

**PRACTICAL NO 1**

**Aim: Implementation of Logic programming using PROLOG DFS for water jug problem**

**Code**:

start(2,0):-write(' 4lit Jug: 2 | 3lit Jug: 0|\n'),

write('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n'),

write('Goal Reached! Congrats!!\n'),

write('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n').

start(X,Y):-write(' Water Jug Game \n'),

write('Intial State: 4lit Jug- 0lit\n'),

write(' 3lit Jug- 0lit\n'),

write('Final State: 4lit Jug- 2lit\n'),

write(' 3lit Jug- 0lit\n'),

write('Follow the Rules: \n'),

write('Rule 1: Fill 4lit Jug\n'),

write('Rule 2: Fill 3lit Jug\n'),

write('Rule 3: Empty 4lit Jug\n'),

write('Rule 4: Empty 3lit Jug\n'),

write('Rule 5: Pour water from 3lit Jug to fill 4lit Jug\n'),

write('Rule 6: Pour water from 4lit Jug to fill 3lit Jug\n'),

write('Rule 7: Pour all of water from 3lit Jug to 4lit Jug\n'),

write('Rule 8: Pour all of water from 4lit Jug to 3lit Jug\n'),

write(' 4lit Jug: 0 | 3lit Jug: 0'),nl,

write(' Current Quantity :'),

write(' 4lit Jug: '),write(X),write('| 3lit Jug: '),

write(Y),write('|\n'),

write(' Enter the move::'),

read(N),

contains(X,Y,N).

contains(\_,Y,1):-start(4,Y).

contains(X,\_,2):-start(X,3).

contains(\_,Y,3):-start(0,Y).

contains(X,\_,4):-start(X,0).

contains(X,Y,5):-N is Y-4+X, start(4,N).

contains(X,Y,6):-N is X-3+Y, start(N,3).

contains(X,Y,7):-N is X+Y, start(N,0).

contains(X,Y,8):-N is X+Y, start(0,N).

**Output Window Commands:**

start(0,0).

Enter Move :: 1.

Enter Move :: 6.

Enter Move :: 4.

Enter Move :: 8.

Enter Move :: 1.

Enter Move :: 6.

Enter Move :: 4.

**PRACTICAL NO 2**

**Program 3- Tic-Tac-Toe using Prolog**

**Code:**

% To play a game with the computer, type

% play.

% Predicates that define the winning conditions:

win(Board, Player) :- rowwin(Board, Player).

win(Board, Player) :- colwin(Board, Player).

win(Board, Player) :- diagwin(Board, Player).

rowwin(Board, Player) :- Board = [Player,Player,Player,\_,\_,\_,\_,\_,\_].

rowwin(Board, Player) :- Board = [\_,\_,\_,Player,Player,Player,\_,\_,\_].

rowwin(Board, Player) :- Board = [\_,\_,\_,\_,\_,\_,Player,Player,Player].

colwin(Board, Player) :- Board = [Player,\_,\_,Player,\_,\_,Player,\_,\_].

colwin(Board, Player) :- Board = [\_,Player,\_,\_,Player,\_,\_,Player,\_].

colwin(Board, Player) :- Board = [\_,\_,Player,\_,\_,Player,\_,\_,Player].

diagwin(Board, Player) :- Board = [Player,\_,\_,\_,Player,\_,\_,\_,Player].

diagwin(Board, Player) :- Board = [\_,\_,Player,\_,Player,\_,Player,\_,\_].

move([b,B,C,D,E,F,G,H,I], Player, [Player,B,C,D,E,F,G,H,I]).

move([A,b,C,D,E,F,G,H,I], Player, [A,Player,C,D,E,F,G,H,I]).

move([A,B,b,D,E,F,G,H,I], Player, [A,B,Player,D,E,F,G,H,I]).

move([A,B,C,b,E,F,G,H,I], Player, [A,B,C,Player,E,F,G,H,I]).

move([A,B,C,D,b,F,G,H,I], Player, [A,B,C,D,Player,F,G,H,I]).

move([A,B,C,D,E,b,G,H,I], Player, [A,B,C,D,E,Player,G,H,I]).

move([A,B,C,D,E,F,b,H,I], Player, [A,B,C,D,E,F,Player,H,I]).

move([A,B,C,D,E,F,G,b,I], Player, [A,B,C,D,E,F,G,Player,I]).

move([A,B,C,D,E,F,G,H,b], Player, [A,B,C,D,E,F,G,H,Player]).

display([A,B,C,D,E,F,G,H,I]) :- write([A,B,C]),nl,write([D,E,F]),nl,

write([G,H,I]),nl,nl.

% Predicates to support playing a game with the user:

x\_can\_win\_in\_one(Board) :- move(Board, x, Newboard), win(Newboard, x).

% The predicate validate generates the computer's (playing o) reponse

% from the current Board.

validate(Board,Newboard) :-

move(Board, o, Newboard),

win(Newboard, o),

!.

validate(Board,Newboard) :-

move(Board, o, Newboard),

not(x\_can\_win\_in\_one(Newboard)).

validate(Board,Newboard) :-

move(Board, o, Newboard).

% The following translates from an integer description

% of x's move to a board transformation.

xmove([b,B,C,D,E,F,G,H,I], 1, [x,B,C,D,E,F,G,H,I]).

xmove([A,b,C,D,E,F,G,H,I], 2, [A,x,C,D,E,F,G,H,I]).

xmove([A,B,b,D,E,F,G,H,I], 3, [A,B,x,D,E,F,G,H,I]).

xmove([A,B,C,b,E,F,G,H,I], 4, [A,B,C,x,E,F,G,H,I]).

xmove([A,B,C,D,b,F,G,H,I], 5, [A,B,C,D,x,F,G,H,I]).

xmove([A,B,C,D,E,b,G,H,I], 6, [A,B,C,D,E,x,G,H,I]).

xmove([A,B,C,D,E,F,b,H,I], 7, [A,B,C,D,E,F,x,H,I]).

xmove([A,B,C,D,E,F,G,b,I], 8, [A,B,C,D,E,F,G,x,I]).

xmove([A,B,C,D,E,F,G,H,b], 9, [A,B,C,D,E,F,G,H,x]).

xmove(Board, \_, Board) :- write('Illegal move.'), nl.

% The 0-place predicate playo starts a game with the user.

play :- explain, playfrom([b,b,b,b,b,b,b,b,b]).

explain :-

write('You play X by entering integer positions followed by a period.'),

nl,

display([1,2,3,4,5,6,7,8,9]).

playfrom(Board) :- win(Board, x), write('You win!').

playfrom(Board) :- win(Board, o), write('I win!').

playfrom(Board) :- read(N),

xmove(Board, N, Newboard),

display(Newboard),

validate(Newboard, Newnewboard),

display(Newnewboard),

playfrom(Newnewboard).

**Output :**



**PRACTICAL NO 3**

**Aim: Implementation of Logic programming using PROLOG Hill-climbing to solve 8- Puzzle Problem.**

**Code:**

ids :-

start(State),

length(Moves, N),

hill([State], Moves, Path), !,

show([start|Moves], Path),

format('~nmoves = ~w~n', [N]).

hill([State|States], [], Path) :-

goal(State), !,

reverse([State|States], Path).

hill([State|States], [Move|Moves], Path) :-

move(State, Next, Move),

not(memberchk(Next, [State|States])),

hill([Next,State|States], Moves, Path).

show([], \_).

show([Move|Moves], [State|States]) :-

State = state(A,B,C,D,E,F,G,H,J),

format('~n~w~n~n', [Move]),

format('~w ~w ~w~n',[A,B,C]),

format('~w ~w ~w~n',[D,E,F]),

format('~w ~w ~w~n',[G,H,J]),

show(Moves, States).

% Empty position is marked with '\*'

start( state(0,1,\*,2,3,4,5,6,7) ).

goal( state(\*,0,1,2,3,4,5,6,7) ).

move( state(A,\*,C,D,E,F,G,H,J), state(\*,A,C,D,E,F,G,H,J), left ).

move( state(A,B,\*,D,E,F,G,H,J), state(A,\*,B,D,E,F,G,H,J), left ).

move( state(A,B,C,D,\*,F,G,H,J), state(A,B,C,\*,D,F,G,H,J), left ).

move( state(A,B,C,D,E,\*,G,H,J), state(A,B,C,D,\*,E,G,H,J), left ).

move( state(A,B,C,D,E,F,G,\*,J), state(A,B,C,D,E,F,\*,G,J), left ).

move( state(A,B,C,D,E,F,G,H,\*), state(A,B,C,D,E,F,G,\*,H), left ).

move( state(\*,B,C,D,E,F,G,H,J), state(B,\*,C,D,E,F,G,H,J), right).

move( state(A,\*,C,D,E,F,G,H,J), state(A,C,\*,D,E,F,G,H,J), right).

move( state(A,B,C,\*,E,F,G,H,J), state(A,B,C,E,\*,F,G,H,J), right).

move( state(A,B,C,D,\*,F,G,H,J), state(A,B,C,D,F,\*,G,H,J), right).

move( state(A,B,C,D,E,F,\*,H,J), state(A,B,C,D,E,F,H,\*,J), right).

move( state(A,B,C,D,E,F,G,\*,J), state(A,B,C,D,E,F,G,J,\*), right).

move( state(A,B,C,\*,E,F,G,H,J), state(\*,B,C,A,E,F,G,H,J), up).

move( state(A,B,C,D,\*,F,G,H,J), state(A,\*,C,D,B,F,G,H,J), up).

move( state(A,B,C,D,E,\*,G,H,J), state(A,B,\*,D,E,C,G,H,J), up).

move( state(A,B,C,D,E,F,\*,H,J), state(A,B,C,\*,E,F,D,H,J), up).

move( state(A,B,C,D,E,F,G,\*,J), state(A,B,C,D,\*,F,G,E,J), up).

move( state(A,B,C,D,E,F,G,H,\*), state(A,B,C,D,E,\*,G,H,F), up).

move( state(\*,B,C,D,E,F,G,H,J), state(D,B,C,\*,E,F,G,H,J), down ).

move( state(A,\*,C,D,E,F,G,H,J), state(A,E,C,D,\*,F,G,H,J), down ).

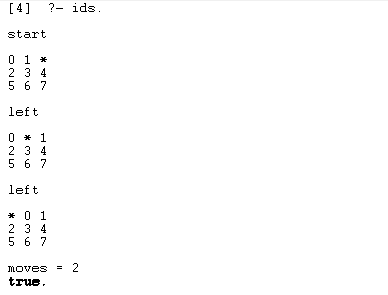
move( state(A,B,\*,D,E,F,G,H,J), state(A,B,F,D,E,\*,G,H,J), down ).

move( state(A,B,C,\*,E,F,G,H,J), state(A,B,C,G,E,F,\*,H,J), down ).

move( state(A,B,C,D,\*,F,G,H,J), state(A,B,C,D,H,F,G,\*,J), down ).

move( state(A,B,C,D,E,\*,G,H,J), state(A,B,C,D,E,J,G,H,\*), down )

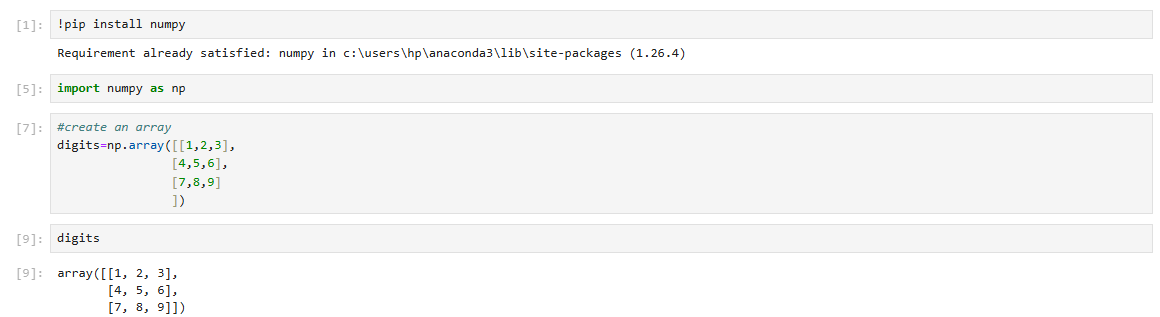
**Output :**

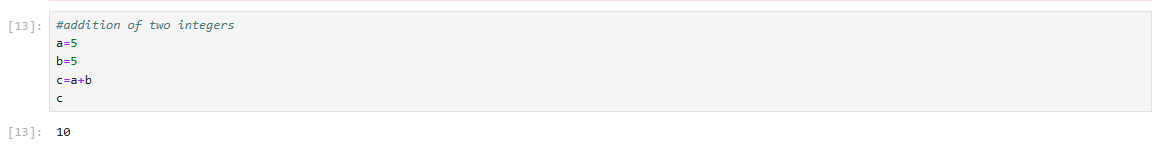
****

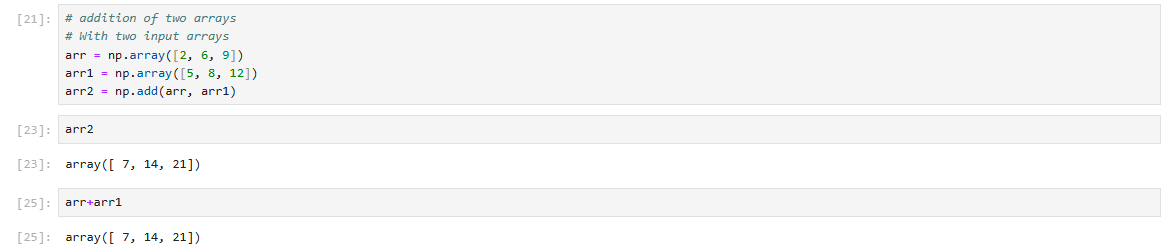
**PRACTICAL NO 4**

**Aim: Introduction to Python Programming: Learn the different libraries - NumPy, Pandas, SciPy, Matplotlib, Scikit Learn.**

**1. NUMPY**

****

****

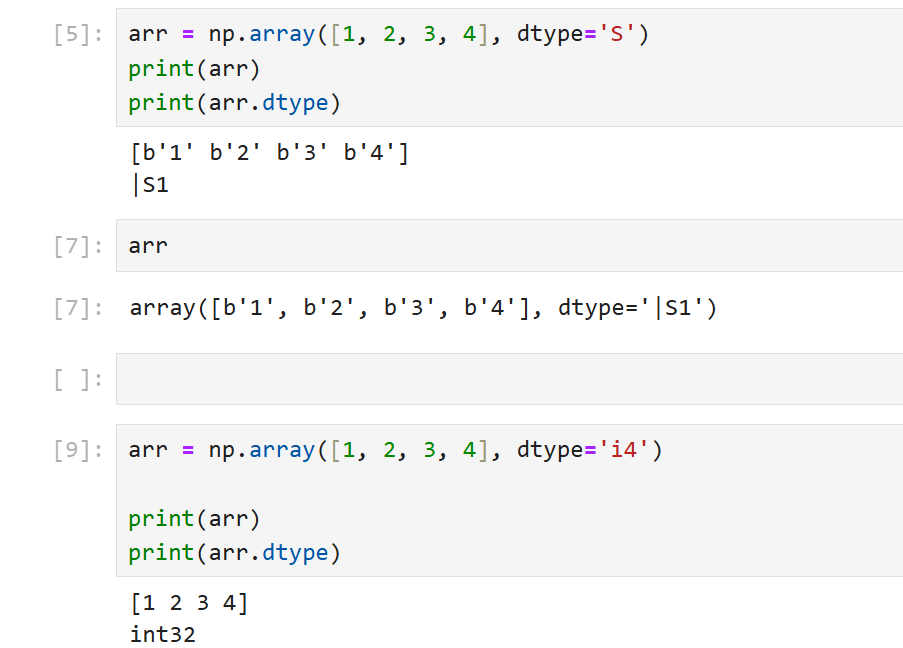
****

****

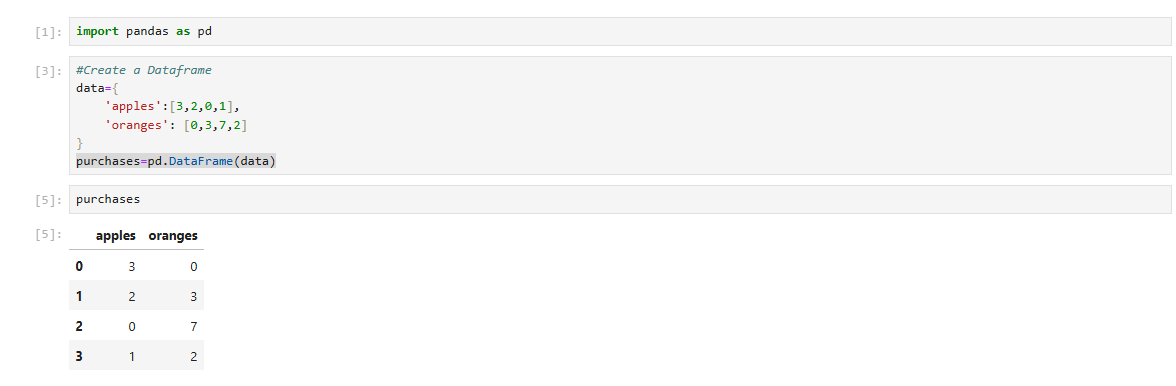
****

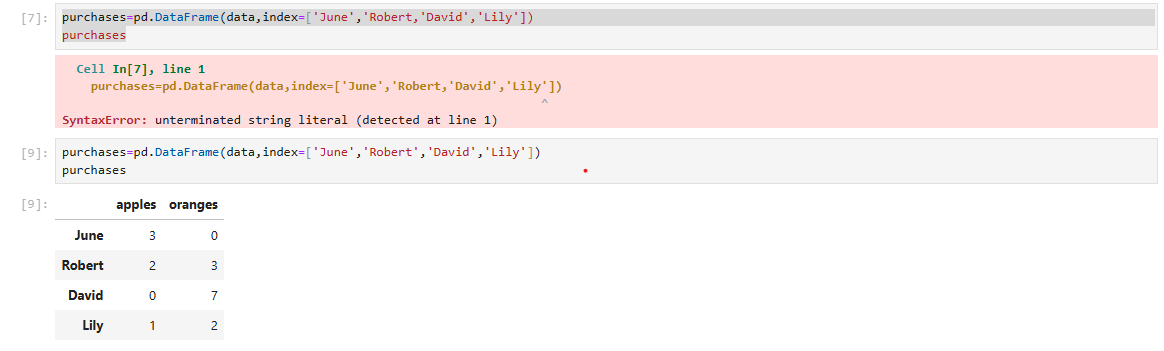
Below is a list of all data types in NumPy and the characters used to represent them.

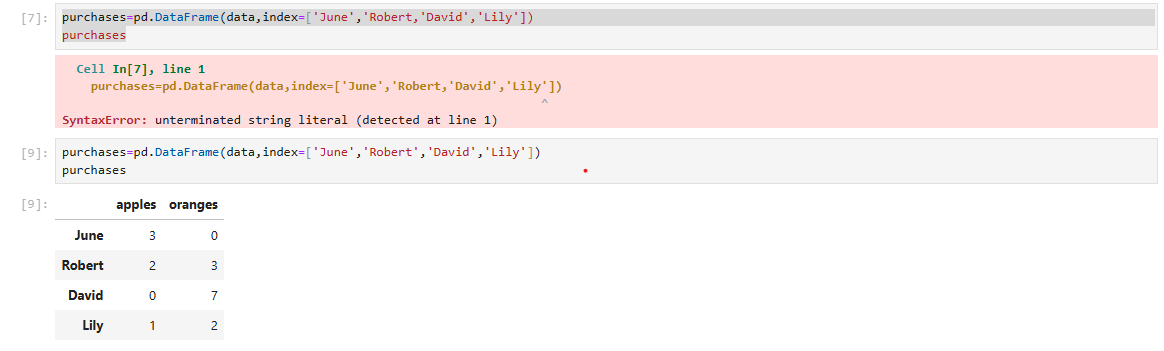
* i - integer
* b - boolean
* f - float
* M - datetime
* S - string



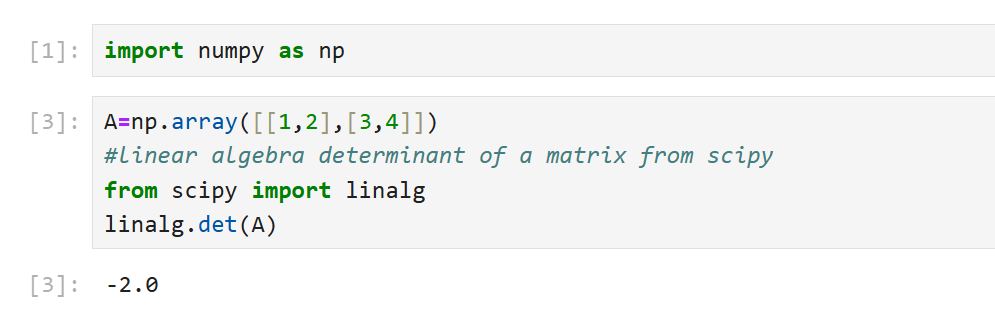
**2. Pandas**

****

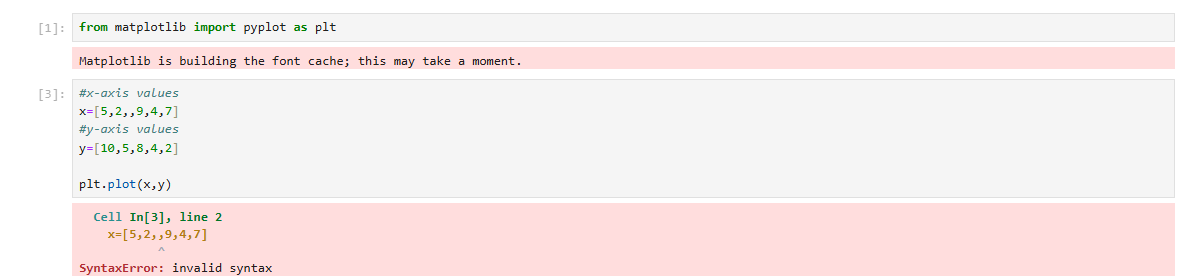
****

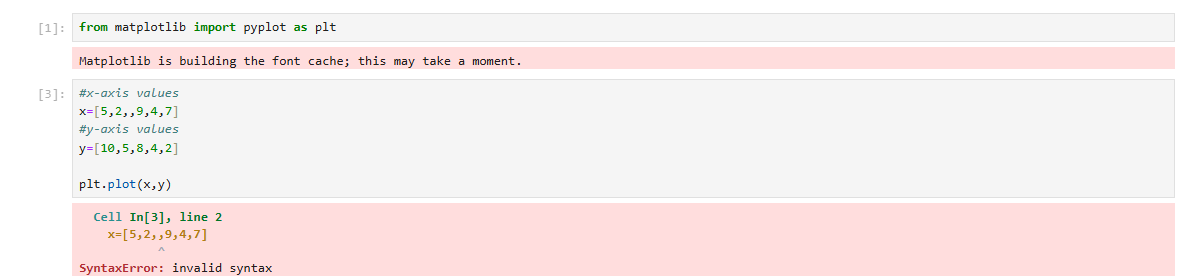
****

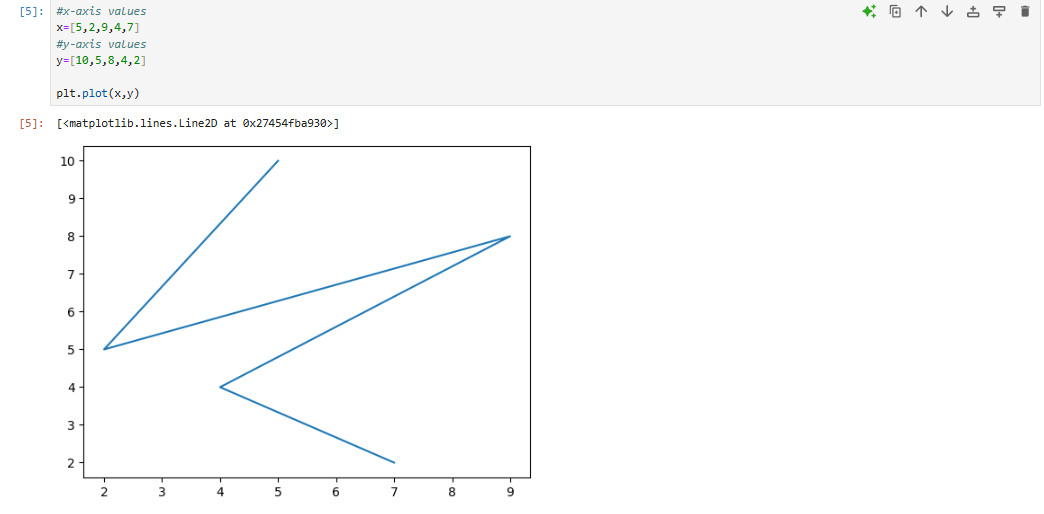
**3. SciPy**

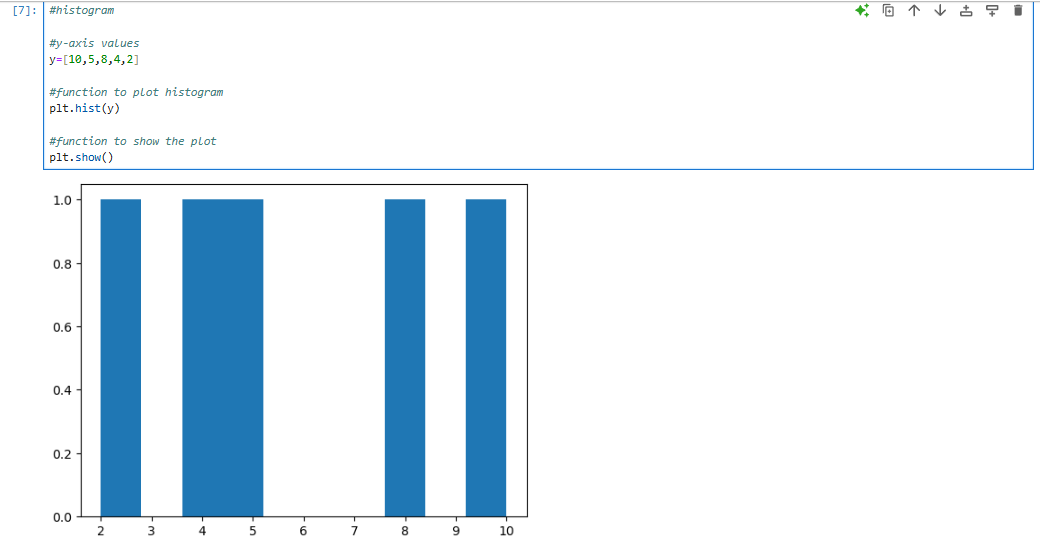
****

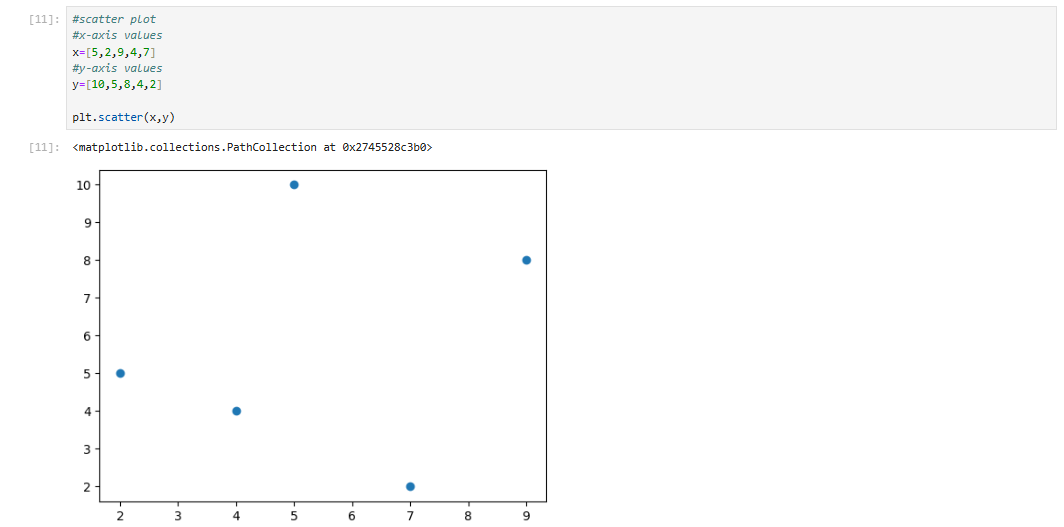
**4. Matplotlib**

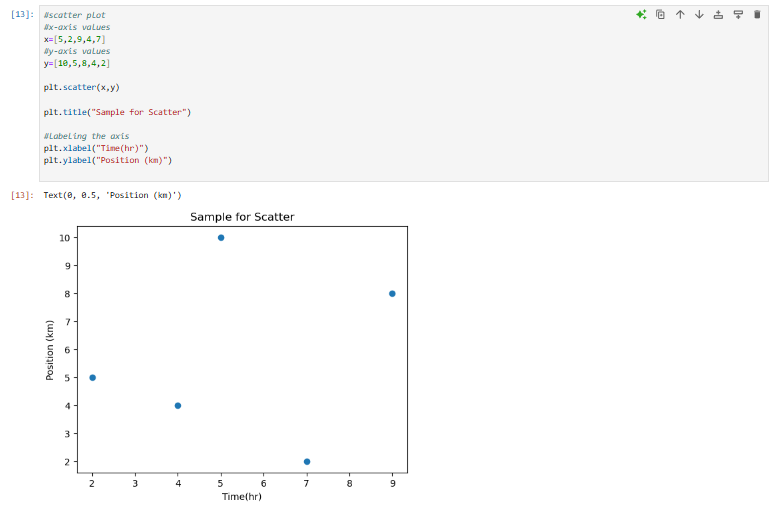
****

****

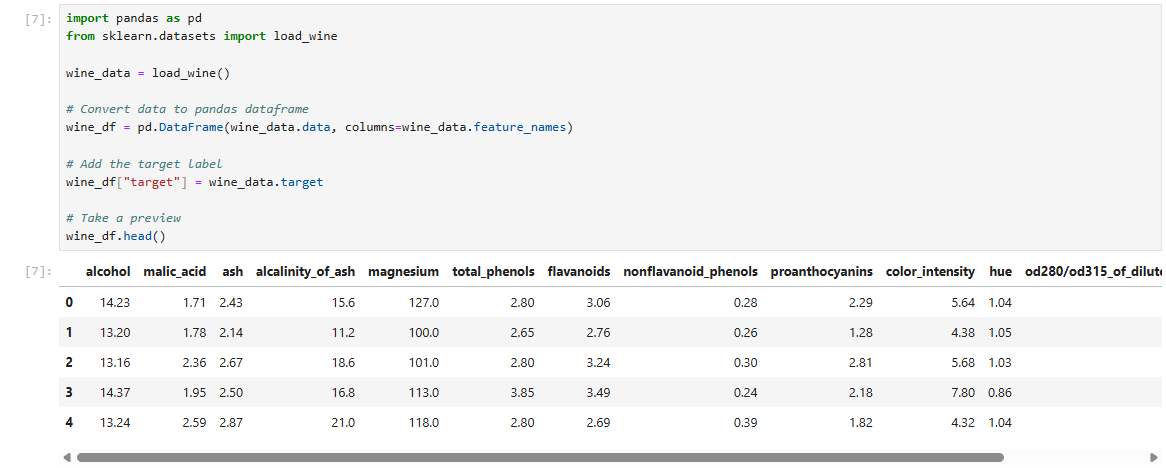
****

****

****

****

**5.SciKit Learn**

****

**Building the model**

Thanks to sklearn, building a machine learning model is extremely simple.

We are going to build three models to predict the class of wine:

1. [Logistic regression](https://www.datacamp.com/tutorial/understanding-logistic-regression-python" \t "_blank)
2. [Support vector machine](https://www.datacamp.com/tutorial/svm-classification-scikit-learn-python" \t "_blank)
3. [Decision tree classifier](https://www.datacamp.com/tutorial/decision-tree-classification-python" \t "_blank)

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier

**PRACTICAL NO 5**

**Aim:Implement Perceptron algorithm for OR operation**

**Code:**

import numpy as np

class Perceptron:

def \_\_init\_\_(self, learning\_rate=0.01, n\_iterations=1):

self.learning\_rate = learning\_rate

self.n\_iterations = n\_iterations

self.weights = None

self.bias = None

def fit(self, X, y):

n\_samples, n\_features = X.shape

self.weights = np.zeros(n\_features)

self.bias = 0

y\_ = np.array([1 if i > 0 else 0 for i in y])

for \_ in range(self.n\_iterations):

for idx, x\_i in enumerate(X):

linear\_output = np.dot(x\_i, self.weights) + self.bias

y\_predicted = self.activation\_function(linear\_output)

update = self.learning\_rate \* (y\_[idx] - y\_predicted)

self.weights += update \* x\_i

self.bias += update

def activation\_function(self, x):

return np.where(x>=0, 1, 0)

def predict(self, X):

linear\_output = np.dot(X, self.weights) + self.bias

y\_predicted = self.activation\_function(linear\_output)

return y\_predicted

# OR gate inputs and outputs

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y = np.array([0, 1, 1, 1])

# Initialize and train the perceptron

perceptron = Perceptron(learning\_rate=0.1, n\_iterations=6)

perceptron.fit(X, y)

# Test the perceptron

predictions = perceptron.predict(X)

predictions

**Output :**

**array([0, 1, 1, 1])**

**PRACTICAL NO 6**

**Aim: Improve the prediction accuracy by estimating the weight values for the training data using stochastic gradient descent. (Perceptron)**

import numpy as np

# Generate synthetic data

np.random.seed(42)

X = 2 \* np.random.rand(100, 1)

y = 4 + 3 \* X + np.random.randn(100, 1)

def sgd(X, y, learning\_rate=0.1, epochs=1000, batch\_size=1):

m = len(X)

theta = np.random.randn(2, 1)

# Add a bias term to X (X\_0 = 1)

X\_bias = np.c\_[np.ones((m, 1)), X]

cost\_history = []

for epoch in range(epochs):

# Shuffle the data at the beginning of each epoch

indices = np.random.permutation(m)

X\_shuffled = X\_bias[indices]

y\_shuffled = y[indices]

for i in range(0, m, batch\_size):

# Select a mini-batch or a single sample

X\_batch = X\_shuffled[i:i+batch\_size]

y\_batch = y\_shuffled[i:i+batch\_size]

# Compute the gradient

gradients = 2 / batch\_size \* X\_batch.T.dot(X\_batch.dot(theta) - y\_batch)

# Update the parameters (theta)

theta -= learning\_rate \* gradients

# Calculate and record the cost (Mean Squared Error)

predictions = X\_bias.dot(theta)

cost = np.mean((predictions - y) \*\* 2)

cost\_history.append(cost)

# Print progress every 100 epochs

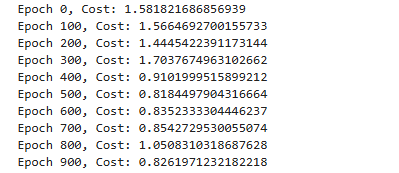
if epoch % 100 == 0:

print(f"Epoch {epoch}, Cost: {cost}")

return theta, cost\_history

# Train the model using SGD

theta\_final, cost\_history = sgd(X, y, learning\_rate=0.1, epochs=1000, batch\_size=1)



import matplotlib.pyplot as plt

# Plot the cost history

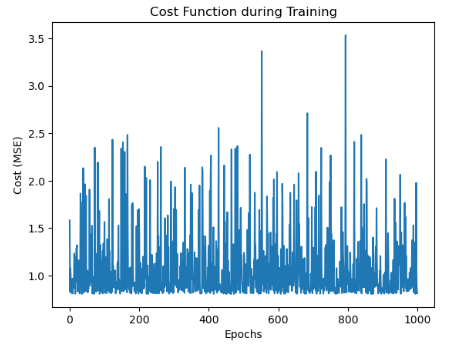
plt.plot(cost\_history)

plt.xlabel('Epochs')

plt.ylabel('Cost (MSE)')

plt.title('Cost Function during Training')

plt.show()



# Plot the data and the regression line

plt.scatter(X, y, color='blue', label='Data points')

plt.plot(X, np.c\_[np.ones((X.shape[0], 1)), X].dot(theta\_final), color='red', label='SGD fit line')

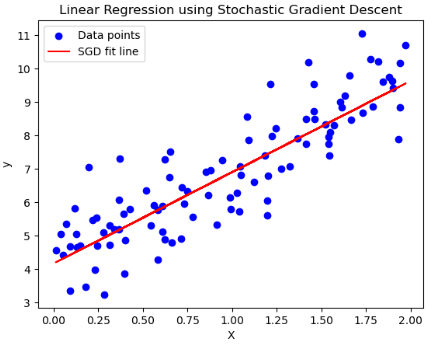
plt.xlabel('X')

plt.ylabel('y')

plt.title('Linear Regression using Stochastic Gradient Descent')

plt.legend()

plt.show()



**PRACTICAL NO 7**

**Aim:Implement Adaline algorithm for AND operation**

import numpy as np

class Adaline:

def \_\_init\_\_(self, input\_size, learning\_rate=0.1, epochs=100):

self.weights = np.zeros(input\_size)

self.bias=0

self.learning\_rate = learning\_rate

self.epochs = epochs

def activation(self, X): # X is input

return X

def predict(self,X):

return self.activation(np.dot(X, self.weights)+self.bias)

def train(self, X,y):

for epoch in range(self.epochs):

for i in range(len(X)):

prediction=self.predict(X[i])

error = y[i]-prediction

self.weights+=self.learning\_rate\*error\*X[i]

self.bias+=self.learning\_rate\*error

def evaluate(self, X):

return np.where(self.predict(X) >=0.5,1,0)

X=np.array([[0,0],[0,1],[1,0],[1,1]])

y=np.array([0,0,0,1])

adaline=Adaline(input\_size=2,learning\_rate=0.1,epochs=100)

adaline.train(X,y)

predictions = adaline.evaluate(X)

for i, prediction in enumerate(predictions):

print(f"Input: {X[i]}=>Predicted: {prediction} => Actual: {y[i]}")

**Output-**

Input: [0 0]=>Predicted: 0 => Actual: 0

Input: [0 1]=>Predicted: 0 => Actual: 0

Input: [1 0]=>Predicted: 0 => Actual: 0

Input: [1 1]=>Predicted: 1 => Actual: 1

**ML**

**PRACTICAL NO 1**

**Aim: Implementation of Features Extraction and Selection, Normalization, Transformation, Principal Components Analysis.**

**Code:**

import pandas as pd

import numpy as np

from sklearn.preprocessing import StandardScaler, MinMaxScaler, PowerTransformer, LabelEncoder

from sklearn.feature\_selection import SelectKBest, f\_classif, VarianceThreshold

from sklearn.decomposition import PCA

# Load the healthcare dataset

file\_path = "health\_dataset.csv"

df = pd.read\_csv(file\_path)

# Display first few rows

display(df.head())

# Identify non-numeric columns

categorical\_columns = df.select\_dtypes(include=['object']).columns

print("Categorical Columns:", categorical\_columns)

# Convert categorical columns using Label Encoding

label\_encoders = {}

for col in categorical\_columns:

    le = LabelEncoder()

    df[col] = le.fit\_transform(df[col])

    label\_encoders[col] = le

# Ensure all columns are numeric

df = df.apply(pd.to\_numeric, errors='coerce')  # Converts non-numeric values to NaN

df.fillna(0, inplace=True)  # Replace NaNs with 0

# Separate features and target

target\_column = 'Disease Risk Score'

if target\_column not in df.columns:

    raise ValueError("Target column not found in dataset")

X = df.drop(columns=[target\_column])

y = df[target\_column]

# Check for missing values in target and encode if necessary

if y.isnull().sum() > 0:

    y.fillna(y.mode()[0], inplace=True)  # Replace NaN with most frequent value

if y.dtype == 'object':

    le = LabelEncoder()

    y = le.fit\_transform(y)

# Remove constant features

var\_thresh = VarianceThreshold(threshold=0)

X = var\_thresh.fit\_transform(X)

# Feature Selection using ANOVA F-score

selector = SelectKBest(score\_func=f\_classif, k=min(10, X.shape[1]))  # Ensure k does not exceed feature count

X\_selected = selector.fit\_transform(X, y)

selected\_features = [col for col, keep in zip(df.drop(columns=[target\_column]).columns, selector.get\_support()) if keep]

print("Selected Features:", selected\_features)

# Normalization using Min-Max Scaling

scaler = MinMaxScaler()

X\_normalized = scaler.fit\_transform(X\_selected)

# Transformation using Power Transform (Box-Cox or Yeo-Johnson)

power\_transformer = PowerTransformer(method='yeo-johnson')  # Use 'box-cox' if no negative values

X\_transformed = power\_transformer.fit\_transform(X\_normalized)

# Dimensionality Reduction using PCA

pca\_components = min(5, X\_transformed.shape[1])  # Ensure PCA components do not exceed feature count

pca = PCA(n\_components=pca\_components)

X\_pca = pca.fit\_transform(X\_transformed)

print("Explained Variance Ratio:", pca.explained\_variance\_ratio\_)

# Convert processed data back to DataFrame

processed\_df = pd.DataFrame(X\_pca, columns=[f'PC{i+1}' for i in range(X\_pca.shape[1])])

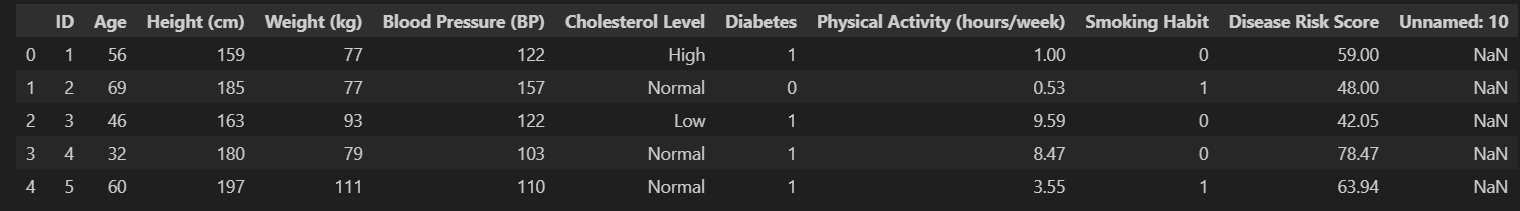
processed\_df['Disease Risk Score'] = y.reset\_index(drop=True)

# Save processed dataset

processed\_df.to\_csv("processed\_health\_dataset.csv", index=False)

print("Processed dataset saved successfully.")

**OUTPUT**



**PRACTICAL NO 2**

**Aim: Implementation of Logistic regression**

Statement 1: Build and train a Logistic Regression Model to do binary classification of iris flowers using the iris dataset.

In particular, the model should predict whether a particular iris flower instance belongs to the class Iris Virginica or not using

#only petal width as the input feature.

**CODE**

import numpy as np

from sklearn import datasets

iris = datasets.load\_iris()

print(type(iris))

print(list(iris.keys()))

X = iris["data"][:,3:] # petal width

y = (iris["target"] == 2).astype(np.int64) # 1 if Iris-Virginica, else 0

**Output:**

<class 'sklearn.utils.\_bunch.Bunch'>

['data', 'target', 'frame', 'target\_names', 'DESCR', 'feature\_names', 'filename', 'data\_module']

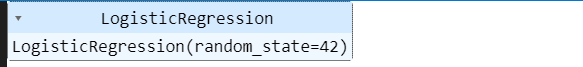
**Code:**

from sklearn.linear\_model import LogisticRegression

log\_reg = LogisticRegression(solver="lbfgs", random\_state=42)

log\_reg.fit(X,y)

**Output**

****

**Code**

import matplotlib.pyplot as plt

X\_new = np.linspace(0,3,1000).reshape(-1,1)

y\_proba = log\_reg.predict\_proba(X\_new)

plt.plot(X\_new, y\_proba[:,1],"g-")

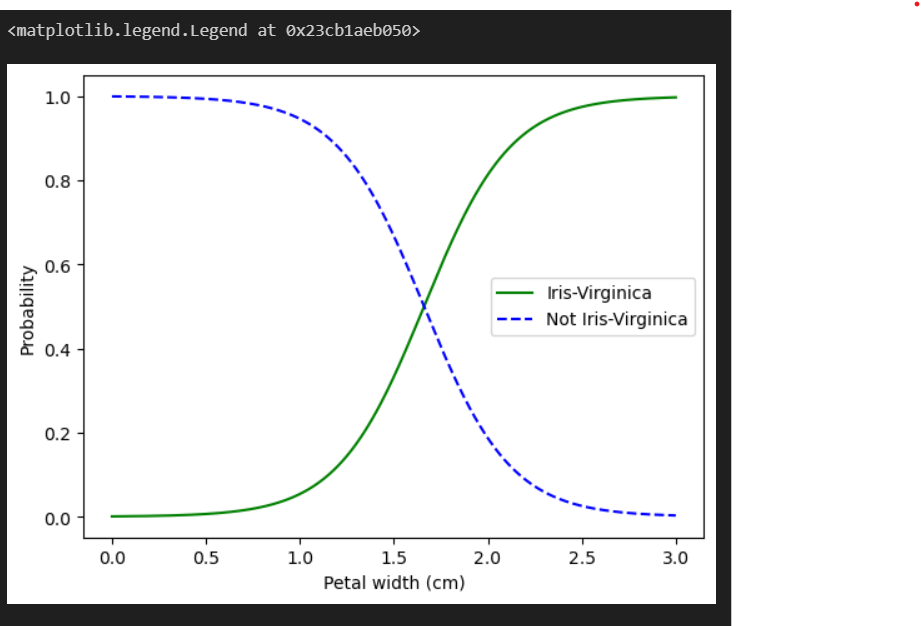
plt.plot(X\_new, y\_proba[:,0], "b--")

plt.xlabel('Petal width (cm)')

plt.ylabel('Probability')

plt.legend(['Iris-Virginica','Not Iris-Virginica'])

**Output**

****

**Code**

log\_reg.predict([[1.7],[1.5]])

**Output**

array([1, 0], dtype=int64)

**Code**

from sklearn.linear\_model import LogisticRegression

X = iris["data"][:, (2, 3)]  # petal length, petal width

y = (iris["target"] == 2).astype(np.int64)

log\_reg2 = LogisticRegression(solver="lbfgs", C=10\*\*10, random\_state=42)

log\_reg2.fit(X, y)

x0, x1 = np.meshgrid(

        np.linspace(2.9, 7, 500).reshape(-1, 1),

        np.linspace(0.8, 2.7, 200).reshape(-1, 1),

    )

X\_new = np.c\_[x0.ravel(), x1.ravel()]

print(X\_new.shape)

y\_proba = log\_reg2.predict\_proba(X\_new)

plt.figure(figsize=(10, 4))

plt.plot(X[y==0, 0], X[y==0, 1], "bs")

plt.plot(X[y==1, 0], X[y==1, 1], "g^")

zz = y\_proba[:, 1].reshape(x0.shape)

contour = plt.contour(x0, x1, zz, cmap=plt.cm.brg)

left\_right = np.array([2.9, 7])

boundary = -(log\_reg2.coef\_[0][0] \* left\_right + log\_reg2.intercept\_[0]) / log\_reg2.coef\_[0][1]

plt.clabel(contour, inline=1, fontsize=12)

plt.plot(left\_right, boundary, "k--", linewidth=3)

plt.text(3.5, 1.5, "Not Iris virginica", fontsize=14, color="b", ha="center")

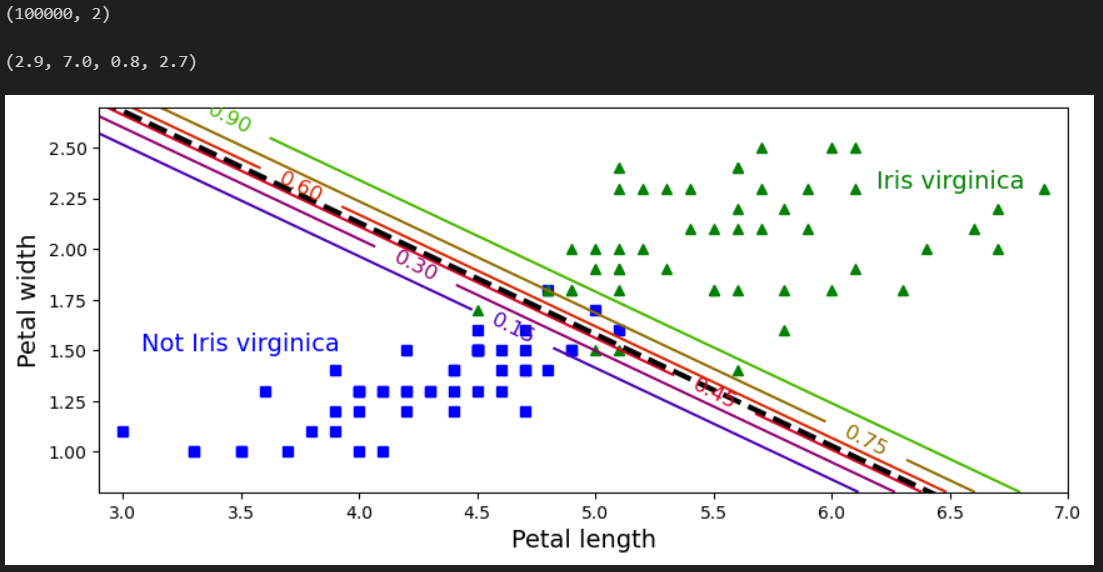
plt.text(6.5, 2.3, "Iris virginica", fontsize=14, color="g", ha="center")

plt.xlabel("Petal length", fontsize=14)

plt.ylabel("Petal width", fontsize=14)

plt.axis([2.9, 7, 0.8, 2.7])

**Output**

****

**PRACTICAL NO 3**

**Aim: Implementation of Classifying data using Support Vector Machine (SVM)- Linear and Non-Linear SVM Classification**

**LINEAR SVM**

**Code:**

%matplotlib inline

import matplotlib

import matplotlib.pyplot as plt

def plot\_svc\_decision\_boundary(svm\_clf, xmin, xmax):

    w = svm\_clf.coef\_[0]

    b = svm\_clf.intercept\_[0]

    # At the decision boundary, w0\*x0 + w1\*x1 + b = 0

    # => x1 = -w0/w1 \* x0 - b/w1

    x0 = np.linspace(xmin, xmax, 200)

    decision\_boundary = -w[0]/w[1] \* x0 - b/w[1]

    margin = 1/w[1]

    gutter\_up = decision\_boundary + margin

    gutter\_down = decision\_boundary - margin

    svs = svm\_clf.support\_vectors\_

    plt.scatter(svs[:, 0], svs[:, 1], s=180, facecolors='#FFAAAA')

    plt.plot(x0, decision\_boundary, "k-", linewidth=2)

    plt.plot(x0, gutter\_up, "k--", linewidth=2)

    plt.plot(x0, gutter\_down, "k--", linewidth=2)

**Code**

from sklearn.svm import SVC

from sklearn import datasets

import numpy as np

# Load Iris dataset

iris = datasets.load\_iris()

X = iris["data"][:, (2, 3)]  # Select petal length and petal width

y = iris["target"]

# Select only Setosa and Versicolor classes

setosa\_or\_versicolor = (y == 0) | (y == 1)

X = X[setosa\_or\_versicolor]

y = y[setosa\_or\_versicolor]

# SVM Classifier model with a large but finite C value

svm\_clf = SVC(kernel="linear", C=1e10)  # Large C approximates a hard margin

svm\_clf.fit(X, y)

# Make a prediction

prediction = svm\_clf.predict([[2.4, 3.1]])

print("Predicted class:", prediction[0])

**Output**

Predicted class: 1

**Code**

#plot the decision boundaries

import numpy as np

plt.figure(figsize=(12,3.2))

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

svm\_clf.fit(X\_scaled, y)

plt.plot(X\_scaled[:, 0][y==1], X\_scaled[:, 1][y==1], "bo")

plt.plot(X\_scaled[:, 0][y==0], X\_scaled[:, 1][y==0], "ms")

plot\_svc\_decision\_boundary(svm\_clf, -2, 2)

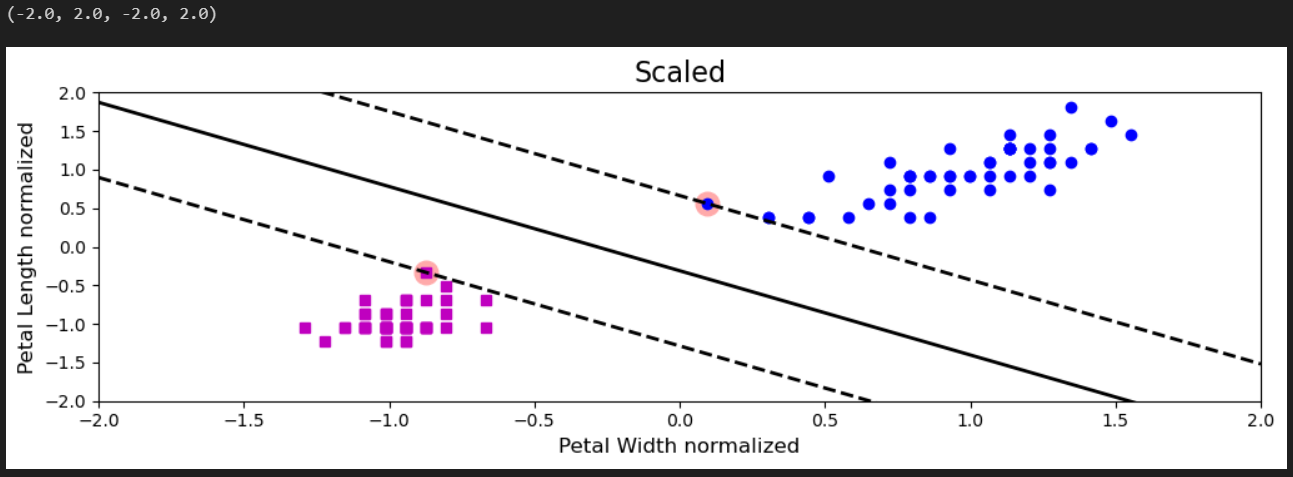
plt.xlabel("Petal Width normalized", fontsize=12)

plt.ylabel("Petal Length normalized", fontsize=12)

plt.title("Scaled", fontsize=16)

plt.axis([-2, 2, -2, 2])

**Output**



**NON-LINEAR SVM**

**Code**

from sklearn.datasets import make\_moons

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import PolynomialFeatures

from sklearn.preprocessing import StandardScaler

from sklearn.svm import SVC

**Code**

import numpy as np

%matplotlib inline

import matplotlib

import matplotlib.pyplot as plt

**Code**

from sklearn.datasets import make\_moons

X, y = make\_moons(n\_samples=100, noise=0.15, random\_state=42)

#define a function to plot the dataset

def plot\_dataset(X, y, axes):

    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")

    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "ms")

    plt.axis(axes)

    plt.grid(True, which='both')

    plt.xlabel(r"$x\_1$", fontsize=20)

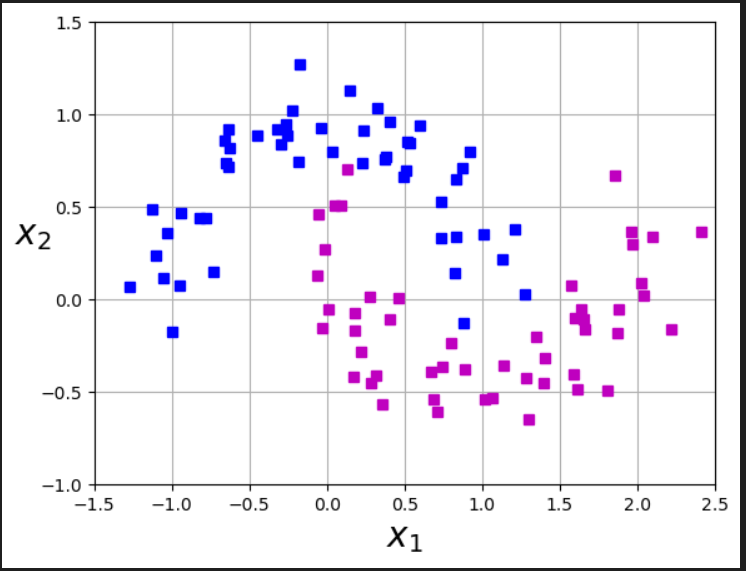
    plt.ylabel(r"$x\_2$", fontsize=20, rotation=0)

#Let's have a look at the data we have generated

plot\_dataset(X, y, [-1.5, 2.5, -1, 1.5])

plt.show()

**Output**



**Code**

#define a function plot the decision boundaries

def plot\_predictions(clf, axes):

    #create data in continous linear space

    x0s = np.linspace(axes[0], axes[1], 100)

    x1s = np.linspace(axes[2], axes[3], 100)

    x0, x1 = np.meshgrid(x0s, x1s)

    X = np.c\_[x0.ravel(), x1.ravel()]

    y\_pred = clf.predict(X).reshape(x0.shape)

    y\_decision = clf.decision\_function(X).reshape(x0.shape)

    plt.contourf(x0, x1, y\_pred, cmap=plt.cm.brg, alpha=0.2)

    plt.contourf(x0, x1, y\_decision, cmap=plt.cm.brg, alpha=0.1)

**Code**

#C controls the width of the street

#Degree of data

#create a pipeline to create features, scale data and fit the model

polynomial\_svm\_clf = Pipeline((

    ("poly\_features", PolynomialFeatures(degree=3)),

    ("scalar", StandardScaler()),

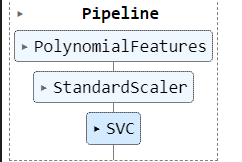
    ("svm\_clf", SVC(kernel="poly", degree=10, coef0=1, C=5))

))

#call the pipeline

polynomial\_svm\_clf.fit(X,y)

**Output**



**Code**

#plot the decision boundaries

plt.figure(figsize=(11, 4))

#plot the decision boundaries

plot\_predictions(polynomial\_svm\_clf, [-1.5, 2.5, -1, 1.5])

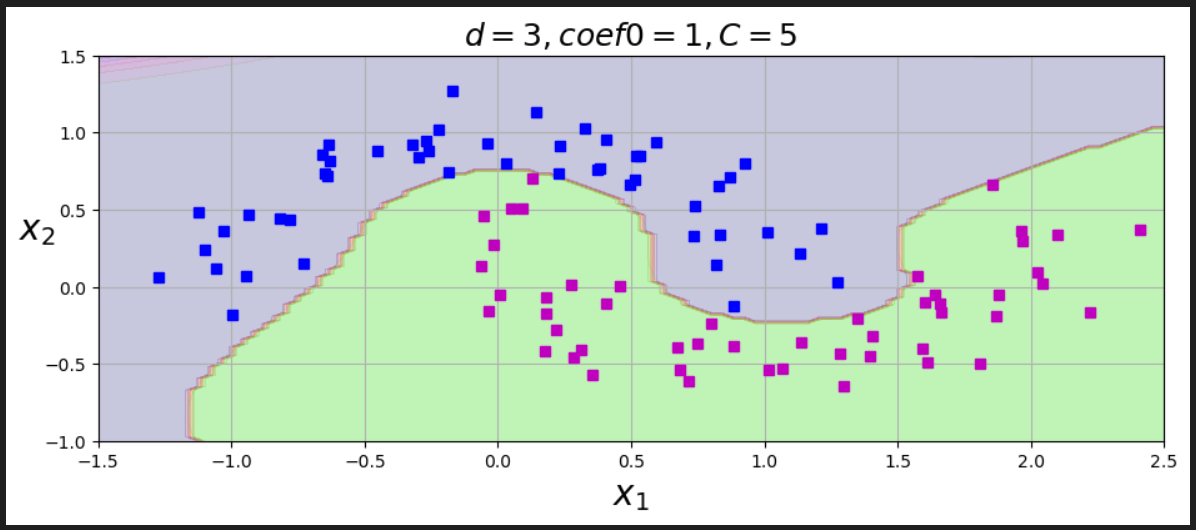
#plot the dataset

plot\_dataset(X, y, [-1.5, 2.5, -1, 1.5])

plt.title(r"$d=3, coef0=1, C=5$", fontsize=18)

plt.show()

**Output**



**PRACTICAL NO 4**

**Aim: Implement Elbow method for K means Clustering**

**Code**

!pip install --user threadpoolctl==3.1.0

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

# Load the dataset

df = pd.read\_csv("clustering.csv")

# Display first few rows of the dataset

print(df.head())

# Drop missing values

df\_cleaned = df.dropna()

# Selecting numerical columns for clustering

numerical\_cols = df\_cleaned.select\_dtypes(include=[np.number]).columns

print("Numerical columns used for clustering:", numerical\_cols.tolist())

# Feature selection for clustering (Modify as needed)

X = df\_cleaned[numerical\_cols]

# Apply the Elbow Method

wcss = []  # Within-cluster sum of squares

for i in range(1, 11):  # Trying different cluster numbers from 1 to 10

    kmeans = KMeans(n\_clusters=i, random\_state=42, n\_init=10)

    kmeans.fit(X)

    wcss.append(kmeans.inertia\_)

# Plot the Elbow Method

plt.plot(range(1, 11), wcss, marker='o', linestyle='--')

plt.xlabel('Number of Clusters')

plt.ylabel('WCSS')

plt.title('Elbow Method for Optimal k')

plt.show()

# Choose optimal k (Modify based on the elbow plot observation)

k\_optimal = 3  # Example choice, change based on your dataset

# Apply K-Means with the optimal number of clusters

kmeans = KMeans(n\_clusters=k\_optimal, random\_state=42, n\_init=10)

df\_cleaned['Cluster'] = kmeans.fit\_predict(X)

# Display clustered data

print(df\_cleaned.head())

# Plot the clusters (for 2D visualization, choose two relevant features)

plt.scatter(df\_cleaned[numerical\_cols[0]], df\_cleaned[numerical\_cols[1]], c=df\_cleaned['Cluster'], cmap='viridis')

plt.xlabel(numerical\_cols[0])

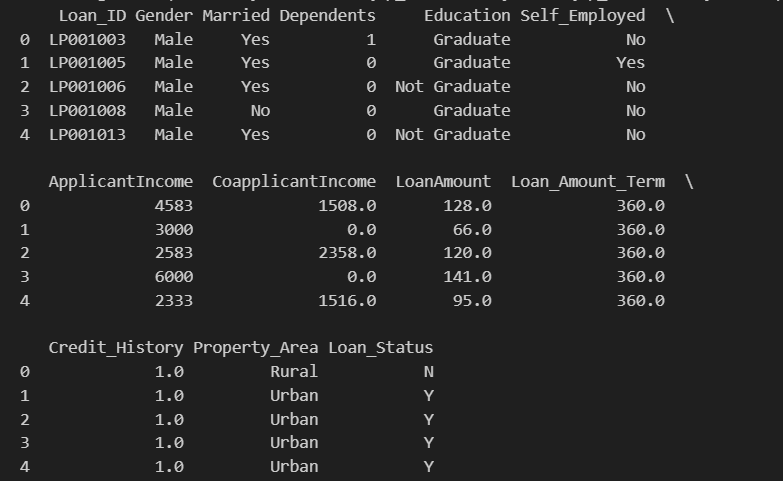
plt.ylabel(numerical\_cols[1])

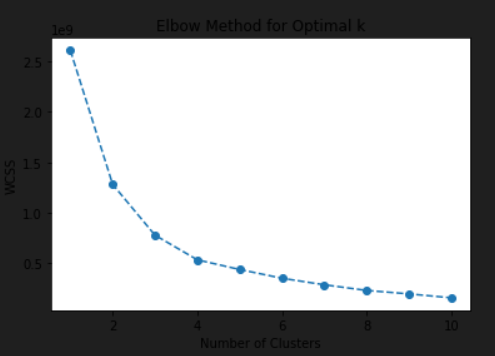
plt.title(f'K-Means Clustering (k={k\_optimal})')

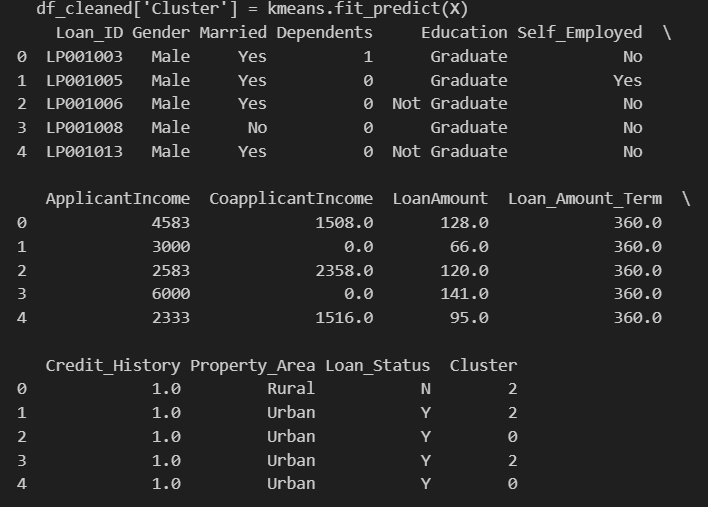
plt.colorbar(label='Cluster')

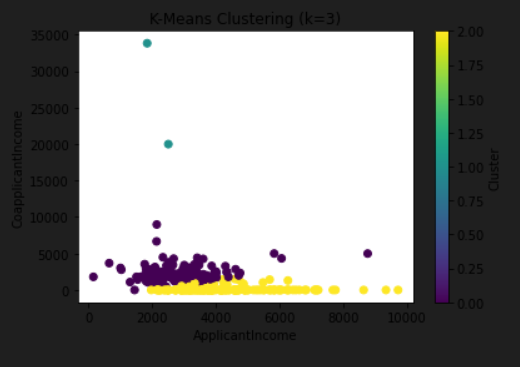
plt.show()

**Output**









**PRACTICAL NO 5**

**Aim: Implementation of Bagging Algorithm: Random Forest**

**Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

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

from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, GradientBoostingClassifier, VotingClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

from sklearn.decomposition import PCA

# Load dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split data into training and testing sets

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

# Random Forest Classifier

rf\_classifier = RandomForestClassifier(n\_estimators=100, random\_state=42)

rf\_classifier.fit(X\_train, y\_train)

y\_pred\_rf = rf\_classifier.predict(X\_test)

accuracy\_rf = accuracy\_score(y\_test, y\_pred\_rf)

print(f'Accuracy of Random Forest Classifier: {accuracy\_rf \* 100:.2f}%')

# AdaBoost Classifier

adaboost = AdaBoostClassifier(base\_estimator=DecisionTreeClassifier(max\_depth=1), n\_estimators=50, random\_state=42)

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

y\_pred\_adaboost = adaboost.predict(X\_test)

accuracy\_adaboost = accuracy\_score(y\_test, y\_pred\_adaboost)

print(f'Accuracy of AdaBoost Classifier: {accuracy\_adaboost \* 100:.2f}%')

# Gradient Boosting Classifier

gb\_classifier = GradientBoostingClassifier(n\_estimators=100, learning\_rate=0.1, random\_state=42)

gb\_classifier.fit(X\_train, y\_train)

y\_pred\_gb = gb\_classifier.predict(X\_test)

accuracy\_gb = accuracy\_score(y\_test, y\_pred\_gb)

print(f'Accuracy of Gradient Boosting Classifier: {accuracy\_gb \* 100:.2f}%')

# Voting Classifier (Ensemble of Logistic Regression, Decision Tree, and Random Forest)

voting\_classifier = VotingClassifier(estimators=[

    ('lr', LogisticRegression()),

    ('dt', DecisionTreeClassifier()),

    ('rf', RandomForestClassifier(n\_estimators=100))

], voting='hard')

voting\_classifier.fit(X\_train, y\_train)

y\_pred\_voting = voting\_classifier.predict(X\_test)

accuracy\_voting = accuracy\_score(y\_test, y\_pred\_voting)

print(f'Accuracy of Voting Classifier: {accuracy\_voting \* 100:.2f}%')

# Reduce dimensions for visualization

pca = PCA(n\_components=2)

X\_reduced = pca.fit\_transform(X)

# Scatter plot of the dataset

plt.figure(figsize=(8, 6))

plt.scatter(X\_reduced[:, 0], X\_reduced[:, 1], c=y, cmap='viridis', edgecolor='k', alpha=0.7)

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.title('Iris Dataset Visualization with PCA')

plt.colorbar(label='Class Labels')

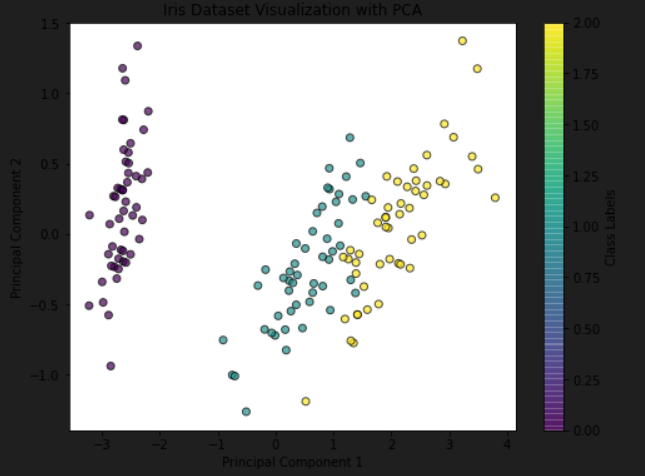
plt.show()

**Output**

Accuracy of Random Forest Classifier: 100.00%

Accuracy of AdaBoost Classifier: 100.00%

Accuracy of Gradient Boosting Classifier: 100.00%



**PRACTICAL NO 6**

**Aim: Implementation of Boosting Algorithms: AdaBoost, Stochastic Gradient Boosting, Voting Ensemble**

**Code**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

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

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score

from sklearn.decomposition import PCA

# Load dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split data into training and testing sets

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

# Initialize and train the Random Forest Classifier

rf\_classifier = RandomForestClassifier(n\_estimators=100, random\_state=42)

rf\_classifier.fit(X\_train, y\_train)

# Make predictions

y\_pred = rf\_classifier.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

print(f'Accuracy of Random Forest Classifier: {accuracy \* 100:.2f}%')

**OUTPUT**

Accuracy of Random Forest Classifier: 100.00%