# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**“JnanaSangama”, Belgaum -590014, Karnataka.**

## LAB

## REPORTON

## MACHINE LEARNING

***Submitted by***


**Imran Wadrali(1BM21CS077)**


***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**


**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019(March 2024 to June 2024)**

**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**” is carried out by **Imran Wadrali (1BM21CS077)** who is 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 Visveswaraya Technological University, Belgaum during the year 2023-2024. The lab report has been approved as it satisfies the academic requirements in respect of **Machine Learning Lab - (22CS3PCMAL)** work prescribed for thesaid degree.

**Dr. K. Panimozhi**
Assistant Professor
BMSCE, Bengaluru

**Dr. Jyothi S Nayak**
Prof.& Head, Dept. of CSE
BMSCE, Bengaluru

## Index



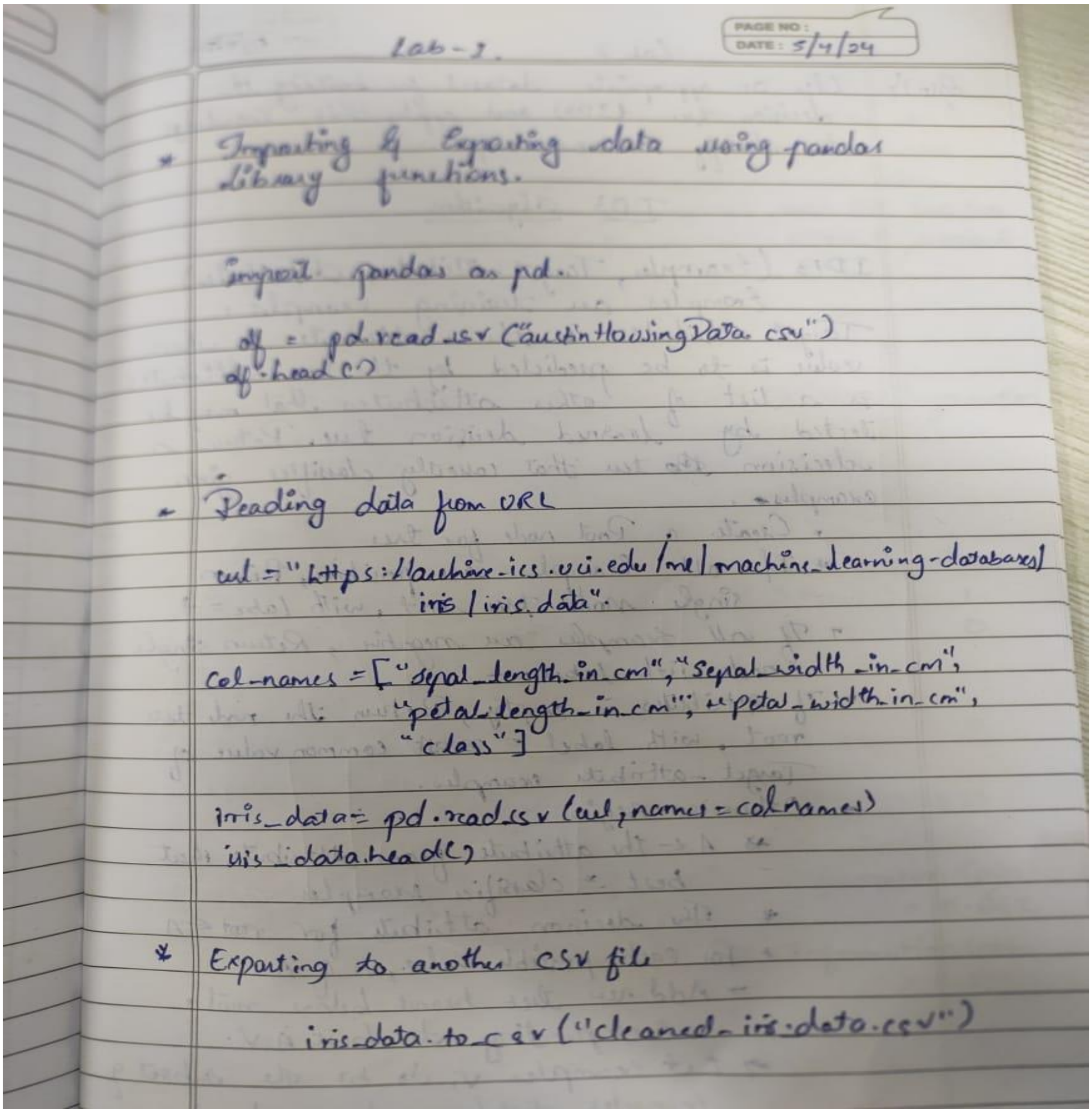
## Course outcomes:

| | |
|---|---|
| CO1 | Apply machine learning techniques in computing systems |
| CO2 | Evaluate the model using metrics |
| CO3 | Design a model using machine learning to solve a problem |
| CO4 | Conduct experiments to solve real-world problems using appropriate machine learning techniques |

# Lab 1

**1)** Write a python program to import and export data using Pandas library functions.

Algorithm (Observation book):



Lab - 1.

PAGE NO :
DATE : 5/4/24

* Importing & Exporting data using pandas library functions.

import pandas as pd.

df = pd.read_csv ("austinHousingData.csv")
df.head()

* Reading data from URL

url = "https://archive.ics.uci.edu/ml/machine_learning-databases/iris/iris.data".

col_names = ["sepal_length_in_cm", "sepal_width_in_cm", "petal_length_in_cm", "petal_width_in_cm", "class"]

iris_data = pd.read_csv (url, names = col_names)
iris_data.head()

* Exporting to another csv file

iris_data.to_csv ("cleaned_iris_data.csv")

Code

```
import pandas as pd
df=pd.read_csv("/content/austinHousingData.csv")
df.head(5)
```

Output:

| | zpid | city | streetAddress | zipcode | description | latitude | longitude | propertyTaxRate | garageSpaces | hasAssociation | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 111373431 | pflugerville | 14424 Lake Victor Dr | 78660 | 14424 Lake Victor Dr, Pflugerville, TX 78660 i... | 30.430632 | -97.663078 | 1.98 | 2 | True | ... |
| **1** | 120900430 | pflugerville | 1104 Strickling Dr | 78660 | Absolutely GORGEOUS 4 Bedroom home with 2 full... | 30.432673 | -97.661697 | 1.98 | 2 | True | ... |
| **2** | 2084491383 | pflugerville | 1408 Fort Dessau Rd | 78660 | Under construction - estimated completion in A... | 30.409748 | -97.639771 | 1.98 | 0 | True | ... |
| **3** | 120901374 | pflugerville | 1025 Strickling Dr | 78660 | Absolutely darling one story home in charming ... | 30.432112 | -97.661659 | 1.98 | 2 | True | ... |
| **4** | 60134862 | pflugerville | 15005 Donna Jane Loop | 78660 | Brimming with appeal & warm livability! Sleek ... | 30.437368 | -97.656860 | 1.98 | 0 | True | ... |

5 rows × 47 columns

**2)** Use an appropriate dataset for building the decision tree (ID3) and apply this knowledge to classify a new sample.

Algorithm (Observation book):

PAGE NO:
DATE: 02/4/24

Lab-2

Q(n)) Use an appropriate dataset for building the decision tree (ID3) and apply this knowledge to classify a new sample.

ID3 Algorithm

ID3 (Examples, Target-attribute, Attributes)
Examples are training examples.
Target attributes is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by learned decision tree. Returns a decision tree that correctly classifies given examples.

- Create a Root node for tree
- If all Examples are positive, Return single node tree root, with label = +
- If all Examples are negative, Return single node with label = -
- If Attributes is empty, Return the node tree root, with label = most common value of Target-attribute examples.
- Otherwise Begin
  - A ← the attribute from Attribute that best classifies examples
  - The decision attribute for root ← A
  - For each possible value, $v_i$ of A
    - → Add new tree branch below root, corresponding to best A = $v_i$.
    - → Let examples $v_i$ to be the subset of examples that have value $v_i$ for A

- If example $v_i$ is empty
  - Then below this new branch add a leaf node with label = most common value of Target-attribute in Examples.
  - → Else below this new branch add the sub tree ID3 ( Examples $v_i$, target-attribute, Attribute {A}) )
- End
- → Return Root

4 The Best attribute is one with highest information gain.

Output:

→ df

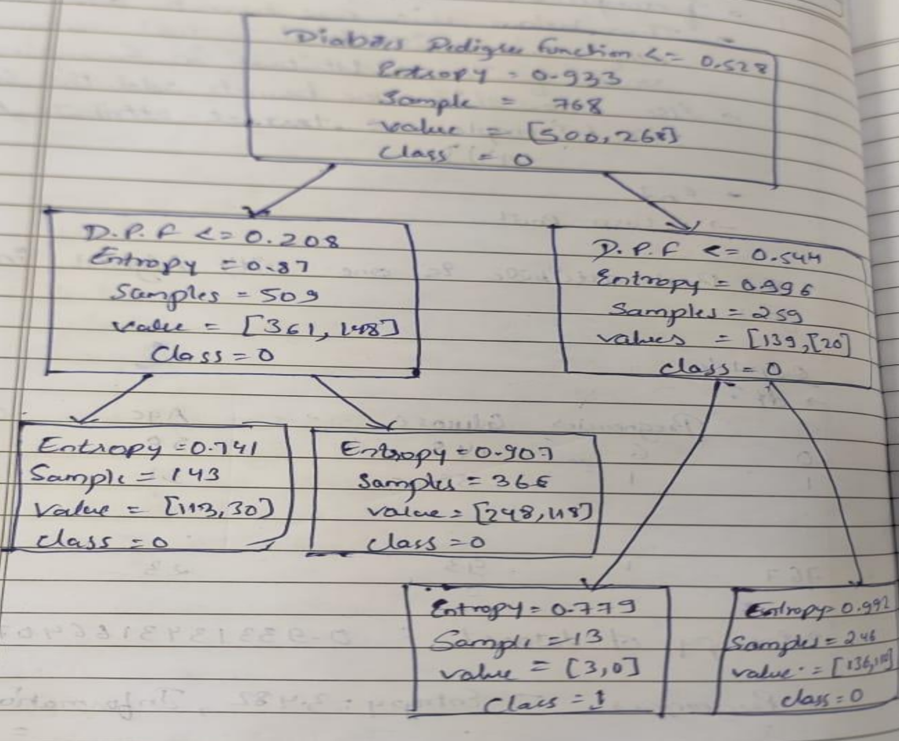
| | Pregnencies | Glucose | ... | Age | outcome |
|---|---|---|---|---|---|
| 0 | 6 | 148 | | 50 | 1 |
| 1 | 1 | 85 | | 31 | 0 |
| : | | | | | |
| : | | | | | |
| 767 | 1 | 93 | | 23 | 0 |

→ Entropy of dataset : 0.9331343166407831

→ Pregnencies – Entropy : 3.482 , Information Gain = 0.662

Age – Entropy : 5.029 , Information Gain = 0.141

Decision tree :

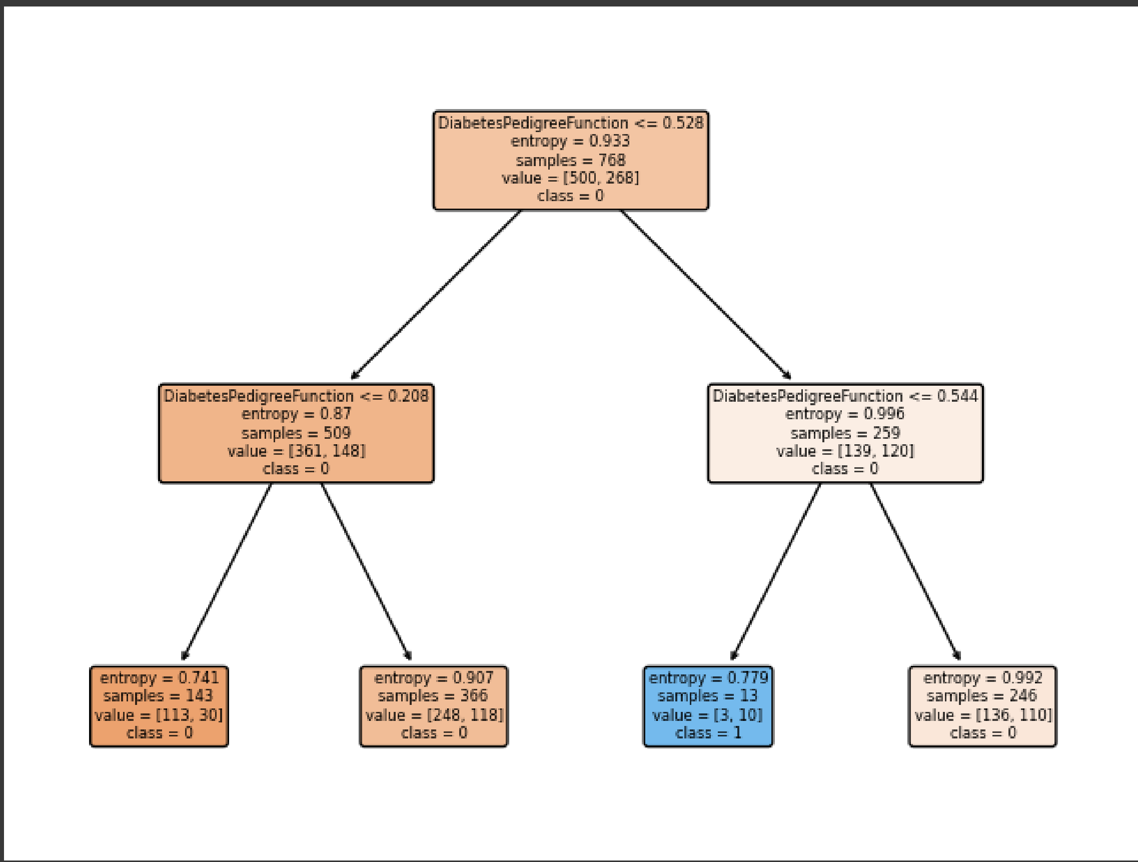


Code:

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt
import math

df = pd.read_csv('/content/diabetes.csv')

def calculate_entropy(data, target_column):
    total_rows = len(data)
    target_values = data[target_column].unique()

    entropy = 0
    for value in target_values:
        # Calculate the proportion of instances with the current value
        value_count = len(data[data[target_column] == value])
        proportion = value_count / total_rows
        entropy -= proportion * math.log2(proportion)

    return entropy

entropy_outcome = calculate_entropy(df, 'Outcome')
print(f"Entropy of the dataset: {entropy_outcome}")

def calculate_entropy(data, target_column): # for each categorical variable
    total_rows = len(data)
    target_values = data[target_column].unique()

    entropy = 0
    for value in target_values:
        # Calculate the proportion of instances with the current value
        value_count = len(data[data[target_column] == value])
        proportion = value_count / total_rows
        entropy -= proportion * math.log2(proportion) if proportion != 0 else 0

    return entropy

def calculate_information_gain(data, feature, target_column):

    # Calculate weighted average entropy for the feature
    unique_values = data[feature].unique()
    weighted_entropy = 0

    for value in unique_values:
        subset = data[data[feature] == value]
        proportion = len(subset) / len(data)
        weighted_entropy += proportion * calculate_entropy(subset, target_column)
```

```
    # Calculate information gain
    information_gain = entropy_outcome - weighted_entropy

    return information_gain

for column in df.columns[:-1]:
    entropy = calculate_entropy(df, column)
    information_gain = calculate_information_gain(df, column, 'Outcome')
    print(f"{column} - Entropy: {entropy:.3f}, Information Gain: {information_gain:.3f}")

# Feature selection for the first step in making decision tree
selected_feature = 'DiabetesPedigreeFunction'

# Create a decision tree
clf = DecisionTreeClassifier(criterion='entropy', max_depth=1)
X = df[[selected_feature]]
y = df['Outcome']
clf.fit(X, y)

plt.figure(figsize=(8, 6))
plot_tree(clf, feature_names=[selected_feature], class_names=['0', '1'], filled=True,
rounded=True)
plt.show()

def id3(data, target_column, features):
    if len(data[target_column].unique()) == 1:
        return data[target_column].iloc[0]


    if len(features) == 0:
        return data[target_column].mode().iloc[0]

    best_feature = max(features, key=lambda x: calculate_information_gain(data, x,
target_column))

    tree = {best_feature: {}}

    features = [f for f in features if f != best_feature]

    for value in data[best_feature].unique():
        subset = data[data[best_feature] == value]
        tree[best_feature][value] = id3(subset, target_column, features)

    return tree

id3(df, 'Outcome', ['Pregnancies',  'Glucose',  'BloodPressure',    'SkinThickness',    'Insulin',
'BMI',  'DiabetesPedigreeFunction', 'Age'] )
```

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




```
Pregnancies - Entropy: 3.482, Information Gain: 0.062
Glucose - Entropy: 6.751, Information Gain: 0.304
BloodPressure - Entropy: 4.792, Information Gain: 0.059
SkinThickness - Entropy: 4.586, Information Gain: 0.082
Insulin - Entropy: 4.682, Information Gain: 0.277
BMI - Entropy: 7.594, Information Gain: 0.344
DiabetesPedigreeFunction - Entropy: 8.829, Information Gain: 0.651
Age - Entropy: 5.029, Information Gain: 0.141
```

**3)** Implement Linear and Multi-Linear Regression algorithm using appropriate dataset

**Algorithm**

Lab-3

Q) Implement Linear & Multiple Linear Regression algorithm using appropriate dataset.

* Linear Regression
  1. Import dataset (2 features)
  2. Store independent feature to 'x' & dependent feature to 'y'
  3. Estimate $b_0$ & $b_1$ coefficient using linear Regression
  4. Plot regression line
  5. Print $b_0$, $b_1$ & predict the value of y for new value of x.

* Multiple Linear Regression
  1. split dataset into training & testing set
  2. We can see multiple independent variables
  3. Create regression model
     regression = LinearRegression()
  4. Fit train set
  5. Test model using test set
  6. Compare actual value & predicted value.

* Build KNN classification model for given dataset

Algorithm:

1] Define value of K and a distance metric

2] For given point, calculate distance between given point & every other point in dataset.

3] choose K closest points.

4] The class / value of given point is majority of that K points.

**Code:**

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

```
[ ] def estimate_coef(x, y):
      # number of observations/points
      n = np.size(x)

      # mean of x and y vector
      m_x = np.mean(x)
      m_y = np.mean(y)

      # calculating cross-deviation and deviation about x
      SS_xy = np.sum(y*x) - n*m_y*m_x
      SS_xx = np.sum(x*x) - n*m_x*m_x

      # calculating regression coefficients
      b_1 = SS_xy / SS_xx
      b_0 = m_y - b_1*m_x

      return (b_0, b_1)
```

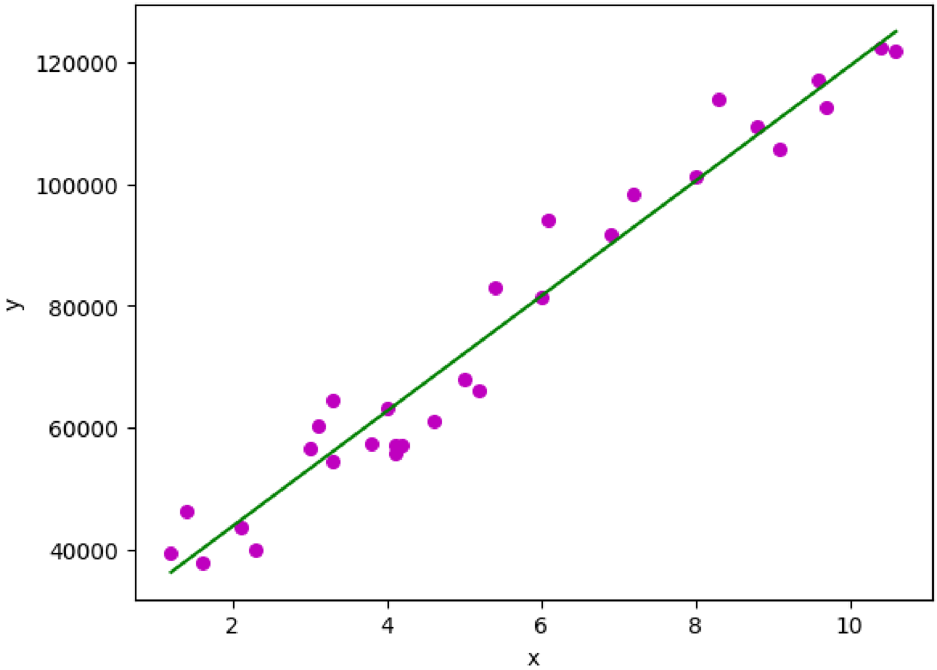
```
def plot_regression_line(x, y, b):
  # plotting the actual points as scatter plot
  plt.scatter(x, y, color = "m",
      marker = "o", s = 30)

  # predicted response vector
  y_pred = b[0] + b[1]*x

  # plotting the regression line
  plt.plot(x, y_pred, color = "g")

  # putting labels
  plt.xlabel('x')
  plt.ylabel('y')
```

```
[ ]  import pandas as pd
     def main():
       # observations / data
       df=pd.read_csv("Salary_dataset.csv")
       x = np.array(df['YearsExperience'])
       y = np.array(df['Salary'])

       # estimating coefficients
       b = estimate_coef(x, y)
       plot_regression_line(x,y,b)
       print("Estimated coefficients:\nb_0 = {}\nb_1 = {}".format(b[0],b[1]))
       x=int(input("Enter X value:"))
       y=b[0]+b[1]*x
       print(f"The predicted salary is:{y}")
```

```
[ ]  main()
```

OUTPUT:

```
Estimated coefficients:
b_0 = 24848.203966523113
b_1 = 9449.962321455092
Enter X value45
The predicted salary is:450096.5084320023
```



b) Implement Multi-Linear Regression algorithm using appropriate dataset

```
[6]     """
        Standardizes the input data using mean and standard deviation.

        Parameters:
            X_train (numpy.ndarray): Training data.
            X_test (numpy.ndarray): Testing data.

        Returns:
            Tuple of standardized training and testing data.
        """
        # Calculate the mean and standard deviation using the training data
        mean = np.mean(X_train, axis=0)
        std = np.std(X_train, axis=0)

        # Standardize the data
        X_train = (X_train - mean) / std
        X_test = (X_test - mean) / std

        return X_train, X_test

    X_train, X_test = standardize_data(X_train, X_test)
```

```
[7] X_train = np.expand_dims(X_train, axis=-1)
    X_test = np.expand_dims(X_test, axis=-1)
```

```
[8] class LinearRegression:
        """
        Linear Regression Model with Gradient Descent

        Linear regression is a supervised machine learning algorithm used for modeling the relationship
        between a dependent variable (target) and one or more independent variables (features) by fitting
        a linear equation to the observed data.

        This class implements a linear regression model using gradient descent optimization for training.
        It provides methods for model initialization, training, prediction, and model persistence.

        Parameters:
            learning_rate (float): The learning rate used in gradient descent.
            convergence_tol (float, optional): The tolerance for convergence (stopping criterion). Defaults to 1e-6.

        Attributes:
```

```
        Attributes:
[8]         W (numpy.ndarray): Coefficients (weights) for the linear regression model.
            b (float): Intercept (bias) for the linear regression model.

        Methods:
            initialize_parameters(n_features): Initialize model parameters.
            forward(X): Compute the forward pass of the linear regression model.
            compute_cost(predictions): Compute the mean squared error cost.
            backward(predictions): Compute gradients for model parameters.
            fit(X, y, iterations, plot_cost=True): Fit the linear regression model to training data.
            predict(X): Predict target values for new input data.
            save_model(filename=None): Save the trained model to a file using pickle.
            load_model(filename): Load a trained model from a file using pickle.

        Examples:
            >>> from linear_regression import LinearRegression
            >>> model = LinearRegression(learning_rate=0.01)
            >>> model.fit(X_train, y_train, iterations=1000)
            >>> predictions = model.predict(X_test)
        """

        def __init__(self, learning_rate, convergence_tol=1e-6):
            self.learning_rate = learning_rate
            self.convergence_tol = convergence_tol
            self.W = None
            self.b = None

        def initialize_parameters(self, n_features):
            """
            Initialize model parameters.

            Parameters:
                n_features (int): The number of features in the input data.
            """
            self.W = np.random.randn(n_features) * 0.01
            self.b = 0
        def forward(self, X):
            """
            Compute the forward pass of the linear regression model.

            Parameters:
                X (numpy.ndarray): Input data of shape (m, n_features).
```

[8]

```python
        Returns:
            numpy.ndarray: Predictions of shape (m,).
        """
        return np.dot(X, self.W) + self.b

    def compute_cost(self, predictions):
        """
        Compute the mean squared error cost.

        Parameters:
            predictions (numpy.ndarray): Predictions of shape (m,).

        Returns:
            float: Mean squared error cost.
        """
        m = len(predictions)
        cost = np.sum(np.square(predictions - self.y)) / (2 * m)
        return cost

    def backward(self, predictions):
        """
        Compute gradients for model parameters.

        Parameters:
            predictions (numpy.ndarray): Predictions of shape (m,).

        Updates:
            numpy.ndarray: Gradient of W.
            float: Gradient of b.
        """
        m = len(predictions)
        self.dW = np.dot(predictions - self.y, self.X) / m
        self.db = np.sum(predictions - self.y) / m
    def fit(self, X, y, iterations, plot_cost=True):
        """
        Fit the linear regression model to the training data.

        Parameters:
            X (numpy.ndarray): Training input data of shape (m, n_features).
            y (numpy.ndarray): Training labels of shape (m,).
            iterations (int): The number of iterations for gradient descent.
            plot_cost (bool, optional): Whether to plot the cost during training. Defaults to True.
```

[8]

```python
        Raises:
            AssertionError: If input data and labels are not NumPy arrays or have mismatched shapes.

        Plots:
            Plotly line chart showing cost vs. iteration (if plot_cost is True).
        """
        assert isinstance(X, np.ndarray), "X must be a NumPy array"
        assert isinstance(y, np.ndarray), "y must be a NumPy array"
        assert X.shape[0] == y.shape[0], "X and y must have the same number of samples"
        assert iterations > 0, "Iterations must be greater than 0"

        self.X = X
        self.y = y
        self.initialize_parameters(X.shape[1])
        costs = []

        for i in range(iterations):
            predictions = self.forward(X)
            cost = self.compute_cost(predictions)
            self.backward(predictions)
            self.W -= self.learning_rate * self.dW
            self.b -= self.learning_rate * self.db
            costs.append(cost)

            if i % 100 == 0:
                print(f'Iteration: {i}, Cost: {cost}')

            if i > 0 and abs(costs[-1] - costs[-2]) < self.convergence_tol:
                print(f'Converged after {i} iterations.')
                break

        if plot_cost:
            fig = px.line(y=costs, title="Cost vs Iteration", template="plotly_dark")
            fig.update_layout(
                title_font_color="#41BEE9",
                xaxis=dict(color="#41BEE9", title="Iterations"),
                yaxis=dict(color="#41BEE9", title="Cost")
            )

            fig.show()
```

```
            fig.show()
    def predict(self, X):
        """
        Predict target values for new input data.

        Parameters:
            X (numpy.ndarray): Input data of shape (m, n_features).

        Returns:
            numpy.ndarray: Predicted target values of shape (m,).
        """
        return self.forward(X)


    def save_model(self, filename=None):
        """
        Save the trained model to a file using pickle.

        Parameters:
            filename (str): The name of the file to save the model to.
        """
        model_data = {
            'learning_rate': self.learning_rate,
            'convergence_tol': self.convergence_tol,
            'W': self.W,
            'b': self.b
        }

        with open(filename, 'wb') as file:
            pickle.dump(model_data, file)

    @classmethod
    def load_model(cls, filename):
        """
        Load a trained model from a file using pickle.

        Parameters:
            filename (str): The name of the file to load the model from.
```

```
8]      with open(filename, 'wb') as file:
            pickle.dump(model_data, file)

    @classmethod
    def load_model(cls, filename):
        """
        Load a trained model from a file using pickle.

        Parameters:
            filename (str): The name of the file to load the model from.

        Returns:
            LinearRegression: An instance of the LinearRegression class with loaded parameters.
        """
        with open(filename, 'rb') as file:
            model_data = pickle.load(file)

        # Create a new instance of the class and initialize it with the loaded parameters
        loaded_model = cls(model_data['learning_rate'], model_data['convergence_tol'])
        loaded_model.W = model_data['W']
        loaded_model.b = model_data['b']

        return loaded_model
```

```
9]  lr = LinearRegression(0.01)
    lr.fit(X_train, y_train, 10000)
```

```
Iteration: 0, Cost: 1670.0184887161677
Iteration: 100, Cost: 227.15535101517312
Iteration: 200, Cost: 33.84101696145528
Iteration: 300, Cost: 7.9408253395546575
Iteration: 400, Cost: 4.4707260872934835
Iteration: 500, Cost: 4.005803317750673
Iteration: 600, Cost: 3.943513116253261
Iteration: 700, Cost: 3.9351674953098015
Iteration: 800, Cost: 3.9340493517293096
Converged after 863 iterations.
```

## 4) Build KNN Classification model for a given dataset.

Algorithm

* Build KNN classification model for given dataset

Algorithm:

1] Define value of K and a distance metric

2] For given point, calculate distance between given point & every other point in dataset.

3] choose K closest points.

4] The class / value of given point is majority of that K points.

Code:

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix
from sklearn import datasets
iris = datasets.load_iris()

x = iris.data
y = iris.target

print('sepal-length', 'sepal-width', 'petal-length', 'petal-width')
print(x)
print('class: 0 - Iris-Setosa, 1 - Iris-Versicolour, 2 - Iris-Virginica')
print(y)
```

```python
In [5]:
x_train, x_test, y_train, y_test = train_test_split(x,y,test_size=0.3)

#To Training the model and Nearest nighbors K=5
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(x_train, y_train)

#to make predictions on our test data
y_pred=classifier.predict(x_test)

print('Prediction -')

for i,test in enumerate(x_test) :
  print(f'{test} - {y_pred[i]}')

# print('Confusion Matrix')
# print(confusion_matrix(y_test,y_pred))
# print('Accuracy Metrics')
# print(classification_report(y_test,y_pred))
```

Results:

```
Prediction -
[5.2 4.1 1.5 0.1] - 0
[5.5 2.3 4.  1.3] - 1
[6.7 3.1 4.7 1.5] - 1
[7.  3.2 4.7 1.4] - 1
[6.2 2.8 4.8 1.8] - 2
[5.7 2.8 4.5 1.3] - 1
[6.  3.4 4.5 1.6] - 1
[5.1 3.8 1.6 0.2] - 0
[5.5 2.5 4.  1.3] - 1
[4.8 3.1 1.6 0.2] - 0
[6.1 3.  4.9 1.8] - 2
[4.7 3.2 1.6 0.2] - 0
[5.6 2.9 3.6 1.3] - 1
[5.4 3.9 1.3 0.4] - 0
[5.  3.2 1.2 0.2] - 0
[6.1 2.9 4.7 1.4] - 1
[5.  3.4 1.5 0.2] - 0
[7.7 2.8 6.7 2. ] - 2
[4.6 3.2 1.4 0.2] - 0
[5.7 2.9 4.2 1.3] - 1
[4.6 3.6 1.  0.2] - 0
[6.8 2.8 4.8 1.4] - 1
[6.8 3.2 5.9 2.3] - 2
```

# 5) Build Logistic Regression Model for a given dataset

Algorithm:

Lab - 4

* Implement Logistic Regression using appropriate dataset

Algorithm:

1) Import required libraries
2) Load, visualize & explore the dataset
3) Clean the dataset
4) Deal with the outlines
5) Define dependent & independent variable & then split the data into a training set & testing set
6) A logistic regression modulating.

Output:

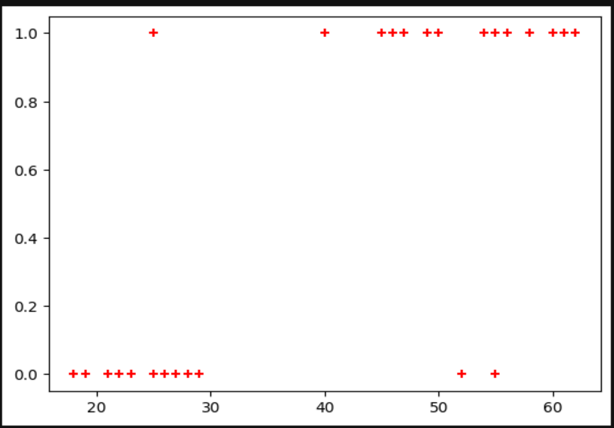
Regression co-efficients obtained are

$b_0 = -68.83$

$b_1 = 0.192671$

Code:

```
In [1]: import pandas as pd
        from matplotlib import pyplot as plt
        %matplotlib inline

In [4]: df = pd.read_csv("insurance_data.csv")
        df.head()
        plt.scatter(df.age,df.bought_insurance,marker ='+',color ='red')

Out[4]: <matplotlib.collections.PathCollection at 0x7e3b7e4c9cf0>
```




```
In [31]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(df[['age']],df.bought_insurance,train_size=0.7)
         print(X_test)
```

```
    age
12   27
13   29
4    46
23   45
3    52
8    62
1    25
11   28
25   54
```

In [32]:

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X_train, y_train)
print(X_test)
y_predicted = model.predict(X_test)
probab = model.predict_proba(X_test)
score = model.score(X_test,y_test)
print(y_predicted)
print("\nprobability: ")
print(probab)
print("\naccuracy: ")
print(score)
```

```
    age
12   27
13   29
4    46
23   45
3    52
8    62
1    25
11   28
25   54
[0 0 1 1 1 1 0 0 1]
```

```
probability:
[[0.75147045 0.24852955]
 [0.69990447 0.30009553]
 [0.20424226 0.79575774]
 [0.2261509  0.7738491 ]
 [0.10537592 0.89462408]
 [0.03115876 0.96884124]
 [0.79674889 0.20325111]
 [0.7264443  0.2735557 ]
 [0.08328741 0.91671259]]

accuracy:
0.8888888888888888
```

In [33]:

```
print(X_test)
```

```
    age
12   27
13   29
4    46
23   45
3    52
8    62
1    25
11   28
25   54
```

In [38]:

```
import math
def sigmoid(x):
  return 1 / (1 + math.exp(-x))
```

In [39]:

```
def prediction_function(age):
  z = 0.042 * age - 1.53 # 0.04150133 ~ 0.042 and -1.52726963 ~ -1.53
  y = sigmoid(z)
  return y
```

```
In [40]: def insure(probability):
           if (probability > 0.5):
             print("The customer will get insurance.")
           else:
             print("The customer will not get insurance.")

         age = 35
         probability = prediction_function(age)
         print("Probability of buying insurance for age 35:", probability)
         insure(probability)

         Probability of buying insurance for age 35: 0.4850044983805899
         The customer will not get insurance.

In [41]: age = 43
         probability = prediction_function(age)
         print("Probability of buying insurance for age 43:", probability)
         insure(probability)

         Probability of buying insurance for age 43: 0.568565299077705
         The customer will get insurance.
```

Results:

```
         Probability of buying insurance for age 35: 0.4850044983805899
         The customer will not get insurance.

In [41]: age = 43
         probability = prediction_function(age)
         print("Probability of buying insurance for age 43:", probability)
         insure(probability)

         Probability of buying insurance for age 43: 0.568565299077705
         The customer will get insurance.
```

**6)** Build Support vector machine model for a given dataset

Algorithm:

Lab-5

1) Implement Support Vector machine

Algorithm:

1. Define Kernel function
   eg: $K(x_1, x_2) = x_1 \cdot x_2$
2. Solve quadratic programming problem to find $\alpha$
3. Compute weight & bias
4. Identify the support vector
5. Make predictions

Output:

→ model = svm(x)
model.fit(x_train, y_train)
prediction = model.predict(x_test)
accuracy(y_test, predictions)
⇒ 0.93023

→ model.predict([[-0.47069, -0.1604, ..., 0.19695]])

array(0).

Code:

```
In [1]:
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
```

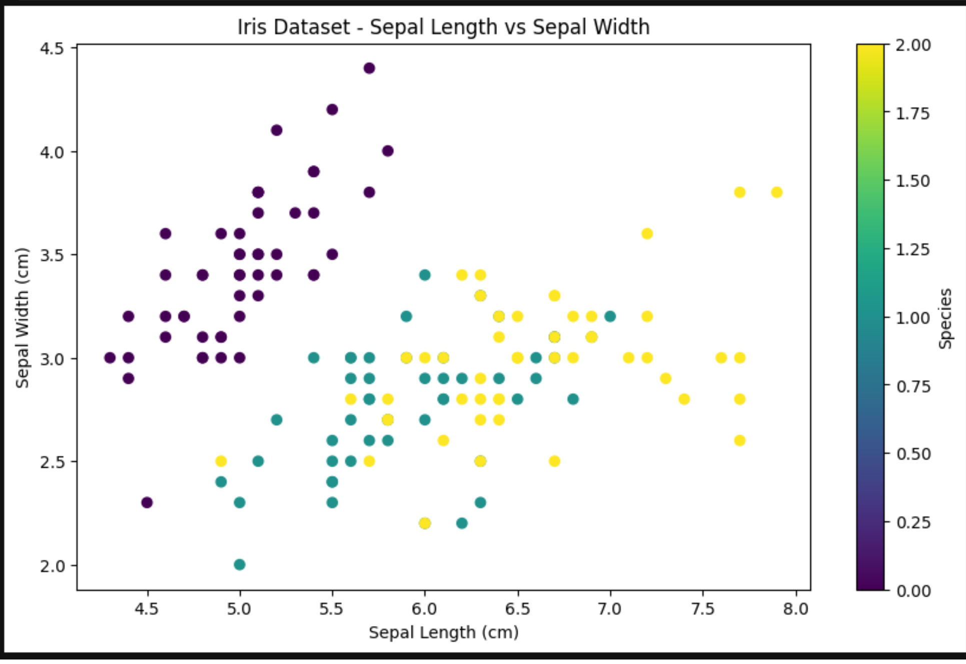
```
In [2]:
# Load the Iris dataset
iris = load_iris()
```

```
In [3]:
# Convert the dataset into a pandas DataFrame
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['target'] = iris.target
```

```
In [4]:
# Display the first few rows of the DataFrame
print(iris_df.head())
```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target
0       0
1       0
2       0
3       0
4       0
```

```
In [5]:
# Plotting to visualize the data
plt.figure(figsize=(10, 6))
plt.scatter(iris_df['sepal length (cm)'], iris_df['sepal width (cm)'],
            c=iris_df['target'], cmap='viridis')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.title('Iris Dataset - Sepal Length vs Sepal Width')
plt.colorbar(label='Species')
plt.show()
```




```
In [6]:
# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.3, random_state=42)
```

```
In [7]:
# Creating and training the SVM classifier
svm_classifier = SVC(kernel='linear')
svm_classifier.fit(X_train, y_train)

# Predicting the labels for the test set
y_pred = svm_classifier.predict(X_test)

# Calculating the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of SVM Classifier:", accuracy)
```

# Results:

```
Accuracy of SVM Classifier: 1.0

In [8]: y_pred

Out[8]: array([1, 0, 2, 1, 1, 0, 1, 2, 1, 1, 2, 0, 0, 0, 0, 1, 2, 1, 1, 2, 0, 2,
        0, 2, 2, 2, 2, 2, 0, 0, 0, 0, 1, 0, 0, 2, 1, 0, 0, 0, 2, 1, 1, 0,
        0])
```

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

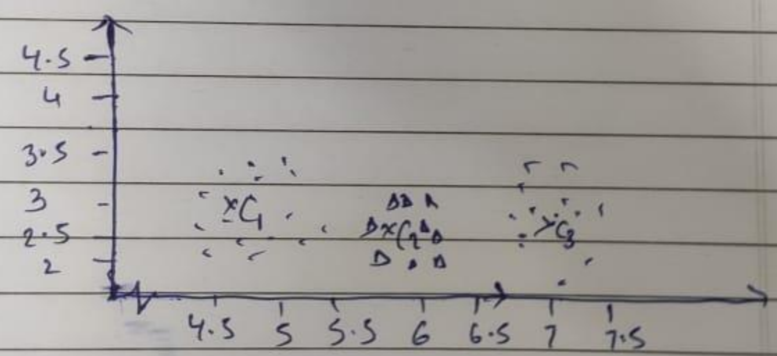
Algorithm:

7) K-means clustering

Algorithm:

1. Select number of K to decide no. of clusters.
2. Select random 'K' points [Centroids]
3. Assign each point to closest centroids, which will form the predefined 'K' clusters
4. Calculate variance and place a new centroid of each cluster
5. Repeat step 3, reassign the centroid.
6. If any reassignment go to step (4) else go to finish
7. The model is trained.

Output:



Code:

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import pandas as pd
import numpy as np
```

```
# import some data to play with
iris = datasets.load_iris()
X = pd.DataFrame(iris.data)
X.columns = ['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width']
y = pd.DataFrame(iris.target)
y.columns = ['Targets']
```

```
# Build the K Means Model
model = KMeans(n_clusters=3)
model.fit(X) # model.labels _: Gives cluster no for which samples belongs
```

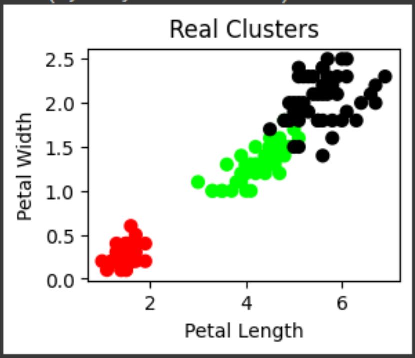
```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
KMeans
KMeans(n_clusters=3)
```

```
# # Visualise the clustering results
plt.figure(figsize=(14,14))
colormap = np.array(['red', 'lime', 'black'])
```

```
<Figure size 1400x1400 with 0 Axes>
```

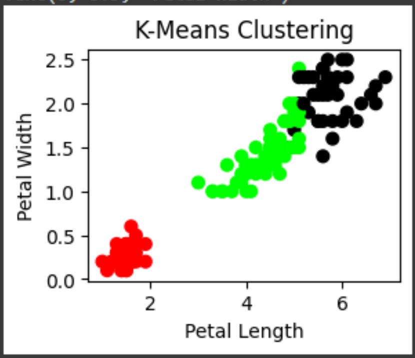
```
# Plot the Original Classifications using Petal features
plt.subplot(2, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Clusters')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
```

```
Text(0, 0.5, 'Petal Width')
```




```
# Plot the Models Classifications
plt.subplot(2, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K-Means Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
```

```
Text(0, 0.5, 'Petal Width')
```



**8)** Implement Dimensionality reduction using Principle Component Analysis (PCA) method.

Algorithm:

PAGE NO :
DATE :

3) Principal component Analysis:

Algorithm :

1) Calculate mean
2) Calculation of Covariance matrix
3) Eigen values of covariance matrix
4) Computation of eigen-vector (unit-eigen vector)
5) Computation of first principal components
6) Geometric meaning of first principal component

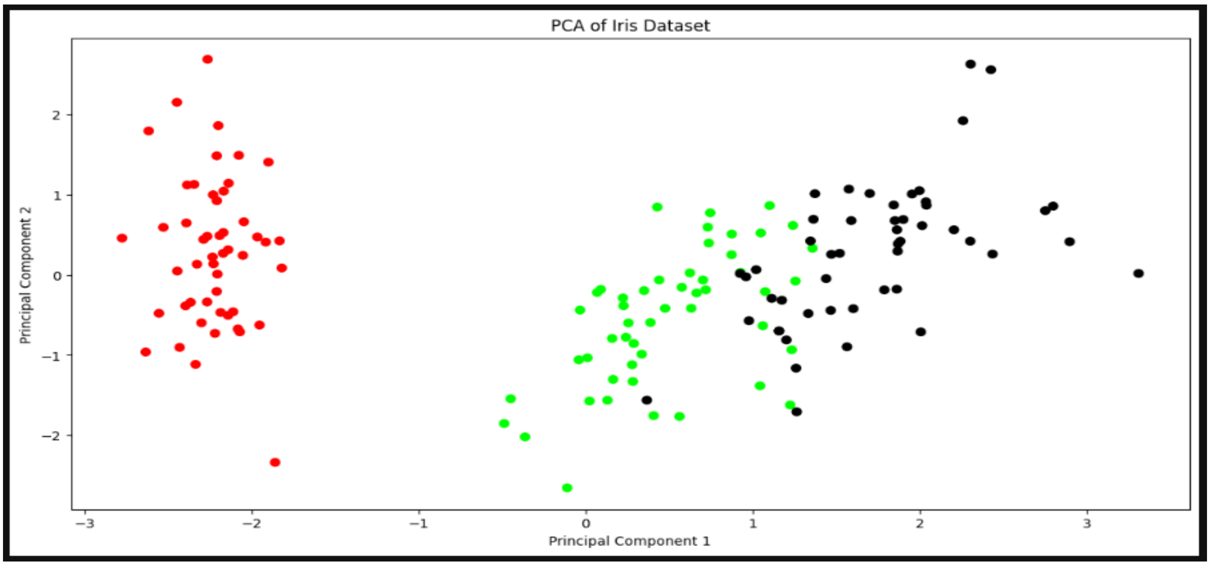
Output :

pca.explained_variance_ratio
array([0.9839448, 0.01620498

Code:

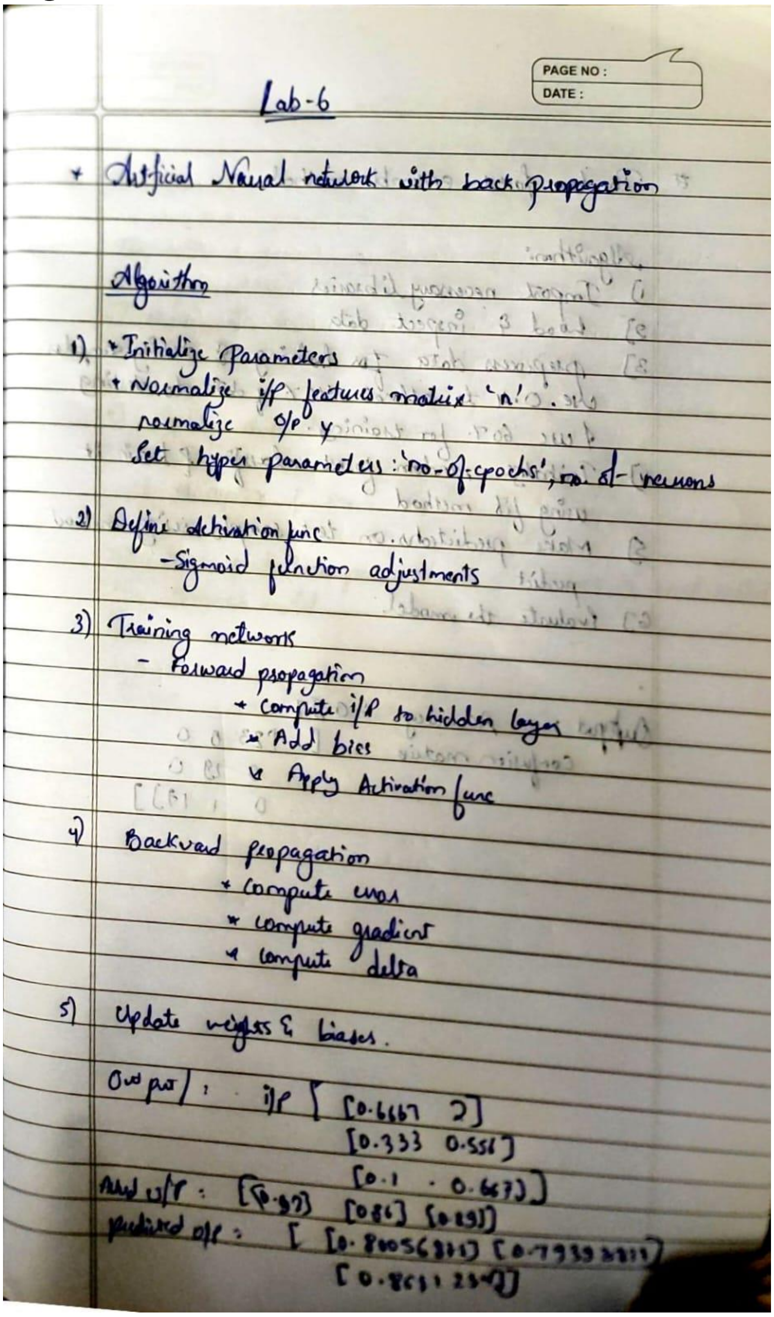
```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import pandas as pd
import numpy as np

# Load the iris dataset
iris = datasets.load_iris()
X = pd.DataFrame(iris.data, columns=['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width'])
y = pd.DataFrame(iris.target, columns=['Targets'])

# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Convert PCA result to a DataFrame
X_pca_df = pd.DataFrame(X_pca, columns=['PCA1', 'PCA2'])

# Add the target column for visualization
X_pca_df['Targets'] = y.Targets

# Visualize the PCA result
plt.figure(figsize=(14, 7))
colormap = np.array(['red', 'lime', 'black'])

# Plot the PCA transformed data
plt.scatter(X_pca_df.PCA1, X_pca_df.PCA2, c=colormap[X_pca_df.Targets], s=40)
plt.title('PCA of Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()
```

Results:




**9)** Build Artificial Neural Network model with back propagation on a given dataset

Algorithm:



PAGE NO :
DATE :

Lab-6

* Artificial Neural network with back propagation

Algorithm

1) * Initialize Parameters
 * Normalize i/p features matrix 'n'.
 normalize o/p y
 Set hyper parameters: 'no-of-epochs', no-of-neurons

2) Define activation func
 - Sigmoid function adjustments

3) Training network
 - Forward propagation
 * compute i/p to hidden layer
 * Add bias
 * Apply Activation func

4) Backward propagation
 * compute error
 * compute gradient
 * compute delta

5) Update weights & biases.

Output: i/p [ [0.6667 1]
 [0.333 0.551]
 [0.1 0.667] ]
Actual o/p: [[0.92] [0.86] [0.89]]
predicted o/p: [ [0.80056811] [0.79393811]
 [0.8631234]]

Code:

```
In [10]:
import numpy as np
x = np.array(([2,9],[1,5],[3,6]),dtype = float)
y = np.array(([92],[86],[89]),dtype = float)

x = x/np.amax(x,axis=0)
y = y/100
```

```
In [11]:
#Varialble Initialization
epoch = 5000
lr = 0.1
inputlayer_neurons = 2
hiddenlayer_neurons = 3
output_neurons = 1
```

```
In [12]:
# weight and bias Initialization

wh = np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh = np.random.uniform(size=(1,hiddenlayer_neurons))
wout = np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout = np.random.uniform(size=(1,output_neurons))
```

```
In [13]:
#sigmoid function

def sigmoid(x):
  return 1/(1+np.exp(-x))

# Derivative of Sigmoid

def der_sigmoid(x):
  return x*(1-x)
```

```
In [14]:
# Draws a random range of numbers uniformly of dim x*y

for i in range(epoch):

  # forward propagation
  hinp1 = np.dot(x,wh)
  hinp = hinp1 + bh
  hlayer_act = sigmoid(hinp)
  outinp1 = np.dot(hlayer_act,wout)
  outinp = outinp1 + bout
  output = sigmoid(outinp)

  # Backpropagation
  EO = y - output
  outgrad = der_sigmoid(output)
  d_output = EO*outgrad
  EH = d_output.dot(wout.T)
```

```
In [15]:
# how much hidden layer weights contributed to error

hiddengrad = der_sigmoid(hlayer_act)
d_hiddenlayer = EH*hiddengrad

#dotproduct of nextlayererror and current layer op

wout += hlayer_act.T.dot(d_output)*lr
wh += x.T.dot(d_hiddenlayer)*lr

print("Input: \n" + str(x))
print("Actual output: \n" + str(y))
print("Predicted Output: \n",output)
```

# Results

```
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.80056875]
 [0.79393831]
 [0.80112347]]
```

**10)** Implement Random forest ensemble method on a given dataset.

Algorithm:

PAGE NO :
DATE :

Random forest ensemble method

Algorithm:

1) Import necessary libraries
2) Load & inspect data
3) preprocess data for training & testing samples use 0.4 to allocate 40% of data to testing & use 60% for training.
4) Initialize random forest classifier & train it using fit method
5) Make predictions on test samples using method predict
6) Evaluate the model

Output : Accuracy : 0.97

Confusion matrix : [ [23 0 0
0 19 0
0 1 17] ]

Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn import datasets

# Load the data
iris_data = datasets.load_iris()

X = pd.DataFrame(iris_data.data, columns=['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width'])
y = pd.DataFrame(iris_data.target, columns=['Targets'])


# Check the info of the modified data
# print(iris_data.info())

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)

# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the classifier to the training data
rf_classifier.fit(X_train, y_train)

# Predict on the test data
y_pred = rf_classifier.predict(X_test)

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Print classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))

# Print confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

Results:

```
Accuracy: 0.98
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        23
           1       0.95      1.00      0.97        19
           2       1.00      0.94      0.97        18

    accuracy                           0.98        60
   macro avg       0.98      0.98      0.98        60
weighted avg       0.98      0.98      0.98        60

Confusion Matrix:
[[23  0  0]
 [ 0 19  0]
 [ 0  1 17]]
```

**11)** Implement Boosting ensemble method on a given dataset.

Algorithm

PAGE NO :
DATE :

# Boosting ensemble method

Algorithm:

1) Import libraries
2) Load the dataset
3) Data preprocessing involves separations of features & dataset
4) Split dataset to train samples
5) Initialize adaboost classifier with specified no. of estimator & base estimator
6) Train model using training data
7) Make predictions for test sample using trained model
8) Evaluate model.

Output : matrix accuracy score : 0.833.

Code:

```
In [4]:

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import AdaBoostClassifier
from sklearn import metrics
from sklearn import datasets

In [5]:

import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

In [6]:
# Load the iris dataset
iris = datasets.load_iris()
X = pd.DataFrame(iris.data, columns=['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width'])
y = pd.DataFrame(iris.target, columns=['Targets'])

In [7]:
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.4,random_state=42)

In [9]:
mylogregmodel = LogisticRegression()

In [10]:
adabc = AdaBoostClassifier(n_estimators = 150, base_estimator = mylogregmodel, learning_rate = 1)

In [11]:
model = adabc.fit(X_train, y_train)

/usr/local/lib/python3.10/dist-packages/sklearn/utils/validation.py:1143: DataConversionWarning: A column-vector y was passed
when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_base.py:166: FutureWarning: `base_estimator` was renamed to `estima
tor` in version 1.2 and will be removed in 1.4.
  warnings.warn(

In [12]:
y_pred = model.predict(X_test)

In [13]:
metrics.accuracy_score(y_test, y_pred)
```

Results:

```
Out[13]: 0.9833333333333333
```