VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Machine Learning (23CS6PCMAL)

Submitted by

Preeti T Korishettar (1BM22CS208)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Machine Learning (23CS6PCMAL)" carried out by **Preeti T Korishettar (1BM22CS208),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Machine Learning (23CS6PCMAL) work prescribed for the said degree.

Lab Faculty Incharge Name: **Ms. Saritha A N** Assistant Professor Department of CSE, BMSCE	 **Dr. Kavitha Sooda** Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	21-2-2025	Write a python program to import and export data using Pandas library functions	
2	3-3-2025	Demonstrate various data pre-processing techniques for a given dataset	
3	10-3-2025	Implement Linear and Multi-Linear Regression algorithm using appropriate dataset	
4	17-3-2025	Build Logistic Regression Model for a given dataset	
5	24-3-2025	Use an appropriate data set for building the decision tree (ID3) and apply this knowledge to classify a new sample	
6	7-4-2025	Build KNN Classification model for a given dataset	
7	21-4-2025	Build Support vector machine model for a given dataset	
8	5-5-2025	Implement Random forest ensemble method on a given dataset	
9	5-5-2025	Implement Boosting ensemble method on a given dataset	
10	12-5-2025	Build k-Means algorithm to cluster a set of data stored in a .CSV file	
11	12-5-2025	Implement Dimensionality reduction using Principal Component Analysis (PCA) method	

Github Link: [Preeti_T_Korishettar]

Program 1

Write a python program to import and export data using Pandas library functions

PAGE NO:
DATE: 05/3/25

LAB - 0

Method 1:

Creating and initializing Data in dataframe directly

code:

```
import pandas as pd.
data = {
        "Name" : ["Preeti", "Prajval", "Prem"],
        "USN" : ["1BM22CS208", "1BM22CS220",
                 "1BM22CS341"],
        "Grade" : ["O", "S", "A+"]
        }
df = pd.DataFrame(data, index=[1, 2, 3])
df
```

output:

	Name	USN	Grade
1	Preeti	1BM22CS208	O
2	Prajval	1BM22CS220	S
3	Prem	1BM22CS341	A+

Method 2:

Import dataset from sklearn datasets

code:

```
from sklearn.datasets import load_iris
iris = load_iris()
df = pd.DataFrame(iris.data, columns = iris.feature_
                                        names)
```

PAGE NO :
DATE :

```
df['target'] = iris.target_names[iris.target]
df.head()
```

output:

	sepal length (cm)	sepal width (cm)	petal length (cm)
0	5.1	3.5	1.4
1	4.9	3.0	1.4
2	4.7	3.2	1.3
3	4.6	3.1	1.5
4	5.0	3.6	1.4

petal width (cm)	target
0.2	setosa
0.2	setosa
0.2	setosa
0.2	setosa
0.2	setosa.

Method 3:

Importing dataset in .csv form.

code:

```
file_path = '/content/sample_data/california_housing_train.csv'
df = pd.read_csv(file_path)
df.head()
```

PAGE NO :
DATE :

output:

	longitude	lat	housing_age	total_rooms	total_bedr	population
0	-114.31	34.19	15.0	5612.0	1283.0	1015.0
	⋮	⋮	⋮	⋮	⋮	⋮

Method 4:

Importing dataset from platforms like UCI, Kaggle etc.

code:

```
df = pd.read_csv('/content/Dataset_of_Diabetes')
print(df.head())
```

output:

	ID	No_Pation	Gender	AGE	Urea	Cr	HbA1c
0	502	17975	F	50	4.7	46	4.9
	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Yahoo Finance api

```
import yfinance as yf
import pandas as pd
import matplotlib.pyplot as plt

tickers = ["Reliance.NS", "TCS.NS", "INFY.NS"]
data = yf.download(tickers, start='2022-10-01',
                   end='2023-10-01', group_by='ticker')
```

```
print(data.head())
print(data.shape)
print(data.columns)
reliance_data = data['RELIANCE.NS']
print(reliance_data.describe())
reliance_data['Daily Return'] = reliance_data['close'].pct_change()
reliance_data['Dai
plt.figure(fig.size = (12,6))
plt.subplot(2,1,1)
plt.subplot(2,1,2)
plt.tight_layout()
plt.show()
reliance_data.to_csv('reliance_stock_data.csv')
```

3/3/25

Code:

```
import pandas as pd

data={

    "Name":["Preeti","Prajval","Prem"],

    "USN":["1BM22CS208","1BM22CS220","1BM22CS341"],

    "Grade":["O","S","A+"]

}

df=pd.DataFrame(data,index=[1,2,3])

from sklearn.datasets import load_iris

iris=load_iris()

df=pd.DataFrame(iris.data,columns=iris.feature_names)
```

```
df['target']=iris.target_names[iris.target]

df.head()

file_path='/content/sample_data/california_housing_train.csv'

df=pd.read_csv(file_path)

df.head()

df = pd.read_csv('/content/Dataset of Diabetes .csv')

print("Sample data:")

df.head()

import yfinance as yf

import pandas as pd

import matplotlib.pyplot as plt

tickers = ["RELIANCE.NS", "TCS.NS", "INFY.NS"]

data = yf.download(tickers, start="2022-10-01", end="2023-10-01",group_by='ticker')

print("First 5 rows of the dataset:")

print(data.head())

print("\nShape of the dataset:")

print(data.shape)

print("\nColumn names:")

print(data.columns)

reliance_data = data['RELIANCE.NS']

print("\nSummary statistics for Reliance Industries:")

print(reliance_data.describe())

reliance_data['Daily Return'] = reliance_data['Close'].pct_change()

plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)

reliance_data['Close'].plot(title="Reliance Industries - Closing Price")
```

```
plt.subplot(2, 1, 2)

reliance_data['Daily Return'].plot(title="Reliance Industries - Daily Returns",
color='orange')

plt.tight_layout()

plt.show()

reliance_data.to_csv('reliance_stock_data.csv')

print("\nReliance stock data saved to 'reliance_stock_data.csv'.")
```

Program 2

Demonstrate various data pre-processing techniques for a given dataset

10/3/2025

PAGE NO :
DATE :

Data Pre-processing is the transformations applied to data before feeding into algorithm.

1. Data cleaning: Handling Missing values,
   Handling categorical data,
   handling Outliers.

2. Data Transformation: Min-max scaler,
   standard scaler.

Implementation:

```
import pandas as pd.
# loading.
df = pd.read_csv('housing.csv')

# display information
df.head()  df.info()

# display statistical information
df.describe()

# display count of unique labels for 'Ocean Proximity'
df.count (returns df[Ocean Proximity]). unique()
df['Ocean_proximity'].value_count()
# Display which attributes in dataset have missing value.
df.isnull.sum()
```

PAGE NO :
DATE :

1. Adult Income Dataset (adult.csv)
   - No columns had missing values

Diabetes Dataset
   - No columns had missing values.

Handling Method : Since no missing values was found, no imputation was performed.

2. Which categorical columns did you identify in the dataset? How did you encode them?

Adult Income Dataset (adult.csv)
   - Categorical columns:
     workclass, education, marital-status, occupation, relationship, race, gender, native-country, income

> Encoding Method : One-hot encoding was applied using pd.get_dummies (df, drop_first = True), which converted these categorical columns into numerical format.

Diabetes Dataset

   - Categorial columns :
     Gender, class

Encoding : One-hot encoding was applied using pd.get_dummies (df, drop_first = True) to transform them into numeric format.

PAGE NO :
DATE :

3. What is difference b/w Min-Max scaling and standard scaling? when would you use one over the other.

Feature
Definition Scales values b/w a fixed range.
Formula:
Effect
when to use

MinMax Scaling:

Scales values b/w a fixed range.
~~Xscaled = Xmax - Xmin~~
x_std = (x - x.min) / (x.max - x.min)
x_scaled = x_std * (max - min) + min

Preserves original data distribution but scales it within a limited range.

when preserving relationships and original range of data is important.

Standard scaling

centers data around mean with a standard deviation of 1

z = (x - u) / s
changes data distribution to have zero mean and unit variance.

PAGE NO :
DATE :

when data follows a normal distribution or contains outliers.

- Min - max
  → you need data to be within fixed range
  → the dataset does not contain extreme outliers.

- standard scale
  → the dataset has varying units and a Gaussian
  → There are significant outliers, as standardization is less sensitive to them.

10/3/25

Code:

```
import pandas as pd
df=pd.read_csv('/content/housing.csv.zip')
df.info()
df.describe()
df['ocean_proximity'].value_counts()
missing_values=df.isnull().sum()
print(missing_values[missing_values>0])
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler
# Load the datasets
adult_df = pd.read_csv('/content/adult.csv.zip')
diabetes_df = pd.read_csv('/content/Dataset of Diabetes .csv')
# Function to handle missing values
def handle_missing_values(df):
    df = df.fillna(df.median(numeric_only=True))  # Fill numeric NaNs with median
    df = df.fillna(df.mode().iloc[0])  # Fill categorical NaNs with mode
    return df
```

```
# Function to handle categorical data
def encode_categorical(df):
    df = pd.get_dummies(df, drop_first=True)  # One-hot encoding
    return df
# Function to handle outliers using IQR method
def remove_outliers(df):
    numeric_cols = df.select_dtypes(include=[np.number]).columns
    for col in numeric_cols:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        df[col] = np.where((df[col] < lower_bound) | (df[col] > upper_bound),
np.nan, df[col])
    df = handle_missing_values(df)  # Refill outlier values
    return df
# Function to apply normalization and standardization
def apply_scaling(df):
    scaler_minmax = MinMaxScaler()
    scaler_standard = StandardScaler()
    numeric_cols = df.select_dtypes(include=[np.number]).columns

    df[numeric_cols] = scaler_minmax.fit_transform(df[numeric_cols])  # Min-Max
Scaling
    df[numeric_cols] = scaler_standard.fit_transform(df[numeric_cols])  #
Standardization
    return df
# Apply preprocessing to both datasets
adult_df = handle_missing_values(adult_df)
adult_df = encode_categorical(adult_df)
adult_df = remove_outliers(adult_df)
adult_df = apply_scaling(adult_df)
diabetes_df = handle_missing_values(diabetes_df)
diabetes_df = encode_categorical(diabetes_df)
diabetes_df = remove_outliers(diabetes_df)
diabetes_df = apply_scaling(diabetes_df)
# Create the directory to save the preprocessed datasets
!mkdir -p /mnt/data
```

```
# Save the preprocessed datasets
adult_df.to_csv('/mnt/data/adult_preprocessed.csv', index=False)
diabetes_df.to_csv('/mnt/data/diabetes_preprocessed.csv', index=False)
print("Preprocessing completed. Processed files saved as 'adult_preprocessed.csv'
and 'diabetes_preprocessed.csv'")
```

Program 3

Implement Linear and Multi-Linear Regression algorithm using appropriate dataset

PAGE NO :
DATE : 24/3/23

LAB - 3

LINEAR REGRESSION

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

x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])

x_mean = np.mean(x)
y_mean = np.mean(y)

numerator = np.sum((x - x_mean) * (y - y_mean))
denominator = np.sum((x - x_mean) ** 2)
b1 = numerator / denominator
b0 = y_mean - (b1 * x_mean)

print(f" Linear Regression Equation : y = {b0:.2f}
        + {b1:.2f} x")

y_pred = b0 + b1 * x

plt.scatter(x, y, color = 'blue', label = 'Data
        Points')
plt.plot(x, y_pred, color = 'red', label = 'Regression
        Line')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Linear Regression')
plt.legend()
plt.show()
```

Code:

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

# Sample Dataset: Hours studied vs Marks scored
X = np.array([1, 2, 3, 4, 5, 6])  # Independent variable
Y = np.array([40, 45, 50, 55, 60, 65])  # Dependent variable

# Initialize parameters
m = 0  # slope
c = 0  # intercept
alpha = 0.01  # learning rate
epochs = 1000  # number of iterations
n = len(X)

# Gradient Descent
for i in range(epochs):
    Y_pred = m * X + c
    error = Y_pred - Y

    dm = (2/n) * np.sum(error * X)
    dc = (2/n) * np.sum(error)

    m -= alpha * dm
    c -= alpha * dc

# Final values
print(f"Slope (m): {m}")
print(f"Intercept (c): {c}")

# Prediction function
def predict(x):
    return m * x + c

# Plotting
plt.scatter(X, Y, color='blue', label='Actual Data')
```

```
plt.plot(X, predict(X), color='red', label='Best Fit Line')
plt.xlabel("Hours Studied")
plt.ylabel("Marks Scored")
plt.title("Linear Regression from Scratch")
plt.legend()
plt.grid()
plt.show()

# Predicting for new value
hours = 7
print(f"Predicted Marks for {hours} hours studied: {predict(hours):.2f}")


# MULTI-LINEAR REGRESSION

import numpy as np

# Dataset: [Area, Bedrooms] -> Price
X = np.array([
    [1000, 2],
    [1200, 3],
    [1500, 3],
    [1800, 4],
    [2000, 4]
])
Y = np.array([300000, 350000, 400000, 450000, 500000])  # Prices

# Normalize features for better training
X = X / np.max(X, axis=0)
Y = Y / 1000000  # Normalize to millions

n_samples, n_features = X.shape

# Initialize weights and bias
W = np.zeros(n_features)  # weights
b = 0  # bias
alpha = 0.01
epochs = 1000

# Gradient Descent
for i in range(epochs):
    Y_pred = np.dot(X, W) + b
    error = Y_pred - Y

    dW = (2 / n_samples) * np.dot(X.T, error)
    db = (2 / n_samples) * np.sum(error)

    W -= alpha * dW
```

```
    b -= alpha * db

# Final model
print(f"Weights: {W}")
print(f"Bias: {b}")

# Prediction function
def predict(area, bedrooms):
    area_norm = area / 2000
    bed_norm = bedrooms / 4
    return (W[0] * area_norm + W[1] * bed_norm + b) * 1000000

# Predicting for a new house
area = 1500
bedrooms = 3
print(f"Predicted price for {area} sq ft and {bedrooms} BHK: ₹{predict(area, bedrooms):.2f}")
```

Program 4

Build Logistic Regression Model for a given dataset

LOGISTIC REGRESSION

```
import numpy as np.

class LogisticRegression:
    def __init__(self, learning_rate = 0.01,
                 n = 1000):
        self.learning_rate = learning_rate
        self.num_iterions = num_iterations
        self.weights = None
        self.bias = None.

def sigmoid (self, z):
    return 1/(1+np.exp(-z))

def fit (self, X, y):
    n_samples, n_features = X.shape
    self.weights = np.zeros(n_features)
    self.bias = 0

for _ in range (self.num_n):
    linear_model = np.dot(X, self.weights)
                   + self.bias
```

PAGE NO :
DATE :

```
y_predicted = self.sigmoid(linear_model)

dw = (1/n_samples) * np.dot(x.T,
                    (y_predicted - y))
db = (1/n_samples) * np.sum(y_predicted - y)

self.weights -= self.learning_rate * dw
self.bias -= self.learning_rate * db

def predict (self, x):
    linear_model = np.dot(x, self.weights) +
                   self.bias
    y_p = self.sigmoid(linear_model)
    y_p_cls = [1 if i > 0.5 else 0 for i
                   in y_p]

    return np.array(y_p_cls)

if __name__ == "__main__":

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

    model = LogisticRegression()
    model.fit(x,y)

    y_pred = model.predict(x)

    print("Predictions:", y_pred)
```

Code:

```
import numpy as np

# Sample dataset
# Features: [hours_studied, attendance_percentage]
X = np.array([
    [1, 50],
    [2, 60],
    [3, 65],
    [4, 70],
    [5, 75],
    [6, 80],
    [7, 85],
    [8, 90]
])

# Labels: 0 = Fail, 1 = Pass
Y = np.array([0, 0, 0, 0, 1, 1, 1, 1])

# Normalize data (optional but helps)
X = X / np.max(X, axis=0)

# Initialize weights and bias
n_samples, n_features = X.shape
W = np.zeros(n_features)
b = 0

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Training hyperparameters
lr = 0.1
epochs = 1000

# Training loop (Gradient Descent)
for i in range(epochs):
    Z = np.dot(X, W) + b
    Y_pred = sigmoid(Z)

    error = Y_pred - Y

    dW = (1 / n_samples) * np.dot(X.T, error)
    db = (1 / n_samples) * np.sum(error)

    W -= lr * dW
```

```
    b -= lr * db

# Final weights
print(f"Weights: {W}")
print(f"Bias: {b}")

# Prediction function
def predict(inputs):
    inputs = np.array(inputs) / np.max(X, axis=0)  # normalize
    z = np.dot(inputs, W) + b
    prob = sigmoid(z)
    return 1 if prob >= 0.5 else 0, prob

# Try predicting
sample = [6, 85]  # 6 hrs studied, 85% attendance
label, probability = predict(sample)
print(f"Prediction for input {sample}: {'Pass' if label == 1 else 'Fail'} with probability {probability:.2f}")
```

Program 5

Use an appropriate data set for building the decision tree (ID3) and apply this knowledge to classify a new sample

PAGE NO :
DATE : 17/03/25

LAB 2

ID3 algorithm implementation

```
import pandas as pd
import numpy as np
import math
from graphviz import Digraph

def cal_entropy (data, target):
    total_samples = len(data)
    class_counts = data [target].value_counts()
    entropy = 0
    for count in class_counts:
        probability = count / total_samples
        entropy -= probability * math.log2(probability
    return entropy

def cal_info_gain(data, feature, target):
    total_samples = len(data)
    weighted_entropy = 0
    for value in data[feature].unique():
        subset = data[data[feature] == value]
        subset_entropy = cal_entropy(subset, target)
        weighted_entropy += (len(subset)/
            total_samples) * subset_entropy
    return cal_entropy(data, target) - weight
                                    _entropy

def build_tree (data, target, features, parent_
                node_class = None):
    if len(data[target].unique()) <= 1:
        return data[target].unique()[0]
```

PAGE NO :
DATE :

```
    if len(features) == 0:
        return parent_node_class

    parent_node_class = data[target].mode()[0]

    best_feature = max(features, key = lambda
        feature: calculate_info_gain(data, feature,
        target))
    tree = {best_feature: {}}
    features.remove(best_feature)

    for value in data[best_feature].unique():
        subset = data[data[best_feature] == value]
        subtree = build_tree(subset, target, features.
            copy(), parent_node_class)

    return tree

def visualize_tree(tree, dot=None, node_name
    =Fa

    if not dot:
        dot = Digraph(comment = 'Decision
            tree')

    if isinstance(tree, dict):
        feature = list(tree.keys())[0]
        dot.node(node_name, label = feature)

    else:
        dot.node(node_name, label = str(tr
    return dot
```

Code:

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

# Dataset
```

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

columns = ['Outlook', 'Temperature', 'Humidity', 'Wind', 'Play']
df = pd.DataFrame(data, columns=columns)

# Calculate entropy
def entropy(target_col):
    values, counts = np.unique(target_col, return_counts=True)
    entropy = 0
    for i in range(len(values)):
        prob = counts[i] / np.sum(counts)
        entropy -= prob * math.log2(prob)
    return entropy

# Info Gain
def info_gain(data, split_attribute_name, target_name="Play"):
    total_entropy = entropy(data[target_name])
    vals, counts = np.unique(data[split_attribute_name], return_counts=True)
    weighted_entropy = 0

    for i in range(len(vals)):
        subset = data[data[split_attribute_name] == vals[i]]
        weighted_entropy += (counts[i]/np.sum(counts)) * entropy(subset[target_name])

    IG = total_entropy - weighted_entropy
    return IG

# ID3 Algorithm
def ID3(data, original_data, features, target_attribute_name="Play", parent_node_class=None):
    unique_classes = np.unique(data[target_attribute_name])

    # Stopping conditions
    if len(unique_classes) == 1:
```

```
        return unique_classes[0]
    elif len(data) == 0:
        return np.unique(original_data[target_attribute_name])[0]
    elif len(features) == 0:
        return parent_node_class
    else:
        parent_node_class =
np.unique(data[target_attribute_name])[np.argmax(np.unique(data[target_attribute_name],
return_counts=True)[1])]

        # Choose best feature
        gains = [info_gain(data, feature, target_attribute_name) for feature in features]
        best_feature = features[np.argmax(gains)]

        # Create tree
        tree = {best_feature: {}}
        feature_values = np.unique(data[best_feature])

        for value in feature_values:
            sub_data = data[data[best_feature] == value]
            subtree = ID3(sub_data, data, [f for f in features if f != best_feature], target_attribute_name,
parent_node_class)
            tree[best_feature][value] = subtree

        return tree

# Build tree
features = list(df.columns)
features.remove("Play")
tree = ID3(df, df, features)
print("\nDecision Tree (ID3):")
print(tree)

# Prediction
def predict(query, tree):
    for key in query.keys():
        if key in tree.keys():
            try:
                result = tree[key][query[key]]
            except:
                return "Unknown"
            if isinstance(result, dict):
                return predict(query, result)
            else:
                return result

# Test with new sample
sample = {'Outlook': 'Sunny', 'Temperature': 'Cool', 'Humidity': 'High', 'Wind': 'Strong'}
```

```
prediction = predict(sample, tree)
print(f"\nPrediction for {sample}: {prediction}")
```

Program 6

Build KNN Classification model for a given dataset

KNN algorithm:

```
import numpy as np
import pandas as pd
from collections import Counter

data = {'feature 1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        'feature 2': [2, 4, 1, 3, 5, 7, 6, 8, 10, 9],
        'class' : ['A', 'A', 'B', 'B', 'A', 'B', 'A',
                   'B', 'B', 'A']}
df = pd.DataFrame(data)

def euclidean_distance(x1, x2):
    return np.sqrt(np.sum(x1 - x2) **2))

def Knn(x_train, y_train, test_point, k = 3):
    distances = [euclidean_dist(test_point, x) for
                 x in X_train]
    k_indices = np.argsort(distances)[:k]
    k_n_l = [y_train[i] for i in k_indices]
    most_common = Counter(k_n_l).most_common(1
    return most_common[0][0]

x = df[['feature 1', 'feature 2']].values
y = df['class'].values

test_point = np.array([6, 6])
predicted_class = Knn(X, y, test_point, k=3)
```

test_points = [[6,6],[2,3],[9,10]]

output:

Predicted class for [6,6]: A
Predicted class for [9,3]: B
Predicted class for [2,3]: A

Code:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from collections import Counter
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Load dataset
iris = load_iris()
X = iris.data[:, :2]  # using only 2 features for 2D visualization
y = iris.target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Euclidean distance function
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2))

# KNN from scratch
def knn_predict(X_train, y_train, x_test, k=3):
```

```
    distances = []
    for i in range(len(X_train)):
        dist = euclidean_distance(X_train[i], x_test)
        distances.append((dist, y_train[i]))
    distances.sort()
    neighbors = distances[:k]
    classes = [label for _, label in neighbors]
    prediction = Counter(classes).most_common(1)[0][0]
    return prediction

# Predicting on test set
predictions = []
k = 3
for test_point in X_test:
    pred = knn_predict(X_train, y_train, test_point, k)
    predictions.append(pred)

# Accuracy
accuracy = np.sum(predictions == y_test) / len(y_test)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

Program 7

Build Support vector machine model for a given dataset

SVM

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

data = {'feature1' : [1,2,3,4,5,6,7,8,9,10],
        'feature2' : [2,4,1,3,5,7,6,8,10,9],
        }
df = pd.DataFrame(data)

x = df[['feature1', 'feature2']].values
y = df['class'].values

class SVM :
    def __init__(self, learning = 0.001, l = 0.01,
                 n = 1000):
        self.lr = learning
        self.l = l
        self.n = n
        self.w = None
        self.b = None
```

PAGE NO :
DATE :

```
def fit(self, x, y):
    n_s, n_f = x.shape
    y_ = np.where(y<=0,-1,1)

    self.w = np.zeros(n_f)
    self.b = 0

    for _ in range(self.n):
        for idx, x_i in enumerate(x):
            condition = y_[idx]*(np.dot(x_i, self.w)
                        - self.b)>=1
            if condition:
                self.w = self.lr*(2*self.l*self.w)
            else:
                self.w = self.lr*(2*self.lambda*
                        self.w-np.dot(x_i,
                        y_[idx]))
                self.b = -self.lr * y_[idx]

    def predict(self, x):
        app = np.dot(x, self.w) - self.b
        return np.sign(app)

clf = SVM()
clf.fit(x,y)

test_points = [[6,6], [2,3], [9,10]]
for point in test_points:
    predictn = clf.predict(np.array(point))
    print(f"Predicted class for {point}:
                {prediction}")
```

Code:

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Generate linearly separable dataset
X, y = make_classification(n_samples=100, n_features=2, n_informative=2,
                           n_redundant=0, n_clusters_per_class=1, random_state=42)
y = np.where(y == 0, -1, 1)  # Convert labels to -1, 1 for SVM

# Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize parameters
```

```
epochs = 1000
learning_rate = 0.001
lambda_param = 0.01
n_features = X_train.shape[1]

w = np.zeros(n_features)
b = 0

# Train the SVM using stochastic gradient descent
for epoch in range(epochs):
    for i in range(len(X_train)):
        x_i = X_train[i]
        y_i = y_train[i]
        if y_i * (np.dot(w, x_i) + b) >= 1:
            w -= learning_rate * (2 * lambda_param * w)
        else:
            w -= learning_rate * (2 * lambda_param * w - np.dot(x_i, y_i))
            b -= learning_rate * y_i

# Prediction function
def predict(X):
    return np.sign(np.dot(X, w) + b)

# Evaluate accuracy
predictions = predict(X_test)
accuracy = np.mean(predictions == y_test)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Visualization
plt.figure(figsize=(8, 6))
for i, label in enumerate(np.unique(y)):
    plt.scatter(X[y == label][:, 0], X[y == label][:, 1], label=f"Class {label}")

# Draw decision boundary
x0 = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
x1 = -(w[0] * x0 + b) / w[1]
plt.plot(x0, x1, color='black', linewidth=2, label='Decision Boundary')
plt.legend()
plt.title("SVM Decision Boundary")
plt.show()
```

Program 8

Implement Random forest ensemble method on a given dataset

```
Random Forest

Function Random_Forest (Dataset, D , Integer
N_Trees , Integer m_features ) :
    Forest ← [ ]
    For i from 1 to N_trees do:
        Sample Di ← Bootsrap_sample(D)
        Tree Ti ← Build_dec_tree (Di, m_features)
        add Ti to forest
    return Forest

Function Build_Decision_Tree (Data Di, Integer m):
    If stopping_condition_met (Di) :
        Return Leaf_Node (Most_common_label(Di))

    Features ← Random_Subset (All_features, m)
    Best_feature, threshold ← Find_Best_Split
            (Di , Features)

    Left_data , Right_data ← Split_Data ( Di ,
            Best_feature , threshold)

    Left_node ← Build_Decision (Left_data , m)
    Right_node ← Build_Decision ( Right_data , m)

    Return Node ( Best_feature , threshold , left_node ,
            Right_node)

Function Predict_Random (Forest , Instance x):
    Predictions ← [Predict_tree ( Ti , x) for each
            Ti in Forest ]
    Return Majority_vote (Predictions)
```

Code:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from collections import Counter
import random

# Load sample dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split into train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Gini index calculation
def gini_index(groups, classes):
    n_instances = float(sum([len(group) for group in groups]))
    gini = 0.0
    for group in groups:
        size = len(group)
        if size == 0: continue
        score = 0.0
        labels = [row[-1] for row in group]
        for class_val in classes:
            p = labels.count(class_val) / size
            score += p * p
        gini += (1 - score) * (size / n_instances)
    return gini

# Split dataset based on a feature and threshold
def test_split(index, value, dataset):
    left, right = [], []
    for row in dataset:
        if row[index] < value: left.append(row)
        else: right.append(row)
    return left, right

# Get the best split
def get_split(dataset):
    class_values = list(set(row[-1] for row in dataset))
    b_index, b_value, b_score, b_groups = None, None, float('inf'), None
    for index in range(len(dataset[0])-1):
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = gini_index(groups, class_values)
```

```
            if gini < b_score:
                b_index, b_value, b_score, b_groups = index, row[index], gini, groups
    return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Create a terminal node
def to_terminal(group):
    outcomes = [row[-1] for row in group]
    return Counter(outcomes).most_common(1)[0][0]

# Recursive split
def split(node, max_depth, min_size, depth):
    left, right = node['groups']
    del(node['groups'])
    if not left or not right:
        node['left'] = node['right'] = to_terminal(left + right)
        return
    if depth >= max_depth:
        node['left'], node['right'] = to_terminal(left), to_terminal(right)
        return
    if len(left) <= min_size:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_split(left)
        split(node['left'], max_depth, min_size, depth+1)
    if len(right) <= min_size:
        node['right'] = to_terminal(right)
    else:
        node['right'] = get_split(right)
        split(node['right'], max_depth, min_size, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size):
    root = get_split(train)
    split(root, max_depth, min_size, 1)
    return root

# Make prediction with a decision tree
def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else: return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else: return node['right']

# Create a subsample with replacement
```

```
def subsample(dataset, ratio):
    sample = []
    n_sample = round(len(dataset) * ratio)
    while len(sample) < n_sample:
        index = random.randrange(len(dataset))
        sample.append(dataset[index])
    return sample

# Random Forest algorithm
def random_forest(train, test, max_depth, min_size, sample_size, n_trees):
    trees = []
    for _ in range(n_trees):
        sample = subsample(train, sample_size)
        tree = build_tree(sample, max_depth, min_size)
        trees.append(tree)
    predictions = [bagging_predict(trees, row) for row in test]
    return predictions

# Make a prediction with a list of trees
def bagging_predict(trees, row):
    predictions = [predict(tree, row) for tree in trees]
    return Counter(predictions).most_common(1)[0][0]

# Prepare full dataset with labels
dataset = np.concatenate((X_train, y_train.reshape(-1,1)), axis=1).tolist()
test_data = X_test.tolist()

# Run Random Forest
predictions = random_forest(dataset, test_data, max_depth=5, min_size=2, sample_size=0.8, n_trees=5)
accuracy = np.mean(predictions == y_test) * 100
print(f"Accuracy: {accuracy:.2f}%")
```

Program 9

Implement Boosting ensemble method on a given dataset

AdaBoost (Boosting)

```
Function :AdaBoost (Dataset D, Integer T):
    Initialize weights wi = 1/n for each training
    sample (xi, yi)

    classifiers ← [ ]
    alphas ← [ ]

    for t from 1 to T do:
        classifier ht ← Train-weak-learner (D, weights)
        Error εt ← Σ [wi * I(ht(xi) ≠ yi)]
        alpha αt ← 0.5 * log((1 - εt)/εt)

    for each i from 1 to n do:
        wi ← wi * exp(-αi * yi * ht(xi))

    Normalize weights: wi ← wi / Σwi

    Append ht to classifiers
    Append αt to Alphas

    Return classifiers, alphas

Function Predict_adaBoost (classifiers,
    alphas, Instance x):

    Total ← 0
    for t from 1 to T do:
        Total ← Total + αt * ht(x)
    Return sign(Total)
```

Code:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer

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

# For binary classification (e.g., class 0 vs others)
y = np.where(y == 0, 1, -1)  # Convert to 1 and -1

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Weak Learner: Decision Stump
def decision_stump(X, y, weights):
    n_samples, n_features = X.shape
    min_error = float('inf')
    best_stump = {}

    for feature in range(n_features):
        thresholds = np.unique(X[:, feature])
        for thresh in thresholds:
            for inequality in ['lt', 'gt']:
                preds = np.ones(n_samples)
                if inequality == 'lt':
                    preds[X[:, feature] <= thresh] = -1
                else:
                    preds[X[:, feature] > thresh] = -1

                error = np.sum(weights[y != preds])
                if error < min_error:
                    min_error = error
                    best_stump['feature'] = feature
                    best_stump['threshold'] = thresh
                    best_stump['inequality'] = inequality
                    best_stump['predictions'] = preds.copy()

    return best_stump, min_error

# AdaBoost algorithm
def adaboost(X, y, n_estimators):
```

```
    n_samples = X.shape[0]
    weights = np.full(n_samples, 1 / n_samples)
    classifiers = []

    for i in range(n_estimators):
        stump, error = decision_stump(X, y, weights)

        # Avoid division by zero
        error = max(error, 1e-10)

        alpha = 0.5 * np.log((1 - error) / error)
        stump['alpha'] = alpha
        classifiers.append(stump)

        # Update weights
        preds = stump['predictions']
        weights *= np.exp(-alpha * y * preds)
        weights /= np.sum(weights)

    return classifiers

# Make prediction with ensemble
def predict(X, classifiers):
    final_pred = np.zeros(X.shape[0])
    for stump in classifiers:
        feature = stump['feature']
        thresh = stump['threshold']
        inequality = stump['inequality']
        alpha = stump['alpha']

        preds = np.ones(X.shape[0])
        if inequality == 'lt':
            preds[X[:, feature] <= thresh] = -1
        else:
            preds[X[:, feature] > thresh] = -1

        final_pred += alpha * preds
    return np.sign(final_pred)

# Train model
classifiers = adaboost(X_train, y_train, n_estimators=10)

# Predict and evaluate
y_pred = predict(X_test, classifiers)
accuracy = np.mean(y_pred == y_test) * 100
print(f"Boosting Accuracy: {accuracy:.2f}%")
```

Program 10

Build k-Means algorithm to cluster a set of data stored in a .CSV file

```
K - Means Clustering.

Function k_Means ( Dataset D , Integer K ,
  Integer max_iter):
    centroids ← Initialize_random (D, K)

  For it from 1 to max_it do:
    clusters ← assign_points_to_centroids(D,
                                  centroids)
    New_centroids ← compute_mean_of_clusters
                                  (clusters)

    If converged (centroids , New_centroids)
      then:
      Break

    centroids ← New_centroids.

  Return clusters , centroids

Function assign_points_to_centroids (Data D,
  centroids):
    clusters ← empty dictionary
    For each point x_i in D do:
      closest ← Find_Nearest_centroid( x_i ,
                  centroids)
      add x_i to clusters[closest]
    Return clusters

Function compute_mean_of_clusters(clusters):
    For each cluster:
    compute mean of all points return updated
                                  centroid
```

Code:

```
import numpy as np
import pandas as pd

# Load data from CSV file
# Make sure to upload your CSV file in Colab or provide the correct path
data = pd.read_csv('/content/your_data.csv')  # Replace with your CSV file path

# Select features (assuming all columns are numeric features)
X = data.values

# Euclidean distance calculation
def euclidean_distance(a, b):
    return np.sqrt(np.sum((a - b) ** 2))

# Initialize centroids randomly
def initialize_centroids(X, k):
    np.random.seed(42)
    random_indices = np.random.permutation(X.shape[0])
    centroids = X[random_indices[:k]]
    return centroids

# Assign clusters based on closest centroid
def assign_clusters(X, centroids):
    clusters = []
    for x in X:
        distances = [euclidean_distance(x, centroid) for centroid in centroids]
        cluster = np.argmin(distances)
        clusters.append(cluster)
    return np.array(clusters)

# Update centroids based on current clusters
def update_centroids(X, clusters, k):
    new_centroids = []
    for i in range(k):
        cluster_points = X[clusters == i]
        if len(cluster_points) == 0:  # Handle empty cluster
            new_centroids.append(X[np.random.choice(len(X))])
        else:
            new_centroids.append(cluster_points.mean(axis=0))
    return np.array(new_centroids)

# k-Means algorithm
def k_means(X, k, max_iters=100):
    centroids = initialize_centroids(X, k)
    for _ in range(max_iters):
        clusters = assign_clusters(X, centroids)
```

```
        new_centroids = update_centroids(X, clusters, k)

        # Check for convergence
        if np.all(centroids == new_centroids):
            break
        centroids = new_centroids
    return clusters, centroids

# Number of clusters (choose based on your problem)
k = 3

# Run k-Means clustering
clusters, centroids = k_means(X, k)

# Add cluster labels to original data
data['Cluster'] = clusters

print("Cluster centroids:")
print(centroids)

print("\nSample clustered data:")
print(data.head())

# Save clustered data with cluster labels to CSV
data.to_csv('/content/clustered_data.csv', index=False)
```

Program 11

Implement Dimensionality reduction using Principal Component Analysis (PCA) method

```
Principal Component Analysis (PCA):

Function PCA ( Data D, Integer K):
    standardized ← standardize_Data(D)

    CovMatrix ← Compute_covariance_matrix
                (standardized)
    Eigenvalues, Eigenvectors ← Eigen_Decomposition
                                (CovMatrix)

    Sorted ← Sort_Eigenvectors_By_Eigenvalues
             (Eigenvalues, Eigenvectors)
    Top-k_vectors ← select_Top_k_Eigenvectors
                    (sorted, k)

    Reduced Data ← Project_Data (standardized,
                   Top-k_vectors)
    Return ReducedData

Function Standardize_Data (Data D):
    For each feature :
        subtract mean and divide by standard
        deviation
    Return standardized D

Function Project_Data (Data, Eigenvectors):
    Return Matrix_Multiplication (Data,
                                 Eigenvectors).
```

Code:

```
import numpy as np
import pandas as pd

# Load your dataset (replace path with your CSV file)
data = pd.read_csv('/content/your_data.csv')  # Update path accordingly

# Extract features as numpy array
X = data.values

# Step 1: Standardize (mean = 0)
X_meaned = X - np.mean(X, axis=0)

# Step 2: Compute covariance matrix
cov_matrix = np.cov(X_meaned, rowvar=False)

# Step 3: Eigen decomposition
eigen_values, eigen_vectors = np.linalg.eigh(cov_matrix)  # eigh for symmetric matrices

# Step 4: Sort eigenvectors by eigenvalues descending
sorted_index = np.argsort(eigen_values)[::-1]
sorted_eigenvalues = eigen_values[sorted_index]
sorted_eigenvectors = eigen_vectors[:, sorted_index]

# Select number of components (k)
k = 2  # choose k as needed
eigenvector_subset = sorted_eigenvectors[:, 0:k]

# Step 5: Project data
X_reduced = np.dot(X_meaned, eigenvector_subset)

print("Original shape:", X.shape)
print("Reduced shape:", X_reduced.shape)

# Optional: create a DataFrame for the reduced data
reduced_df = pd.DataFrame(X_reduced, columns=[f'PC{i+1}' for i in range(k)])
print(reduced_df.head())
```