

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT On MACHINE LEARNING

Submitted by

KAUSHIK POTLURI (1BM21CS089)

**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**

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**" carried out by **KAUSHIK POTLURI (1BM21CS089)**, 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 Visvesvaraya Technological University, Belgaum during the year 2023-24. The Lab report has been approved as it satisfies the academic requirements in respect of Machine Learning Lab - **(22CS6PCMAL)** work prescribed for the said degree.

Sunayana S
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index

Sl. No.	Experiment Title	Page No.
1	Write a python program to import and export data using Pandas library functions	1
2	Demonstrate various data pre-processing techniques for a given dataset	4
3	Use an appropriate data set for building the decision tree (ID3) and apply this knowledge to classify a new sample.	6
4	Build KNN Classification model for a given dataset.	13
5	Implement Linear and Multi-Linear Regression algorithm using appropriate dataset	20
6	Build Logistic Regression Model for a given dataset	27
7	Build Support vector machine model for a given dataset	37
8	Build k-Means algorithm to cluster a set of data stored in a .CSV file.	43
9	Implement Dimensionality reduction using Principle Component Analysis (PCA) method.	47
10	Build Artificial Neural Network model with back propagation on a given dataset	51
11	a) Implement Random forest ensemble method on a given dataset. b) Implement Boosting ensemble method on a given dataset.	54

Course outcomes:

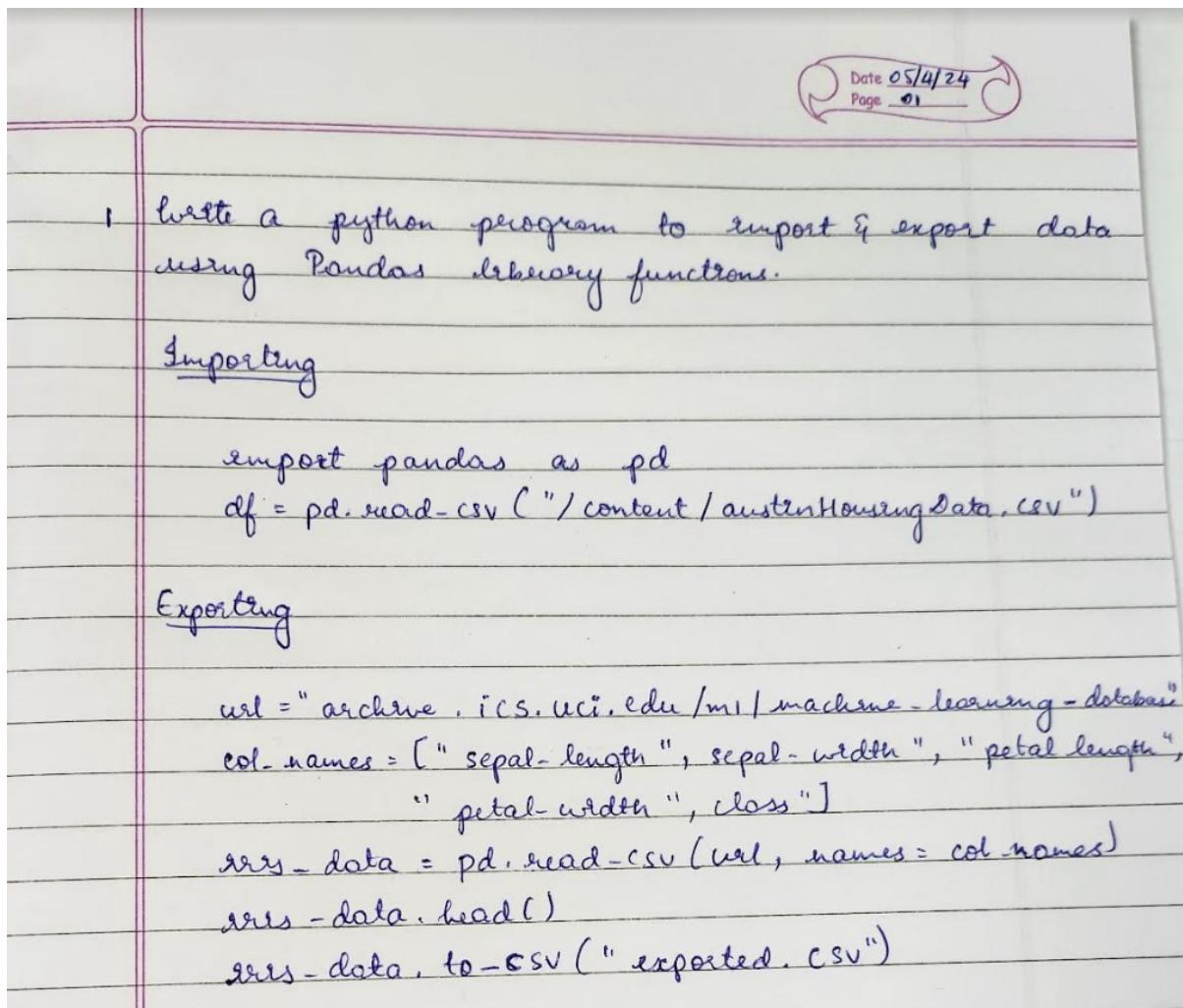
CO1	Apply machine learning techniques in computing systems
CO2	Evaluate the model using metrics
CO3	Design a model using machine learning to solve a problem
CO4	Conduct experiments to solve real-world problems using appropriate machine learning techniques

Observation and report

Date: 05/04/2024

Program Title: Write a python program to import and export data using Pandas library functions.

Algorithm - snapshot from observation



Code :

```
✓ [1] import pandas as pd  
1s ✓ [2] df= pd.read_csv("/content/austinHousingData.csv")  
      df.head()
```

Output:

	zpid	city	streetAddress	zipcode	description	latitude	longitude	propertyTaxRate	garageSpaces	hasAssociation	...	numOfMiddleSchools	numOfHighSchools	avgSchoolDistance	av
0	111373431	pflugerville	14424 Lake Victor Dr	78660	14424 Lake Victor Dr, Pflugerville, TX 78660 ...	30.430632	-97.663078	1.98	2	True	...	1	1	1.266667	
1	120900430	pflugerville	1104 Strickling Dr	78660	Absolutely GORGEOUS 4 Bedroom home with 2 full...	30.432673	-97.661697	1.98	2	True	...	1	1	1.400000	
2	2084491383	pflugerville	1406 Fort Dessau Rd	78660	Under construction - estimated completion in A...	30.409748	-97.639771	1.98	0	True	...	1	1	1.200000	
3	120901374	pflugerville	1025 Strickling Dr	78660	Absolutely darling one story home in charming ...	30.432112	-97.661659	1.98	2	True	...	1	1	1.400000	
4	60134862	pflugerville	15005 Donna Jane Loop	78660	Brimming with appeal & warm livability! Sleek ...	30.437368	-97.656860	1.98	0	True	...	1	1	1.133333	

5 rows x 47 columns

Code :

```

✓ [3] url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
0s 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)

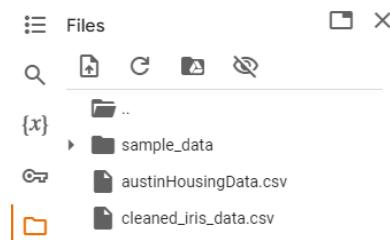
iris_data.head()

✓ ⏎ iris_data.to_csv("cleaned_iris_data.csv")

```

Output:

	sepal_length_in_cm	sepal_width_in_cm	petal_length_in_cm	petal_width_in_cm	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa



Observation and report

Date: 05/04/2024

Program Title: Demonstrate various data pre-processing techniques for a given dataset.

Algorithm - snapshot from observation

2. Data Preprocessing
 - i) Import dataset using pandas
 - ii) Perform dataset shape & analyse dataset
 - iii) Use isnull() function
 - iv) Drop or fill missing values according to your use case
 - v) We can generate dummy variables (as a binary variable that indicates whether a separate categorical variable that indicates a specific value)

Code & Output:

```
✓ [1] %matplotlib inline
    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
    import seaborn as sns
    import sklearn

✓ [3] dataset = pd.read_csv("/content/Data.csv")

✓ [4] df = pd.DataFrame(dataset)

✓ [5] df
```

	Country	Age	Salary	Purchased
0	France	44.0	72000.0	No
1	Spain	27.0	48000.0	Yes
2	Germany	30.0	54000.0	No
3	Spain	38.0	61000.0	No
4	Germany	40.0	NaN	Yes
5	France	35.0	58000.0	Yes
6	Spain	NaN	52000.0	No
7	France	48.0	79000.0	Yes
8	Germany	50.0	83000.0	No
9	France	37.0	67000.0	Yes

Next steps: [View recommended plots](#)

```
✓ [8] X = df.iloc[:, :-1].values
      y = df.iloc[:, -1].values
```

```
✓ [9] print(X)
[[['France' 44.0 72000.0]
 ['Spain' 27.0 48000.0]
 ['Germany' 30.0 54000.0]
 ['Spain' 38.0 61000.0]
 ['Germany' 40.0 nan]
 ['France' 35.0 58000.0]
 ['Spain' nan 52000.0]
 ['France' 48.0 79000.0]
 ['Germany' 50.0 83000.0]
 ['France' 37.0 67000.0]]
```

```
✓ [10] print(y)
[ 'No' 'Yes' 'No' 'No' 'Yes' 'Yes' 'No' 'Yes' 'No' 'Yes']
```

```
✓ [11] df.isnull().sum()
Country      0
Age         1
Salary       1
Purchased    0
dtype: int64
```

```
✓ [12] df1 = df.copy()
```

```
✓ [13] df2 = df.copy()
```

✓ 0s df2

	Country	Age	Salary	Purchased	grid icon	info icon
0	France	44.0	72000.0	No		
1	Spain	27.0	48000.0	Yes		
2	Germany	30.0	54000.0	No		
3	Spain	38.0	61000.0	No		
4	Germany	40.0	Nan	Yes		
5	France	35.0	58000.0	Yes		
6	Spain	Nan	52000.0	No		
7	France	48.0	79000.0	Yes		
8	Germany	50.0	83000.0	No		
9	France	37.0	67000.0	Yes		

Next steps: View recommended plots

✓ 0s pd.get_dummies(df2)

	Age	Salary	Country_France	Country_Germany	Country_Spain	Purchased_No	Purchased_Yes	grid icon	info icon
0	44.0	72000.0	True	False	False	True	False		
1	27.0	48000.0	False	False	True	False	True		
2	30.0	54000.0	False	True	False	True	False		
3	38.0	61000.0	False	False	True	True	False		
4	40.0	Nan	False	True	False	False	True		
5	35.0	58000.0	True	False	False	False	True		
6	Nan	52000.0	False	False	True	True	False		
7	48.0	79000.0	True	False	False	False	True		
8	50.0	83000.0	False	True	False	True	False		
9	37.0	67000.0	True	False	False	False	True		

✓ 0s # summarize the shape of the raw data
print("Before:",df1.shape)

drop rows with missing values
df1.dropna(inplace=True)

summarize the shape of the data with missing rows removed
print("After:",df1.shape)

```
Before: (10, 4)
After: (8, 4)
```

Observation and report

Date: 12/04/2024

Program Title: Write a python program for decision tree algorithm.

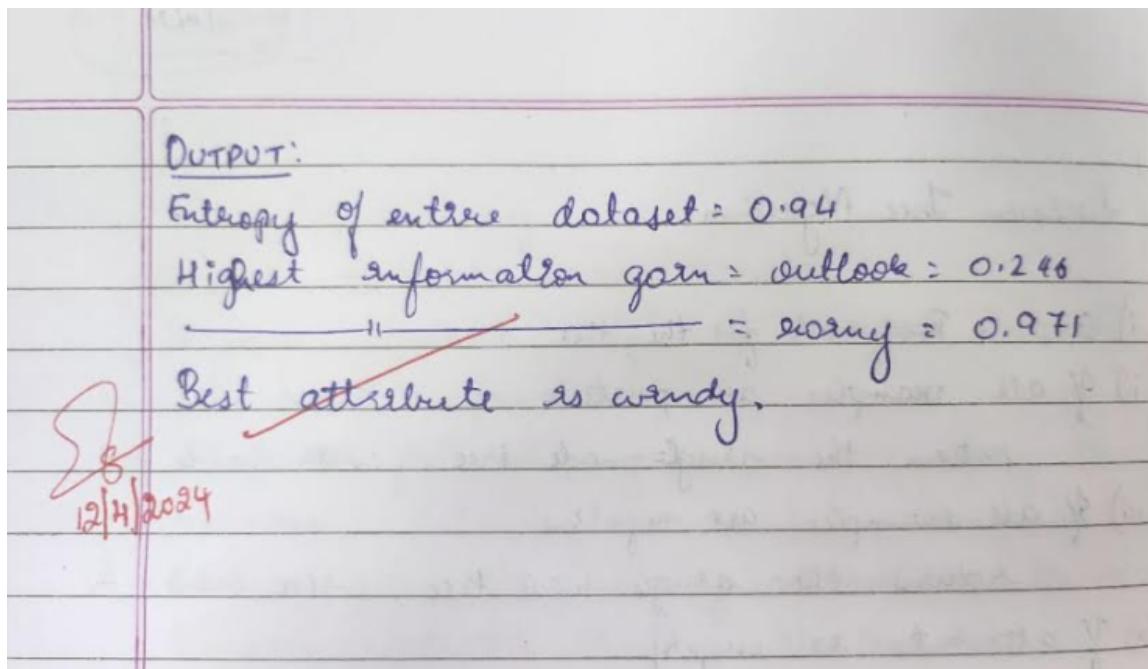
Algorithm - snapshot from observation

3. Decision Tree Algorithm

- i) Create a Root node for the tree
- ii) If all examples are positive
return the single-node tree, with label = +
- iii) If all examples are negative
return the single-node tree, with label = -
- iv) If attributes is empty
return the single node tree root, with label = most common value of Target-attribute
- v) Otherwise begin
 - A \leftarrow the attribute from attributes that best* classifies Example
 - The decision attribute of root \leftarrow A
 - For each possible value, v_i of A,
 - If Add a new tree branch below root,
corresponding to the test $A = v_i$.
 - Let Examples $_{v_i}$ be the subset of examples
that have value v_i for A.
 - If Examples $_{v_i}$ is empty
Then below this new branch add a
leaf node with label = most common
value of Target-attribute in Ex.
 - Else below this new branch add the
subtree ID3(Example $_{v_i}$, Target-attribute,
Attributes - {A})

vi) End

vii) Return Root.



Code & Output :

```

[3] import pandas as pd
import numpy as np
import math

# Reading the dataset (Tennis-dataset)
data = pd.read_csv('/content/PlayTennis.csv')

[8] def highlight(cell_value):
    ...
    Highlight yes / no values in the dataframe
    ...
    color_1 = 'background-color: pink;'
    color_2 = 'background-color: lightgreen;'

    if cell_value == 'no':
        return color_1
    elif cell_value == 'yes':
        return color_2

data.style.applymap(highlight)\n    .set_properties(subset=data.columns, **{'width': '100px'})\n    .set_table_styles([{'selector': 'th', 'props': [('background-color', 'lightgray'), ('border', '1px solid gray'), ('font-weight', 'bold')]}],\n                    {'selector': 'tr:hover', 'props': [('background-color', 'white'), ('border', '1.5px solid black')]}))
```

	outlook	temp	humidity	windy	play
0	sunny	hot	high	False	no
1	sunny	hot	high	True	no
2	overcast	hot	high	False	yes
3	rainy	mild	high	False	yes
4	rainy	cool	normal	False	yes
5	rainy	cool	normal	True	no
6	overcast	cool	normal	True	yes
7	sunny	mild	high	False	no
8	sunny	cool	normal	False	yes
9	rainy	mild	normal	False	yes
10	sunny	mild	normal	True	yes
11	overcast	mild	high	True	yes
12	overcast	hot	normal	False	yes
13	rainy	mild	high	True	no

```

✓ [9] def find_entropy(data):
    """
        Returns the entropy of the class or features
        formula: - Σ P(X)logP(X)
    """
    entropy = 0
    for i in range(data.nunique()):
        x = data.value_counts()[i]/data.shape[0]
        entropy += (- x * math.log(x,2))
    return round(entropy,3)

def information_gain(data, data_):
    """
        Returns the information gain of the features
    """
    info = 0
    for i in range(data_.nunique()):
        df = data[data_ == data_.unique()[i]]
        w_avg = df.shape[0]/data.shape[0]
        entropy = find_entropy(df.play)
        x = w_avg * entropy
        info += x
    ig = find_entropy(data.play) - info
    return round(ig, 3)

def entropy_and_infogain(datax, feature):
    """
        Grouping features with the same class and computing their
        entropy and information gain for splitting
    """
    for i in range(data[feature].nunique()):
        df = datax[datax[feature]==data[feature].unique()[i]]
        if df.shape[0] < 1:
            continue

        display(df[[feature, 'play']].style.applymap(highlight)\.
            .set_properties(subset=[feature, 'play'], **{'width': '80px'})\.
            .set_table_styles([{'selector': 'th', 'props': [('background-color', 'lightgray'),
                ('border', '1px solid gray'),
                ('font-weight', 'bold')]}],
                {'selector': 'td', 'props': [((border', '1px solid gray'))],
                {'selector': 'tr:hover', 'props': [('background-color', 'white'),
                    ('border', '1.5px solid black')]}]))]

        print(f'Entropy of {feature} - {data[feature].unique()[i]} = {find_entropy(df.play)}')
        print(f'Information Gain for {feature} = {information_gain(datax, datax[feature])}')


[10] print(f'Entropy of the entire dataset: {find_entropy(data.play)}')

```

Entropy of the entire dataset: 0.94

```
✓ 0s ⏴ [11] entropy_and_infogain(data, 'outlook')
```

	outlook	play
0	sunny	no
1	sunny	no
7	sunny	no
8	sunny	yes
10	sunny	yes

Entropy of outlook - sunny = 0.971

	outlook	play
2	overcast	yes
6	overcast	yes
11	overcast	yes
12	overcast	yes

Entropy of outlook - overcast = 0.0

	outlook	play
3	rainy	yes
4	rainy	yes
5	rainy	no
9	rainy	yes
13	rainy	no

Entropy of outlook - rainy = 0.971

Information Gain for outlook = 0.246

```
✓ 0s [12] entropy_and_infogain(data, 'temp')
```

	temp	play
0	hot	no
1	hot	no
2	hot	yes
12	hot	yes

Entropy of temp - hot = 1.0

	temp	play
3	mild	yes
7	mild	no
9	mild	yes
10	mild	yes
11	mild	yes
13	mild	no

Entropy of temp - mild = 0.918

	temp	play
4	cool	yes
5	cool	no
6	cool	yes
8	cool	yes

Entropy of temp - cool = 0.811

Information Gain for temp = 0.029

```
✓ 0s [13] entropy_and_infogain(data, 'humidity')
```

	humidity	play
0	high	no
1	high	no
2	high	yes
3	high	yes
7	high	no
11	high	yes
13	high	no

Entropy of humidity - high = 0.985

	humidity	play
4	normal	yes
5	normal	no
6	normal	yes
8	normal	yes
9	normal	yes
10	normal	yes
12	normal	yes

Entropy of humidity - normal = 0.592
Information Gain for humidity = 0.151

```
✓ 0s [14] entropy_and_infogain(data, 'windy')
```

	windy	play
0	False	no
2	False	yes
3	False	yes
4	False	yes
7	False	no
8	False	yes
9	False	yes
12	False	yes

Entropy of windy - False = 0.811

	windy	play
1	True	no
5	True	no
6	True	yes
10	True	yes
11	True	yes
13	True	no

Entropy of windy - True = 1.0
Information Gain for windy = 0.048

```

✓ [15] sunny = data[data['outlook'] == 'sunny']
sunny.style.applymap(highlight)\n    .set_properties(subset=data.columns, **{'width': '100px'})\n    .set_table_styles([{'selector': 'th', 'props': [('background-color', 'lightgray'), ('border', '1px solid gray'), ('font-weight', 'bold')]}],\n                    {'selector': 'tr:hover', 'props': [('background-color', 'white'), ('border', '1.5px solid black')]}])

```

	outlook	temp	humidity	windy	play
0	sunny	hot	high	False	no
1	sunny	hot	high	True	no
7	sunny	mild	high	False	no
8	sunny	cool	normal	False	yes
10	sunny	mild	normal	True	yes

```

✓ [16] print(f'Entropy of the Sunny dataset: {find_entropy(sunny.play)}')

```

Entropy of the Sunny dataset: 0.971

```

✓ [17] entropy_and_infogain(sunny, 'temp')

```

	temp	play
0	hot	no
1	hot	no

Entropy of temp - hot = 0.0

	temp	play
7	mild	no
10	mild	yes

Entropy of temp - mild = 1.0

	temp	play
8	cool	yes

Entropy of temp - cool = 0.0

Information Gain for temp = 0.571

```

✓ [18] entropy_and_infogain(sunny, 'humidity')

```

	humidity	play
0	high	no
1	high	no
7	high	no

Entropy of humidity - high = 0.0

	humidity	play
8	normal	yes
10	normal	yes

Entropy of humidity - normal = 0.0

Information Gain for humidity = 0.971

```

✓ [19] entropy_and_infogain(sunny, 'windy')


|   | windy | play |
|---|-------|------|
| 0 | False | no   |
| 7 | False | no   |
| 8 | False | yes  |


Entropy of windy - False = 0.918


|    | windy | play |
|----|-------|------|
| 1  | True  | no   |
| 10 | True  | yes  |


Entropy of windy - True = 1.0
Information Gain for windy = 0.02

✓ [20] rainy = data[data['outlook'] == 'rainy']
rainy.style.applymap(highlight)\n    .set_properties(subset=data.columns, **{'width': '100px'})\\
    .set_table_styles([{'selector': 'th', 'props': [('background-color', 'lightgray'), ('border', '1px solid gray'), ('font-weight', 'bold')]},\n                    {'selector': 'tr:hover', 'props': [('background-color', 'white'), ('border', '1.5px solid black')]})]


|    | outlook | temp | humidity | windy | play |
|----|---------|------|----------|-------|------|
| 3  | rainy   | mild | high     | False | yes  |
| 4  | rainy   | cool | normal   | False | yes  |
| 5  | rainy   | cool | normal   | True  | no   |
| 9  | rainy   | mild | normal   | False | yes  |
| 13 | rainy   | mild | high     | True  | no   |


✓ [21] print(f'Entropy of the Rainy dataset: {find_entropy(rainy.play)}')
Entropy of the Rainy dataset: 0.971

✓ [22] entropy_and_infogain(rainy, 'temp')


|    | temp | play |
|----|------|------|
| 3  | mild | yes  |
| 9  | mild | yes  |
| 13 | mild | no   |


Entropy of temp - mild = 0.918


|   | temp | play |
|---|------|------|
| 4 | cool | yes  |
| 5 | cool | no   |


Entropy of temp - cool = 1.0
Information Gain for temp = 0.02

✓ [23] entropy_and_infogain(rainy, 'humidity')


|    | humidity | play |
|----|----------|------|
| 3  | high     | yes  |
| 13 | high     | no   |


Entropy of humidity - high = 1.0


|   | humidity | play |
|---|----------|------|
| 4 | normal   | yes  |
| 5 | normal   | no   |
| 9 | normal   | yes  |


Entropy of humidity - normal = 0.918
Information Gain for humidity = 0.02

```

```
✓ [24] entropy_and_infogain(rainy, 'windy')
```

	windy	play
3	False	yes
4	False	yes
9	False	yes

Entropy of windy - False = 0.0

	windy	play
5	True	no
13	True	no

Entropy of windy - True = 0.0

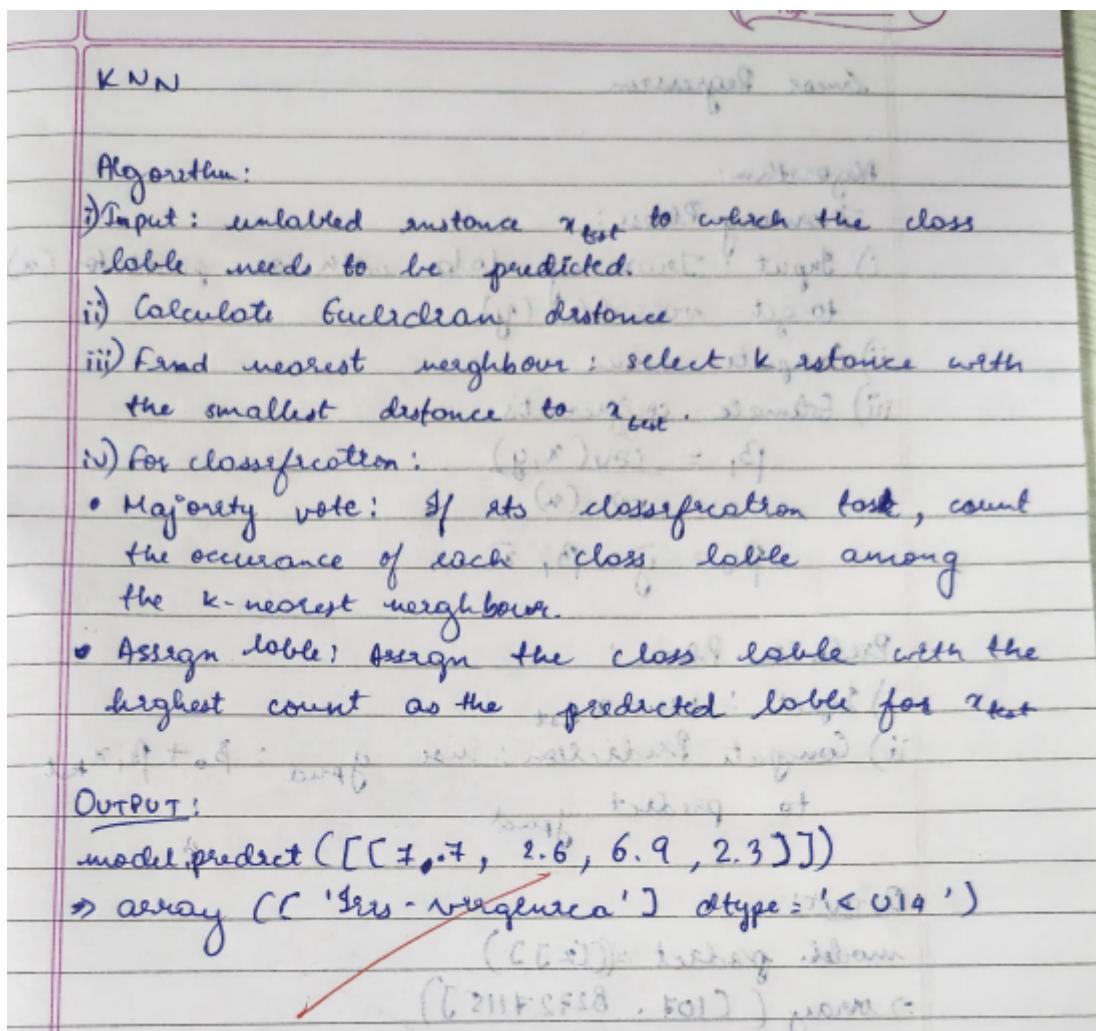
Information Gain for windy = 0.971

Observation and report

Date: 03/05/2024

Program Title: Build KNN Classification model for a given dataset.

Algorithm - snapshot from observation



Code&Output :

```
✓ [1] import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt
    import plotly.express as px
    import seaborn as sns

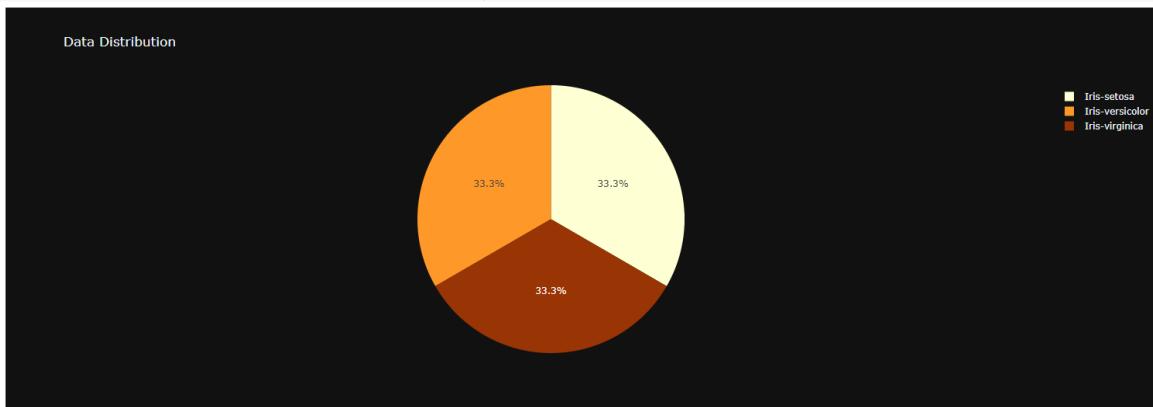
✓ [4] iris = pd.read_csv("/content/Iris.csv") #Load Data
      iris.drop('Id',inplace=True,axis=1) #Drop Id column

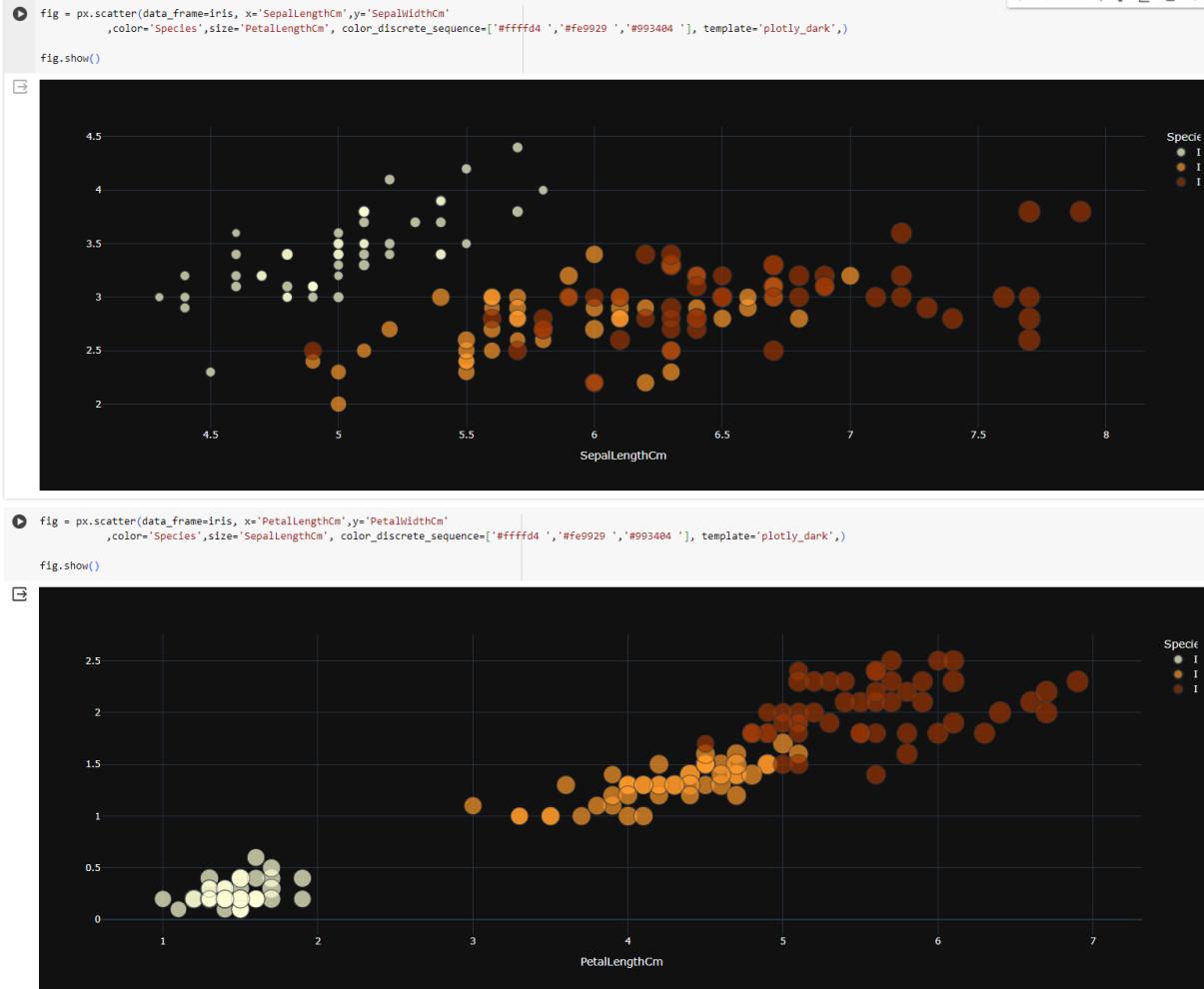
✓ [5] iris.head().style.background_gradient(sns.color_palette("YlOrBr", as_cmap=True))
```

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.100000	3.500000	1.400000	0.200000	Iris-setosa
1	4.900000	3.000000	1.400000	0.200000	Iris-setosa
2	4.700000	3.200000	1.300000	0.200000	Iris-setosa
3	4.600000	3.100000	1.500000	0.200000	Iris-setosa
4	5.000000	3.600000	1.400000	0.200000	Iris-setosa

```
✓ [6] X = iris.iloc[:, :-1] #Set our training data
      y = iris.iloc[:, -1] #Set training labels
```

```
✓ [7] fig = px.pie(iris, 'Species',color_discrete_sequence=['#ffffd4 ','#fe9929 ','#993404 '],title='Data Distribution',template='plotly_dark')
      fig.show()
```





```

[10] class KNN:
    """
    K-Nearest Neighbors (KNN) classification algorithm

    Parameters:
    -----
    n_neighbors : int, optional (default=5)
        Number of neighbors to use in the majority vote.

    Methods:
    -----
    fit(X_train, y_train):
        Stores the values of X_train and y_train.

    predict(X):
        Predicts the class labels for each example in X.

    """
    def __init__(self, n_neighbors=5):
        self.n_neighbors = n_neighbors

    def euclidean_distance(self, x1, x2):
        """
        Calculate the Euclidean distance between two data points.

        Parameters:
        -----
        x1 : numpy.ndarray, shape (n_features,)
            A data point in the dataset.

        x2 : numpy.ndarray, shape (n_features,)
            A data point in the dataset.

        Returns:
        -----
        distance : float
            The Euclidean distance between x1 and x2.
        """
        return np.linalg.norm(x1 - x2)

    def fit(self, X_train, y_train):
        """
        Stores the values of X_train and y_train.

        Parameters:
        -----
        X_train : numpy.ndarray, shape (n_samples, n_features)
            The training dataset.

        y_train : numpy.ndarray, shape (n_samples,)
            The target labels.
        """
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X):
        """
        Predicts the class labels for each example in X.

        Parameters:
        -----
        X : numpy.ndarray, shape (n_samples, n_features)
            The test dataset.

        Returns:
        -----
        predictions : numpy.ndarray, shape (n_samples,)
            The predicted class labels for each example in X.
        """
        # Create empty array to store the predictions
        predictions = []
        # Loop over X examples
        for x in X:
            # Get prediction using the prediction helper function
            prediction = self._predict(x)
            # Append the prediction to the predictions list
            predictions.append(prediction)
        return np.array(predictions)

    def _predict(self, x):
        """
        Predicts the class label for a single example.

        Parameters:
        -----
        x : numpy.ndarray, shape (n_features,)
            A single data point.
        """
        # Calculate distances from x to all training samples
        distances = self.euclidean_distance(x, self.X_train)
        # Find indices of the n_neighbors closest samples
        nearest_indices = np.argsort(distances)[:self.n_neighbors]
        # Extract the labels of these samples
        nearest_labels = self.y_train[nearest_indices]
        # Count the occurrences of each class
        counts = np.bincount(nearest_labels)
        # Find the most frequent class
        predicted_label = np.argmax(counts)
        return predicted_label

```

```

[10]     -----
      x : numpy.ndarray, shape (n_features,)
      A data point in the test dataset.

      Returns:
      -----
      most_occurring_value : int
          The predicted class label for x.
      """

      # Create empty array to store distances
      distances = []
      # Loop over all training examples and compute the distance between x and all the training examples
      for x_train in self.X_train:
          distance = self.euclidean_distance(x, x_train)
          distances.append(distance)
      distances = np.array(distances)

      # Sort by ascendingly distance and return indices of the given n neighbours
      n_neighbors_idxs = np.argsort(distances)[: self.n_neighbors]

      # Get labels of n-neighbour indexes
      labels = self.y_train[n_neighbors_idxs]
      labels = list(labels)
      # Get the most frequent class in the array
      most_occurring_value = max(labels, key=labels.count)
      return most_occurring_value

[11] def train_test_split(X, y, random_state=42, test_size=0.2):
    """
    Splits the data into training and testing sets.

    Parameters:
        X (numpy.ndarray): Features array of shape (n_samples, n_features).
        y (numpy.ndarray): Target array of shape (n_samples,).
        random_state (int): Seed for the random number generator. Default is 42.
        test_size (float): Proportion of samples to include in the test set. Default is 0.2.

    Returns:
        Tuple[numpy.ndarray]: A tuple containing X_train, X_test, y_train, y_test.
    """
    # Get number of samples
    n_samples = X.shape[0]

    # Set the seed for the random number generator
    np.random.seed(random_state)

    # Shuffle the indices
    shuffled_indices = np.random.permutation(np.arange(n_samples))

    # Determine the size of the test set
    test_size = int(n_samples * test_size)

    # Split the indices into test and train
    test_indices = shuffled_indices[:test_size]
    train_indices = shuffled_indices[test_size:]

    # Split the features and target arrays into test and train
    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test

```

```

[12] X_train, X_test, y_train, y_test = train_test_split(X.values, y.values, test_size = 0.2, random_state=42) #split the data into traing and validating

[13] model = KNN(7)
model.fit(X_train, y_train)

[14] def compute_accuracy(y_true, y_pred):
"""
    Computes the accuracy of a classification model.

    Parameters:
    y_true (numpy array): A numpy array of true labels for each data point.
    y_pred (numpy array): A numpy array of predicted labels for each data point.

    Returns:
    float: The accuracy of the model, expressed as a percentage.
"""
    y_true = y_true.flatten()
    total_samples = len(y_true)
    correct_predictions = np.sum(y_true == y_pred)
    return (correct_predictions / total_samples)

[15]
predictions = model.predict(X_test)
accuracy = compute_accuracy(y_test, predictions)
print(f" our model got accuracy score of : {accuracy}")

our model got accuracy score of : 0.9666666666666667

[18] from sklearn.neighbors import KNeighborsClassifier
skmodel = KNeighborsClassifier(n_neighbors=7)
skmodel.fit(X_train, y_train)

+     KNeighborsClassifier
KNeighborsClassifier(n_neighbors=7)

[19] sk_predictions = skmodel.predict(X_test)
sk_accuracy = compute_accuracy(y_test, sk_predictions)
print(f" sklearn-model got accuracy score of : {sk_accuracy}")

sklearn-model got accuracy score of : 0.9666666666666667

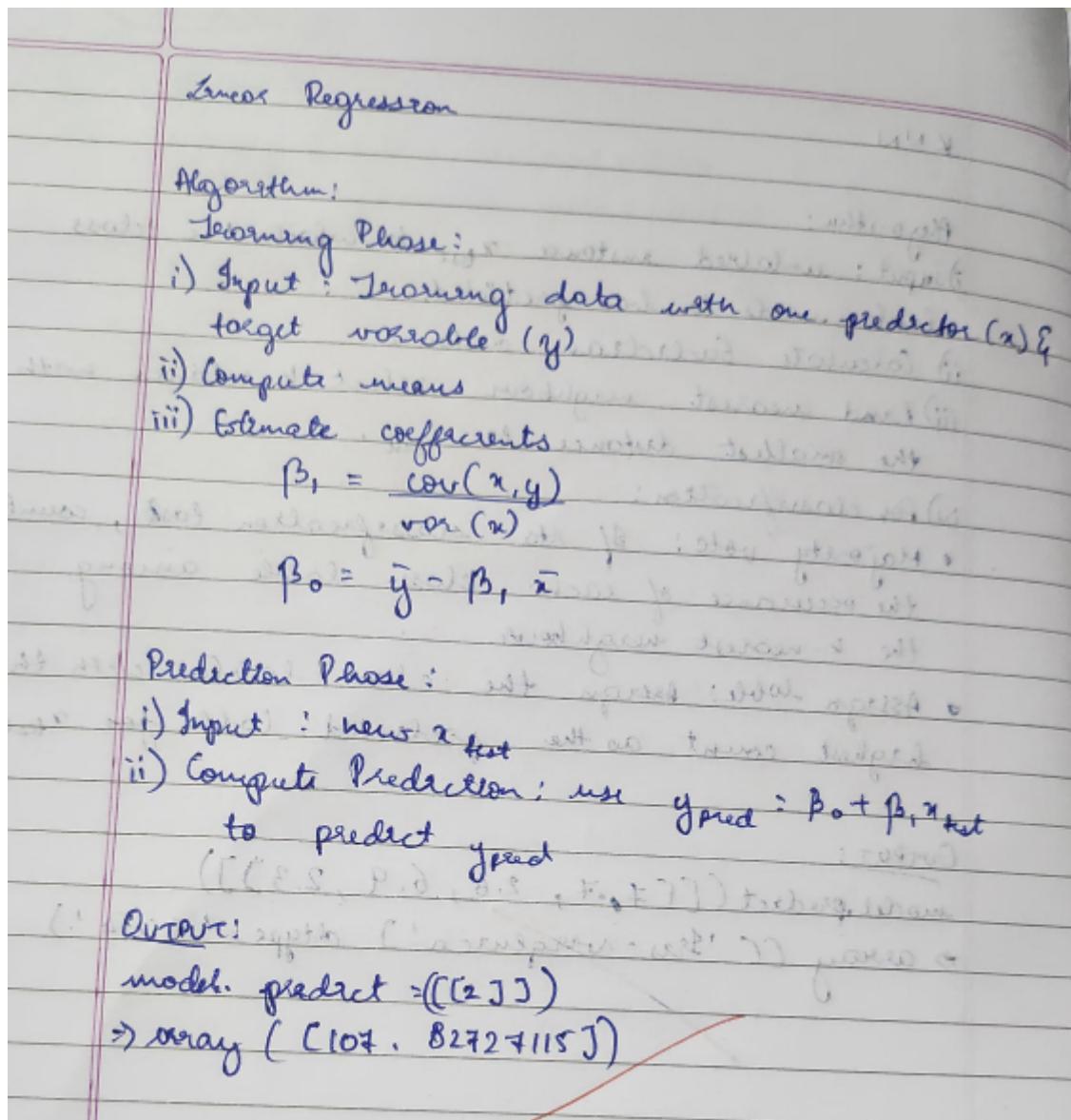
```

Observation and report

Date: 03/05/2024

Program Title: Implement Linear Regression algorithm using appropriate dataset

Algorithm - snapshot from observation



Code & Output:

```
[1] import math
import numpy as np
import pandas as pd
import plotly.express as px
import pickle

[2] # Load the training and test datasets
train_data = pd.read_csv('/content/train.csv')
test_data = pd.read_csv('/content/test.csv')

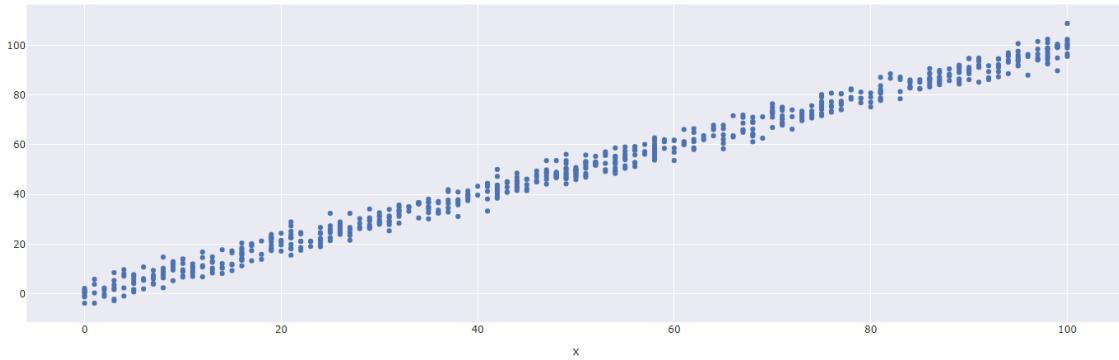
# Remove rows with missing values
train_data = train_data.dropna()
test_data = test_data.dropna()
```

```
[3] train_data.head()
```

	x	y
0	24.0	21.549452
1	50.0	47.464463
2	15.0	17.218656
3	38.0	36.586398
4	87.0	87.288984

```
px.scatter(x=train_data['x'], y=train_data['y'], template='seaborn')
```

```
✉
```



```
[5] # Set training data and target
X_train = train_data['x'].values
y_train = train_data['y'].values

# Set testing data and target
X_test = test_data['x'].values
y_test = test_data['y'].values

[6] def standardize_data(X_train, X_test):
    """
    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)

▶ 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:
        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).

        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.

        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()

    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.

        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

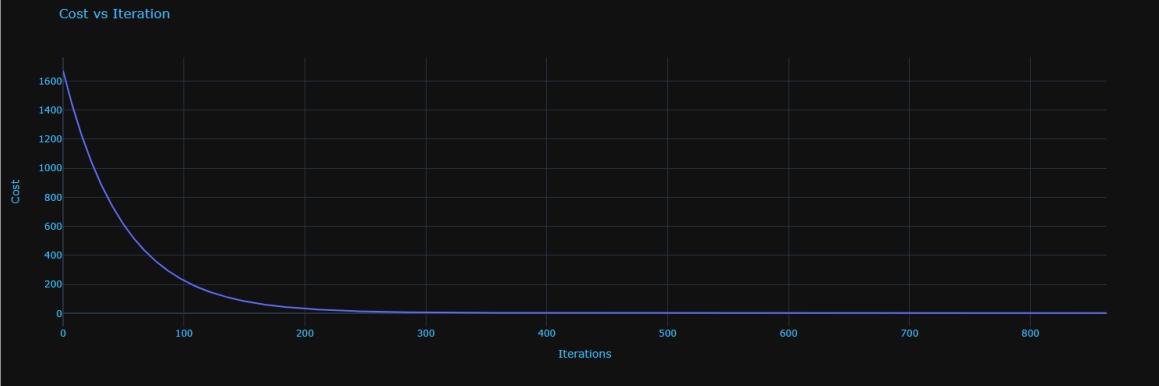
```

```

[8] lr = LinearRegression(0.01)
lr.fit(X_train, y_train, 10000)

[9]
Iteration: 0, Cost: 1670.4939835422822
Iteration: 100, Cost: 227.2198576573727
Iteration: 200, Cost: 33.84955235666352
Iteration: 300, Cost: 7.478079323294246
Iteration: 400, Cost: 4.478079323294246
Iteration: 500, Cost: 4.00582304545451
Iteration: 600, Cost: 3.9435150866540726
Iteration: 700, Cost: 3.93404940109863
Iteration: 800, Cost: 3.93404940109863
Converged after 863 iterations.

```



```
[10] lr.save_model('model.pkl')
```

```
[11] model = LinearRegression.load_model("model.pkl")
```

```

[12] class RegressionMetrics:
    @staticmethod
    def mean_squared_error(y_true, y_pred):
        """
        Calculate the Mean Squared Error (MSE).

        Args:
            y_true (numpy.ndarray): The true target values.
            y_pred (numpy.ndarray): The predicted target values.

        Returns:
            float: The Mean Squared Error.
        """
        assert len(y_true) == len(y_pred), "Input arrays must have the same length."
        mse = np.mean((y_true - y_pred) ** 2)
        return mse

    @staticmethod
    def root_mean_squared_error(y_true, y_pred):
        """
        Calculate the Root Mean Squared Error (RMSE).

        Args:
            y_true (numpy.ndarray): The true target values.
            y_pred (numpy.ndarray): The predicted target values.

        Returns:
            float: The Root Mean Squared Error.
        """
        assert len(y_true) == len(y_pred), "Input arrays must have the same length."
        mse = RegressionMetrics.mean_squared_error(y_true, y_pred)
        rmse = np.sqrt(mse)

```

```

        return rmse

    @staticmethod
    def r_squared(y_true, y_pred):
        """
        Calculate the R-squared (R^2) coefficient of determination.

        Args:
            y_true (numpy.ndarray): The true target values.
            y_pred (numpy.ndarray): The predicted target values.

        Returns:
            float: The R-squared (R^2) value.
        """
        assert len(y_true) == len(y_pred), "Input arrays must have the same length."
        mean_y = np.mean(y_true)
        ss_total = np.sum((y_true - mean_y) ** 2)
        ss_residual = np.sum((y_true - y_pred) ** 2)
        r2 = 1 - (ss_residual / ss_total)
        return r2

```

```

[13] y_pred = model.predict(X_test)
mse_value = RegressionMetrics.mean_squared_error(y_test, y_pred)
rmse_value = RegressionMetrics.root_mean_squared_error(y_test, y_pred)
r_squared_value = RegressionMetrics.r_squared(y_test, y_pred)

print(f"Mean Squared Error (MSE): {mse_value}")
print(f"Root Mean Squared Error (RMSE): {rmse_value}")
print(f"R-squared (Coefficient of Determination): {r_squared_value}")

```

Mean Squared Error (MSE): 9.442671868011633
Root Mean Squared Error (RMSE): 3.072893077868417
R-squared (Coefficient of Determination): 0.9887898698341414

```

[12]     @staticmethod
    def r_squared(y_true, y_pred):
        """
        Calculate the R-squared (R^2) coefficient of determination.

        Args:
            y_true (numpy.ndarray): The true target values.
            y_pred (numpy.ndarray): The predicted target values.

        Returns:
            float: The R-squared (R^2) value.
        """
        assert len(y_true) == len(y_pred), "Input arrays must have the same length."
        mean_y = np.mean(y_true)
        ss_total = np.sum((y_true - mean_y) ** 2)
        ss_residual = np.sum((y_true - y_pred) ** 2)
        r2 = 1 - (ss_residual / ss_total)
        return r2

```

```

[13] y_pred = model.predict(X_test)
mse_value = RegressionMetrics.mean_squared_error(y_test, y_pred)
rmse_value = RegressionMetrics.root_mean_squared_error(y_test, y_pred)
r_squared_value = RegressionMetrics.r_squared(y_test, y_pred)

print(f"Mean Squared Error (MSE): {mse_value}")
print(f"Root Mean Squared Error (RMSE): {rmse_value}")
print(f"R-squared (Coefficient of Determination): {r_squared_value}")

```

Mean Squared Error (MSE): 9.44266965025894
Root Mean Squared Error (RMSE): 3.07289271701095
R-squared (Coefficient of Determination): 0.9887898724670081

```

▶ model.predict([[2]])
array([107.82727115])

```

```
✓ [24] model = LogisticRegression.load_model("model.pkl")  
  
✓ [25] y_pred = model.predict(X_test)  
      accuracy = ClassificationMetrics.accuracy(y_test, y_pred)  
      precision = ClassificationMetrics.precision(y_test, y_pred)  
      recall = ClassificationMetrics.recall(y_test, y_pred)  
      f1_score = ClassificationMetrics.f1_score(y_test, y_pred)  
  
      print(f"Accuracy: {accuracy:.2%}")  
      print(f"Precision: {precision:.2%}")  
      print(f"Recall: {recall:.2%}")  
      print(f"F1-Score: {f1_score:.2%}")  
  
Accuracy: 98.23%  
Precision: 100.00%  
Recall: 95.24%  
F1-Score: 97.56%
```

Observation and report

Date: 03/05/2024

Program Title: Build Logistic Regression Model for a given dataset

Algorithm - snapshot from observation

The image shows a handwritten note on a lined notebook page. At the top right, there is a circular stamp with the date "3/5/24" and the word "Page". The main title "Logistic Regression" is written in cursive at the top left. Below it, the word "Algorithm:" is followed by a list of steps for the "Training phase":

- i) Inputs & Initialize parameter: Start with random or zero values for weights (A) & bias (b)
- ii) Compute peer predictions: Calculate predictions using the equation $\hat{y} = \sigma(A^T x + b)$, where σ is the sigmoid function.
- iii) Compute loss: measure the error between prediction & actual to label using binary cross-entropy loss.
- iv) Update parameters: adjust weights & bias using gradient descent to minimize the loss.
- v) repeat: Iterate step 2-4 until convergence or a max no. of iterations.

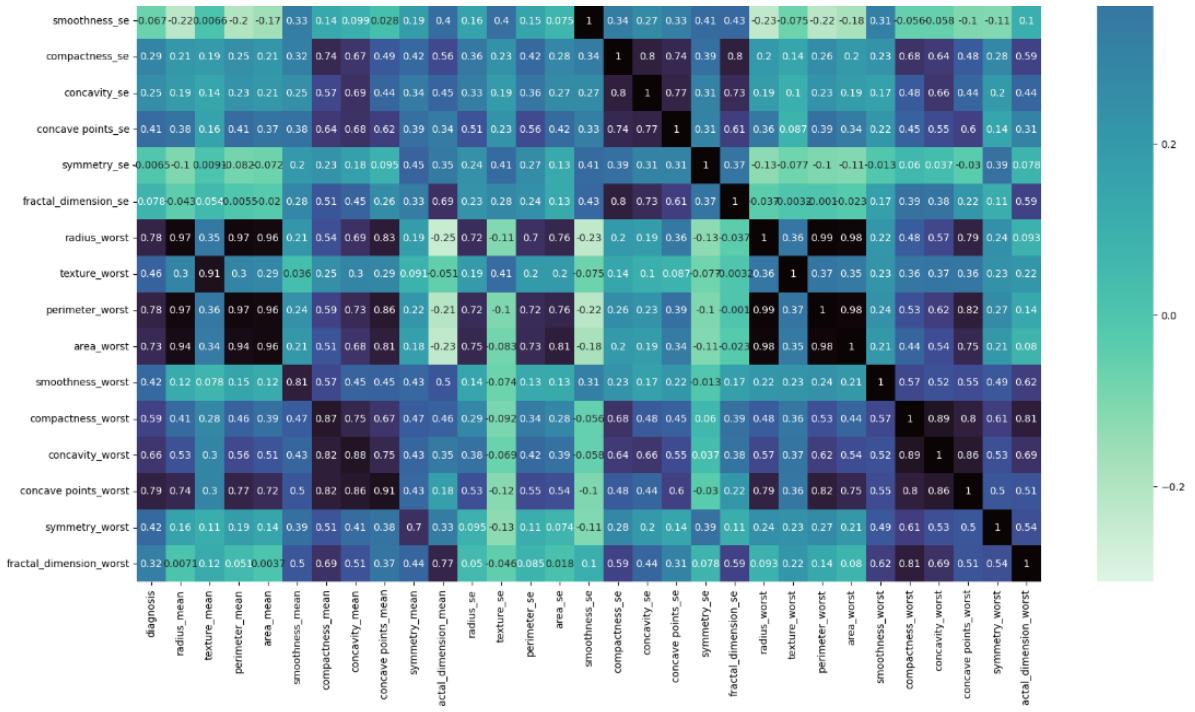
Below this, the word "Prediction Phase:" is written, followed by:

- i) Compute the prediction if the probability is above 0.5 classify as 1 otherwise 0.

In the bottom left corner, there is a red handwritten signature and the date "3/5/2024".

Code & Output:





```
[12] # Get the absolute value of the correlation
cor_target = abs(corr["diagnosis"])

# Select highly correlated features (threshold = 0.2)
relevant_features = cor_target[cor_target>0.2]

# Collect the names of the features
names = [index for index, value in relevant_features.items()]

# Drop the target variable from the results
names.remove('diagnosis')

# Display the results
pprint.pprint(names)

[13] ['radius_mean',
'texture_mean',
'perimeter_mean',
'area_mean',
'smoothness_mean',
'compactness_mean',
'concavity_mean',
'concave points_mean',
'symmetry_mean',
'radius_se',
'perimeter_se',
'area_se',
'compactness_se',
'concavity_se',
'concave points_se',
'radius_worst',
'texture_worst',
'perimeter_worst',
'area_worst',
'smoothness_worst',
'compactness_worst',
'concavity_worst',
'concave points_worst',
'symmetry_worst',
'fractal_dimension_worst']

[14] X = df[names].values
y = df['diagnosis'].values

[15] def train_test_split(X, y, random_state=42, test_size=0.2):
    """
    Splits the data into training and testing sets.

    Parameters:
        X (numpy.ndarray): Features array of shape (n_samples, n_features).
        y (numpy.ndarray): Target array of shape (n_samples,).
        random_state (int): Seed for the random number generator. Default is 42.
        test_size (float): Proportion of samples to include in the test set. Default is 0.2.

    Returns:
        Tuple[numpy.ndarray]: A tuple containing X_train, X_test, y_train, y_test.
    """
    # Get number of samples
    n_samples = X.shape[0]

    # Set the seed for the random number generator
    np.random.seed(random_state)

    # Shuffle the indices
    shuffled_indices = np.random.permutation(np.arange(n_samples))

    # Determine the size of the test set
    test_size = int(n_samples * test_size)

    # Split the indices into test and train
    test_indices = shuffled_indices[:test_size]
    train_indices = shuffled_indices[test_size:]

    # Split the features and target arrays into test and train
    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test

[16] X_train, X_test, y_train, y_test = train_test_split(X,y)

[17] def standardize_data(X_train, X_test):
    """
    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)
```

```
[17] def sigmoid(z):
    """
    Compute the sigmoid function for a given input.

    The sigmoid function is a mathematical function used in logistic regression and neural networks
    to map any real-valued number to a value between 0 and 1.

    Parameters:
        z (float or numpy.ndarray): The input value(s) for which to compute the sigmoid.

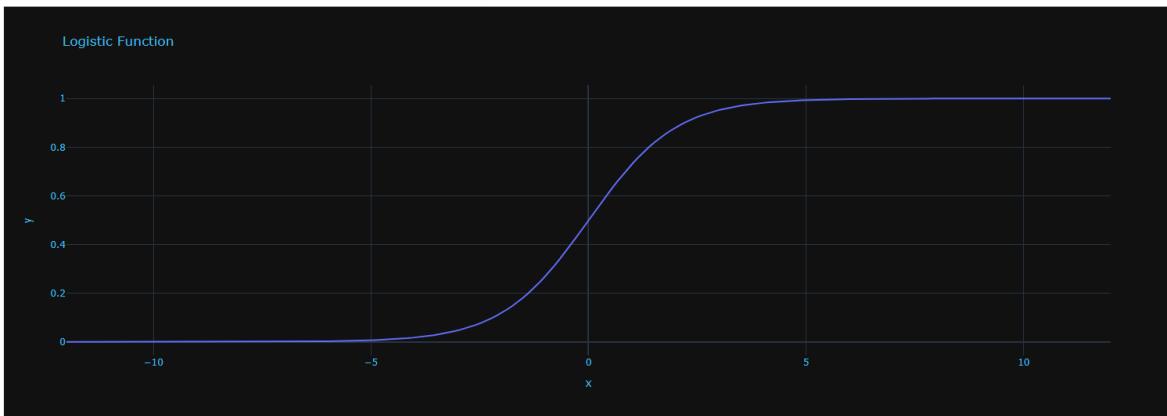
    Returns:
        float or numpy.ndarray: The sigmoid of the input value(s).

    Example:
        >>> sigmoid(0)
        0.5
    """
    # Compute the sigmoid function using the formula: 1 / (1 + e^(-z)).
    sigmoid_result = 1 / (1 + np.exp(-z))

    # Return the computed sigmoid value.
    return sigmoid_result

[18] z = np.linspace(-12, 12, 200)
```

```
fig = px.line(x=z, y=sigmoid(z), title='Logistic Function', template="plotly_dark")
fig.update_layout(
    title_font_color="#41BEE9",
    xaxis=dict(color="#41BEE9"),
    yaxis=dict(color="#41BEE9")
)
fig.show()
```



```
[19]  class LogisticRegression:
        """
        Logistic Regression model.

        Parameters:
            learning_rate (float): Learning rate for the model.

        Methods:
            initialize_parameter(): Initializes the parameters of the model.
            sigmoid(z): Computes the sigmoid activation function for given input z.
            forward(X): Computes forward propagation for given input X.
            compute_cost(predictions): Computes the cost function for given predictions.
            compute_gradient(predictions): Computes the gradients for the model using given predictions.
            fit(X, y, iterations, plot_cost): Trains the model on given input X and labels y for specified iterations.
            predict(X): Predicts the labels for given input X.
        """

        def __init__(self, learning_rate=0.0001):
            np.random.seed(1)
            self.learning_rate = learning_rate

        def initialize_parameter(self):
            """
            Initializes the parameters of the model.
            """
            self.W = np.zeros(self.X.shape[1])
            self.b = 0.0

        def forward(self, X):
            """
            Computes forward propagation for given input X.

            Parameters:
                X (numpy.ndarray): Input array.

            Returns:
                numpy.ndarray: Output array.
            """
            # print(X.shape, self.W.shape)
            Z = np.matmul(X, self.W) + self.b
            A = sigmoid(Z)
            return A

        def compute_cost(self, predictions):
            """
            Computes the cost function for given predictions.

            Parameters:
                predictions (numpy.ndarray): Predictions of the model.
            """

```

```

    Returns:
        float: Cost of the model.
    """
    m = self.X.shape[0] # number of training examples
    # compute the cost
    cost = np.sum((-np.log(predictions + 1e-8) * self.y) + (-np.log(1 - predictions + 1e-8)) * (
        1 - self.y)) # we are adding small value epsilon to avoid log of 0
    cost = cost / m
    return cost

    def compute_gradient(self, predictions):
        """
        Computes the gradients for the model using given predictions.

        Parameters:
            predictions (numpy.ndarray): Predictions of the model.
        """
        # get training shape
        m = self.X.shape[0]

        # compute gradients
        self.dW = np.matmul(self.X.T, (predictions - self.y))
        self.dW = np.array([np.mean(grad) for grad in self.dW])

        self.db = np.sum(np.subtract(predictions, self.y))

        # scale gradients
        self.dW = self.dW * 1 / m
        self.db = self.db * 1 / m

    def fit(self, X, y, iterations, plot_cost=True):
        """
        Trains the model on given input X and labels y for specified iterations.

        Parameters:
            X (numpy.ndarray): Input features array of shape (n_samples, n )
            y (numpy.ndarray): Labels array of shape (n_samples, 1)
            iterations (int): Number of iterations for training.
            plot_cost (bool): Whether to plot cost over iterations or not.

        Returns:
            None.
        """
        self.X = X
        self.y = y

        self.initialize_parameter()

        costs = []
        for i in range(iterations):
            # forward propagation
            predictions = self.forward(self.X)

            # compute cost
            cost = self.compute_cost(predictions)
            costs.append(cost)

```

```

        # compute gradients
        self.compute_gradient(predictions)

        # update parameters
        self.W = self.W - self.learning_rate * self.dW
        self.b = self.b - self.learning_rate * self.db

        # print cost every 100 iterations
        if i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))

    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()

    def predict(self, X):
        """
        Predicts the labels for given input X.

        Parameters:
            X (numpy.ndarray): Input features array.

        Returns:
            numpy.ndarray: Predicted labels.
        """
        predictions = self.forward(X)
        return np.round(predictions)

    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,
            '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.

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

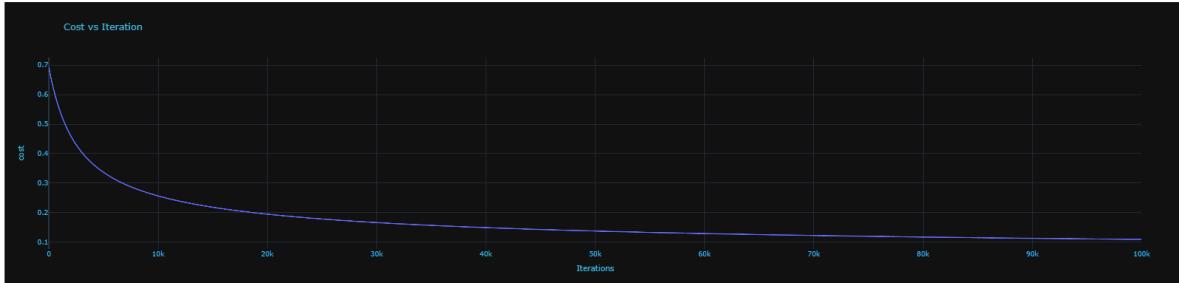
        # Create a new instance of the class and initialize it with the loaded parameters
        loaded_model = cls(model_data['learning_rate'])
        loaded_model.W = model_data['W']
        loaded_model.b = model_data['b']

        return loaded_model

```

[28]: lg = LogisticRegression()
lg.fit(X_train, y_train, 100000)

Cost after iteration 0: 0.693471609599454
Cost after iteration 10000: 0.257977370558246
Cost after iteration 20000: 0.19529178673689726
Cost after iteration 30000: 0.15311701344031552
Cost after iteration 40000: 0.1497939546876498
Cost after iteration 50000: 0.13818761344031554
Cost after iteration 60000: 0.13031876134403154
Cost after iteration 70000: 0.1231144039988139
Cost after iteration 80000: 0.11795163708798082
Cost after iteration 90000: 0.113513771308802



```

[21] lg.save_model("model.pkl")

[22] class ClassificationMetrics:
    @staticmethod
    def accuracy(y_true, y_pred):
        """
        Computes the accuracy of a classification model.

        Parameters:
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        float: The accuracy of the model, expressed as a percentage.
        """
        y_true = y_true.flatten()
        total_samples = len(y_true)
        correct_predictions = np.sum(y_true == y_pred)
        return (correct_predictions / total_samples)

    @staticmethod
    def precision(y_true, y_pred):
        """
        Computes the precision of a classification model.

        Parameters:
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        float: The precision of the model, which measures the proportion of true positive prediction
        out of all positive predictions made by the model.
        """
        true_positives = np.sum((y_true == 1) & (y_pred == 1))
        false_positives = np.sum((y_true == 0) & (y_pred == 1))
        return true_positives / (true_positives + false_positives)

    @staticmethod
    def recall(y_true, y_pred):
        """
        Computes the recall (sensitivity) of a classification model.

        Parameters:
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        float: The recall of the model, which measures the proportion of true positive predictions
        out of all actual positive instances in the dataset.
        """
        true_positives = np.sum((y_true == 1) & (y_pred == 1))
        false_negatives = np.sum((y_true == 1) & (y_pred == 0))
        return true_positives / (true_positives + false_negatives)

[22] @staticmethod
    def f1_score(y_true, y_pred):
        """
        Computes the F1-score of a classification model.

        Parameters:
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        float: The F1-score of the model, which is the harmonic mean of precision and recall.
        """
        precision_value = ClassificationMetrics.precision(y_true, y_pred)
        recall_value = ClassificationMetrics.recall(y_true, y_pred)
        return 2 * (precision_value * recall_value) / (precision_value + recall_value)

[23] model = LogisticRegression.load_model("model.pkl")

[24] y_pred = model.predict(X_test)
accuracy = ClassificationMetrics.accuracy(y_test, y_pred)
precision = ClassificationMetrics.precision(y_test, y_pred)
recall = ClassificationMetrics.recall(y_test, y_pred)
f1_score = ClassificationMetrics.f1_score(y_test, y_pred)

print(f"Accuracy: {accuracy:.2%}")
print(f"Precision: {precision:.2%}")
print(f"Recall: {recall:.2%}")
print(f"F1-Score: {f1_score:.2%}")

Accuracy: 98.23%
Precision: 100.00%
Recall: 95.24%
F1-Score: 97.56%

```

Observation and report

Date: 24/05/2024

Program Title: Build Support vector machine model for a given dataset Algorithm

Algorithm - snapshot from observation

SVM	Implementation
<p>Algorithm</p> <ul style="list-style-type: none">i) Define the kernel function. $k(x, x_i) = x_1 x_2$ii) solve the quadratic programming (QP) problem to find the α.iii) Compute the weight & bias.iv) Identify the support vectors.v) Make predictions. <p>Output:</p> <pre>→ model = SVM() → model.fit(X_train, y_train) → predictions = model.predict(X_test) → accuracy = (y_test, predictions).sum() / len(y_test)</pre> <p>0.9823008 (accuracy with sigmoid)</p> <p>$0.47069438, -0.16048584, -0.44810956, -0.02441212, -0.019956313, 0.18320441, 0.19695794]$</p> <p>array([0])</p>	

Code & Output:

```

✓ [1] from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

✓ [2] import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px

✓ [3] df = pd.read_csv('/content/drive/MyDrive/breast-cancer.csv')
df.head()

id diagnosis radius_mean texture_mean perimeter_mean area_mean smoothness_mean compactness_mean concavity_mean concave points_mean ...
0 842302 M 17.99 10.38 122.80 1001.0 0.11840 0.27760 0.3001 0.14710 ...
1 842517 M 20.57 17.77 132.90 1326.0 0.08474 0.07864 0.0869 0.07017 ...
2 84300903 M 19.69 21.25 130.00 1203.0 0.10960 0.15990 0.1974 0.12790 ...
3 84348301 M 11.42 20.38 77.58 386.1 0.14250 0.28390 0.2414 0.10520 ...
4 84358402 M 20.29 14.34 135.10 1297.0 0.10030 0.13280 0.1980 0.10430 ...

5 rows × 32 columns

```

```

[5] df.drop('id', axis=1, inplace=True) #drop redundant columns

[6] df.describe().T

```

	count	mean	std	min	25%	50%	75%	max
radius_mean	569.0	14.127292	3.524049	6.981000	11.700000	13.370000	15.780000	28.11000
texture_mean	569.0	19.289649	4.301036	9.710000	16.170000	18.840000	21.800000	39.28000
perimeter_mean	569.0	91.969033	24.298981	43.790000	75.170000	86.240000	104.100000	188.50000
area_mean	569.0	654.889104	351.914129	143.500000	420.300000	551.100000	782.700000	2501.00000
smoothness_mean	569.0	0.096360	0.014064	0.052630	0.086370	0.095870	0.105300	0.16340
compactness_mean	569.0	0.104341	0.052813	0.019380	0.064920	0.092630	0.130400	0.34540
concavity_mean	569.0	0.088799	0.079720	0.000000	0.029560	0.061540	0.130700	0.42680
concave points_mean	569.0	0.048919	0.038803	0.000000	0.020310	0.033500	0.074000	0.20120
symmetry_mean	569.0	0.181162	0.027414	0.106000	0.161900	0.179200	0.195700	0.30400
fractal_dimension_mean	569.0	0.062798	0.007060	0.049960	0.057700	0.061540	0.066120	0.09744
radius_se	569.0	0.405172	0.277313	0.111500	0.232400	0.324200	0.478900	2.87300
texture_se	569.0	1.216853	0.551648	0.360200	0.833900	1.108000	1.474000	4.88500
perimeter_se	569.0	2.866059	2.021855	0.757000	1.606000	2.287000	3.357000	21.98000
area_se	569.0	40.337079	45.491006	6.802000	17.850000	24.530000	45.190000	542.20000
smoothness_se	569.0	0.007041	0.003003	0.001713	0.005169	0.006380	0.008146	0.03113
compactness_se	569.0	0.025479	0.017000	0.002252	0.013020	0.020450	0.032450	0.12540

```

[7] df['diagnosis'] = (df['diagnosis'] == 'M').astype(int) #encode the label into 1/0

[8] corr = df.corr()

[10] # Get the absolute value of the correlation
cor_target = abs(corr["diagnosis"])

# Select highly correlated features (threshold = 0.2)
relevant_features = cor_target[cor_target>0.2]

# Collect the names of the features
names = [index for index, value in relevant_features.items()]

# Drop the target variable from the results
names.remove('diagnosis')

# Display the results
print(names)

[11] X = df[names].values
y = df['diagnosis']

[12] def scale(X):
    """
    Standardizes the data in the array X.

    Parameters:
        X (numpy.ndarray): Features array of shape (n_samples, n_features).

    Returns:
        numpy.ndarray: The standardized features array.
    """
    # Calculate the mean and standard deviation of each feature
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)

    # Standardize the data
    X = (X - mean) / std
    return X

[13] X = scale(X)

[14] def train_test_split(X, y, random_state=42, test_size=0.2):
    """
    Splits the data into training and testing sets.

    Parameters:
        X (numpy.ndarray): Features array of shape (n_samples, n_features).
        y (numpy.ndarray): Target array of shape (n_samples,).
        random_state (int): Seed for the random number generator. Default is 42.
        test_size (float): Proportion of samples to include in the test set. Default is 0.2.

    Returns:
        Tuple[numpy.ndarray]: A tuple containing X_train, X_test, y_train, y_test.
    """
    # Get number of samples
    n_samples = X.shape[0]

    # Set the seed for the random number generator
    np.random.seed(random_state)

    # Shuffle the indices
    shuffled_indices = np.random.permutation(np.arange(n_samples))

    # Determine the size of the test set
    test_size = int(n_samples * test_size)

    # Split the indices into test and train
    test_indices = shuffled_indices[:test_size]
    train_indices = shuffled_indices[test_size:]

    # Split the features and target arrays into test and train
    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test

[15] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42) #split the data into traing and validation

```

```

[18] class SVM:
    """
        A Support Vector Machine (SVM) implementation using gradient descent.

    Parameters:
    -----
    iterations : int, default=1000
        The number of iterations for gradient descent.
    lr : float, default=0.01
        The learning rate for gradient descent.
    lambdaa : float, default=0.01
        The regularization parameter.

    Attributes:
    -----
    lambdaa : float
        The regularization parameter.
    iterations : int
        The number of iterations for gradient descent.
    lr : float
        The learning rate for gradient descent.
    w : numpy array
        The weights.
    b : float
        The bias.

    Methods:
    -----
    initialize_parameters(X)
        Initializes the weights and bias.
    gradient_descent(X, y)
        Updates the weights and bias using gradient descent.
    update_parameters(dw, db)
        Updates the weights and bias.
    fit(X, y)
        Fits the SVM to the data.
    predict(X)
        Predicts the labels for the given data.

    """

    def __init__(self, iterations=1000, lr=0.01, lambdaa=0.01):
        """
            Initializes the SVM model.

        Parameters:
        -----
        iterations : int, default=1000
            The number of iterations for gradient descent.
        lr : float, default=0.01
            The learning rate for gradient descent.
        lambdaa : float, default=0.01
            The regularization parameter.
        """
        self.lambdaa = lambdaa
        self.iterations = iterations
        self.lr = lr
        self.w = None
        self.b = None
    def initialize_parameters(self, X):
        """
            Initializes the weights and bias.

        Parameters:
        -----

```

```

✓ [18]      X : numpy array
             The input data.
"""
m, n = X.shape
self.w = np.zeros(n)
self.b = 0

def gradient_descent(self, X, y):
    """
    Updates the weights and bias using gradient descent.

    Parameters:
    -----
    X : numpy array
        The input data.
    y : numpy array
        The target values.
    """
    y_ = np.where(y <= 0, -1, 1)
    for i, x in enumerate(X):
        if y_[i] * (np.dot(x, self.w) - self.b) >= 1:
            dw = 2 * self.lambdab * self.w
            db = 0
        else:
            dw = 2 * self.lambdab * self.w - np.dot(x, y_[i])
            db = y_[i]
        self.update_parameters(dw, db)

    def update_parameters(self, dw, db):
        """
        Updates the weights and bias.

        Parameters:
        -----
        dw : numpy array
            The change in weights.
        db : float
            The change in bias.
        """
        self.w = self.w - self.lr * dw
        self.b = self.b - self.lr * db

```

```

def fit(self, X, y):
    """
    Fits the SVM to the data.

    Parameters:
    -----
    X : numpy array
        The input data.
    y : numpy array
        The target values.
    """
    self.initialize_parameters(X)
    for i in range(self.iterations):
        self.gradient_descent(X, y)

    def predict(self, X):
        """
        Predicts the class labels for the test data.

        Parameters
        -----
        X : array-like, shape (n_samples, n_features)
            The input data.

        Returns

```

```
[18]     Returns
-----
y_pred : array-like, shape (n_samples,)
    The predicted class labels.

"""
# get the outputs
output = np.dot(X, self.w) - self.b
# get the signs of the labels depending on if it's greater/less than zero
label_signs = np.sign(output)
#set predictions to 0 if they are less than or equal to -1 else set them to 1
predictions = np.where(label_signs <= -1, 0, 1)
return predictions
```

```
[19] def accuracy(y_true, y_pred):
"""
Computes the accuracy of a classification model.

Parameters:
-----
y_true (numpy array): A numpy array of true labels for each data point.
y_pred (numpy array): A numpy array of predicted labels for each data point.

Returns:
-----
float: The accuracy of the model
"""

total_samples = len(y_true)
correct_predictions = np.sum(y_true == y_pred)
return (correct_predictions / total_samples)
```

```
[20] model = SVM()
model.fit(X_train,y_train)
predictions = model.predict(X_test)

accuracy(y_test, predictions)
```

```
→ 0.9823008849557522
```

```
[28] model.predict([-0.47069438, -0.16048584, -0.44810956, -0.49199876,  0.23411429,
  0.02765051, -0.10984741, -0.27623152,  0.41394897, -0.03274296,
 -0.18269561, -0.22105292, -0.35591235, -0.16192949, -0.23133322,
 -0.26903951, -0.16890536, -0.33393537, -0.35629925,  0.4485028 ,
 -0.10474068, -0.02441212, -0.19956318,  0.18320441,  0.19695794])
```

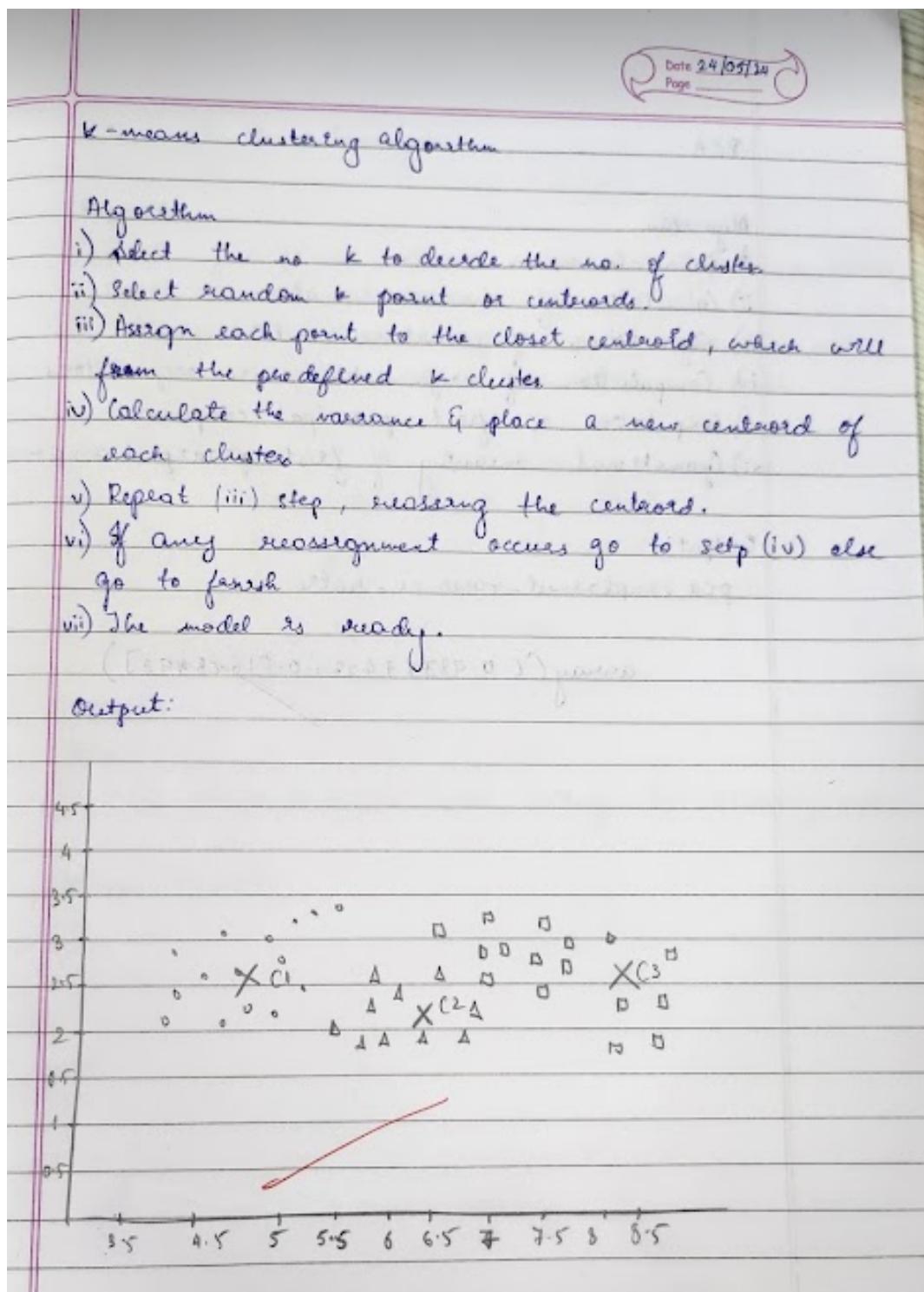
```
→ array(0)
```

Observation and report

Date: 24/05/2024

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

Algorithm - snapshot from observation



Code & Output:

```

[1] from google.colab import drive
drive.mount('/content/drive')

→ Mounted at /content/drive

[2] import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import seaborn as sns
import plotly.graph_objects as go

[3] iris = pd.read_csv("/content/drive/MyDrive/Iris.csv") #Load Data
iris.drop('Id', inplace=True, axis=1) #Drop Id column

[4] X = iris.iloc[:, :-1] #Set our training data
y = iris.iloc[:, -1] #We'll use this just for visualization as clustering doesn't require labels

[5] class Kmeans:
    """
    K-Means clustering algorithm implementation.

    Parameters:
    K (int): Number of clusters

    Attributes:
    K (int): Number of clusters
    centroids (numpy.ndarray): Array containing the centroids of each cluster
    """

    Methods:
        __init__(self, K): Initializes the Kmeans instance with the specified number of clusters.
        initialize_centroids(self, X): Initializes the centroids for each cluster by selecting K random points from the dataset.
        assign_points_centroids(self, X): Assigns each point in the dataset to the nearest centroid.
        compute_mean(self, X, points): Computes the mean of the points assigned to each centroid.
        fit(self, X, iterations=10): Clusters the dataset using the K-Means algorithm.
    """

    def __init__(self, K):
        assert K > 0, "K should be a positive integer."
        self.K = K

    def initialize_centroids(self, X):
        assert X.shape[0] >= self.K, "Number of data points should be greater than or equal to K."

        randomized_X = np.random.permutation(X.shape[0])
        centroid_idx = randomized_X[:self.K] # get the indices for the centroids
        self.centroids = X[centroid_idx] # assign the centroids to the selected points

    def assign_points_centroids(self, X):
        """
        Assign each point in the dataset to the nearest centroid.

        Parameters:
        X (numpy.ndarray): dataset to cluster

        Returns:
        numpy.ndarray: array containing the index of the centroid for each point
        """
        X = np.expand_dims(X, axis=1) # expand dimensions to match shape of centroids
        distance = np.linalg.norm((X - self.centroids), axis=-1) # calculate Euclidean distance between each point and each centroid
        points = np.argmin(distance, axis=1) # assign each point to the closest centroid
        assert len(points) == X.shape[0], "Number of assigned points should equal the number of data points."

```

```

✓ 0s [5]     points = np.argmin(distance, axis=1) # assign each point to the closest centroid
    assert len(points) == X.shape[0], "Number of assigned points should equal the number of data points."
    return points

def compute_mean(self, X, points):
    """
    Compute the mean of the points assigned to each centroid.

    Parameters:
    X (numpy.ndarray): dataset to cluster
    points (numpy.ndarray): array containing the index of the centroid for each point

    Returns:
    numpy.ndarray: array containing the new centroids for each cluster
    """
    centroids = np.zeros((self.K, X.shape[1])) # initialize array to store centroids
    for i in range(self.K):
        centroid_mean = X[points == i].mean(axis=0) # calculate mean of the points assigned to the current centroid
        centroids[i] = centroid_mean # assign the new centroid to the mean of its points
    return centroids

def fit(self, X, iterations=10):
    """
    Cluster the dataset using the K-Means algorithm.

    Parameters:
    X (numpy.ndarray): dataset to cluster
    iterations (int): number of iterations to perform (default=10)

    Returns:
    numpy.ndarray: array containing the final centroids for each cluster
    numpy.ndarray: array containing the index of the centroid for each point
    """
    self.initialize_centroids(X) # initialize the centroids
    for i in range(iterations):
        points = self.assign_points_centroids(X) # assign each point to the nearest centroid
        self.centroids = self.compute_mean(X, points) # compute the new centroids based on the mean of their points

        # Assertions for debugging and validation
        assert len(self.centroids) == self.K, "Number of centroids should equal K."
        assert X.shape[1] == self.centroids.shape[1], "Dimensionality of centroids should match input data."
        assert max(points) < self.K, "Cluster index should be less than K."
        assert min(points) >= 0, "Cluster index should be non-negative."

    return self.centroids, points

✓ 0s [6] X = X.values

✓ 0s [7] kmeans = Kmeans(3)

centroids, points = kmeans.fit(X, 1000)

✓ 2s [8] fig = go.Figure()
fig.add_trace(go.Scatter(
    x=X[points == 0, 0], y=X[points == 0, 1],
    mode='markers', marker_color='#DB4CB2', name='Iris-setosa'
))

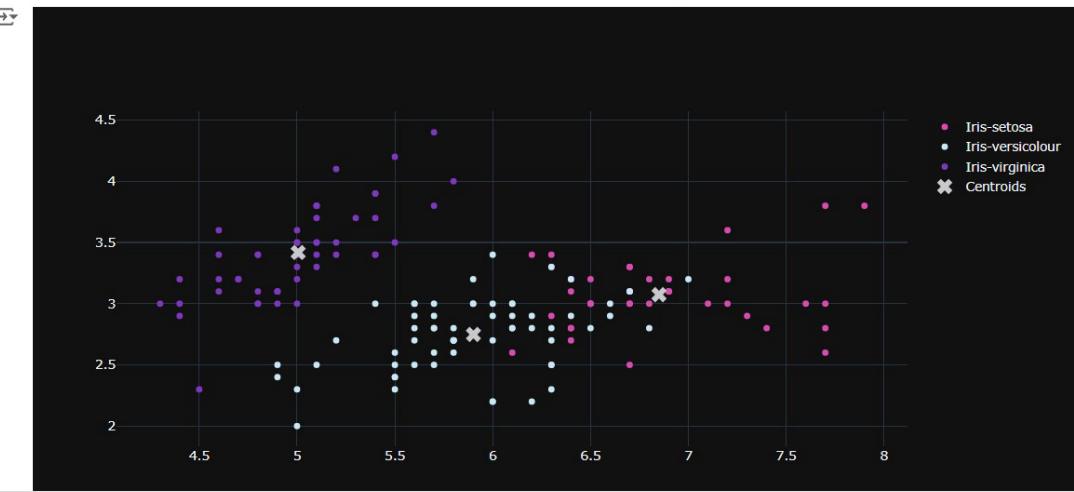
fig.add_trace(go.Scatter(
    x=X[points == 1, 0], y=X[points == 1, 1],
    mode='markers', marker_color='#c9e9f6', name='Iris-versicolour'
))

```

```

[8] ))
fig.add_trace(go.Scatter(
    x=X[points == 2, 0], y=X[points == 2, 1],
    mode='markers',marker_color='#7D3AC1',name='Iris-virginica'
))
fig.add_trace(go.Scatter(
    x=centroids[:, 0], y=centroids[:,1],
    mode='markers',marker_color="#CAC9CD",marker_symbol=4,marker_size=13,name='Centroids'
))
fig.update_layout(template='plotly_dark',width=1000, height=500,)


```

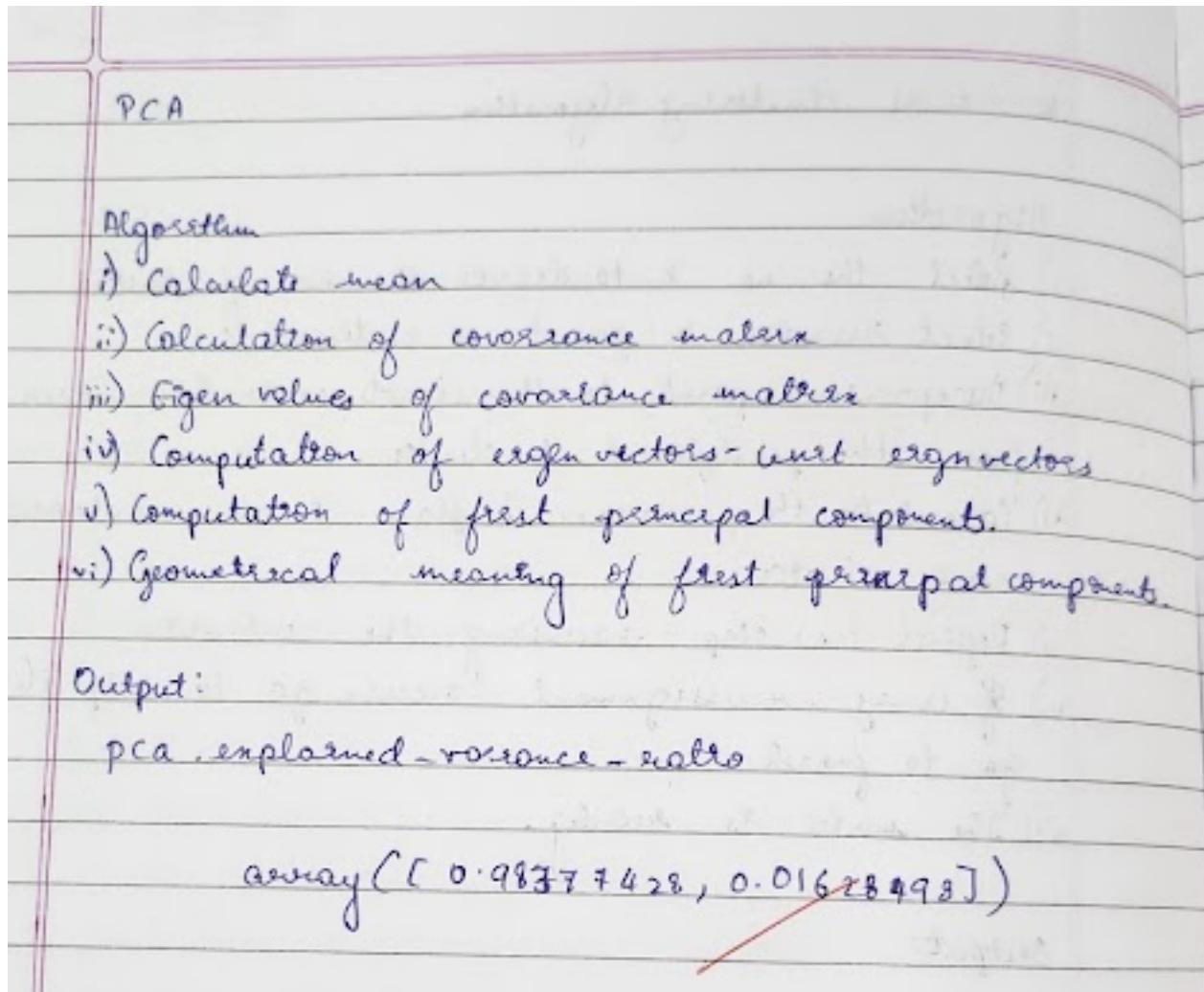


Observation and report

Date: 24/05/2024

Program Title: Implement Dimensionality reduction using Principle Component Analysis (PCA)

Algorithm - snapshot from observation



Code & Output:

+ Code + Text

✓ RAM 
Disk 

```
✓ [1] from google.colab import drive  
drive.mount('/content/drive')
```

→ Mounted at /content/drive

```
✓ [2] import seaborn as sns  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import plotly.express as px  
import plotly.graph_objects as go  
from plotly.subplots import make_subplots
```

```
✓ [3] df = pd.read_csv('/content/drive/MyDrive/breast-cancer.csv')  
df.head()
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave_points_mean	... radius_worst	
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	25.38
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54

5 rows × 32 columns

```

✓ [4] df.drop('id', axis=1, inplace=True) #drop redundant columns

✓ [5] df['diagnosis'] = (df['diagnosis'] == 'M').astype(int) #encode the label into 1/0

✓ [6] corr = df.corr()

✓ [8] # Get the absolute value of the correlation
cor_target = abs(corr["diagnosis"])

# Select highly correlated features (threshold = 0.2)
relevant_features = cor_target[cor_target>0.2]

# Collect the names of the features
names = [index for index, value in relevant_features.items()]

# Drop the target variable from the results
names.remove('diagnosis')

# Display the results
print(names)

→ ['radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean', 'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean', 'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se', 'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se', 'fractal_dimension_se', 'radius_worst', 'texture_worst', 'perimeter_worst', 'area_worst', 'smoothness_worst', 'compactness_worst', 'concavity_worst', 'concave points_worst', 'symmetry_worst', 'fractal_dimension_worst']

✓ [9] X = df[names].values

✓ [9] X = df[names].values

[11] class PCA:
    """
        Principal Component Analysis (PCA) class for dimensionality reduction.
    """

    def __init__(self, n_components):
        """
            Constructor method that initializes the PCA object with the number of components to retain.

            Args:
            - n_components (int): Number of principal components to retain.
        """
        self.n_components = n_components
    def fit(self, X):
        """
            Fits the PCA model to the input data and computes the principal components.

            Args:
            - X (numpy.ndarray): Input data matrix with shape (n_samples, n_features).
        """
        # Compute the mean of the input data along each feature dimension.
        mean = np.mean(X, axis=0)

        # Subtract the mean from the input data to center it around zero.
        X = X - mean

        # Compute the covariance matrix of the centered input data.
        cov = np.cov(X.T)

```

```

✓ [11]      # Compute the covariance matrix of the centered input data.
cov = np.cov(X.T)

# Compute the eigenvectors and eigenvalues of the covariance matrix.
eigenvalues, eigenvectors = np.linalg.eigh(cov)
# Reverse the order of the eigenvalues and eigenvectors.
eigenvalues = eigenvalues[::-1]
eigenvectors = eigenvectors[:,::-1]

# Keep only the first n_components eigenvectors as the principal components.
self.components = eigenvectors[:, :self.n_components]

# Compute the explained variance ratio for each principal component.
# Compute the total variance of the input data
total_variance = np.sum(np.var(X, axis=0))

# Compute the variance explained by each principal component
self.explained_variances = eigenvalues[:self.n_components]

# Compute the explained variance ratio for each principal component
self.explained_variance_ratio_ = self.explained_variances / total_variance
def transform(self, X):
"""
Transforms the input data by projecting it onto the principal components.

Args:
- X (numpy.ndarray): Input data matrix with shape (n_samples, n_features).

Returns:
- transformed_data (numpy.ndarray): Transformed data matrix with shape (n_samples, n_components).
"""
# Center the input data around zero using the mean computed during the fit step.
X = X - np.mean(X, axis=0)

```

```

✓ [11] # Project the centered input data onto the principal components.
Ctrl+M B     transformed_data = np.dot(X, self.components)

return transformed_data

def fit_transform(self, X):
"""
Fits the PCA model to the input data and computes the principal components then
transforms the input data by projecting it onto the principal components.

Args:
- X (numpy.ndarray): Input data matrix with shape (n_samples, n_features).
"""
self.fit(X)
transformed_data = self.transform(X)
return transformed_data

```

```
✓ [12] pca = PCA(2)
```

```
✓ [13] pca.fit(X)
```

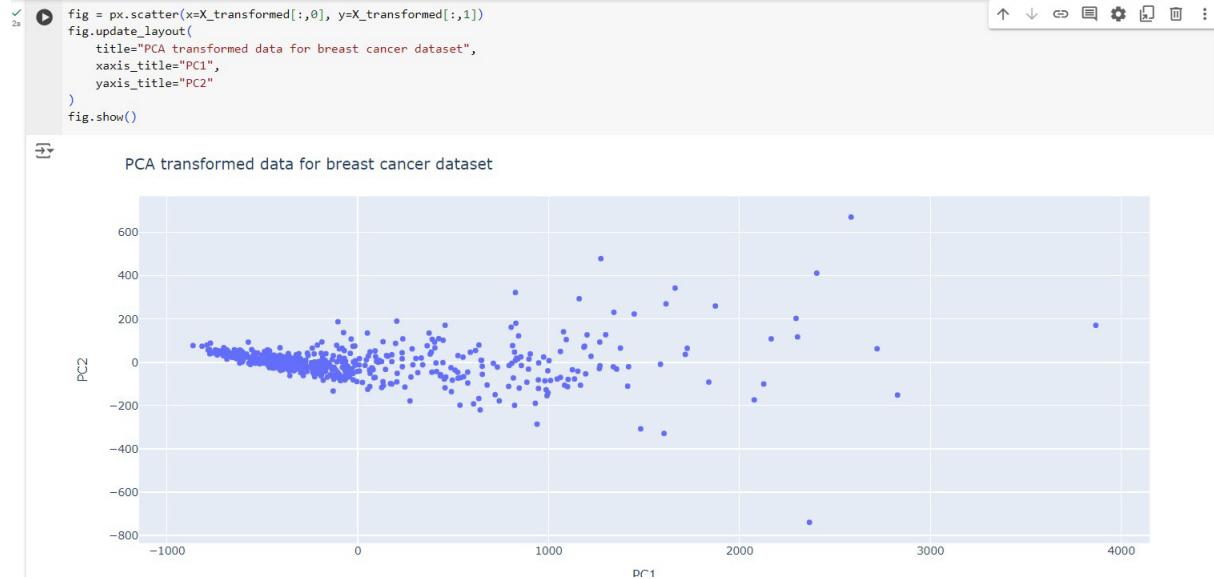
```
✓ [14] pca.explained_variance_ratio_
```

```
→ array([0.98377428, 0.01620498])
```

```
✓ [15] X_transformed = pca.transform(X)
```

```
✓ [16] X_transformed[:,1].shape
```

```
→ (569,)
```

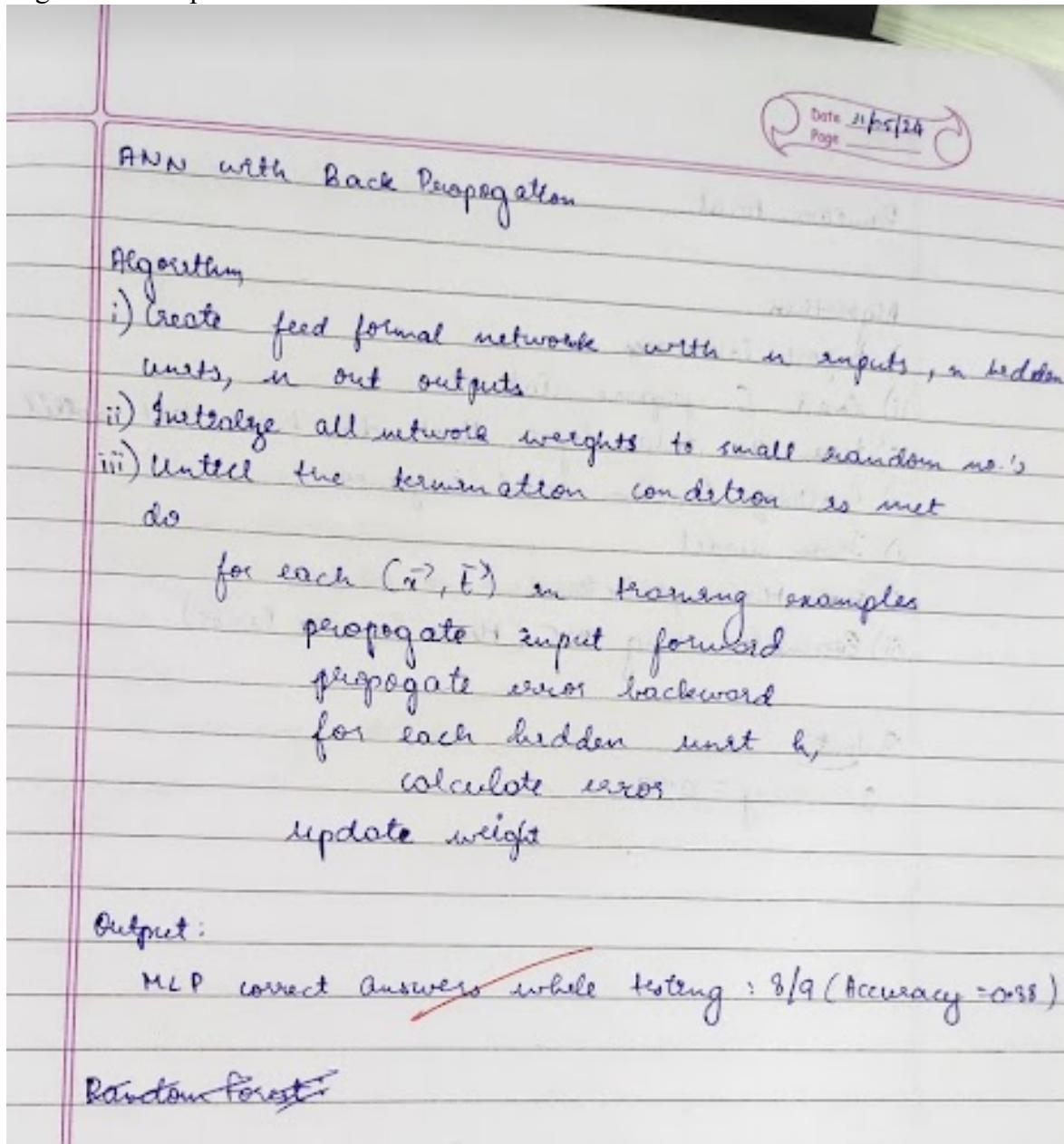


Observation and report

Date: 31/05/2024

Program Title: Build Artificial Neural Network model with back propagation on a given dataset

Algorithm - snapshot from observation



Code & Output:

```

✓ 0s  import numpy as np
    from sklearn.model_selection import train_test_split

    db = np.loadtxt("/content/duke-breast-cancer.txt")
    print("Database raw shape (%s,%s)" % np.shape(db))

    ↵ Database raw shape (86,7130)

✓ 0s  [3]  np.random.shuffle(db)
        y = db[:, 0]
        x = np.delete(db, [0], axis=1)
        x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.1)
        print(np.shape(x_train),np.shape(x_test))

    ↵ (77, 7129) (9, 7129)

✓ 0s  [4]  hidden_layer = np.zeros(72)
        weights = np.random.random((len(x[0]), 72))
        output_layer = np.zeros(2)
        hidden_weights = np.random.random((72, 2))

✓ 0s  [5]  def sum_function(weights, index_locked_col, x):
            result = 0
            for i in range(0, len(x)):
                result += x[i] * weights[i][index_locked_col]
            return result

✓ 0s  [6]  def activate_layer(layer, weights, x):
            for i in range(0, len(layer)):
                layer[i] = 1.7159 * np.tanh(2.0 * sum_function(weights, i, x) / 3.0)

✓ 0s  [7]  def soft_max(layer):
            soft_max_output_layer = np.zeros(len(layer))
            for i in range(0, len(layer)):
                denominator = 0
                for j in range(0, len(layer)):
                    denominator += np.exp(layer[j] - np.max(layer))
                soft_max_output_layer[i] = np.exp(layer[i] - np.max(layer)) / denominator
            return soft_max_output_layer

✓ 0s  [8]  def recalculate_weights(learning_rate, weights, gradient, activation):
            for i in range(0, len(weights)):
                for j in range(0, len(weights[i])):
                    weights[i][j] = (learning_rate * gradient[j] * activation[i]) + weights[i][j]

✓ 0s  [9]  def back_propagation(hidden_layer, output_layer, one_hot_encoding, learning_rate, x):
            output_derivative = np.zeros(2)
            output_gradient = np.zeros(2)
            for i in range(0, len(output_layer)):
                output_derivative[i] = (1.0 - output_layer[i]) * output_layer[i]
            for i in range(0, len(output_layer)):
                output_gradient[i] = output_derivative[i] * (one_hot_encoding[i] - output_layer[i])
            hidden_derivative = np.zeros(72)
            hidden_gradient = np.zeros(72)
            for i in range(0, len(hidden_layer)):
                hidden_derivative[i] = (1.0 - hidden_layer[i]) * (1.0 + hidden_layer[i])
            for i in range(0, len(hidden_layer)):
                sum_ = 0
                for j in range(0, len(output_gradient)):
                    sum_ += output_gradient[j] * hidden_weights[i][j]
                hidden_gradient[i] = sum_ * hidden_derivative[i]
            recalculate_weights(learning_rate, hidden_weights, output_gradient, hidden_layer)
            recalculate_weights(learning_rate, weights, hidden_gradient, x)

```

```
✓ [10] one_hot_encoding = np.zeros((2,2))
    for i in range(0, len(one_hot_encoding)):
        one_hot_encoding[i][1] = 1
    training_correct_answers = 0
    for i in range(0, len(x_train)):
        activate_layer(hidden_layer, weights, x_train[i])
        activate_layer(output_layer, hidden_weights, hidden_layer)
        output_layer = soft_max(output_layer)
        training_correct_answers += 1 if y_train[i] == np.argmax(output_layer) else 0
        back_propagation(hidden_layer, output_layer, one_hot_encoding[int(y_train[i])], -1, x_train[i])
    print("MLP Correct answers while learning: %s / %s (Accuracy = %s) on %s database." % (training_correct_answers, len(x_train),
                                                                                           training_correct_answers/len(x_train), "Duke breast cancer"))
```

⤵ MLP Correct answers while learning: 44 / 77 (Accuracy = 0.5714285714285714) on Duke breast cancer database.

```
✓ [11] testing_correct_answers = 0
    for i in range(0, len(x_test)):
        activate_layer(hidden_layer, weights, x_test[i])
        activate_layer(output_layer, hidden_weights, hidden_layer)
        output_layer = soft_max(output_layer)
        testing_correct_answers += 1 if y_test[i] == np.argmax(output_layer) else 0
    print("MLP Correct answers while testing: %s / %s (Accuracy = %s) on %s database." % (testing_correct_answers, len(x_test),
                                                                                           testing_correct_answers/len(x_test), "Duke breast cancer"))
```

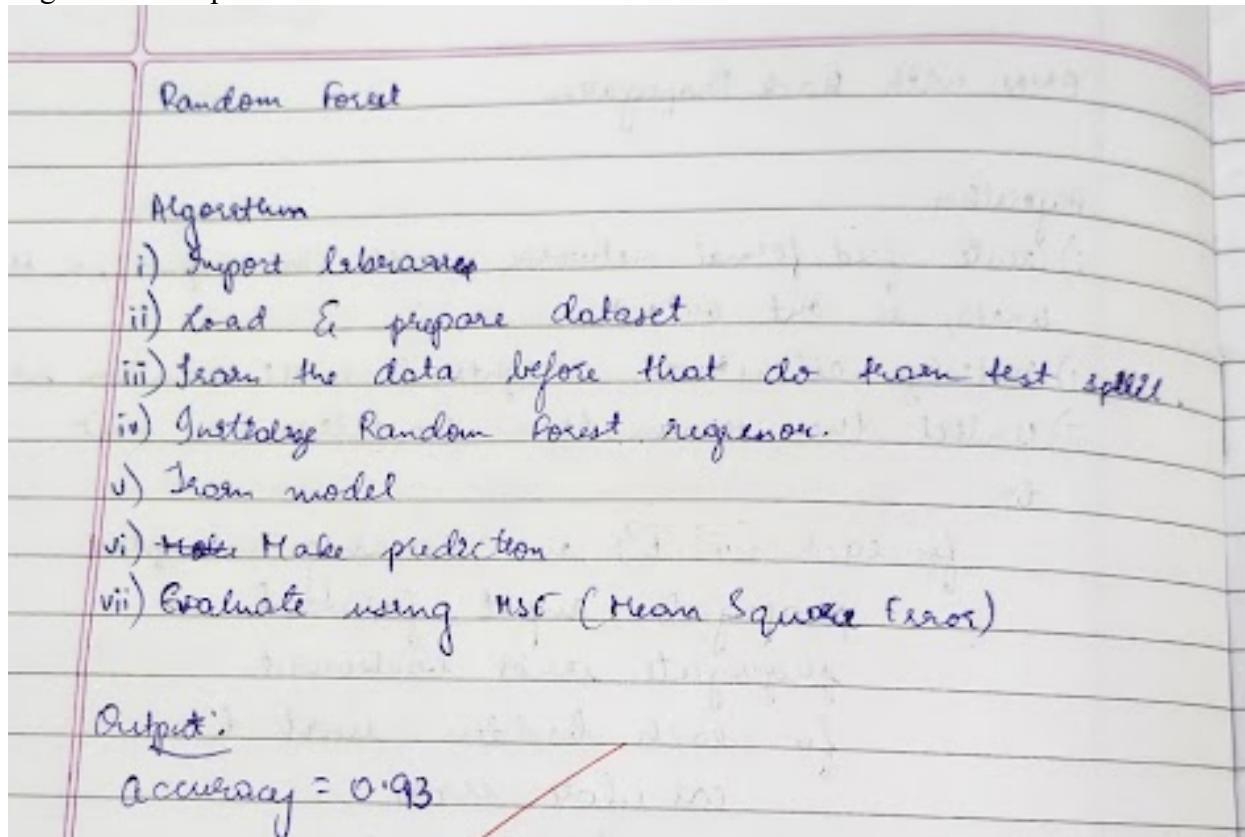
⤵ MLP Correct answers while testing: 8 / 9 (Accuracy = 0.8888888888888888) on Duke breast cancer database

Observation and report

Date: 31/05/2024

Program Title: Implement Random forest ensemble method on a given dataset

Algorithm - snapshot from observation



Code & Output:

```

✓ [17] from google.colab import drive
      drive.mount('/content/drive')

→ Mounted at /content/drive

✓ 0s [18] import math
      import numpy as np
      import pandas as pd
      import matplotlib.pyplot as plt
      import seaborn as sns
      import plotly.express as px

✓ 0s [19] iris = pd.read_csv("/content/drive/MyDrive/Iris.csv") #Load Data
      iris.drop('Id',inplace=True,axis=1) #Drop Id column

✓ 0s [20] iris.head().style.background_gradient(cmap =sns.light_palette("seagreen", as_cmap=True)
      )

→ SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
 0 5.100000 3.500000 1.400000 0.200000 Iris-setosa
 1 4.900000 3.000000 1.400000 0.200000 Iris-setosa
 2 4.700000 3.200000 1.300000 0.200000 Iris-setosa
 3 4.600000 3.100000 1.500000 0.200000 Iris-setosa
 4 5.000000 3.600000 1.400000 0.200000 Iris-setosa

✓ 0s [21] X_df = iris.iloc[:, :-1] #Set our training dataframe
      y_df = iris.iloc[:, -1] # Set our training labels dataframe

✓ 0s [22] fig = px.pie(iris, 'species',color_discrete_sequence=['#3dec84 ','#009688 ','#218857 '],title='Data Distribution',template='plotly')
      fig.show()

→ Data Distribution



| Species         | Percentage |
|-----------------|------------|
| Iris-setosa     | 33.3%      |
| Iris-versicolor | 33.3%      |
| Iris-virginica  | 33.3%      |



✓ 0s [23] iris['Species'] = iris['Species'].astype("category")
      codes = iris['Species'].cat.codes

```

```

✓ [23] iris['Species'] = iris['Species'].astype("category")
      codes = iris['Species'].cat.codes

✓ [24] def train_test_split(X, y, random_state=42, test_size=0.2):
      """
      Splits the data into training and testing sets.

      Parameters:
          X (numpy.ndarray): Features array of shape (n_samples, n_features).
          y (numpy.ndarray): Target array of shape (n_samples,).
          random_state (int): Seed for the random number generator. Default is 42.
          test_size (float): Proportion of samples to include in the test set. Default is 0.2.

      Returns:
          Tuple[numpy.ndarray]: A tuple containing X_train, X_test, y_train, y_test.
      """
      # Get number of samples
      n_samples = X.shape[0]

      # Set the seed for the random number generator
      np.random.seed(random_state)

      # Shuffle the indices
      shuffled_indices = np.random.permutation(np.arange(n_samples))

      # Determine the size of the test set
      test_size = int(n_samples * test_size)

      # Split the indices into test and train
      test_indices = shuffled_indices[:test_size]
      train_indices = shuffled_indices[test_size:]

      # Split the features and target arrays into test and train
      X_train, X_test = X[train_indices], X[test_indices]
      y_train, y_test = y[train_indices], y[test_indices]

      return X_train, X_test, y_train, y_test

[25] X = iris.iloc[:, :-1].values
      y = iris.iloc[:, -1].values.reshape(-1,1)
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2, random_state=41)

[26] from sklearn.tree import DecisionTreeClassifier
      m = DecisionTreeClassifier()

[27] class RandomForest:
      """
      A random forest classifier.

      Parameters
      -----
      n_trees : int, default=7
          The number of trees in the random forest.
      max_depth : int, default=7
          The maximum depth of each decision tree in the random forest.
      min_samples : int, default=2
          The minimum number of samples required to split an internal node
          of each decision tree in the random forest.

      Attributes
      -----
      n_trees : int
          The number of trees in the random forest.
      max_depth : int
          The maximum depth of each decision tree in the random forest.
      min_samples : int
          The minimum number of samples required to split an internal node
          of each decision tree in the random forest.
      trees : list of DecisionTreeClassifier
          The decision trees in the random forest.
      """

      def __init__(self, n_trees=7, max_depth=7, min_samples=2):
          """
          Initialize the random forest classifier.
      """

```

```

[27]      The decision trees in the random forest.

      """
      Initialize the random forest classifier.

      Parameters
      -----
      n_trees : int, default=7
          The number of trees in the random forest.
      max_depth : int, default=7
          The maximum depth of each decision tree in the random forest.
      min_samples : int, default=2
          The minimum number of samples required to split an internal node
          of each decision tree in the random forest.
      """
      self.n_trees = n_trees
      self.max_depth = max_depth
      self.min_samples = min_samples
      self.trees = []

  def fit(self, X, y):
      """
      Build a random forest classifier from the training set (X, y).

      Parameters
      -----
      X : array-like of shape (n_samples, n_features)
          The training input samples.
      y : array-like of shape (n_samples,)
          The target values.

      Returns
      -----
      self : object
          Returns self.
      """
      # Create an empty list to store the trees.
      self.trees = []
      # Concatenate X and y into a single dataset.
      dataset = np.concatenate((X, y.reshape(-1, 1)), axis=1)
      # Loop over the number of trees.
      """
      # Create an empty list to store the trees.
      self.trees = []
      # Concatenate X and y into a single dataset.
      dataset = np.concatenate((X, y.reshape(-1, 1)), axis=1)
      # Loop over the number of trees.
      for _ in range(self.n_trees):
          # Create a decision tree instance.
          tree = DecisionTreeClassifier(max_depth=self.max_depth, min_samples_split=self.min_samples)
          # Sample from the dataset with replacement (bootstrapping).
          dataset_sample = self.bootstrap_samples(dataset)
          # Get the X and y samples from the dataset sample.
          X_sample, y_sample = dataset_sample[:, :-1], dataset_sample[:, -1]
          # Fit the tree to the X and y samples.
          tree.fit(X_sample, y_sample)
          # Store the tree in the list of trees.
          self.trees.append(tree)
      return self

  def bootstrap_samples(self, dataset):
      """
      Bootstrap the dataset by sampling from it with replacement.

      Parameters
      -----
      dataset : array-like of shape (n_samples, n_features + 1)
          The dataset to bootstrap.

      Returns
      -----
      dataset_sample : array-like of shape (n_samples, n_features + 1)
          The bootstrapped dataset sample.
      """
      # Get the number of samples in the dataset.
      n_samples = dataset.shape[0]
      # Generate random indices to index into the dataset with replacement.
      np.random.seed(1)
      indices = np.random.choice(n_samples, n_samples, replace=True)
      # Return the bootstrapped dataset sample using the generated indices.
      dataset_sample = dataset[indices]
      return dataset_sample

  def most_common_label(self, v):

```

```
[27] def most_common_label(self, y):
    """
        Return the most common label in an array of labels.

    Parameters
    -----
    y : array-like of shape (n_samples,)
        The array of labels.

    Returns
    -----
    most_occuring_value : int or float
        The most common label in the array.
    """
    y = list(y)
    # get the highest present class in the array
    most_occuring_value = max(y, key=y.count)
    return most_occuring_value

def predict(self, X):
    """
        Predict class for X.

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)
        The input samples.

    Returns
    -----
    majority_predictions : array-like of shape (n_samples,)
        The predicted classes.
    """
    #get prediction from each tree in the tree list on the test data
    predictions = np.array([tree.predict(X) for tree in self.trees])
    # get prediction for the same sample from all trees for each sample in the test data
    preds = np.swapaxes(predictions, 0, 1)
    #get the most voted value by the trees and store it in the final predictions array
    majority_predictions = np.array([self.most_common_label(pred) for pred in preds])
    return majority_predictions
```

```
[28] def accuracy(y_true, y_pred):
    """
        Computes the accuracy of a classification model.

    Parameters:
    y_true (numpy array): A numpy array of true labels for each data point.
    y_pred (numpy array): A numpy array of predicted labels for each data point.

    Returns:
    float: The accuracy of the model, expressed as a percentage.
    """
    y_true = y_true.flatten()
    total_samples = len(y_true)
    correct_predictions = np.sum(y_true == y_pred)
    return (correct_predictions / total_samples)
```

```
[35] from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_label.py:116: 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().
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_label.py:134: 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().
```

```
[39] from sklearn.preprocessing import LabelEncoder  
  
label_encoder = LabelEncoder()  
y_train_encoded = label_encoder.fit_transform(y_train.ravel())  
y_test_encoded = label_encoder.transform(y_test.ravel())  
model = RandomForest(10, 10, 2)  
model.fit(X_train, y_train_encoded)  
  
predictions = model.predict(X_test)  
accuracy(y_test_encoded, predictions)
```

→ 0.9333333333333333

```
▶ from sklearn.tree import DecisionTreeClassifier  
  
# Create and train the decision tree model  
dt = DecisionTreeClassifier()  
dt.fit(X_train, y_train_encoded)  
  
# Make predictions on the test data  
predictions = dt.predict(X_test)  
  
# Calculate accuracy  
accuracy(y_test_encoded, predictions)
```

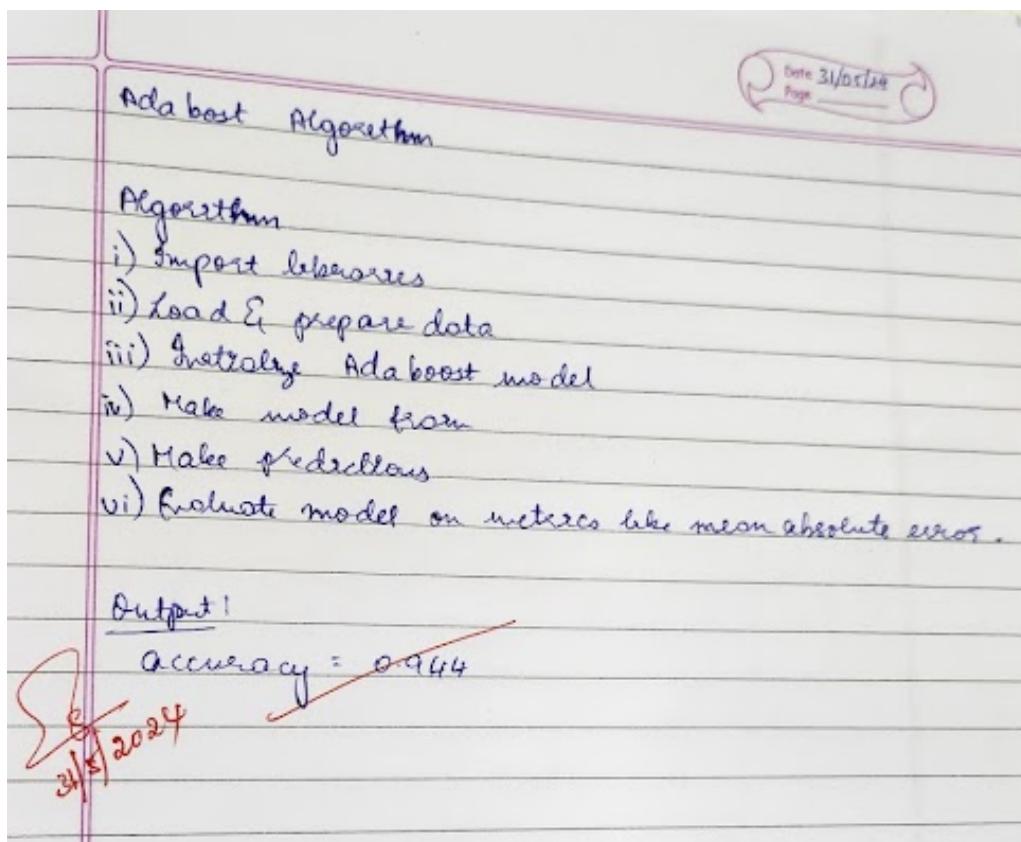
→ 0.9

Observation and report

Date: 31/05/2024

Program Title: Implement Boosting ensemble method on a given dataset

Algorithm - snapshot from observation



```
[15] # Compute error rate, alpha and w
def compute_error(y, y_pred, w_i):
    ...
    Calculate the error rate of a weak classifier m. Arguments:
    y: actual target value
    y_pred: predicted value by weak classifier
    w_i: individual weights for each observation
    Note that all arrays should be the same length
    ...
    return (sum(w_i * (np.not_equal(y, y_pred)).astype(int)))/sum(w_i)

def compute_alpha(error):
    ...
    Calculate the weight of a weak classifier m in the majority vote of the final classifier. This is called
    alpha in chapter 10.1 of The Elements of Statistical Learning. Arguments:
    error: error rate from weak classifier m
    ...
    return np.log((1 - error) / error)

def update_weights(w_i, alpha, y, y_pred):
    ...
    Update individual weights w_i after a boosting iteration. Arguments:
    w_i: individual weights for each observation
    y: actual target value
    y_pred: predicted value by weak classifier
    alpha: weight of weak classifier used to estimate y_pred
    ...
    return w_i * np.exp(alpha * (np.not_equal(y, y_pred)).astype(int))
```

```

[16] # Define AdaBoost class
class AdaBoost:

    def __init__(self):
        self.alphas = []
        self.G_M = []
        self.M = None
        self.training_errors = []
        self.prediction_errors = []

    def fit(self, X, y, M = 100):
        ...
        Fit model. Arguments:
        X: independent variables - array-like matrix
        y: target variable - array-like vector
        M: number of boosting rounds. Default is 100 - integer
        ...

        # Clear before calling
        self.alphas = []
        self.training_errors = []
        self.M = M

        # Iterate over M weak classifiers
        for m in range(0, M):

            # Set weights for current boosting iteration
            if m == 0:
                w_i = np.ones(len(y)) * 1 / len(y) # At m = 0, weights are all the same and equal to 1 / N
            else:
                # (d) Update w_i
                w_i = update_weights(w_i, alpha_m, y, y_pred)

            # (a) Fit weak classifier and predict labels
            G_m = DecisionTreeClassifier(max_depth = 1)      # Stump: Two terminal-node classification tree
            G_m.fit(X, y, sample_weight = w_i)
            y_pred = G_m.predict(X)

            self.G_M.append(G_m) # Save to list of weak classifiers

            # (b) Compute error
            error_m = compute_error(y, y_pred, w_i)
            w_i = update_weights(w_i, alpha_m, y, y_pred)

            # (a) Fit weak classifier and predict labels
            G_m = DecisionTreeClassifier(max_depth = 1)      # Stump: Two terminal-node classification tree
            G_m.fit(X, y, sample_weight = w_i)
            y_pred = G_m.predict(X)

            self.G_M.append(G_m) # Save to list of weak classifiers

            # (b) Compute error
            error_m = compute_error(y, y_pred, w_i)
            self.training_errors.append(error_m)

            # (c) Compute alpha
            alpha_m = compute_alpha(error_m)
            self.alphas.append(alpha_m)

        assert len(self.G_M) == len(self.alphas)

    def predict(self, X):
        ...
        Predict using fitted model. Arguments:
        X: independent variables - array-like
        ...

        # Initialise dataframe with weak predictions for each observation
        weak_preds = pd.DataFrame(index = range(len(X)), columns = range(self.M))

        # Predict class label for each weak classifier, weighted by alpha_m
        for m in range(self.M):
            y_pred_m = self.G_M[m].predict(X) * self.alphas[m]
            weak_preds.iloc[:,m] = y_pred_m

        # Calculate final predictions
        y_pred = (1 * np.sign(weak_preds.T.sum())).astype(int)

        return y_pred

```

```
[17] import pandas as pd
     import numpy as np
     from sklearn.model_selection import train_test_split
     from sklearn.tree import DecisionTreeClassifier

# Dataset
df = pd.read_csv('/content/spambase.data', header = None)

# Column names
names = pd.read_csv('/content/spambase.names', sep = ':', skiprows=range(0, 33), header = None)
col_names = list(names[0])
col_names.append('Spam')

# Rename df columns
df.columns = col_names

# Convert classes in target variable to {-1, 1}
df['Spam'] = df['Spam'] * 2 - 1

# Train - test split
X_train, X_test, y_train, y_test = train_test_split(df.drop(columns = 'Spam').values,
                                                    df['Spam'].values,
                                                    train_size = 3065,
                                                    random_state = 2)
```

```
# Fit model
ab = AdaBoost()
ab.fit(X_train, y_train, M = 400)

# Predict on test set
y_pred = ab.predict(X_test)

from sklearn.metrics import accuracy_score

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Accuracy: 0.9440104166666666