified k-fold cross-validation solved the problem of an imbalanced dataset.

## Leave one out cross-validation

Leave-one-out cross-validation (LOOCV) is an exhaustive cross-validation technique. It is a category of Lp OCV with the case of p=1.

## Leave p out cross-validation

Leave-p-out cross-validation (LpOCV) is an exhaustive cross-validation technique, that involve using p-observation as validation data, and remaining data is used to train the model. This is repeated in all ways to cut the original sample on a validation set of p observations and a training set.

# Practical – Cross-Validation in machine learning

cv\_dataset = pd.read\_csv('simple\_linear\_regression\_dataset.csv')

cv\_dataset.head(3)

X = cv\_dataset.iloc[:,:-1]

y = cv\_dataset['Target']

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import cross\_val\_score, LeaveOneOut, LeavePOut, KFold, StratifiedKFold

p = cross\_val\_score(LinearRegression(),X,y, cv=5) # cv=5, cv=LeaveOneOut(), LeavePOut(p=2), KFold(n\_splits=10), StratifiedKFold(n\_splits=10), cv=5 mean five time splitting five time model training and also same n\_splits parameter in KFlod

p.sort()

p

# Unsupervised Learning

* Unsupervised learning is a type of machine learning that learns from unlabeled data.
* This means that the data does not have any pre-existing labels or categories.
* The goal of unsupervised learning is to discover patterns and relationships in the data without any explicit guidance.

**Unsupervised Learning**

Clustering Association

## Popular unsupervised machine learning algorithm:

* K-means Clustering
* Hierarchal Clustering
* DBSCAN Clustering
* Apriori Algorithm
* Principal Component Analysis

# K-Means Clustering

K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabeled dataset into different clusters.

K defines the number of pre-defined clusters that need to be created in the process.

Used on linearly separable data

**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.

WCSS = ∑pi in cluster1 distance(PiC1)2 + ∑pi in cluster2 distance(PiC2)2 + ∑pi in cluster3 distance(PiC3)2

## Practical – k-means clustering

km\_dataset = pd.read\_csv('kmeans\_clustering\_dataset.csv')

km\_dataset.head()

sns.pairplot(data=km\_dataset)

plt.show()

from sklearn.cluster import Kmeans

wcss = []

for i in range(2,21):

km = KMeans(n\_clusters=i, init='k-means++')

km.fit(km\_dataset)

wcss.append(km.inertia\_)

plt.figure(figsize=(10,5))

plt.plot([i for i in range(2,21)],wcss, marker="o")

plt.xlabel("no of clusters")

plt.xticks([i for i in range(2,21)])

plt.ylabel("wcss")

plt.grid(axis ='x')

plt.show()

knn = KMeans(n\_clusters=3)

km\_dataset["Predict"] = knn.fit\_predict(km\_dataset)

sns.pairplot(data=km\_dataset, hue="Predict")

plt.show()

# Hierarchical Clustering

* It is used to group the unlabeled datasets into a cluster and as known as hierarchical cluster analysis and HCA.
* In this algorithm, we develop the hierarchy of clusters in the form of tree, and this tree-shaped structure is known as the dendrogram.

## 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.
* The dendrogram plot, the y-axis shows the Euclidean distance between data points, and the x-axis shows all the data points of the given dataset.

**Hierarchical Clustering technique has two approaches:**

**Agglomerative:** Agglomerative is a bottom-up approach, in which the algorithm starts with taking all data point as single clusters and merging them until one cluster is left.

**Divisive:** Divisive is the reverse of the agglomerative as it is a top-down approach.

## Measure for the distance between two clusters:

The closest distance between the two clusters is crucial for the hierarchical clustering. There are various way to calculate the distance between two clusters, and these ways decide the rule of clustering. These measures are called linkage methods.

* **Single Linkage**
* **Complete Linkage**
* **Average Linkage**
* **Centroid Linkage**

# Practical - Hierarchical Clustering

hc\_dataset = pd.read\_csv('iris.csv')

hc\_dataset.head()

sns.pairplot(data=hc\_dataset)

plt.show()

import scipy.cluster.hierarchy as sc #for dendrogram to find no of cluster

plt.figure(figsize=(20,10))

sc.dendrogram(sc.linkage(hc\_dataset, method='single', metric='euclidean'))

plt.show()

from sklearn.cluster import AgglomerativeClustering #making model

ac = AgglomerativeClustering(n\_clusters=2,linkage='single')

hc\_dataset['Predict'] = ac.fit\_predict(hc\_dataset)

hc\_dataset.head(3)

sns.pairplot(data=hc\_dataset, hue='Predict')

plt.show()

# DBSCAN Clustering

Density-Based Spatial Clustering of Applications with Noise, The clusters found by DBSCAN can be any shape, which can deal with some special cases that other methods cannot.

Use for non-linear datasets

# Practical – DBSCAN Clustering

db\_dataset = pd.read\_csv('dbscan\_clustering\_dataset.csv')

db\_dataset.head()

db\_dataset.isnull().sum()

sns.scatterplot(x='Latitude', y='Longitude', data=db\_dataset)

plt.show()

from sklearn.cluster import DBSCAN

db = DBSCAN(eps=0.4, min\_samples=5)

db\_dataset['Predict'] = db.fit\_predict(db\_dataset)

sns.scatterplot(x='Latitude', y='Longitude', data=db\_dataset, hue='Predict')

plt.show()

# Silhouette Score

* Silhouette score refers to a method of interpretation and validation of consistency within clusters of data.
* Silhouette Coefficient or Silhouette Score is a metric used to calculate the goodness of a clustering techniques.
* Its value ranges from -1 to 1.

Formula of silhouette score is note in notebook.

# Practical – Silhouette Score

km\_dataset = pd.read\_csv('kmeans\_clustering\_dataset.csv')

km\_dataset.head()

sns.pairplot(data=km\_dataset)

plt.show()

from sklearn.cluster import KMeans

wcss = []

for i in range(2,21):

km = KMeans(n\_clusters=i, init='k-means++')

km.fit(km\_dataset)

wcss.append(km.inertia\_)

plt.figure(figsize=(10,5))

plt.plot([i for i in range(2,21)],wcss, marker="o")

plt.xlabel("no of clusters")

plt.xticks([i for i in range(2,21)])

plt.ylabel("wcss")

plt.grid(axis ='x')

plt.show()

knn = KMeans(n\_clusters=3)

km\_dataset["Predict"] = knn.fit\_predict(km\_dataset)

**Main part – silhouette score**

from sklearn.metrics import silhouette\_score

silhouette\_score(km\_dataset, labels=knn.labels\_)

ss = []

no\_c = [j for j in range(2,21)]

for i in range(2,21):

km1 = KMeans(n\_clusters=i)

km1.fit(km\_dataset)

ss.append(silhouette\_score(km\_dataset,km1.labels\_))

plt.plot(no\_c,ss)

plt.xlabel('no of cluster')

plt.ylabel('silhouette\_score')

plt.xticks(no\_c)

plt.grid(axis='x')

plt.show()

­­**----------------**

sns.pairplot(data=km\_dataset, hue="Predict")

plt.show()

# Association Rule Learning

Association rule learning is a learning technique used to discover interesting relationships and patterns in large datasets.

It is often apply to transactional data, where items are bought or used together.

## How does association rule learning work

* Support
* Confidence
* Lift

## Support

The first step for us and the algorithm is to find frequently bought items.

It is a straightforward calculation that is based on frequently.

**Support (A) = Transactions (A) / Total Transactions**

## Confidence

We have identified frequently bought items let’s calculate confidence. This will tell us how confident (based on our data) we can be that an item will be purchased, given that item has been purchased.

**Conf (X => Y) = P(X|Y) = supp(X ∩ Y) / supp(X) = no. of transactions containing X and Y / no. of transactions containing X**

## Lift

Given that different items are bought at different frequencies (Strong association)

**Lift (X => Y) = supp(X ∩ Y) / supp(X) \* supp(Y)**

**Lift > 1:** means that the two items are more likely to be bought together.

**Lift < 1:** means that the two items are more likely to be bought separately.

**Lift = 1:** means that there is no association between the two items.

# Types of Association Rule Learning

* **Apriori**
* **Éclat**
* **F-P Growth Algorithm**

## Applications of Association Rule Learning

* Market Basket Analysis
* Medical Diagnosis
* Protein Sequence

# Apriori Algorithm Learning

The Apriori algorithm is an unsupervised machine learning algorithm used for association rule learning. Association rule learning is a data mining technique that identifies frequent patterns, connections and dependencies among different groups of items called itemsets in data.

## Practical – Apriori Algorithm Learning

from mlxtend.preprocessing import TransactionEncoder

from mlxtend.frequent\_patterns import apriori, association\_rules

# Step 1: Load the dataset

df = pd.read\_csv("apriori\_raw\_transactions.csv")

# Step 2: Split items by comma to create list of transactions

transactions = df["Items"].apply(lambda x: x.split(", ")).tolist()

# Step 3: One-hot encode the transactions

te = TransactionEncoder()

te\_array = te.fit\_transform(transactions)

df\_encoded = pd.DataFrame(te\_array, columns=te.columns\_)

# Step 4: Apply Apriori algorithm

frequent\_itemsets = apriori(df\_encoded, min\_support=0.05, use\_colnames=True)

# Step 5: Generate association rules

rules = association\_rules(frequent\_itemsets, metric="lift", min\_threshold=1)

# Display top rules

print("Frequent Itemsets:\n", frequent\_itemsets.head())

print("\nAssociation Rules:\n", rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']].head())

# Frequent Pattern Growth Algorithm (Association Rule) Learning

* **Frequent Pattern set**
* **Ordered-item set**
* **Ordered-item set & Conditional Frequent Pattern Tree is built**
* **Frequent Pattern Rules**

## Practical - Frequent Pattern Growth Algorithm

from mlxtend.preprocessing import TransactionEncoder

from mlxtend.frequent\_patterns import fpgrowth, association\_rules

# Step 1: Load the dataset

df = pd.read\_csv("apriori\_raw\_transactions.csv")

# Step 2: Split items by comma to create list of transactions

transactions = df["Items"].apply(lambda x: x.split(", ")).tolist()

# Step 3: One-hot encode the transactions

te = TransactionEncoder()

te\_array = te.fit\_transform(transactions)

df\_encoded = pd.DataFrame(te\_array, columns=te.columns\_)

# Step 4: Apply Apriori algorithm

frequent\_itemsets = fpgrowth(df\_encoded, min\_support=0.05, use\_colnames=True)

# Step 5: Generate association rules

rules = association\_rules(frequent\_itemsets, metric="lift", min\_threshold=1)

# Display top rules

print("Frequent Itemsets:\n", frequent\_itemsets.head())

print("\nAssociation Rules:\n", rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']].head())

# Ensemble Learning

The Ensemble methods in machine learning combine the insights obtained from multiple learning models to facilitate accurate and improved decisions.

There are two main types of ensemble learning:

1. Bagging (Bootstrap Aggregating)
2. Boosting

**Basic Ensemble Techniques** **Algorithms based on Bagging and Boosting**

* Max Voting • Bagging Algorithms:
* Averaging • Bagging meta-estimator
* Weighted Average • Random forest

• Boosting Algorithms:

**Advanced Ensemble Techniques** • AdaBoost

* Stacking • GBM
* Blending • XGBM
* Bagging • Light GBM
* Boosting • CatBoost

# Max Voting

The max voting method is generally used for classification problems. In this technique, multiple models are used to make predictions for each data point.

The predictions by each model are considered as a ‘vote’. The predictions which we get from the majority of models are used as the final prediction.

# Averaging & Weighted Average Voting

Take an average of predictions from all the models and use it to make the final prediction.

Averaging can be used for making predictions in regression problems while calculating probabilities for classification problems.

## Classification Practice – Max Voting, Averaging & Weighted Average Voting

from sklearn.datasets import make\_moons

x, y = make\_moons(n\_samples=1000, noise=0.2)

df = {"x1":x[:,0], "x2":x[:,1],"y":y}

dataset = pd.DataFrame(df)

sns.scatterplot(x="x1", y="x2", data=dataset, hue="y")

plt.show()

x\_a = dataset.iloc[:,:-1]

y\_a = dataset["y"]

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x\_a, y\_a, test\_size=0.2, random\_state=42)

from sklearn.tree import DecisionTreeClassifier

from sklearn.svm import SVC

from sklearn.naive\_bayes import GaussianNB

dt = DecisionTreeClassifier()

dt.fit(x\_train, y\_train)

dt.score(x\_train, y\_train)\*100, dt.score(x\_test, y\_test)\*100

sv = SVC()

sv.fit(x\_train, y\_train)

sv.score(x\_train, y\_train)\*100, sv.score(x\_test, y\_test)\*100

gnb = GaussianNB()

gnb.fit(x\_train, y\_train)

gnb.score(x\_train, y\_train)\*100, gnb.score(x\_test, y\_test)\*100

from sklearn.ensemble import VotingClassifier

li = [('dt1',DecisionTreeClassifier()), ('sv1',SVC()), ('gnb1',GaussianNB())]

vc = VotingClassifier(li) # weights=[2,5,10]

vc.fit(x\_train, y\_train)

vc.score(x\_train, y\_train)\*100, vc.score(x\_test, y\_test)\*100

pred = {"dt": dt.predict(x\_test), "svm": sv.predict(x\_test), "gnb":gnb.predict(x\_test), "vote":vc.predict(x\_test)}

pd.DataFrame(pred)

## Regression Practice – Max Voting, Averaging & Weighted Average Voting

dataset = pd.read\_csv("knn\_regression\_car\_dataset.csv")

dataset.head(3)

sns.pairplot(dataset)

plt.show()

x = dataset.iloc[:,:-1]

y = dataset["FuelEfficiency"]

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2, random\_state=42)

from sklearn.linear\_model import LinearRegression

from sklearn.tree import DecisionTreeRegressor

from sklearn.svm import SVR

from sklearn.neighbors import KNeighborsRegressor

lr = LinearRegression()

lr.fit(x\_train, y\_train)

lr.score(x\_train, y\_train)\*100, lr.score(x\_test, y\_test)\*100

dt = DecisionTreeRegressor()

dt.fit(x\_train, y\_train)

dt.score(x\_train, y\_train)\*100, dt.score(x\_test, y\_test)\*100

sv = SVR()

sv.fit(x\_train, y\_train)

sv.score(x\_train, y\_train)\*100, sv.score(x\_test, y\_test)\*100

knn = KNeighborsRegressor()

knn.fit(x\_train, y\_train)

knn.score(x\_train, y\_train)\*100, knn.score(x\_test, y\_test)\*100

from sklearn.ensemble import VotingRegressor

li = [("dt1", DecisionTreeRegressor()), ("sv1", SVR()), ("knn1",KNeighborsRegressor()), ("lr",LinearRegression())]

vr = VotingRegressor(li, weights=[10,20,15, 18])

vr.fit(x\_train, y\_train)

vr.score(x\_train, y\_train)\*100, vr.score(x\_test, y\_test)\*100

pred = {"dt": dt.predict(x\_test),"svm": sv.predict(x\_test),"knn": knn.predict(x\_test), "lr": lr.predict(x\_test),"vote": vr.predict(x\_test)}

pd.DataFrame(pred)

# Bagging (Bagging Meta-Estimator, Random Forest)

Bagging (Bootstrap Aggregating) technique uses these subset (bags) to get a fair idea of the distribution (complete set). The size of subsets created for bagging may be less than the original set.

## Bagging Meta-Estimator

Bagging meta-estimator is an ensembling algorithm that can be used for both classification (Bagging Classifier) and regression (Bagging Regressor) problems. It follows the typical bagging techniques to make predictions.

## Random Forest

Random forest is another ensemble machine learning algorithm that follows the bagging technique. It is an extension of the bagging estimator algorithm.

The base estimators in random forest are decision trees.

## Practical - Bagging (Bagging meta-estimator, Random forest) Classification

from sklearn.datasets import make\_moons

x, y = make\_moons(n\_samples=1000, noise=0.2)

df = {"x1":x[:,0], "x2":x[:,1],"y":y}

dataset = pd.DataFrame(df)

sns.scatterplot(x="x1", y="x2", data=dataset, hue="y")

plt.show()

x\_a = dataset.iloc[:,:-1]

y\_a = dataset["y"]

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x\_a, y\_a, test\_size=0.2, random\_state=42)

from sklearn.ensemble import BaggingClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.neighbors import KNeighborsClassifier

bck = BaggingClassifier(estimator=KNeighborsClassifier(), n\_estimators=50)

bck.fit(x\_train, y\_train)

bck.score(x\_train, y\_train)\*100, bck.score(x\_test, y\_test)\*100

rfc = RandomForestClassifier(n\_estimators=50)

rfc.fit(x\_train, y\_train)

rfc.score(x\_train, y\_train)\*100, rfc.score(x\_test, y\_test)\*100

## Practical - Bagging (Bagging meta-estimator, Random forest) Regression

dataset = pd.read\_csv("knn\_regression\_car\_dataset.csv")

dataset.head(3)

x = dataset.iloc[:,:-1]

y = dataset["FuelEfficiency"]

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2, random\_state=42)

from sklearn.ensemble import BaggingRegressor

from sklearn.ensemble import RandomForestRegressor

from sklearn.linear\_model import LinearRegression

bg = BaggingRegressor(estimator=LinearRegression(), n\_estimators=10)

bg.fit(x\_train, y\_train)

bg.score(x\_train, y\_train)\*100, bg.score(x\_test, y\_test)\*100

rf = RandomForestRegressor(n\_estimators=10)

rf.fit(x\_train, y\_train)

bg.score(x\_train, y\_train)\*100, bg.score(x\_test, y\_test)\*100

complete machine learning