

MACHINE LEARNING

LAB FILE



Delhi Technological University

Submitted by

Prashant Tiwari

Roll Number :- 2K20/IT/103

Batch :- IT-B

Submitted to : Assistant Professor Gull Kaur

INDEX

S. No.	Program	Date
1.	Python Basics	
2.	Linear Regression with Gradient Descent	
3.	Linear Regression using normal equation	
4.	Decision Tree	
5.	K-nearest neighbours	
6.	Naive Bayes classifier	
7.	Support Vector Machine	
8.	K-means clustering	

Lab 1

Aim : To explore and demonstrate python to perform basic linear algebra functions

Theory :-

Transpose :- In linear algebra, the **transpose** of a matrix is an operator which flips a matrix over its diagonal; that is, it switches the row and column indices of the matrix **A** by producing another matrix, often denoted by **A^T**

Dot Product :- The **dot product** or **scalar product** is an algebraic operation that takes two equal-length sequences of numbers (usually coordinate vectors), and returns a single number. In Euclidean geometry, the dot product of the Cartesian coordinates of two vectors is widely used. It is often called the **inner product** (or rarely **projection product**) of Euclidean space, even though it is not the only inner product that can be defined on Euclidean space. Algebraically, the dot product is the sum of the products of the corresponding entries of the two sequences of numbers. Geometrically, it is the product of the Euclidean magnitudes of the two vectors and the cosine of the angle between them.

Cross Product :- In mathematics, the **cross product** or **vector product** (occasionally **directed area product**, to emphasize its geometric significance) is a binary operation on two vectors in a three-dimensional oriented Euclidean vector space. Given two linearly independent vectors **a** and **b**, the cross product, **a × b**, is a vector that is perpendicular to both **a** and **b**, and thus normal to the plane containing them.

Determinant :- In mathematics, the **determinant** is a scalar value that is a function of the entries of a square matrix. It allows characterising some properties of the matrix and the linear map represented by the matrix. In particular, the determinant is nonzero if and only if the matrix is invertible and the linear map represented by the matrix is an isomorphism. The determinant of a product of matrices is the product of their determinants. The determinant of a matrix **A** is denoted **det(A)**,

Program and Output :-

CREATING ROW AND COLUMN VECTOR AND SHOWING THEIR SHAPE

```
vector_row = array([[1,-5,3,2,4]])  
vector_column = array([[1],[2],[3],[4]])  
print(vector_row.shape)  
print(vector_column.shape)
```

(1, 5)

(4, 1)

```
v_row = array([[1,2,3],[4,5,6],[7,8,9]])  
v_column = array([[9,8],[7,6]])  
print(v_row.shape)  
print(v_column.shape)
```

(3, 3)

(2, 2)

DOT PRODUCT

```
#FINDING ANGLE BETWEEN TWO VECTORS USING DOT PRODUCT  
v = array([[10, 9, 3]])  
w = array([[2, 5, 12]])  
theta = arccos(dot(v, w.T)/(norm(v)*norm(w)))  
print(theta)
```

[[0.97992471]]

Transpose of a Vector

```
new_vector = vector_row
print(new_vector)
norm_1 = norm(new_vector, 1)
norm_2 = norm(new_vector, 2)
norm_inf = norm(new_vector, inf)
print("L_1 is: %.1f"%norm_1)
print("L_2 is: %.1f"%norm_2)
print("L_inf is: %.1f"%norm_inf)
```

```
[[ 1 -5  3  2  4]]
L_1 is: 5.0
L_2 is: 7.4
L_inf is: 15.0
```

CROSS PRODUCT

```
v = array([[0, 2, 0]])
w = array([[3, 0, 0]])
print(cross(v, w))
```

```
[[ 0  0 -6]]
```

MATRICES

```
P = array([[1, 7], [2, 3], [5, 0], [7, 7]])
Q = array([[2, 6, 3, 1], [1, 2, 3, 4]])
print(P)
print(Q)
print(dot(P, Q))
print(dot(Q, P))
```

```
[[1 7]
 [2 3]
 [5 0]
 [7 7]]
[[2 6 3 1]
 [1 2 3 4]]
[[ 9 20 24 29]
 [ 7 18 15 14]
 [10 30 15  5]
 [21 56 42 35]]
[[36 39]
 [48 41]]
```

#SINGULAR NON SINGULAR MATRICES

```
from numpy.linalg import inv
P = array([[0, 1, 0], [0, 0, 0], [1, 0, 1]])
print("Inverse M:\n", inv(M))
print("Det(p) :\n", det(P))
```

Inverse M:

```
[[-1.57894737 -0.07894737  1.23684211  1.10526316]
 [-0.63157895 -0.13157895  0.39473684  0.84210526]
 [ 0.68421053  0.18421053 -0.55263158 -0.57894737]
 [ 0.52631579  0.02631579 -0.07894737 -0.36842105]]
```

Det(p) :

0.0

```
#SQAURE MATRICES
```

```
from numpy.linalg import det
```

```
M = array([[0,2,1,3], [3,2,8,1], [1,0,0,3], [0,3,2,1]])
```

```
print("M :\n",M)
```

```
print("Determinant : %.1f"%det(M))
```

```
I = eye(4)
```

```
print("I :\n",I)
```

```
print("M*I :\n", dot(M,I))
```

M :

```
[[0 2 1 3]
```

```
[3 2 8 1]
```

```
[1 0 0 3]
```

```
[0 3 2 1]]
```

Determinant : -38.0

I :

```
[[1. 0. 0. 0.]
```

```
[0. 1. 0. 0.]
```

```
[0. 0. 1. 0.]
```

```
[0. 0. 0. 1.]]
```

M*I :

```
[[0. 2. 1. 3.]
```

```
[3. 2. 8. 1.]
```

```
[1. 0. 0. 3.]
```

```
[0. 3. 2. 1.]]
```

#CONDITION NUMBER AND RANK

```
from numpy.linalg import cond, matrix_rank
A = array([[1,1,0],
[0,1,0],
[1,0,1]])

print("Condition number:\n", cond(A))
print("Rank:\n", matrix_rank(A))
y = array([[1], [2], [1]])
A_y = concatenate((A, y), axis = 1)
print("Augmented matrix:\n", A_y)
```

Condition number:

4.048917339522305

Rank:

3

Augmented matrix:

[[1 1 0 1]

[0 1 0 2]

[1 0 1 1]]

SOLVING LINEAR EQUATIONS

```
A = array([[4,3,5],[-2,-4,5],[8,8,0]])
y = array([2,5,-3])
x = linalg.solve(A,y)
print(x)
```

[0.375 -0.75 0.55]

Lab 2

Implementation of Linear regression algorithm with gradient descent in python.

Theory :-

In linear regression, the model targets to get the best-fit regression line to predict the value of y based on the given input value (x). While training the model, the model calculates the cost function which measures the Root Mean Squared error between the predicted value (pred) and true value (y). The model targets to minimise the cost function.

$$Y = mX + c$$

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model.

A learning rate is used for each pair of input and output values. It is a scalar factor and coefficients are updated in direction towards minimizing error. The process is repeated until a minimum sum squared error is achieved or no further improvement is possible.

Let's try applying gradient descent to **m** and **c** and approach it step by step:

1. Initially let $m = 0$ and $c = 0$. Let L be our learning rate. This controls how much the value of **m** changes with each step. L could be a small value like 0.0001 for good accuracy.
2. Calculate the partial derivative of the loss function with respect to m , and plug in the current values of x , y , m and c in it to obtain the derivative value **D**.

$$D_m = \frac{1}{n} \sum_{i=0}^n 2(y_i - (mx_i + c))(-x_i)$$

$$D_m = \frac{-2}{n} \sum_{i=0}^n x_i(y_i - \bar{y}_i)$$

D_m is the value of the partial derivative with respect to m . Similarly let's find the partial derivative with respect to c , D_c :

$$D_c = \frac{-2}{n} \sum_{i=0}^n (y_i - \bar{y}_i)$$

3. Now we update the current value of m and c using the following equation:

$$m = m - L \times D_m$$

$$c = c - L \times D_c$$

4. We repeat this process until our loss function is a very small value or ideally 0 (which means 0 error or 100% accuracy). The value of m and c that we are left with now will be the optimum values.

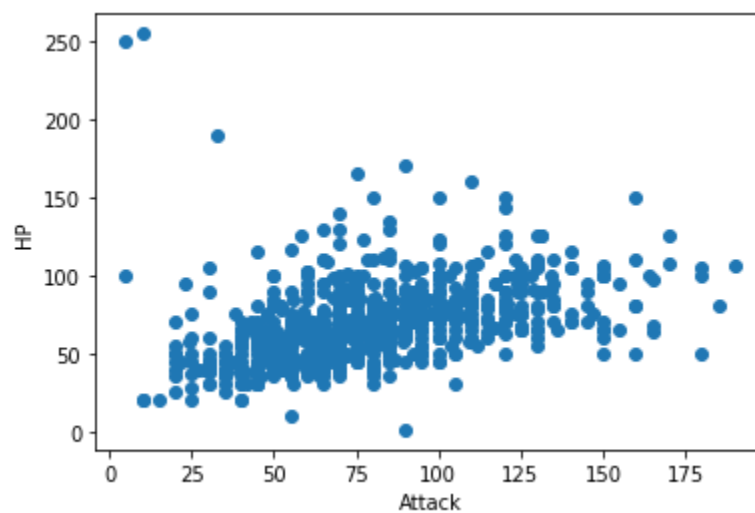
Program and Output

IMPORTING LIBRARIES

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

PROCESSING DATA

```
data = pd.read_csv("pokemon.csv")
x = data.Attack
y = data.HP
plt.xlabel("Attack")
plt.ylabel("HP")
plt.scatter(x,y)
plt.show()
```



BUILDING MODEL

```
#y = mx+c
m=0
c=0

#learning rate
L = 0.0001

#number of operations to perform on gradient descent
epochs = 5000

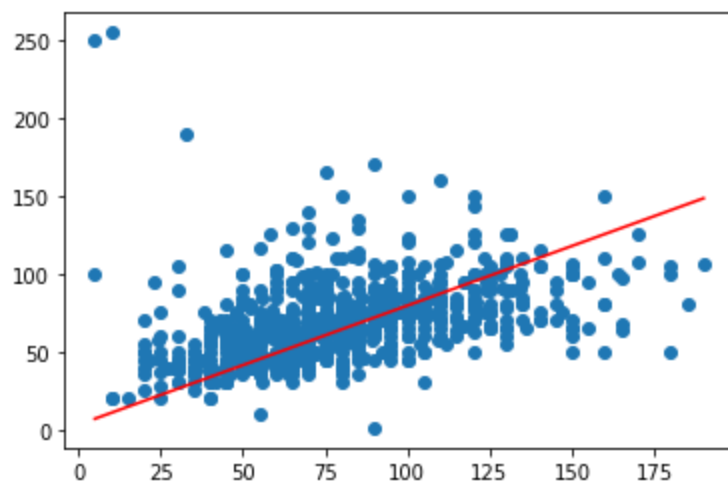
n = float(len(x))
# Performing Gradient Descent
for i in range(epochs):
    y_pred = m*x + c # The current predicted value of Y
    D_m = (-2/n) * sum(x * (y - y_pred)) # Derivative wrt m
    D_c = (-2/n) * sum(y - y_pred) # Derivative wrt c
    m = m - L * D_m # Update m
    c = c - L * D_c # Update c

print (m, c)
```

0.735495697973632 5.784479191969768

MAKING PREDICTIONS

```
y_pred = m*x+c
plt.scatter(x,y)
# regression line
plt.plot([min(x), max(x)], [min(y_pred), max(y_pred)], color='red')
plt.show()
```



Lab 3

Implementation of Linear regression algorithm using normal equation in python.

Theory :-

In linear regression, the model targets to get the best-fit regression line to predict the value of y based on the given input value (x). While training the model, the model calculates the cost function which measures the Root Mean Squared error between the predicted value (pred) and true value (y). The model targets to minimize the cost function.

$$Y = mX + c$$

Normal Equation is the Closed-form solution for the Linear Regression algorithm which means that we can obtain the optimal parameters by just using a formula that includes a few matrix multiplications and inversions.

To calculate θ , we take the partial derivative of the MSE loss function (equation 2) with respect to θ and set it equal to zero. Then, do a little bit of linear algebra to get the value of θ .

$$\theta = (X^T X)^{-1} X^T \vec{y}.$$

Program and Output

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
data = pd.read_table("dat.txt", delimiter = ',')
x = data.col1
y = data.col2
```

```
x_bias = np.ones((len(x), 1))
x = np.array(x)
y = np.array(y)
```

```
print(x.shape, x_bias.shape)
```

(100,) (100, 1)

```
x_new = np.reshape(x,(100, 1))
```

```
x_new = np.append(x_bias,x_new,axis=1)
x_new
```

Output exceeds the [size limit](#). Open the full output data [in a text editor](#)

```
array([[ 1.          , 32.50234527],
       [ 1.          , 53.42680403],
       [ 1.          , 61.53035803],
       [ 1.          , 47.47563963],
       [ 1.          , 59.81320787],
       [ 1.          , 55.14218841],
       [ 1.          , 52.21179669],
       [ 1.          , 39.29956669],
       [ 1.          , 48.10504169],
       [ 1.          , 52.55001444],
       [ 1.          , 45.41973014],
       [ 1.          , 54.35163488],
       [ 1.          , 44.1640495 ],
       [ 1.          , 58.16847072],
       [ 1.          , 56.72720806],
       [ 1.          , 48.95588857],
       [ 1.          , 44.68719623],
       [ 1.          , 60.29732685],
       [ 1.          , 45.61864377],
       [ 1.          , 38.81681754],
       [ 1.          , 66.18981661],
       [ 1.          , 65.41605175],
       [ 1.          , 47.48120861],
       [ 1.          , 41.57564262],
       [ 1.          , 51.84518691],
```

...


```
x_transpose = np.transpose(x_new)
print(x_transpose.shape, x_new.shape)
```

(2, 100) (100, 2)

```
transpose_dot_new = x_transpose.dot(x_new)
transpose_dot_new
```

array([[1.00000000e+02, 4.89583415e+03],
 [4.89583415e+03, 2.49096119e+05]])

```
temp1 = np.linalg.inv(transpose_dot_new)
temp1
```

array([[2.64877549e-01, -5.20600864e-03],
 [-5.20600864e-03, 1.06335478e-04]])

```
y.shape
```

(100,)

```
temp2 = x_transpose.dot(y)
temp2
```

```
array([ 7273.50505537, 368535.14867955])
```

```
theta = temp1.dot(temp2)
theta
```

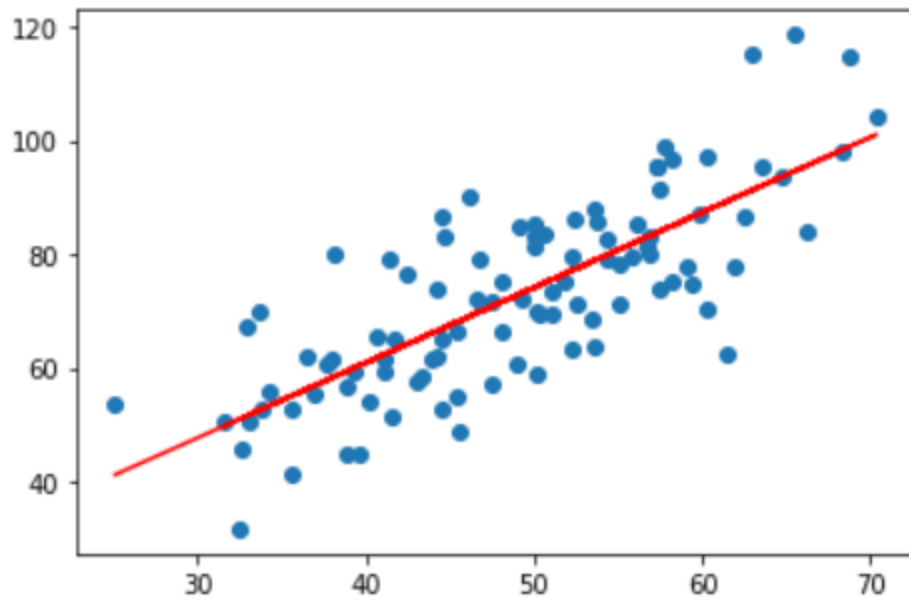
```
array([7.99102098, 1.32243102])
```

```
intercept = theta[0]
slope = theta[1]
print("Intercept : ", intercept)
print("Slope : ", slope)
```

```
Intercept : 7.9910209822691876
```

```
Slope : 1.3224310227553855
```

```
plt.scatter(x,y)
plt.plot(x,slope*x+intercept, color="red")
```



```
def predict(inp, slope, intercept):  
    pred_value = slope*inp+intercept  
    return pred_value
```

```
print(predict(3,slope, intercept))
```

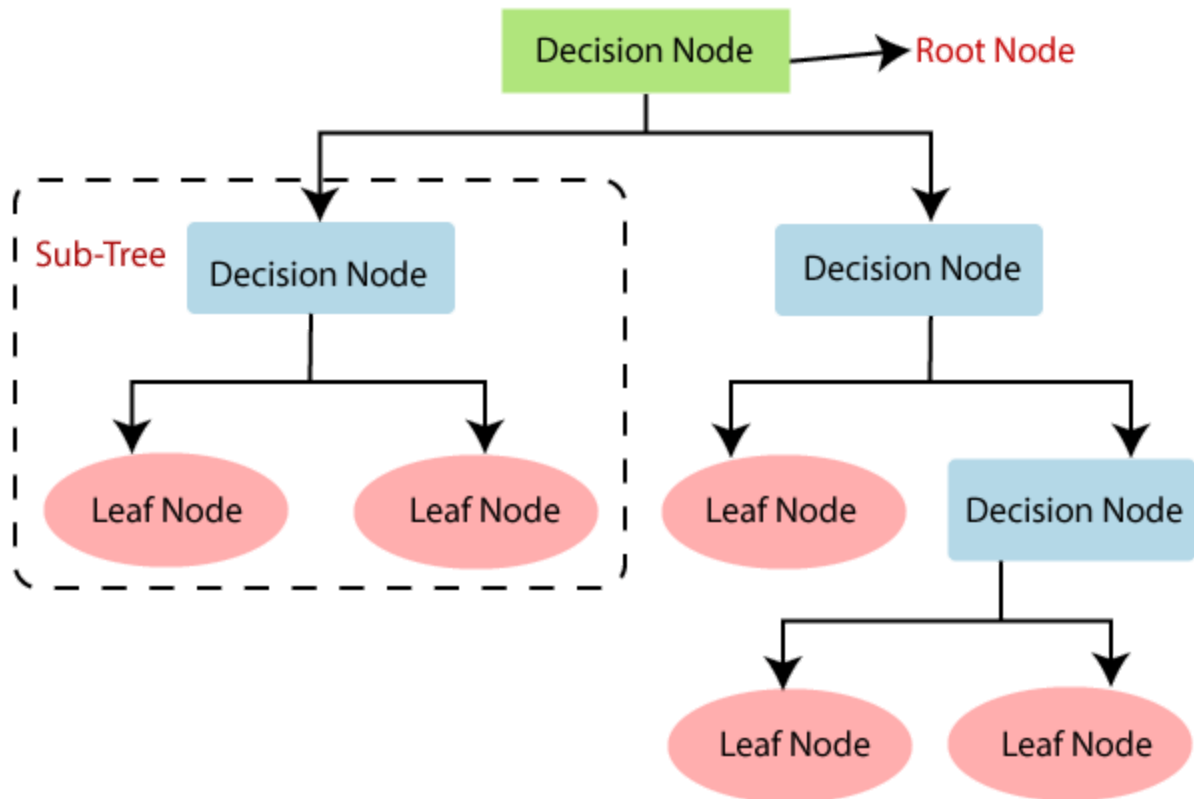
11.958314050535344

Lab 4

Implementation of Decision Tree based on ID3 and CART Algorithms

Theory :-

- 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:



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/Subtree:** 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.

How does the Decision Tree algorithm Work?

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 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.

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**

1. 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:

$$1. \text{ Information Gain} = \text{Entropy}(S) - [(\text{Weighted Avg}) * \text{Entropy}(\text{each feature})]$$

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

$$\text{Entropy}(s) = -P(\text{yes}) \log_2 P(\text{yes}) - P(\text{no}) \log_2 P(\text{no})$$

Where,

- **S= Total number of samples**
- **P(yes)= probability of yes**

- **P(no)= probability of no**

2. 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:

$$\text{Gini Index} = 1 - \sum_j P_j^2$$

PROGRAM AND OUTPUT :

ID3 –


```
import pandas as pd
import numpy as np
import pprint
```

```
df1 = pd.read_csv("ML Lab 4 Data - Sheet1.csv")
df1 = df1.drop(['Day'], axis=1)
df1
```

	Outlook	Temperature	Humidity	Wind	Decision
0	Sunny	Hot	High	Weak	No
1	Sunny	Hot	High	Strong	No
2	Overcast	Hot	High	Weak	Yes
3	Rain	Mild	High	Weak	Yes
4	Rain	Cool	Normal	Weak	Yes
5	Rain	Cool	Normal	Strong	No
6	Overcast	Cool	Normal	Strong	Yes
7	Sunny	Mild	High	Weak	No
8	Sunny	Cool	Normal	Weak	Yes
9	Rain	Mild	Normal	Weak	Yes
10	Sunny	Mild	Normal	Strong	Yes
11	Overcast	Mild	High	Strong	Yes
12	Overcast	Hot	Normal	Weak	Yes
13	Rain	Mild	High	Strong	No

```
def find_entropy(df):
    Class = df.keys()[-1]
    values = df[Class].unique()
    entropy = 0
    for value in values:
        prob = df[Class].value_counts()[value]/len(df[Class])
        entropy += -prob * np.log2(prob)
    return np.float(entropy)
```

```
def find_entropy_attribute(df, attribute):
    Class = df.keys()[-1]
    target_values = df[Class].unique()
    attribute_values = df[attribute].unique()
    avg_entropy = 0
    for value in attribute_values:
        entropy = 0
        for value1 in target_values:
            num = len(df[attribute][df[attribute] == value][df[Class] == value1])
            den = len(df[attribute][df[attribute] == value])
            prob = num/den
            entropy += -prob * np.log2(prob + 0.000001)
        avg_entropy += (den/len(df))*entropy
    return np.float(avg_entropy)
```

```
def find_winner(df):
    IG = []
    for key in df.keys()[:-1]:
        IG.append(find_entropy(df) - find_entropy_attribute(df, key))
    return df.keys()[:-1][np.argmax(IG)]
```

```
def get_subtable(df, attribute, value):  
    return df[df[attribute] == value].reset_index(drop = True)
```

```
def buildtree(df, tree = None):  
    node = find_winner(df)  
    attvalue = np.unique(df[node])  
    Class = df.keys()[-1]  
    if tree is None:  
        tree = {}  
        tree[node] = {}  
    for value in attvalue:  
        subtable = get_subtable(df, node, value)  
        Clvalue, counts = np.unique(subtable[Class], return_counts = True)  
        if len(counts) == 1:  
            tree[node][value] = Clvalue[0]  
        else:  
            tree[node][value] = buildtree(subtable)  
    return tree
```

```
tree = buildtree(df1)  
pprint.pprint(tree)
```

```
{'Outlook': {'Overcast': 'Yes',  
             'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}},  
             'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}}}
```

```
def predict(inst, tree):
    for node in tree.keys():
        value = inst[node]
        tree = tree[node][value]
        prediction = 0
        if type(tree) is dict:
            prediction = predict(inst, tree)
        else:
            prediction = tree
    return prediction
```

```
Y_label = []
for i in range(len(df1)):
    inst = df1.iloc[i,:]
    prediction = predict(inst, tree)
    Y_label.append(prediction)
print(Y_label)
```

['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']

CART –

```
def unique_vals(dataset, col):  
    return set([row[col] for row in dataset])
```

```
header = data.columns  
header
```

```
Index(['Outlook', 'Temperature', 'Humidity', 'Wind', 'Decision'], dtype='object')
```

```
data = data.values.tolist()
```

```
unique_vals(data,0)
```

```
{'Overcast', 'Rain', 'Sunny'}
```

```
def class_counts(dataset):  
    counts = {}  
    for row in dataset:  
        label = row[-1]  
        if label not in counts:  
            counts[label] = 0  
        counts[label] += 1  
    return counts
```

```
class_counts(data)
```

```
{'No': 5, 'Yes': 9}
```

```
def is_numeric(value):  
    return isinstance(value, int) or isinstance(value, float)
```

```
class Question:  
    def __init__(self, column, value):  
        self.column = column  
        self.value = value  
  
    def match(self, example):  
        val = example[self.column]  
        if is_numeric(val):  
            return val >= self.value  
        else:  
            return val == self.value  
  
    def __repr__(self):  
        condition = "=="  
        if is_numeric(self.value):  
            condition = ">="   
        return "Is %s %s %s?" % (  
            header[self.column], condition, str(self.value))
```

```
Question(2, 'Normal')
```

Is Humidity == Normal?

```
q = Question(0, 'Sunny')
```

```
eg = data[0]  
q.match(eg)
```

True

```
def partition(datset, question):  
    true_rows, false_rows = [], []  
    for row in datset:  
        if question.match(row):  
            true_rows.append(row)  
        else:  
            false_rows.append(row)  
    return true_rows, false_rows
```

```
true_rows, false_rows = partition(data, Question(2, 'Normal'))
true_rows
```

```
[['Rain', 'Cool', 'Normal', 'Weak', 'Yes'],
 ['Rain', 'Cool', 'Normal', 'Strong', 'No'],
 ['Overcast', 'Cool', 'Normal', 'Strong', 'Yes'],
 ['Sunny', 'Cool', 'Normal', 'Weak', 'Yes'],
 ['Rain', 'Mild', 'Normal', 'Weak', 'Yes'],
 ['Sunny', 'Mild', 'Normal', 'Strong', 'Yes'],
 ['Overcast', 'Hot', 'Normal', 'Weak', 'Yes']]
```

```
false_rows
```

```
[['Sunny', 'Hot', 'High', 'Weak', 'No'],
 ['Sunny', 'Hot', 'High', 'Strong', 'No'],
 ['Overcast', 'Hot', 'High', 'Weak', 'Yes'],
 ['Rain', 'Mild', 'High', 'Weak', 'Yes'],
 ['Sunny', 'Mild', 'High', 'Weak', 'No'],
 ['Overcast', 'Mild', 'High', 'Strong', 'Yes'],
 ['Rain', 'Mild', 'High', 'Strong', 'No']]
```

```
def gini(dataset):
    counts = class_counts(dataset)
    impurity = 1
    for lbl in counts:
        prob_of_lbl = counts[lbl] / float(len(dataset))
        impurity -= prob_of_lbl**2
    return impurity
```



```
current_uncertainty = gini(data)
current_uncertainty
```

0.4591836734693877

```
def find_best_split(dataset):
    best_gain = 0 # keep track of the best information gain
    best_question = None # keep train of the feature / value that produced it
    current_uncertainty = gini(dataset)
    n_features = len(dataset[0]) - 1 # number of columns

    for col in range(n_features): # for each feature

        values = set([row[col] for row in dataset]) # unique values in the column

        for val in values: # for each value

            question = Question(col, val)

            # try splitting the dataset
            true_rows, false_rows = partition(dataset, question)

            # Skip this split if it doesn't divide the
            # dataset.
            if len(true_rows) == 0 or len(false_rows) == 0:
                continue

            # Calculate the information gain from this split
            gain = info_gain(true_rows, false_rows, current_uncertainty)

            if gain >= best_gain:
                best_gain, best_question = gain, question

    return best_gain, best_question
```

```
best_gain, best_question = find_best_split(data)
```

```
print(best_gain)  
print(best_question)
```

0.10204081632653056

Is Outlook == Overcast?

```
class Leaf:  
    def __init__(self, dataset):  
        self.predictions = class_counts(dataset)
```

```
class Decision_Node:  
    def __init__(self, question, true_branch, false_branch):  
        self.question = question  
        self.true_branch = true_branch  
        self.false_branch = false_branch
```

```

def build_tree(dataset):
    gain, question = find_best_split(dataset)
    if gain == 0:
        return Leaf(dataset)
    true_rows, false_rows = partition(dataset, question)

    # Recursively build the true branch.
    true_branch = build_tree(true_rows)

    # Recursively build the false branch.
    false_branch = build_tree(false_rows)
    return Decision_Node(question, true_branch, false_branch)

```

```

def print_tree(node, spacing=""):
    # Base case: we've reached a leaf
    if isinstance(node, Leaf):
        print (spacing + "Predict", node.predictions)
        return

    # Print the question at this node
    print (spacing + str(node.question))

    # Call this function recursively on the true branch
    print (spacing + '--> True:')
    print_tree(node.true_branch, spacing + " ")

    # Call this function recursively on the false branch
    print (spacing + '--> False:')
    print_tree(node.false_branch, spacing + " ")

```

```
my_tree = build_tree(data)
```

```
print_tree(my_tree)
```

```
Is Outlook == Overcast?
--> True:
    Predict {'Yes': 4}
--> False:
    Is Humidity == Normal?
    --> True:
        Is Wind == Strong?
        --> True:
            Is Temperature == Mild?
            --> True:
                Predict {'Yes': 1}
            --> False:
                Predict {'No': 1}
        --> False:
            Predict {'Yes': 3}
    --> False:
        Is Outlook == Sunny?
        --> True:
            Predict {'No': 3}
        --> False:
            Is Wind == Strong?
            --> True:
                Predict {'No': 1}
            --> False:
                Predict {'Yes': 1}
```

```
def classify(row, node):  
    if isinstance(node, Leaf):  
        return node.predictions  
    if node.question.match(row):  
        return classify(row, node.true_branch)  
    else:  
        return classify(row, node.false_branch)
```

```
test_data = ['Sunny', 'Cool', 'High', 'Weak']  
classify(test_data, my_tree)
```

{'No': 3}

```
test_data = ['Sunny', 'Cool', 'Normal', 'Weak']  
classify(test_data, my_tree)
```

{'Yes': 3}

Lab 5

Implementation of K-nearest neighbour algorithm on a dataset

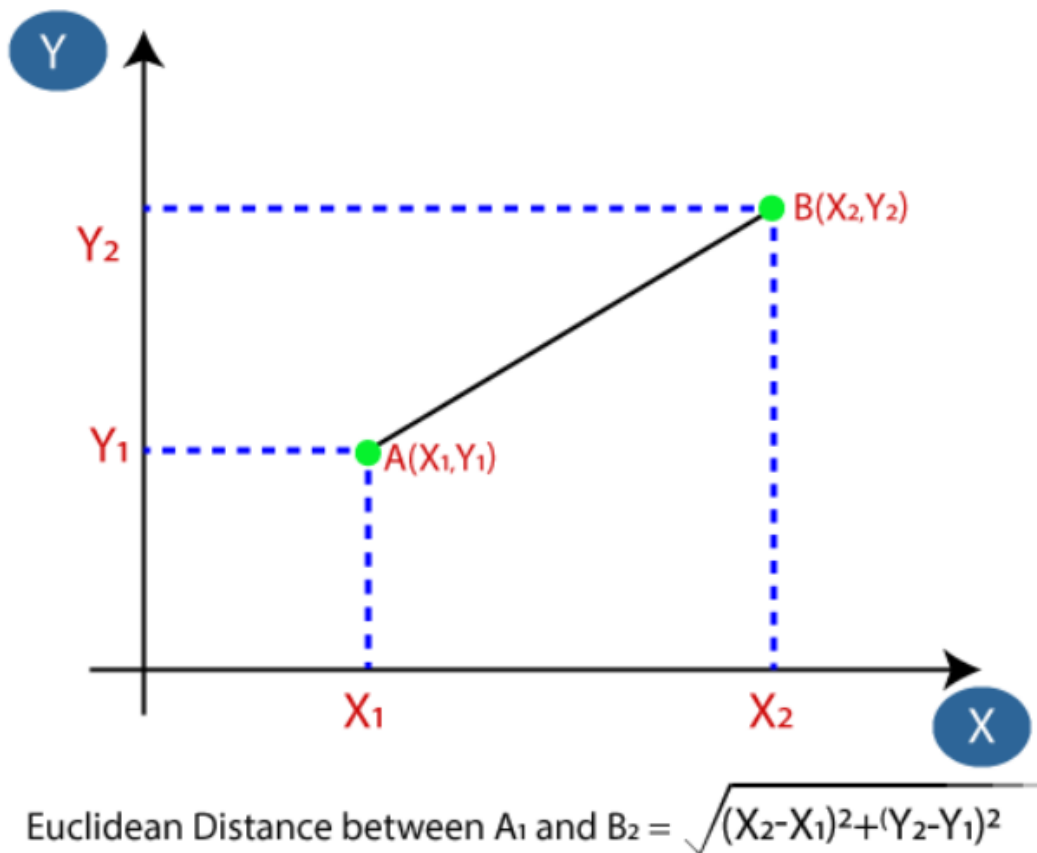
Theory :-

- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- The 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.
- The 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.
- The KNN 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.

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

- **Step-1:** Select the number K of the neighbours

- **Step-2:** Calculate the Euclidean distance of **K number of neighbours**
- **Step-3:** Take the K nearest neighbours as per the calculated Euclidean distance.
- **Step-4:** Among these k neighbours, 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 neighbour is maximum.
- **Step-6:** Our model is ready.



The value of K is taken as the square root of the number of observations in the training dataset. If the square root comes out to be even we add 1 to make it odd. It

is done so that there is no possibility of having an equal number of neighbours belonging to different classes.

PROGRAM AND OUTPUT :

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

IMPORTING DATASET

```
data = pd.read_csv("knndata.csv")
data = data.drop(['Unnamed: 0'], axis=1)
data
```

	WTT	PTI	EQW	SBI	LQE	QWG	FDJ	PJF	HQE	NXJ	TARGET CLASS
0	0.913917	1.162073	0.567946	0.755464	0.780862	0.352608	0.759697	0.643798	0.879422	1.231409	1
1	0.635632	1.003722	0.535342	0.825645	0.924109	0.648450	0.675334	1.013546	0.621552	1.492702	0
2	0.721360	1.201493	0.921990	0.855595	1.526629	0.720781	1.626351	1.154483	0.957877	1.285597	0
3	1.234204	1.386726	0.653046	0.825624	1.142504	0.875128	1.409708	1.380003	1.522692	1.153093	1
4	1.279491	0.949750	0.627280	0.668976	1.232537	0.703727	1.115596	0.646691	1.463812	1.419167	1
...
995	1.010953	1.034006	0.853116	0.622460	1.036610	0.586240	0.746811	0.319752	1.117340	1.348517	1
996	0.575529	0.955786	0.941835	0.792882	1.414277	1.269540	1.055928	0.713193	0.958684	1.663489	0
997	1.135470	0.982462	0.781905	0.916738	0.901031	0.884738	0.386802	0.389584	0.919191	1.385504	1
998	1.084894	0.861769	0.407158	0.665696	1.608612	0.943859	0.855806	1.061338	1.277456	1.188063	1
999	0.837460	0.961184	0.417006	0.799784	0.934399	0.424762	0.778234	0.907962	1.257190	1.364837	1

1000 rows × 11 columns

SPLITTING DATASET INTO TRAIN AND TEST DATA

```
x = data.iloc[:, :-1]
y = data.iloc[:, -1]
xtrain, xtest, ytrain, ytest = train_test_split(x, y, random_state=0, test_size=0.2)
```

STANDARDIZING DATA

```
scaler = StandardScaler()
xtrain = scaler.fit_transform(xtrain)
xtest = scaler.transform(xtest)
ytrain = np.array(ytrain)
ytest = np.array(ytest)
```

CALCULATING K

```
K = round(np.sqrt(len(xtrain)))+1
K
```

GETTING DISTANCE OF A POINT FROM NEIGHBOURS

```
def get_distance_array(xxtrain, test_point):  
    ans_array = np.array([])  
    for n in xxtrain:  
        ans_array = np.append(ans_array, np.sum((n-test_point)**2))  
    return ans_array
```

✓ 0.3s

CLASSIFYING THE POINT BASED ON K-NEAREST NEIGHBOURS

```
def classify(xxtrain, test_point, yytrain):  
    ans_array = pd.DataFrame(get_distance_array(xxtrain, test_point))  
    yytrain = pd.DataFrame(yytrain)  
    smallest_dist = ans_array.nsmallest(K, [0])  
    ones = 0  
    for i in smallest_dist.index:  
        if yytrain.iloc[i,0] == 1:  
            ones+=1  
    if ones > int(K/2):  
        return 1  
    return 0
```

✓ 0.3s

```
print("Predicted : ",classify(xtrain,xtest[1],ytrain),"\nActual : " ,ytest[1])
```

✓ 0.5s

Predicted : 0

Actual : 0

CALCULATING ACCURACY

```
def accuracy_score(xxtrain, xxtest, yytrain, yytest):  
    accurate_ans = 0  
    for i in range (len(xxtest)):  
        if yytest[i] == classify(xxtrain, xxtest[i], yytrain):  
            accurate_ans += 1  
    return (accurate_ans/len(xxtest))*100
```

✓ 0.4s

```
accuracy_score(xtrain, xtest, ytrain, ytest)
```

✓ 2.3s

93.5

Lab 6

Implementation of Naive Bayes for classification

THEORY :

- The Naïve Bayes algorithm is a supervised learning algorithm, which is based on **Bayes theorem** and used for solving classification problems.
- It is mainly used in *text classification* that includes a high-dimensional training dataset.
- The Naïve Bayes Classifier is one of the simple 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.

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 bases of colour, 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.

The formula for bayes' theorem is given by :

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

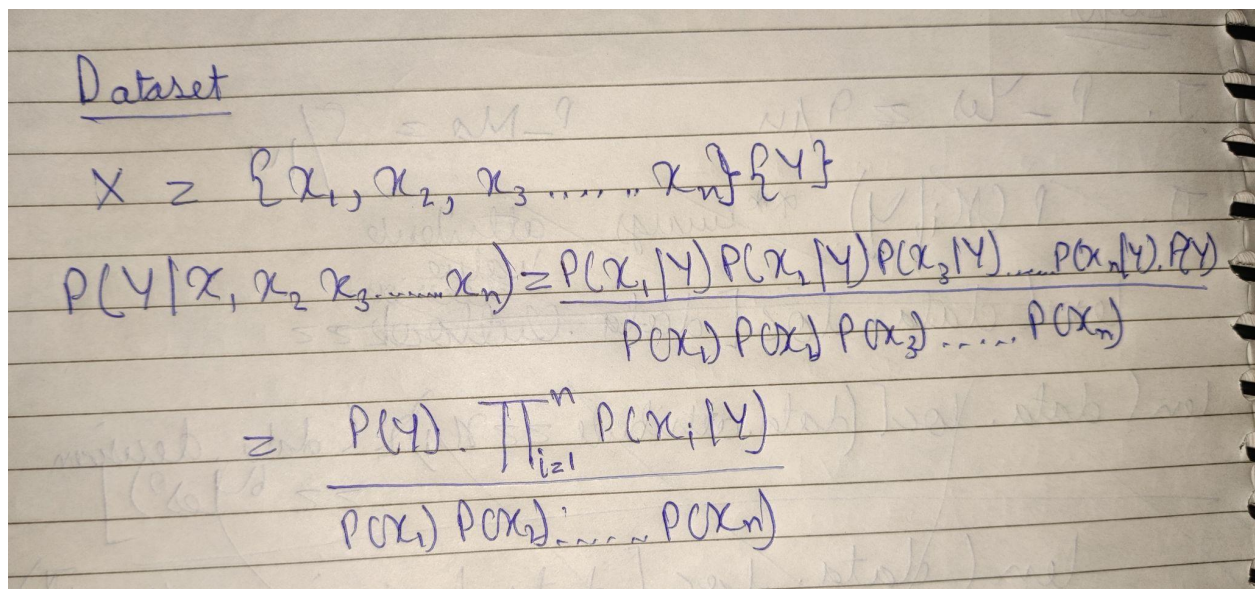
Where,

P(A|B) is Posterior probability: Probability of hypothesis A on the observed event B.

P(B|A) is Likelihood probability: Probability of the evidence given that the probability of a hypothesis is true.

P(A) is Prior Probability: Probability of hypothesis before observing the evidence.

P(B) is Marginal Probability: Probability of Evidence.



The image shows a handwritten derivation of Bayes' theorem on lined paper. It starts with the label 'Dataset' underlined. Below it, the dataset is defined as $X = \{x_1, x_2, x_3, \dots, x_n\} \in \{Y\}$. The main equation is $P(Y|x_1, x_2, x_3, \dots, x_n) = \frac{P(x_1|Y)P(x_2|Y)P(x_3|Y) \dots P(x_n|Y)P(Y)}{P(x_1)P(x_2)P(x_3) \dots P(x_n)}$. This is then simplified to $= \frac{P(Y) \prod_{i=1}^n P(x_i|Y)}{P(x_1)P(x_2) \dots P(x_n)}$.

PROGRAM AND OUTPUT :

```
import pandas as pd
import numpy as np
```

IMPORTING DATASET

```
data = pd.read_csv("ML Lab 4 Data - Sheet1.csv")
data
```

	Day	Outlook	Temperature	Humidity	Wind	Decision
0	1	Sunny	Hot	High	Weak	No
1	2	Sunny	Hot	High	Strong	No
2	3	Overcast	Hot	High	Weak	Yes
3	4	Rain	Mild	High	Weak	Yes
4	5	Rain	Cool	Normal	Weak	Yes
5	6	Rain	Cool	Normal	Strong	No
6	7	Overcast	Cool	Normal	Strong	Yes
7	8	Sunny	Mild	High	Weak	No
8	9	Sunny	Cool	Normal	Weak	Yes
9	10	Rain	Mild	Normal	Weak	Yes
10	11	Sunny	Mild	Normal	Strong	Yes
11	12	Overcast	Mild	High	Strong	Yes
12	13	Overcast	Hot	Normal	Weak	Yes
13	14	Rain	Mild	High	Strong	No

CALCULATING PROBABILITY OF DECISION AND THAT OF A PARAMETER GIVEN THE DECISION

```
def p_decision(decision):
    return len(data.loc[data.Decision == decision])/len(data)

def P_XiY(**kwargs):
    return len(data.loc[(data[kwargs["attribute"]] == kwargs["attribute_value"]) & (data.Decision == kwargs["decision"])]/len(data.loc[data.Decision == kwargs["decision"]]))
```

MAKING DECISION BASED ON GIVEN INFORMATION

```
def making_decision(**kwargs):
    con_probs_yes = []
    if kwargs["Outlook"] != "":
        con_probs_yes.append(P_XiY(attribute = "Outlook", attribute_value = kwargs["Outlook"], decision = "Yes"))
    if kwargs["Temperature"] != "":
        con_probs_yes.append(P_XiY(attribute = "Temperature", attribute_value = kwargs["Temperature"], decision = "Yes"))
    if kwargs["Humidity"] != "":
        con_probs_yes.append(P_XiY(attribute = "Humidity", attribute_value = kwargs["Humidity"], decision = "Yes"))
    if kwargs["Wind"] != "":
        con_probs_yes.append(P_XiY(attribute = "Wind", attribute_value = kwargs["Wind"], decision = "Yes"))

    p_yes = np.prod(con_probs_yes)*p_decision("Yes")

    con_probs_No = []
    if kwargs["Outlook"] != "":
        con_probs_No.append(P_XiY(attribute = "Outlook", attribute_value = kwargs["Outlook"], decision = "No"))
    if kwargs["Temperature"] != "":
        con_probs_No.append(P_XiY(attribute = "Temperature", attribute_value = kwargs["Temperature"], decision = "No"))
    if kwargs["Humidity"] != "":
        con_probs_No.append(P_XiY(attribute = "Humidity", attribute_value = kwargs["Humidity"], decision = "No"))
    if kwargs["Wind"] != "":
        con_probs_No.append(P_XiY(attribute = "Wind", attribute_value = kwargs["Wind"], decision = "No"))

    p_no = np.prod(con_probs_No)*p_decision("No")

    ans_prob = p_yes/(p_yes+p_no)
    if ans_prob > 0.5:
        return "Yes"
    return "No"
```

```
print(making_decision(Outlook = "Sunny", Temperature = "Cool", Humidity = "", Wind = ""))
print(making_decision(Outlook = "Rain", Temperature = "Mild", Humidity = "High", Wind = "Strong"))
```

Yes

No

CACULATING ACCURACY

```
correct = 0
for i in range(len(data)):
    if data.iloc[i,-1] == making_decision(Outlook = data.iloc[i,1], Temperature = data.iloc[i,2], Humidity = data.iloc[i,3], Wind = data.iloc[i,4]):
        correct += 1
print("Accuracy : ",(correct/len(data))*100)
```

Accuracy : 92.85714285714286

Lab 7

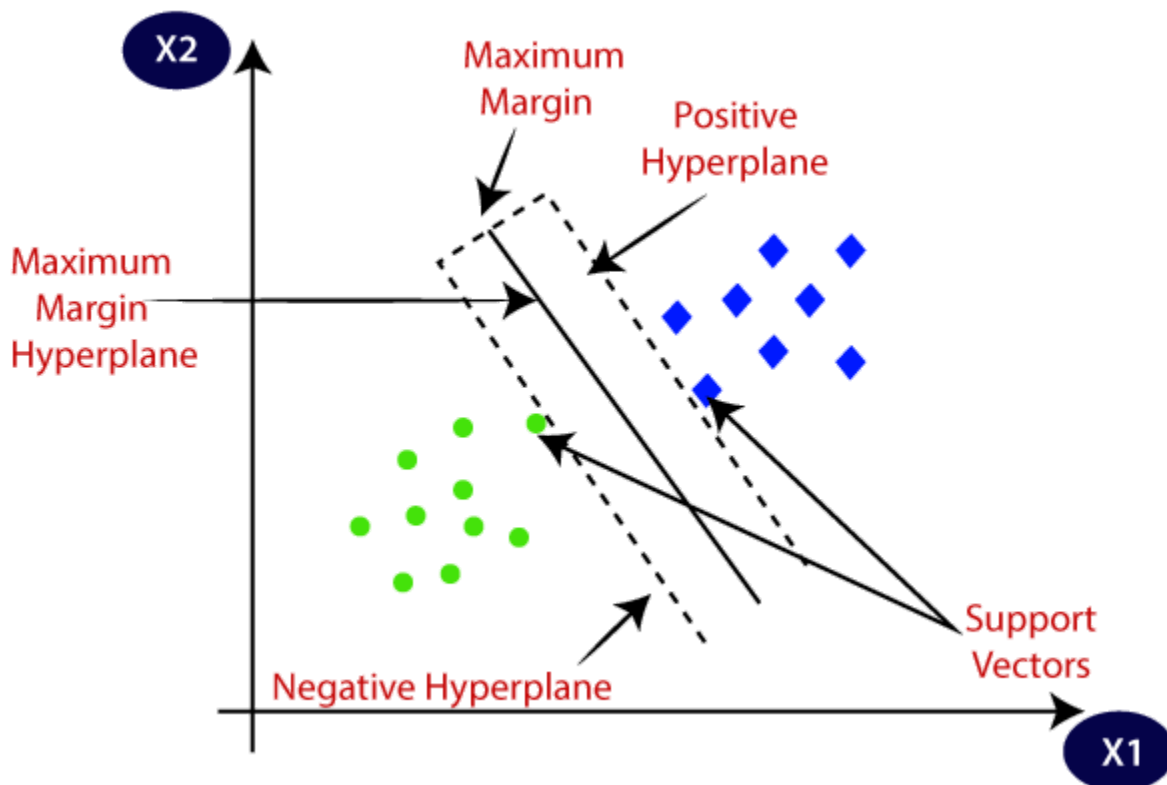
Implementation of Support Vector Machine Algorithm

THEORY :

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

SVM can be of two types:

- **Linear SVM:** Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- **Non-linear SVM:** Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

Hyperplane and Support Vectors in the SVM algorithm:

Hyperplane:

There can be multiple lines/decision boundaries to segregate the classes in n-dimensional space, but we need to find out the best decision boundary that helps to classify the data points. This best boundary is known as the hyperplane of SVM.

The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features (as shown in image), then the hyperplane will be a straight line. And if there are 3 features, then the hyperplane will be a 2-dimension plane.

We always create a hyperplane that has a maximum margin, which means the maximum distance between the data points.

Support Vectors:

The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vectors. These vectors support the hyperplane, hence called a Support vector.

PROGRAM AND OUTPUT :

Importing Libraries

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

✓ 0.7s

Support Vector Machine Classifier

```
class SVM_classifier():

    def __init__(self, learning_rate, no_of_iterations, lambda_parameter):
        self.learning_rate = learning_rate
        self.no_of_iterations = no_of_iterations
        self.lambda_parameter = lambda_parameter

    def fit(self, X, Y):
        self.m, self.n = X.shape
        self.w = np.zeros(self.n)
        self.b = 0
        self.X = X
        self.Y = Y

        for i in range(self.no_of_iterations):
            self.update_weights()
```

```

def update_weights(self):
    y_label = np.where(self.Y <= 0, -1, 1)

    for index, x_i in enumerate(self.X):
        condition = y_label[index] * (np.dot(x_i, self.w) - self.b) >= 1
        if (condition == True):
            dw = 2 * self.lambda_parameter * self.w
            db = 0
        else:
            dw = 2 * self.lambda_parameter * self.w - np.dot(x_i, y_label[index])
            db = y_label[index]
        self.w = self.w - self.learning_rate * dw
        self.b = self.b - self.learning_rate * db

def predict(self, X):
    output = np.dot(X, self.w) - self.b
    predicted_labels = np.sign(output)
    y_hat = np.where(predicted_labels <= -1, 0, 1)
    return y_hat

```

Data Collection & Processing

```
diabetes_data = pd.read_csv("diabetes.csv")
diabetes_data.head()
```

✓ 0.6s

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

File Editor | File Explorer

```
diabetes_data['Outcome'].value_counts()
```

✓ 0.7s

0 500

1 268

Name: Outcome, dtype: int64

0 --> Non-diabetic

1 --> Diabetic

```
features = diabetes_data.drop(columns='Outcome', axis=1)
target = diabetes_data['Outcome']
```

✓ 0.4s

```
print(features)
```

✓ 0.5s

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	\
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	
..	
763	10	101	76	48	180	32.9	
764	2	122	70	27	0	36.8	
765	5	121	72	23	112	26.2	
766	1	126	60	0	0	30.1	
767	1	93	70	31	0	30.4	

	DiabetesPedigreeFunction	Age
0	0.627	50
1	0.351	31
2	0.672	32
3	0.167	21
4	2.288	33
..
763	0.171	63
764	0.340	27
765	0.245	30
766	0.349	47
767	0.315	23

[768 rows x 8 columns]

Data Standardization

```
scaler = StandardScaler()  
scaler.fit(features)  
standardized_data = scaler.transform(features)  
print(standardized_data)
```

✓ 0.8s

```
[[ 0.63994726  0.84832379  0.14964075 ...  0.20401277  0.46849198  
   1.4259954 ]  
 [-0.84488505 -1.12339636 -0.16054575 ... -0.68442195 -0.36506078  
  -0.19067191]  
 [ 1.23388019  1.94372388 -0.26394125 ... -1.10325546  0.60439732  
  -0.10558415]  
 ...  
 [ 0.3429808   0.00330087  0.14964075 ... -0.73518964 -0.68519336  
  -0.27575966]  
 [-0.84488505  0.1597866  -0.47073225 ... -0.24020459 -0.37110101  
   1.17073215]  
 [-0.84488505 -0.8730192   0.04624525 ... -0.20212881 -0.47378505  
  -0.87137393]]
```

```
features = standardized_data  
target = diabetes_data['Outcome']
```

✓ 0.4s

Train Test Split

```
X_train, X_test, Y_train, Y_test = train_test_split(features, target, test_size=0.2, random_state = 2)
print(features.shape, X_train.shape, X_test.shape)
```

✓ 0.4s

(768, 8) (614, 8) (154, 8)

Training the Model

Support Vector Machine Classifier

```
classifier = SVM_classifier(learning_rate=0.001, no_of_iterations=1000, lambda_parameter=0.01)
classifier.fit(X_train, Y_train)
```

✓ 7.2s

Accuracy Score

```
# accuracy on training data
X_train_prediction = classifier.predict(X_train)
training_data_accuracy = accuracy_score(Y_train, X_train_prediction)
print('Accuracy score on training data = ', training_data_accuracy*100)
# accuracy on training data
X_test_prediction = classifier.predict(X_test)
test_data_accuracy = accuracy_score(Y_test, X_test_prediction)
print('Accuracy score on test data = ', test_data_accuracy*100)
```

✓ 0.4s

Accuracy score on training data = 77.68729641693811

Accuracy score on test data = 75.32467532467533

Building a Predictive System

```
input_data = (5,166,72,19,175,25.8,0.587,51)
input_data_as_numpy_array = np.asarray(input_data)
input_data_reshaped = input_data_as_numpy_array.reshape(1,-1)

std_data = scaler.transform(input_data_reshaped)
print(std_data)

prediction = classifier.predict(std_data)
print(prediction)

if (prediction[0] == 0):
    print('The person is not diabetic')
else:
    print('The Person is diabetic')
```

✓ 0.6s

```
[[ 0.3429808  1.41167241  0.14964075 -0.09637905  0.82661621 -0.78595734
  0.34768723  1.51108316]]
```

```
[1]
```

```
The Person is diabetic
```

Lab 8

Implementation of K means clustering Algorithm

THEORY :

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.

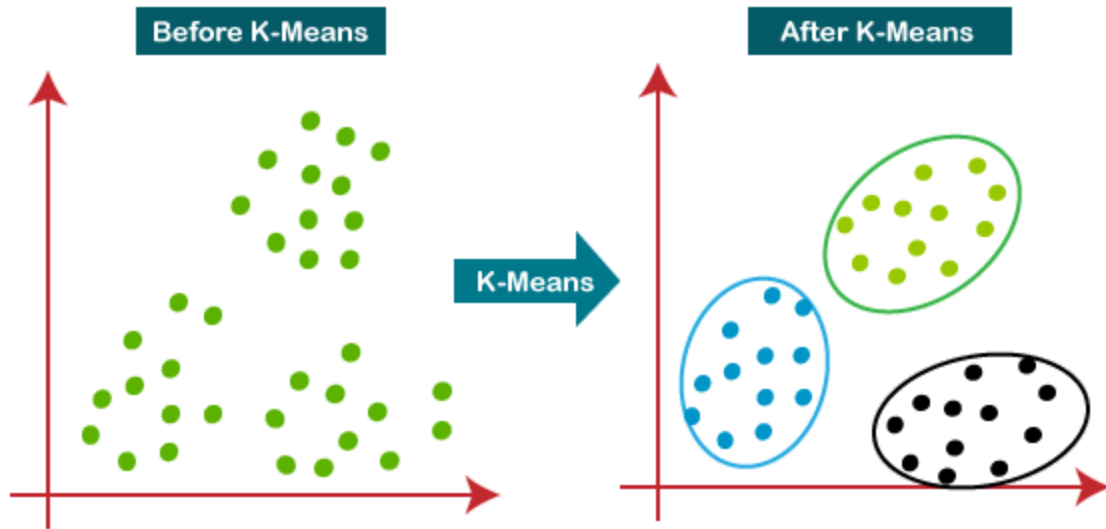
The algorithm takes the unlabeled dataset as input, divides the dataset into k-number of clusters, and repeats the process until it does not find the best clusters. The value of k should be predetermined in this algorithm.

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:



How does the K-Means Algorithm Work?

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 assign 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.

PROGRAM AND OUTPUT :

IMPORTING LIBRARIES

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

IMPORTING DATASET

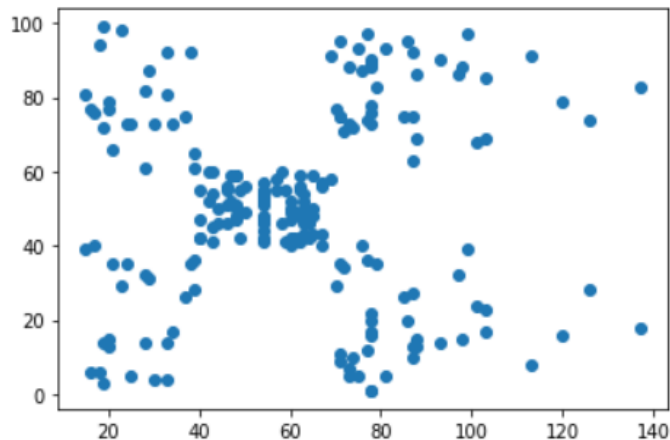
```
data = pd.read_csv("segmented_customers.csv")
data
```

	CustomerID	Gender	Age	Annual Income (k\$)	Spending Score (1-100)	cluster
0	1	Male	19	15	39	4
1	2	Male	21	15	81	3
2	3	Female	20	16	6	4
3	4	Female	23	16	77	3
4	5	Female	31	17	40	4
...
195	196	Female	35	120	79	1
196	197	Female	45	126	28	2
197	198	Male	32	126	74	1
198	199	Male	32	137	18	2
199	200	Male	30	137	83	1

200 rows × 6 columns

```
plt.scatter(data["Annual Income (k$)", data["Spending Score (1-100)"]])
```

<matplotlib.collections.PathCollection at 0x1f0ca990e80>



```
df = data[["Annual Income (k$)", "Spending Score (1-100)"]]
df_use = df.sample(n=5, replace=False)
df.drop(df_use.index, axis=0, inplace=True)
best_var = df.var() * 10
df_use.reset_index(drop=True, inplace=True)
df.reset_index(drop=True, inplace=True)
BFIRST_CENTROID = pd.DataFrame(columns=df.columns)
BSECOND_CENTROID = pd.DataFrame(columns=df.columns)
BTHIRD_CENTROID = pd.DataFrame(columns=df.columns)
BFOURTH_CENTROID = pd.DataFrame(columns=df.columns)
BFIFTH_CENTROID = pd.DataFrame(columns=df.columns)
for i in range(30):
    FIRST_CENTROID = pd.DataFrame(columns=df.columns)
    SECOND_CENTROID = pd.DataFrame(columns=df.columns)
    THIRD_CENTROID = pd.DataFrame(columns=df.columns)
    FOURTH_CENTROID = pd.DataFrame(columns=df.columns)
    FIFTH_CENTROID = pd.DataFrame(columns=df.columns)
    FIRST_CENTROID = pd.concat([FIRST_CENTROID, df_use.iloc[0].to_frame().T], ignore_index=True)
    SECOND_CENTROID = pd.concat([SECOND_CENTROID, df_use.iloc[1].to_frame().T], ignore_index=True)
    THIRD_CENTROID = pd.concat([THIRD_CENTROID, df_use.iloc[2].to_frame().T], ignore_index=True)
    FOURTH_CENTROID = pd.concat([FOURTH_CENTROID, df_use.iloc[3].to_frame().T], ignore_index=True)
    FIFTH_CENTROID = pd.concat([FIFTH_CENTROID, df_use.iloc[4].to_frame().T], ignore_index=True)
    for i in range(len(df)):
        d1 = (df.iloc[i, 0] - FIRST_CENTROID.iloc[0,0])**2 + (df.iloc[i,1] - FIRST_CENTROID.iloc[0,1])**2
        d2 = (df.iloc[i, 0] - SECOND_CENTROID.iloc[0,0])**2 + (df.iloc[i,1] - SECOND_CENTROID.iloc[0,1])**2
        d3 = (df.iloc[i, 0] - THIRD_CENTROID.iloc[0,0])**2 + (df.iloc[i,1] - THIRD_CENTROID.iloc[0,1])**2
        d4 = (df.iloc[i, 0] - FOURTH_CENTROID.iloc[0,0])**2 + (df.iloc[i,1] - FOURTH_CENTROID.iloc[0,1])**2
        d5 = (df.iloc[i, 0] - FIFTH_CENTROID.iloc[0,0])**2 + (df.iloc[i,1] - FIFTH_CENTROID.iloc[0,1])**2
        if d1 == min(d1,d2,d3,d4,d5):
            FIRST_CENTROID = pd.concat([FIRST_CENTROID, df.iloc[i].to_frame().T], ignore_index=True)
            FIRST_CENTROID.iloc[0] = (FIRST_CENTROID.iloc[0] + FIRST_CENTROID.iloc[-1])/2
        elif d2 == min(d1,d2,d3,d4,d5):
            SECOND_CENTROID = pd.concat([SECOND_CENTROID, df.iloc[i].to_frame().T], ignore_index=True)
            SECOND_CENTROID.iloc[0] = (SECOND_CENTROID.iloc[0] + SECOND_CENTROID.iloc[-1])/2
```

```

elif d3 == min(d1,d2,d3,d4,d5):
    THIRD_CENTROID = pd.concat([THIRD_CENTROID, df.iloc[i].to_frame().T], ignore_index=True)
    THIRD_CENTROID.iloc[0] = (THIRD_CENTROID.iloc[0] + THIRD_CENTROID.iloc[-1])/2
elif d4 == min(d1,d2,d3,d4,d5):
    FOURTH_CENTROID = pd.concat([FOURTH_CENTROID, df.iloc[i].to_frame().T], ignore_index=True)
    FOURTH_CENTROID.iloc[0] = (FOURTH_CENTROID.iloc[0] + FOURTH_CENTROID.iloc[-1])/2
elif d5 == min(d1,d2,d3,d4,d5):
    FIFTH_CENTROID = pd.concat([FIFTH_CENTROID, df.iloc[i].to_frame().T], ignore_index=True)
    FIFTH_CENTROID.iloc[0] = (FIFTH_CENTROID.iloc[0] + FIFTH_CENTROID.iloc[-1])/2
FIRST_CENTROID.iloc[0] = FIRST_CENTROID.mean()
SECOND_CENTROID.iloc[0] = SECOND_CENTROID.mean()
THIRD_CENTROID.iloc[0] = THIRD_CENTROID.mean()
FOURTH_CENTROID.iloc[0] = FOURTH_CENTROID.mean()
FIFTH_CENTROID.iloc[0] = FIFTH_CENTROID.mean()
curr_var = FIRST_CENTROID.var()+SECOND_CENTROID.var()+THIRD_CENTROID.var()+FOURTH_CENTROID.var()+FIFTH_CENTROID.var()
if best_var.iloc[0] > curr_var.iloc[0]:
    BFIRST_CENTROID = FIRST_CENTROID
    BSECOND_CENTROID = SECOND_CENTROID
    BTHIRD_CENTROID = THIRD_CENTROID
    BFOURTH_CENTROID = FOURTH_CENTROID
    BFIFTH_CENTROID = FIFTH_CENTROID
    best_var = curr_var
df_use = df.sample(n=5, replace=False)
df = data[["Annual Income (k$)", "Spending Score (1-100)"]]
df.drop(df_use.index, axis=0, inplace=True)
df_use.reset_index(drop=True, inplace=True)
df.reset_index(drop=True, inplace=True)

```

```

plt.scatter(BFIRST_CENTROID["Annual Income (k$)"], BFIRST_CENTROID["Spending Score (1-100)"])
plt.scatter(BSECOND_CENTROID["Annual Income (k$)"], BSECOND_CENTROID["Spending Score (1-100)"])
plt.scatter(BTHIRD_CENTROID["Annual Income (k$)"], BTHIRD_CENTROID["Spending Score (1-100)"])
plt.scatter(BFOURTH_CENTROID["Annual Income (k$)"], BFOURTH_CENTROID["Spending Score (1-100)"])
plt.scatter(BFIFTH_CENTROID["Annual Income (k$)"], BFIFTH_CENTROID["Spending Score (1-100)"])
plt.show()

```

