**CNN**

import numpy as np

from google.colab import drive

drive.mount('/content/drive')

!unzip -q "drive/My Drive/flowers.zip"

import numpy as np

import tensorflow as tf

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Define paths

train\_data\_dir = 'flowers/train'

test\_data\_dir = 'flowers/test'

# Define image parameters

img\_width, img\_height = 150, 150

input\_shape = (img\_width, img\_height, 3)

epochs = 20

batch\_size = 32

# Data Augmentation for training data

train\_datagen = ImageDataGenerator(rescale=1. / 255,

shear\_range=0.2,

zoom\_range=0.2,

horizontal\_flip=True)

# Only rescaling for testing data

test\_datagen = ImageDataGenerator(rescale=1. / 255)

# Generating data from directories

train\_generator = train\_datagen.flow\_from\_directory(train\_data\_dir,

target\_size=(img\_width, img\_height),

batch\_size=batch\_size,

class\_mode='categorical')

test\_generator = test\_datagen.flow\_from\_directory(test\_data\_dir,

target\_size=(img\_width, img\_height),

batch\_size=batch\_size,

class\_mode='categorical')

# Building the CNN model

model = Sequential()

model.add(Conv2D(32, (3, 3), input\_shape=input\_shape, activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(128, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(128, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(512, activation='relu'))

model.add(Dense(5, activation='softmax')) # 5 classes: daisy, dandelion, rose, sunflower, tulip

model.compile(loss='categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

# Training the model

model.fit(train\_generator,

steps\_per\_epoch=train\_generator.samples // batch\_size,

epochs=epochs,

validation\_data=test\_generator,

validation\_steps=test\_generator.samples // batch\_size)

# Saving the model

model.save("flower\_classification\_model.h5")

import numpy as np

import tensorflow as tf

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Define paths

train\_data\_dir = 'flowers/train'

test\_data\_dir = 'flowers/test'

# Define image parameters

img\_width, img\_height = 150, 150

input\_shape = (img\_width, img\_height, 3)

epochs = 5

batch\_size = 32

# Data Augmentation for training data

train\_datagen = ImageDataGenerator(rescale=1. / 255,

shear\_range=0.2,

zoom\_range=0.2,

horizontal\_flip=True)

# Only rescaling for testing data

test\_datagen = ImageDataGenerator(rescale=1. / 255)

# Generating data from directories

train\_generator = train\_datagen.flow\_from\_directory(train\_data\_dir,

target\_size=(img\_width, img\_height),

batch\_size=batch\_size,

class\_mode='categorical')

test\_generator = test\_datagen.flow\_from\_directory(test\_data\_dir,

target\_size=(img\_width, img\_height),

batch\_size=batch\_size,

class\_mode='categorical')

# Building the CNN model

model = Sequential()

model.add(Conv2D(32, (3, 3), input\_shape=input\_shape, activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(128, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(128, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(512, activation='relu'))

model.add(Dense(5, activation='softmax')) # 5 classes: daisy, dandelion, rose, sunflower, tulip

model.compile(loss='categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

# Training the model

model.fit(train\_generator,

steps\_per\_epoch=train\_generator.samples // batch\_size,

epochs=epochs,

validation\_data=test\_generator,

validation\_steps=test\_generator.samples // batch\_size)

# Saving the model

model.save("flower\_classification\_model.h5")

from tensorflow.keras.models import load\_model

from tensorflow.keras.preprocessing.image import img\_to\_array, load\_img

# Define image path

img\_path = '/content/flowers/test/Image\_5.jpg' # Replace with your image path

# Load the saved model

model = load\_model('flower\_classification\_model.h5')

# Preprocess the image

img = load\_img(img\_path, target\_size=(150, 150)) # Adjust if your image size differs

img = img\_to\_array(img)

img /= 255.0 # Rescale for model input

img = np.expand\_dims(img, axis=0) # Add a dimension for batch processing

# Make prediction

predictions = model.predict(img)

# Get the flower class with the highest probability

predicted\_class\_index = np.argmax(predictions[0])

# Load flower class labels (assuming you have a list named 'class\_names')

flower\_names = ['daisy', 'dandelion', 'rose', 'sunflower', 'tulip'] # Adjust class names

# Print the predicted flower class

print(f"Predicted flower: {flower\_names[predicted\_class\_index]}")

**ANN**

import numpy as np

from sklearn.datasets import load\_iris

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

# Load the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split the dataset 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)

class NeuralNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, learning\_rate=0.1):

self.input\_size = input\_size

self.hidden\_size = hidden\_size

self.output\_size = output\_size

self.learning\_rate = learning\_rate

# Initialize weights and biases

self.W1 = np.random.randn(self.input\_size, self.hidden\_size)

self.b1 = np.zeros((1, self.hidden\_size))

self.W2 = np.random.randn(self.hidden\_size, self.output\_size)

self.b2 = np.zeros((1, self.output\_size))

def sigmoid(self, x):

return 1 / (1 + np.exp(-x))

def tanh(self, x):

return np.tanh(x)

def relu(self, x):

return np.maximum(0, x)

def softmax(self, x):

exp\_values = np.exp(x - np.max(x, axis=1, keepdims=True))

return exp\_values / np.sum(exp\_values, axis=1, keepdims=True)

def forward(self, X):

self.z1 = np.dot(X, self.W1) + self.b1

self.a1 = self.relu(self.z1) # Change activation function here

self.z2 = np.dot(self.a1, self.W2) + self.b2

self.output = self.softmax(self.z2)

return self.output

def backward(self, X, y):

m = X.shape[0]

delta3 = self.output

delta3[range(m), y] -= 1

dW2 = np.dot(self.a1.T, delta3)

db2 = np.sum(delta3, axis=0, keepdims=True)

delta2 = np.dot(delta3, self.W2.T) \* (self.a1 > 0) # Change activation function here

dW1 = np.dot(X.T, delta2)

db1 = np.sum(delta2, axis=0)

# Update weights and biases

self.W2 -= self.learning\_rate \* dW2

self.b2 -= self.learning\_rate \* db2

self.W1 -= self.learning\_rate \* dW1

self.b1 -= self.learning\_rate \* db1

def train(self, X, y, epochs=100):

for epoch in range(epochs):

# Forward pass

output = self.forward(X)

# Backward pass

self.backward(X, y)

# Compute loss

loss = self.cross\_entropy\_loss(output, y)

if epoch % 100 == 0:

print(f'Epoch {epoch}, Loss: {loss}')

def cross\_entropy\_loss(self, y\_pred, y\_true):

m = y\_true.shape[0]

log\_likelihood = -np.log(y\_pred[range(m), y\_true])

loss = np.sum(log\_likelihood) / m

return loss

def predict(self, X):

return np.argmax(self.forward(X), axis=1)

# Parameters

input\_size = X\_train.shape[1]

hidden\_size = 64 # Change the number of hidden nodes here

output\_size = len(np.unique(y\_train))

learning\_rate = 0.001

epochs = 1000

# Create and train the neural network

model = NeuralNetwork(input\_size, hidden\_size, output\_size, learning\_rate)

model.train(X\_train, y\_train, epochs=epochs)

# Evaluate the model

predictions = model.predict(X\_test)

accuracy = np.mean(predictions == y\_test)

print(f'Test Accuracy: {accuracy}')

# Parameters

input\_size = X\_train.shape[1]

hidden\_size = 128 # Increase the number of nodes in the hidden layer

output\_size = len(np.unique(y\_train))

learning\_rate = 0.001

epochs = 1000

# Create and train the neural network

model = NeuralNetwork(input\_size, hidden\_size, output\_size, learning\_rate)

model.train(X\_train, y\_train, epochs=epochs)

# Evaluate the model

predictions = model.predict(X\_test)

accuracy = np.mean(predictions == y\_test)

print(f'Test Accuracy with {hidden\_size} hidden nodes: {accuracy}')

class NeuralNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, learning\_rate=0.1):

self.input\_size = input\_size

self.hidden\_size = hidden\_size

self.output\_size = output\_size

self.learning\_rate = learning\_rate

# Initialize weights and biases

self.W1 = np.random.randn(self.input\_size, self.hidden\_size)

self.b1 = np.zeros((1, self.hidden\_size))

self.W2 = np.random.randn(self.hidden\_size, self.output\_size)

self.b2 = np.zeros((1, self.output\_size))

def sigmoid(self, x):

return 1 / (1 + np.exp(-x))

def tanh(self, x):

return np.tanh(x)

def relu(self, x):

return np.maximum(0, x)

def softmax(self, x):

exp\_values = np.exp(x - np.max(x, axis=1, keepdims=True))

return exp\_values / np.sum(exp\_values, axis=1, keepdims=True)

def forward(self, X, activation='relu'):

self.z1 = np.dot(X, self.W1) + self.b1

if activation == 'sigmoid':

self.a1 = self.sigmoid(self.z1)

elif activation == 'tanh':

self.a1 = self.tanh(self.z1)

elif activation == 'relu':

self.a1 = self.relu(self.z1)

self.z2 = np.dot(self.a1, self.W2) + self.b2

self.output = self.softmax(self.z2)

return self.output

def backward(self, X, y):

m = X.shape[0]

delta3 = self.output

delta3[range(m), y] -= 1

dW2 = np.dot(self.a1.T, delta3)

db2 = np.sum(delta3, axis=0, keepdims=True)

delta2 = np.dot(delta3, self.W2.T) \* (self.a1 > 0) # Change activation function here

dW1 = np.dot(X.T, delta2)

db1 = np.sum(delta2, axis=0)

# Update weights and biases

self.W2 -= self.learning\_rate \* dW2

self.b2 -= self.learning\_rate \* db2

self.W1 -= self.learning\_rate \* dW1

self.b1 -= self.learning\_rate \* db1

def train(self, X, y, epochs=100):

for epoch in range(epochs):

# Forward pass

output = self.forward(X)

# Backward pass

self.backward(X, y)

# Compute loss

loss = self.cross\_entropy\_loss(output, y)

if epoch % 100 == 0:

print(f'Epoch {epoch}, Loss: {loss}')

def cross\_entropy\_loss(self, y\_pred, y\_true):

m = y\_true.shape[0]

log\_likelihood = -np.log(y\_pred[range(m), y\_true])

loss = np.sum(log\_likelihood) / m

return loss

def predict(self, X):

return np.argmax(self.forward(X), axis=1)

# Parameters

input\_size = X\_train.shape[1]

hidden\_size = 64

output\_size = len(np.unique(y\_train))

learning\_rate = 0.001

epochs = 1000

# Activation functions to test

activation\_functions = ['sigmoid', 'tanh', 'relu']

for activation in activation\_functions:

# Create and train the neural network

model = NeuralNetwork(input\_size, hidden\_size, output\_size, learning\_rate)

model.train(X\_train, y\_train, epochs=epochs)

# Evaluate the model

predictions = model.predict(X\_test)

accuracy = np.mean(predictions == y\_test)

print(f'Test Accuracy with {activation.capitalize()} activation function: {accuracy}')

**SVM**

import matplotlib.pyplot as plt

from sklearn import svm

# Sample data points

positive\_data = [(3, 1), (3, -1), (6, 1), (6, -1)]

negative\_data = [(1, 0), (0, 1), (0, -1), (-1, 0)]

# Extract x and y coordinates separately

positive\_x = [point[0] for point in positive\_data]

positive\_y = [point[1] for point in positive\_data]

negative\_x = [point[0] for point in negative\_data]

negative\_y = [point[1] for point in negative\_data]

# Plot the data points

plt.scatter(positive\_x, positive\_y, label='Positive')

plt.scatter(negative\_x, negative\_y, label='Negative')

plt.xlabel('X')

plt.ylabel('Y')

plt.title('Data Points for SVM Classification')

plt.legend()

# Create a linear SVM classifier

clf = svm.SVC(kernel='linear')

# Train the classifier with the data

clf.fit(positive\_data + negative\_data, [1] \* len(positive\_data) + [-1] \* len(negative\_data))

# Identify the nearest data points (support vectors)

support\_vectors = clf.support\_vectors\_

# Extract x and y coordinates of support vectors

support\_x = [point[0] for point in support\_vectors]

support\_y = [point[1] for point in support\_vectors]

# Plot the support vectors

plt.scatter(support\_x, support\_y, marker='o', c='red', label='Support Vectors')

plt.legend()

# Get the hyperplane coefficients from the model

w = clf.coef\_[0]

b = clf.intercept\_[0]

# Calculate the equation of the hyperplane

decision\_boundary = lambda x: (-w[0] \* x - b) / w[1]

# Plot the hyperplane boundary

x\_values = range(min(positive\_x + negative\_x) - 2, max(positive\_x + negative\_x) + 2)

plt.plot(x\_values, [decision\_boundary(x) for x in x\_values], label='Hyperplane')

plt.legend()

plt.grid(True)

plt.show()

import matplotlib.pyplot as plt

from sklearn import svm

positive\_data = [(3, 1), (3, -1), (6, 1), (6, -1)]

negative\_data = [(1, 0), (0, 1), (0, -1), (-1, 0)]

positive\_x = [point[0] for point in positive\_data]

positive\_y = [point[1] for point in positive\_data]

negative\_x = [point[0] for point in negative\_data]

negative\_y = [point[1] for point in negative\_data]

plt.scatter(positive\_x, positive\_y, label='Positive')

plt.scatter(negative\_x, negative\_y, label='Negative')

plt.xlabel('X')

plt.ylabel('Y')

plt.title('Data Points for SVM Classification')

plt.legend()

clf = svm.SVC(kernel='linear')

clf.fit(positive\_data + negative\_data, [1] \* len(positive\_data) + [-1] \* len(negative\_data))

support\_vectors = clf.support\_vectors\_

support\_x = [point[0] for point in support\_vectors]

support\_y = [point[1] for point in support\_vectors]

plt.scatter(support\_x, support\_y, marker='o', c='green', label='Support Vectors')

plt.legend()

w = clf.coef\_[0]

b = clf.intercept\_[0]

decision\_boundary = lambda x: (-w[0] \* x - b) / w[1]

print("Equation of the hyperplane: y = ", "{:.2f}x".format(-w[0]/w[1]), " + ", "{:.2f}".format(-b/w[1]))

x\_values = range(min(positive\_x + negative\_x) - 2, max(positive\_x + negative\_x) + 2)

plt.plot(x\_values, [decision\_boundary(x) for x in x\_values], label='Hyperplane')

plt.legend()

plt.grid(True)

plt.show()

import matplotlib.pyplot as plt

from sklearn import svm

positive\_x = [point[0] for point in positive\_data]

positive\_y = [point[1] for point in positive\_data]

negative\_x = [point[0] for point in negative\_data]

negative\_y = [point[1] for point in negative\_data]

plt.scatter(positive\_x, positive\_y, label='Positive')

plt.scatter(negative\_x, negative\_y, label='Negative')

plt.xlabel('X')

plt.ylabel('Y')

plt.title('Data Points for SVM Classification')

plt.legend()

clf = svm.SVC(kernel='linear')

clf.fit(positive\_data + negative\_data, [1] \* len(positive\_data) + [-1] \* len(negative\_data))

support\_vectors = clf.support\_vectors\_

support\_x = [point[0] for point in support\_vectors]

support\_y = [point[1] for point in support\_vectors]

plt.scatter(support\_x, support\_y, marker='o', c='violet', label='Support Vectors')

plt.legend()

w = clf.coef\_[0]

b = clf.intercept\_[0]

decision\_boundary = lambda x: (-w[0] \* x - b) / w[1]

print("Equation of the hyperplane: y = ", "{:.2f}x".format(-w[0]/w[1]), " + ", "{:.2f}".format(-b/w[1]))

x\_values = range(min(positive\_x + negative\_x) - 2, max(positive\_x + negative\_x) + 2)

plt.plot(x\_values, [decision\_boundary(x) for x in x\_values], label='Hyperplane')

plt.legend()

plt.ylim([-2, 2])

plt.grid(True)

plt.show()

decision\_boundary = lambda x: (-w[0] \* x - b) / w[1]

import numpy as np

import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

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

# Load your dataset

# Replace 'your\_dataset.csv' with the path to your dataset file

dataset = pd.read\_csv

import numpy as np

import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

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

# Load your dataset

# Replace '' with the path to your dataset file

dataset = pd.read\_csv("D:\Users\sfl22\Downloads\iris.csv")

# Assuming your dataset has features in columns 0 to n-2 and labels in column n-1

X = dataset.iloc[:, :-1].values # Features

y = dataset.iloc[:, -1].values # Labels

# 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)

# Create Decision Tree model

dt\_model = DecisionTreeClassifier()

# Train the model

dt\_model.fit(X\_train, y\_train)

# Predict the outputs for the test set

predictions = dt\_model.predict(X\_test)

# Evaluate the model

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

print("Accuracy:", accuracy)csv')

# Assuming your dataset has features in columns 0 to n-2 and labels in column n-1

X = dataset.iloc[:, :-1].values # Features

y = dataset.iloc[:, -1].values # Labels

# 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)

# Create Decision Tree model

dt\_model = DecisionTreeClassifier()

# Train the model

dt\_model.fit(X\_train, y\_train)

# Predict the outputs for the test set

predictions = dt\_model.predict(X\_test)

# Evaluate the model

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

print("Accuracy:", accuracy)

import numpy as np

import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

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

# Load your dataset

# Replace 'your\_dataset.csv' with the path to your dataset file

dataset = pd.read\_csv("D:\\Users\\sfl22\\Downloads\\iris.csv")

# Assuming your dataset has features in columns 0 to n-2 and labels in column n-1

X = dataset.iloc[:, :-1].values # Features

y = dataset.iloc[:, -1].values # Labels

# 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)

# Create Decision Tree model

dt\_model = DecisionTreeClassifier()

# Train the model

dt\_model.fit(X\_train, y\_train)

# Predict the outputs for the test set

predictions = dt\_model.predict(X\_test)

# Evaluate the model

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

print("Accuracy:", accuracy)

import pandas as pd

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

from sklearn.preprocessing import StandardScaler

# Load the dataset

df = pd.read\_csv("D:\\Users\\sfl22\\Downloads\\iris.csv")

# Preprocess the data

X = df.iloc[:, :4].values

y = df.iloc[:, 4].values

# Split the data into training and testing sets

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

# Scale the data

scaler = StandardScaler()

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

X\_test = scaler.transform(X\_test)

# Preprocess the new data point

new\_data = np.array([[5.2, 3.1, 1.4, 0.2]])

new\_data\_scaled = scaler.transform(new\_data)

import pandas as pd

import numpy as np

from sklearn.tree import DecisionTreeClassifier

# Load the dataset

dataset = pd.read\_csv("D:\\Users\\sfl22\\Downloads\\iris.csv")

# Assuming your dataset has features in columns 0 to n-2 and labels in column n-1

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

# Create Decision Tree model

clf = DecisionTreeClassifier()

# Train the model

clf.fit(X, y)

# Predict the new species

new\_species\_df = clf.predict\_proba(np.array([[5.2, 3.1, 1.4, 0.2]]))

new\_species\_class\_index = np.argmax(new\_species\_df)

new\_species\_decision\_function = new\_species\_df[0, new\_species\_class\_index]

# Print the prediction

print("Decision Tree prediction: ", clf.classes\_[new\_species\_class\_index], ", decision function: ", round(new\_species\_decision\_function, 3))

import pandas as pd

import numpy as np

import os

import matplotlib.pyplot as plt

import seaborn as sns

df = pd.read\_csv("D:\\Users\\sfl22\\Downloads\\iris.csv")

df.head()

df.describe()

n\_train = 150

n\_test = 1000

noise = 0.1

def f(x):

x = x.ravel()

return np.exp(-(x \*\* 2)) + 1.5 \* np.exp(-((x - 2) \*\* 2))

def generate(n\_samples, noise):

X = np.random.rand(n\_samples) \* 10 - 5

X = np.sort(X).ravel()

y = (

np.exp(-(X \*\* 2))

+ 1.5 \* np.exp(-((X - 2) \*\* 2))

+ np.random.normal(0.0, noise, n\_samples)

)

X = X.reshape((n\_samples, 1))

return X, y

X\_train, y\_train = generate(n\_samples=n\_train, noise=noise)

X\_test, y\_test = generate(n\_samples=n\_test, noise=noise)

from sklearn.tree import DecisionTreeRegressor

reg\_tree = DecisionTreeRegressor(max\_depth=5, random\_state=17)

reg\_tree.fit(X\_train, y\_train)

reg\_tree\_pred = reg\_tree.predict(X\_test)

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

plt.plot(X\_test, f(X\_test), "b")

plt.scatter(X\_train, y\_train, c="b", s=20)

plt.plot(X\_test, reg\_tree\_pred, "g", lw=2)

plt.xlim([-5, 5])

plt.title(

"Decision tree regressor, MSE = %.2f"

% (np.sum((y\_test - reg\_tree\_pred) \*\* 2) / n\_test)

)

plt.show()

(

tree=tree\_grid.best\_estimator\_,

feature\_names=df.columns,

png\_file\_to\_save="iris.csv",

)

# Import necessary libraries

import pandas as pd

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import classification\_report

# Load the dataset

data = pd.read\_csv("D:\\Users\\sfl22\\Downloads\\iris.csv")

# Define features and target variable

X = data.iloc[:, :-1] # Features: sepal length, sepal width, petallength, petal width

y = data.iloc[:, -1] # Target variable: Ground truth class name

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,

test\_size=0.2, random\_state=42)

# Implement Decision Tree Classifier

dt\_classifier = DecisionTreeClassifier()

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

# Implement KNN Classifier

knn\_classifier = KNeighborsClassifier()

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

# Predict the species of a new flower with given attributes

new\_flower = [[5.2, 3.1, 1.4, 0.2]]

dt\_prediction = dt\_classifier.predict(new\_flower)

knn\_prediction = knn\_classifier.predict(new\_flower)

# Output the predicted species

print("\nDecision Tree Prediction:", dt\_prediction)

print("KNN Prediction:", knn\_prediction)

# Calculate accuracy for Decision Tree

dt\_y\_pred = dt\_classifier.predict(X\_test)

dt\_accuracy = accuracy\_score(y\_test, dt\_y\_pred)

print("\nDecision Tree Accuracy:", dt\_accuracy)

# # Calculate confusion matrix for Decision Tree

# dt\_conf\_matrix = confusion\_matrix(y\_test, dt\_y\_pred)

# print("Decision Tree Confusion Matrix:")

# print(dt\_conf\_matrix)

# Calculate accuracy for KNN

knn\_y\_pred = knn\_classifier.predict(X\_test)

knn\_accuracy = accuracy\_score(y\_test, knn\_y\_pred)

print("\nKNN Accuracy:", knn\_accuracy)

# # Calculate confusion matrix for KNN

# knn\_conf\_matrix = confusion\_matrix(y\_test, knn\_y\_pred)

# print("KNN Confusion Matrix:")

# print(knn\_conf\_matrix)

# Generate classification report for Decision Tree

dt\_y\_pred = dt\_classifier.predict(X\_test)

dt\_classification\_report = classification\_report(y\_test, dt\_y\_pred)

print("\nDecision Tree Classification Report:")

print(dt\_classification\_report)

# Generate classification report for KNN

knn\_y\_pred = knn\_classifier.predict(X\_test)

knn\_classification\_report = classification\_report(y\_test, knn\_y\_pred)

print("\nKNN Classification Report:")

print(knn\_classification\_report)

# Visualize the Decision Tree as a tree structure

# Visualize the Decision Tree as a tree structure

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

plot\_tree(dt\_classifier,feature\_names=X.columns, class\_names=['Setosa', 'Versicolour','Virginica'], filled=True)

plt.show()

**DECISION TREE**

# Import necessary libraries

import pandas as pd

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import classification\_report

# Load the dataset

data = pd.read\_csv('iris.csv')

# Define features and target variable

X = data.iloc[:, :-1] # Features: sepal length, sepal width, petal length, petal width

y = data.iloc[:, -1] # Target variable: Ground truth class name

# Split the data into training and testing sets

X\_train, X\_test, y\_traihttp://localhost:8888/notebooks/decision%20knn.ipynb#n, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Implement Decision Tree Classifier

dt\_classifier = DecisionTreeClassifier()

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

# Implement KNN Classifier

knn\_classifier = KNeighborsClassifier()

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

# Predict the species of a new flower with given attributes

new\_flower = [[5.2, 3.1, 1.4, 0.2]]

dt\_prediction = dt\_classifier.predict(new\_flower)

knn\_prediction = knn\_classifier.predict(new\_flower)

# Output the predicted species

print("\nDecision Tree Prediction:", dt\_prediction)

print("KNN Prediction:", knn\_prediction)

# Calculate accuracy for Decision Tree

dt\_y\_pred = dt\_classifier.predict(X\_test)

dt\_accuracy = accuracy\_score(y\_test, dt\_y\_pred)

print("\nDecision Tree Accuracy:", dt\_accuracy)

# Calculate confusion matrix for Decision Tree

dt\_conf\_matrix = confusion\_matrix(y\_test, dt\_y\_pred)

print("Decision Tree Confusion Matrix:")

print(dt\_conf\_matrix)

# Calculate accuracy for KNN

knn\_y\_pred = knn\_classifier.predict(X\_test)

knn\_accuracy = accuracy\_score(y\_test, knn\_y\_pred)

print("\nKNN Accuracy:", knn\_accuracy)

# Calculate confusion matrix for KNN

knn\_conf\_matrix = confusion\_matrix(y\_test, knn\_y\_pred)

print("KNN Confusion Matrix:")

print(knn\_conf\_matrix)

# Generate classification report for Decision Tree

dt\_y\_pred = dt\_classifier.predict(X\_test)

dt\_classification\_report = classification\_report(y\_test, dt\_y\_pred)

print("\nDecision Tree Classification Report:")

print(dt\_classification\_report)

# Generate classification report for KNN

knn\_y\_pred = knn\_classifier.predict(X\_test)

knn\_classification\_report = classification\_report(y\_test, knn\_y\_pred)

print("\nKNN Classification Report:")

print(knn\_classification\_report)

# Visualize the Decision Tree as a tree structure

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

plot\_tree(dt\_classifier, feature\_names=X.columns.tolist(), class\_names=['Setosa', 'Versicolour', 'Virginica'], filled=True)

plt.show()

import matplotlib.pyplot as plt

# Predicted species from Decision Tree and KNN

species = ['setosa']

counts = [1] # Assuming only one prediction for simplicity

# Create a pie chart

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

plt.pie(counts, labels=species, autopct='%1.1f%%', startangle=140)

plt.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.

# Add a title

plt.title('Predicted Species of the New Flower')

# Display the plot

plt.show()

**KNN**

# Setup

import numpy as np

from sklearn import datasets

from sklearn import neighbors

import pylab as pl

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

iris = datasets.load\_iris()

print(iris.keys())

n\_samples, n\_features = iris.data.shape

print((n\_samples, n\_features))

print(iris.data[0])

print(iris.target.shape)

print(iris.target)

print(iris.target\_names)

x\_index = 0

y\_index = 1

# this formatter will label the colorbar with the correct target names

formatter = plt.FuncFormatter(lambda i, \*args: iris.target\_names[int(i)])

plt.scatter(iris.data[:, x\_index], iris.data[:, y\_index],

c=iris.target, cmap=plt.cm.get\_cmap('RdYlBu', 3))

plt.colorbar(ticks=[0, 1, 2], format=formatter)

plt.clim(-0.5, 2.5)

plt.xlabel(iris.feature\_names[x\_index])

plt.ylabel(iris.feature\_names[y\_index]);

X, y = iris.data, iris.target

clf = neighbors.KNeighborsClassifier(n\_neighbors=5)

clf.fit(X, y)

result = clf.predict([[3, 5, 4, 2],])

print(iris.target\_names[result])

clf.predict\_proba([[3, 5, 4, 2],])

cmap\_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])

cmap\_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

def plot\_iris\_knn():

iris = datasets.load\_iris()

X = iris.data[:, :2] # we only take the first two features.

y = iris.target

knn = neighbors.KNeighborsClassifier(n\_neighbors=3)

knn.fit(X, y)

x\_min, x\_max = X[:, 0].min() - .1, X[:, 0].max() + .1

y\_min, y\_max = X[:, 1].min() - .1, X[:, 1].max() + .1

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100),

np.linspace(y\_min, y\_max, 100))

Z = knn.predict(np.c\_[xx.ravel(), yy.ravel()])

# Put the result into a color plot

Z = Z.reshape(xx.shape)

pl.figure()

pl.pcolormesh(xx, yy, Z, cmap=cmap\_light)

# Plot also the training points

pl.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap\_bold)

pl.xlabel('sepal length (cm)')

pl.ylabel('sepal width (cm)')

pl.axis('tight')

plot\_iris\_knn()

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

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

from sklearn.neighbors import KNeighborsClassifier

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

# Load the Iris dataset

iris = load\_iris()

X, y = iris.data, iris.target

# Split data into train and test sets

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

# Create and fit the KNN classifier

k = 3 # You can choose any value of k

knn = KNeighborsClassifier(n\_neighbors=k)

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

# Make predictions

y\_pred = knn.predict(X\_test)

# Calculate evaluation metrics

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

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

precision = precision\_score(y\_test, y\_pred, average='weighted')

recall = recall\_score(y\_test, y\_pred, average='weighted')

# Print the results

print(f"Confusion Matrix:\n{conf\_matrix}")

print(f"Accuracy: {accuracy:.2f}")

print(f"Precision: {precision:.2f}")

print(f"Recall: {recall:.2f}")

# Plot the confusion matrix

plt.figure(figsize=(6, 4))

plt.imshow(conf\_matrix, cmap='Blues', interpolation='nearest')

plt.title("Confusion Matrix")

plt.colorbar()

plt.xticks(np.arange(3), iris.target\_names, rotation=45)

plt.yticks(np.arange(3), iris.target\_names)

plt.xlabel("Predicted Class")

plt.ylabel("True Class")

plt.show()

**Kmeans**

import numpy as np

from sklearn.cluster import KMeans

# Generate 5 random data points (you can replace these with your own data)

data\_points = np.array([[1, 1], [2, 1], [4, 3], [5, 4], [3, 2]])

# Initialize the KMeans model with the desired number of clusters (e.g., 2)

num\_clusters = 2

kmeans = KMeans(n\_clusters=num\_clusters)

# Fit the model to the data

kmeans.fit(data\_points)

# Get the cluster centers and labels

cluster\_centers = kmeans.cluster\_centers\_

labels = kmeans.labels\_

print(f"Cluster centers: {cluster\_centers}")

print(f"Labels for each data point: {labels}")

import matplotlib.pyplot as plt

# Plot the data points

plt.scatter(data\_points[:, 0], data\_points[:, 1], c=labels, cmap='viridis', marker='o', s=100)

plt.scatter(cluster\_centers[:, 0], cluster\_centers[:, 1], c='red', marker='x', s=200, label='Cluster Centers')

# Add labels and title

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('K-Means Clustering')

# Add legend

plt.legend()

# Show the plot

plt.show()

**PCA**

import matplotlib.pyplot as plt

from sklearn import svm

import numpy as np # Import NumPy for array creation

# Define some data points

data = np.array([[-1, 1], [-1, -1], [1, 1], [1, -1]]) # Use NumPy array

labels = [1, 1, -1, -1]

# Rest of the code (visualization, training, prediction) remains the same

import numpy as np

import matplotlib.pyplot as plt

def pca(data):

"""

Performs Principal Component Analysis on a given dataset.

Args:

data: A NumPy array representing the dataset.

Returns:

A tuple containing:

eigenvalues: A NumPy array of eigenvalues.

eigenvectors: A NumPy array of eigenvectors.

transformed\_data\_single: Transformed data using the first eigenvector.

transformed\_data\_all: Transformed data using all eigenvectors.

"""

data\_centered = data - np.mean(data, axis=0)

# Print centered data

print("Adjusted Data:")

print(data\_centered)

# Calculate the covariance matrix

covariance = np.cov(data\_centered.T)

# Find eigenvalues and eigenvectors

eigenvalues, eigenvectors = np.linalg.eig(covariance)

# Sort eigenvalues and eigenvectors in descending order by eigenvalue

idx = eigenvalues.argsort()[::-1]

eigenvalues = eigenvalues[idx]

eigenvectors = eigenvectors[:, idx]

# Transform the data using the first eigenvector

transformed\_data\_single = data\_centered.dot(eigenvectors[:, 0:1])

# Transform the data using all eigenvectors

transformed\_data\_all = data\_centered.dot(eigenvectors)

return eigenvalues, eigenvectors, transformed\_data\_single, transformed\_data\_all, data\_centered

# Replace with your actual raw dataset

raw\_data = np.array([

[2.5, 2.4],

[0.5, 0.7],

[2.2, 2.9],

[1.9, 2.2],

[3.1, 3.0],

[2.3, 2.7],

[2.0, 1.6],

[1.0, 1.1],

[1.5, 1.6],

[1.1, 0.9],

])

# Perform PCA

eigenvalues, eigenvectors, transformed\_data\_single, transformed\_data\_all, data\_centered = pca(raw\_data)

# Print results (similar to solution)

print("\nEigenvalues:", eigenvalues,"\n")

print("Eigenvectors:", eigenvectors,"\n")

# Transformed Data (Single eigenvector)

print("Transformed Data (Single eigenvector):")

print(transformed\_data\_single.flatten())

print("\n")

# Final Data (using all eigenvectors, for comparison)

print("Transformed Data (All eigenvectors):")

print(transformed\_data\_all)

import numpy as np

class PCA:

def \_\_init\_\_(self, n\_components):

self.n\_components = n\_components

self.components = None

self.mean = None

def fit(self, X):

# Mean centering

self.mean = np.mean(X, axis=0)

X = X - self.mean

# Compute covariance matrix

cov\_matrix = np.cov(X.T)

# Compute eigenvectors and eigenvalues

eigenvalues, eigenvectors = np.linalg.eig(cov\_matrix)

# Sort eigenvalues and corresponding eigenvectors in descending order

idx = np.argsort(eigenvalues)[::-1]

eigenvectors = eigenvectors[:, idx]

# Select top n\_components eigenvectors

self.components = eigenvectors[:, :self.n\_components]

def transform(self, X):

# Mean centering

X = X - self.mean

return np.dot(X, self.components)

def inverse\_transform(self, X):

return np.dot(X, self.components.T) + self.mean

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Example dataset

X = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Number of principal components

n\_components = 2

# Perform PCA

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

pca.fit(X)

X\_pca = pca.transform(X)

X\_original = pca.inverse\_transform(X\_pca)

print("Original Data:")

print(X)

print("\nTransformed Data (Principal Components):")

print(X\_pca)

print("\nReconstructed Data:")

print(X\_original)

plt.scatter(data\_centered[:, 0], data\_centered[:, 1], alpha=0.7, label='Standardized Data')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('Standadized Data Visualization')

plt.legend()

plt.show()

plt.scatter(data\_centered[:, 0], data\_centered[:, 1], alpha=0.7, label='Centered Data')

plt.xlim(-1.0, 1.5)

plt.ylim(-0.5, 2.5)

plt.grid(True)

plt.gca().set\_aspect('equal')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.title('Centered Data')

# Legend

plt.legend()

# Show the plot

plt.show()

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

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

from sklearn.decomposition import PCA

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix

from matplotlib.colors import ListedColormap

dataset = pd.read\_csv('Wine.csv')

# distributing the dataset into two components X and Y

X = dataset.iloc[:, 0:13].values

y = dataset.iloc[:, 13].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state = 0)

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

pca = PCA(n\_components = 2)

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

X\_test = pca.transform(X\_test)

explained\_variance = pca.explained\_variance\_ratio\_

classifier = LogisticRegression(random\_state = 0)

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

y\_pred = classifier.predict(X\_test)

cm = confusion\_matrix(y\_test, y\_pred)

from matplotlib.colors import ListedColormap

X\_set, y\_set = X\_train, y\_train

X1, X2 = np.meshgrid(np.arange(start = X\_set[:, 0].min() - 1,stop = X\_set[:, 0].max() + 1, step = 0.01),np.arange(start = X\_set[:, 1].min() - 1,stop = X\_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),X2.ravel()]).T).reshape(X1.shape), alpha = 0.75,cmap = ListedColormap(('yellow', 'white', 'aquamarine')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

plt.title('Logistic Regression (Training set)')

plt.xlabel('PC1') # for Xlabel

plt.ylabel('PC2') # for Ylabel

plt.legend() # to show legend

# show scatter plot

plt.show()

from matplotlib.colors import ListedColormap

X\_set, y\_set = X\_test, y\_test

X1, X2 = np.meshgrid(np.arange(start = X\_set[:, 0].min() - 1,stop = X\_set[:, 0].max() + 1, step = 0.01),np.arange(start = X\_set[:, 1].min() - 1,stop = X\_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),X2.ravel()]).T).reshape(X1.shape), alpha = 0.75,cmap = ListedColormap(('yellow', 'white', 'aquamarine')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

# title for scatter plot

plt.title('Logistic Regression (Test set)')

plt.xlabel('PC1') # for Xlabel

plt.ylabel('PC2') # for Ylabel

plt.legend()

# show scatter plot

plt.show()

**PREPROCESS**

import pandas as pd

f=pd.read\_csv('iris1.csv')

print(data.head())

data.head()

print(data)

f.to\_csv(r"D:\Users\sfl22\Downloads\dataset\y.csv")

print (data. isnull().sum())

from PIL import Image

image\_path ="RR.jpg"

image = Image.open(image\_path)

display(image)

image.save(r"D:\Users\sfl22\Downloads\dataset\rrr.jpg")

import pandas as pd

import numpy as np

from sklearn import preprocessing

import pandas as pd

df = pd.read\_csv('iris1.csv')

loan\_dataset= pd.read\_csv("train\_u6lujuX\_CVtuZ9i (1).csv")

loan\_dataset.head(10)

loan\_dataset.info()

loan\_dataset.describe().T

print(f'Shape of Loan Dataset before drop duplicated Row is: {loan\_dataset.shape}')

loan\_dataset = loan\_dataset.drop\_duplicates()

print(f'Shape of Loan Dataset After Drop Duplicated Row is: {loan\_dataset.shape}')

loan\_dataset = loan\_dataset.drop("Loan\_ID", axis=1)

categorical\_features = ['Gender', 'Married', 'Dependents', 'Education', 'Self\_Employed','Property\_Area','Credit\_History']

continuous\_features = ['ApplicantIncome', 'CoapplicantIncome', 'LoanAmount', 'Loan\_Amount\_Term']

X = loan\_dataset.drop(columns='Loan\_Status')

Y = loan\_dataset['Loan\_Status']

# Splitting into Train and Validation Sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, Y, test\_size=0.2,stratify=Y, shuffle=True, random\_state = 40)

# Splitting Validation Set into Validation and Test Sets

X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_val, y\_val, test\_size=0.5, stratify=y\_val, shuffle=True, random\_state = 40)

# Missing Value Analysis

def calculate\_missing\_values(X\_train, X\_val, X\_test):

Miss\_Train = X\_train.isna().sum()

Miss\_Val = X\_val.isna().sum()

Miss\_test = X\_test.isna().sum()

# Convert the series to dataframes

output\_train = pd.DataFrame(Miss\_Train, columns=['Missing Values X\_train'])

output\_val = pd.DataFrame(Miss\_Val, columns=['Missing Values X\_val'])

output\_test = pd.DataFrame(Miss\_test, columns=['Missing Values X\_test'])

# Concatenate the dataframes output\_train and output\_val

output = pd.concat([output\_train, output\_val,output\_test], axis=1, join='inner')

return output

output = calculate\_missing\_values(X\_train, X\_val, X\_test)

# Define a function to apply the desired styling

def color\_cell(value):

if value >=20 :

return 'background-color:#2e9ee8'

elif value >=10 and value <20 :

return 'background-color:#7ac1f0'

elif value >=1 and value <10 :

return 'background-color:#bdddf2'

return ''

# Apply the styling to the DataFrame

styled\_df = output.style.applymap(color\_cell)

styled\_df

def find\_rows\_with\_high\_null\_values(df):

threshold = 0.5

# Getting DataFrame Name

df\_name = [name for name in globals() if globals()[name] is df][0]

null\_threshold = int(threshold \* len(df.columns))

null\_rows = df[df.apply(lambda x: x.isnull().sum(), axis=1) >= null\_threshold]

num\_null\_rows = len(null\_rows)

print(f"The number of rows consisting of more than 50% missing values in {df\_name} is: {num\_null\_rows}")

# Example usage:

find\_rows\_with\_high\_null\_values(X\_train)

find\_rows\_with\_high\_null\_values(X\_val)

find\_rows\_with\_high\_null\_values(X\_test)

# calculate mean columns

def calculate\_mean(df, column):

mean = df[column].mean().round()

return mean

for col in ["LoanAmount","Loan\_Amount\_Term"]:

print(f'Mean {col} in Trainset is: {calculate\_mean(X\_train, col)}')

print(f'Mean {col} in Valset is: {calculate\_mean(X\_val, col)}')

print(f'Mean {col} in Testset is: {calculate\_mean(X\_test, col)}')

def fill\_missing\_values\_by\_mean(df, column):

for col in cols:

df[col].fillna(df[col].mean(), inplace=True)

return df

cols = ["LoanAmount","Loan\_Amount\_Term"]

X\_train = fill\_missing\_values\_by\_mean(X\_train, cols)

X\_val = fill\_missing\_values\_by\_mean(X\_val, cols)

X\_test = fill\_missing\_values\_by\_mean(X\_test, cols)

Isna\_cate = ['Gender', 'Married','Education', 'Dependents', 'Self\_Employed', 'Property\_Area', 'Credit\_History']

def Get\_Mode(df, cols):

list\_of\_most\_frequent = {}

for col in cols:

f = df[col].mode().iloc[0]

list\_of\_most\_frequent[col] = f

return pd.DataFrame(list\_of\_most\_frequent, index=['Most Frequent'

X\_train.isna().sum()

X\_val.isna().sum()

X\_test.isna().sum()

Nominal\_fetaures = ['Gender', 'Married','Education', 'Self\_Employed', 'Property\_Area']

# Encoding Categorical Features

encoder = {'0': 1/4, '1': 2/4, '2': 3/4, '3+': 4/4}

for df in [X\_train, X\_val, X\_test]:

df["Dependents"] = df['Dependents'].map(encoder)

X\_train.head(10)

# Convet target column to int

mapping = {'Y': 1,'N': 0}

y\_train = pd.Series(y\_train).map(mapping)

y\_val = pd.Series(y\_val).map(mapping)

y\_test = pd.Series(y\_test).map(mapping)

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

dataset= pd.read\_csv("Q1\_Nutrition\_Dataset.csv")

dataset

dataset.duplicated().sum()

df

dataset.isnull()

dataset.isnull().sum()

dataset=dataset.fillna(dataset.mean())

print(dataset.columns.tolist())

pip install numpy matplotlib

import numpy as np

import matplotlib.pyplot as plt

# Function to estimate coefficients

def estimate\_coef(x, y):

# Calculate the coefficients of the linear regression line

b\_1 = np.sum((x - np.mean(x)) \* (y - np.mean(y))) / np.sum((x - np.mean(x)) \*\* 2)

b\_0 = np.mean(y) - b\_1 \* np.mean(x)

return b\_0, b\_1

# Main function

def main():

# Observations / data

x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

# Estimating coefficients

b\_0, b\_1 = estimate\_coef(x, y)

print("Estimated coefficients:\nb\_0 = {}\nb\_1 = {}".format(b\_0, b\_1))

# Plotting the data points and the regression line

plt.scatter(x, y, color='red')

plt.plot(x, b\_0 + b\_1 \* x, color='blue')

plt.title('Simple Linear Regression')

plt.xlabel('x')

plt.ylabel('y')

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

import pandas as pd

import matplotlib.pyplot as plt

# Load the dataset from a CSV file using pandas

df = pd.read\_csv('Q1\_Nutrition\_Dataset.csv', delimiter=',', na\_values='?')

# Assuming the first column is the independent variable (x) and the second column is the dependent variable (y)

x = df.iloc[:, 0].values

y = df.iloc[:, 1].values

# Perform linear regression

coefficients = np.polyfit(x, y, 1)

# The coefficients variable now contains the slope and intercept of the line of best fit

slope, intercept = coefficients

# Plot the data points

plt.scatter(x, y, color='green')

# Plot the line of best fit

plt.plot(x, slope \* x + intercept, color='blue')

# Labels and title

plt.title('Linear Regression')

plt.xlabel('x')

plt.ylabel('y')

# Display the plot

plt.show()

**LINEAR REG**

import pandas as pd

import matplotlib.pyplot as plt

# Load the dataset from a CSV file using pandas

df = pd.read\_csv('Q1\_Nutrition\_Dataset.csv', delimiter=',')

# Assuming the first column is the independent variable (x) and the second column is the dependent variable (y)

x = df.iloc[:, 0].values

y = df.iloc[:, 1].values

# Perform linear regression

coefficients = np.polyfit(x, y, 1)

# The coefficients variable now contains the slope and intercept of the line of best fit

slope, intercept = coefficients

# Print the estimated coefficients

print("Estimated coefficients:\nb\_0 = {:.10f}\nb\_1 = {:.10f}".format(intercept, slope))

# Plot the data points

plt.scatter(x, y, color='GREEN')

# Plot the line of best fit

plt.plot(x, slope \* x + intercept, color='blue')

# Labels and title

plt.title('Linear Regression')

plt.xlabel('x')

plt.ylabel('y')

# Display the plot

plt.show()

**LOGISTIC REG**

**1.**

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.linear\_model import LogisticRegression

import numpy as np

# Load the dataset from a CSV file using pandas

df = pd.read\_csv('Book1.csv', delimiter=',')

# Assuming the first column is the independent variable (x) and the second column is the dependent variable (y)

x = df.iloc[:, 0].values.reshape(-1, 1) # Ensure x is reshaped to be compatible with the model's requirement

y = df.iloc[:, 1].values

# Create an instance of the LogisticRegression model

model = LogisticRegression()

# Fit the model to the data

model.fit(x, y)

# Create a mesh grid for plotting

x\_min, x\_max = x.min() - 1, x.max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.02), np.arange(-1, 2, 0.02))

Z = model.predict\_proba(xx.ravel().reshape(-1, 1))[:, 1] # Ensure the input matches the model's expectation

Z = Z.reshape(xx.shape)

# Plot the data points

plt.scatter(x, y, color='GREEN', label='Data Points')

# Plot the decision boundary

contours = plt.contour(xx, yy, Z, levels=[0.5], colors='blue', linestyles='--')

# Add labels to the contour plot

plt.clabel(contours, inline=True, fontsize=8, colors='blue')

# Add a legend

plt.legend()

# Labels and title

plt.title('Logistic Regression')

plt.xlabel('x')

plt.ylabel('y')

# Display the plot

plt.show()

class MyLogisticRegression():

def \_\_init\_\_(self, lr=1e-2, max\_iter=100):

self.lr = lr

self.max\_iter = max\_iter11

self.w = None

self.b = None

def fit(self, x, y):

self.w = 2\*np.random.random((1, x.shape[1])) - 1

self.b = 2\*np.random.random() - 1

for step in range(self.max\_iter):

z = np.dot(x, self.w.T) + self.b

y\_pred = self.\_\_sigmoid(z)

error = y - y\_pred

self.w = self.w + self.lr\*np.dot(error.T, x)

self.b = self.b + self.lr\*error.sum()

def predict(self, x):

return self.\_\_sigmoid(np.dot(x, self.w.T) + self.b)

def \_\_sigmoid(self, x):

return 1.0 / (1.0 + np.exp(-x))

np.random.seed(42)

clf\_sk = LogisticRegression(C=1e15, max\_iter=1000, solver='newton-cg')

clf\_sk.fit(x, y.ravel())

print('coef\_:', clf\_sk.coef\_)

print('intercept\_:', clf\_sk.intercept\_)

print('acurácia: {:.2f}%'.format(clf\_sk.score(x, y)\*100))

x\_test = np.linspace(x.min(), x.max(), 100).reshape(-1,1)

y\_pred\_sk = clf\_sk.predict\_proba(x\_test)

y\_pred = clf.predict(x\_test)

plt.scatter(x, y, c=y, cmap='bwr')

plt.plot(x\_test, y\_pred\_sk[:,1], color='blue', linewidth=7.0)

plt.plot(x\_test, y\_pred, color='green')

plt.xlabel('idade')

plt.ylabel('comprou?')

**2.**

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.linear\_model import LogisticRegression

# Load the dataset from a CSV file using pandas

df = pd.read\_csv('Q1\_Nutrition\_Dataset.csv', delimiter=',')

# Assuming the first column is the independent variable (x) and the second column is the dependent variable (y)

# For logistic regression, we need to ensure y is binary. If not, you might need to preprocess your data.

# This example assumes y is already binary.

x = df.iloc[:, 0].values.reshape(-1, 1) # Reshape to be compatible with the model's requirement

y = df.iloc[:, 1].values

# Create an instance of the LogisticRegression model

model = LogisticRegression()

# Fit the model to the data

model.fit(x, y)

# Print the estimated coefficients

print("Estimated coefficients:\nIntercept = {:.10f}\nSlope = {:.10f}".format(model.intercept\_[0], model.coef\_[0][0]))

# Plot the data points

plt.scatter(x, y, color='GREEN')

# Plot the decision boundary

# Since logistic regression outputs probabilities, we can plot the decision boundary as the line where the probability is 0.5

decision\_boundary = model.predict(x)

plt.plot(x, decision\_boundary, color='blue')

# Labels and title

plt.title('Logistic Regression')

plt.xlabel('x')

plt.ylabel('y')

# Display the plot

plt.show()

**3.**

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.linear\_model import LogisticRegression

import numpy as np

# Load the dataset from a CSV file using pandas

df = pd.read\_csv('Android\_Malware.csv', delimiter=',')

# Assuming the first column is the independent variable (x) and the second column is the dependent variable (y)

x = df.iloc[:, 0].values.reshape(-1, 1) # Ensure x is reshaped to be compatible with the model's requirement

y = df.iloc[:, 1].values

# Create an instance of the LogisticRegression model

model = LogisticRegression()

# Fit the model to the data

model.fit(x, y)

# Create a mesh grid for plotting

x\_min, x\_max = x.min() - 1, x.max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.02), np.arange(-1, 2, 0.02))

Z = model.predict\_proba(xx.ravel().reshape(-1, 1))[:, 1] # Ensure the input matches the model's expectation

Z = Z.reshape(xx.shape)

# Plot the data points

plt.scatter(x, y, color='GREEN', label='Data Points')

# Plot the decision boundary

contours = plt.contour(xx, yy, Z, levels=[0.5], colors='blue', linestyles='--')

# Add labels to the contour plot

plt.clabel(contours, inline=True, fontsize=8, colors='blue')

# Add a legend

plt.legend()

# Labels and title

plt.title('Logistic Regression')

plt.xlabel('x')

plt.ylabel('y')

# Display the plot

plt.show()

**mean**

df['col\_name']=df['col\_name'].fillna(df['col\_name'].mean().astype(int))