

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.



LAB REPORT

ON

MACHINE LEARNING

Submitted by

HARSH GHIYA(1BM21CS073)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING



B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(March 2024 to June 2024)



**B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019**
(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**MACHINE LEARNING**" is carried out by **HARSH GHIYA (1BM21CS073)** who is bonafide student of **B.M.S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visveswaraya Technological University, Belgaum during the year 2023-2024. The lab report has been approved as it satisfies the academic requirements in respect of **Machine Learning Lab - (22CS3PCMAL)** work prescribed for the said degree.

Dr. K. Panimozhi
Assistant Professor
BMSCE, Bengaluru

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

Index Sheet

| Sl. No. | Experiment Title | Page No. |
|--------------------|---|-----------------|
| 1 | Write a python program to import and export data using Pandas library functions | 4 |
| 2 | Demonstrate various data pre-processing techniques for a given dataset | 7 |
| 3 | Use an appropriate data set for building the decision tree (ID3) and apply this knowledge to classify a new sample. | 10 |
| 4 | Build KNN Classification model for a given dataset. | 14 |
| 5 | Implement Linear and Multi-Linear Regression algorithm using appropriate dataset | 21 |
| 6 | Build Logistic Regression Model for a given dataset | 28 |
| 7 | Build Support vector machine model for a given dataset. | 38 |
| 8 | Build k-Means algorithm to cluster a set of data stored in a .CSV file. | 44 |
| 9 | Implement Dimensionality reduction using Principle Component Analysis (PCA) method. | 48 |
| 10 | Build Artificial Neural Network model with back propagation on a given dataset | 52 |
| 11 | a) Implement Random forest ensemble method on a given dataset. b) Implement Boosting ensemble method on a given dataset. | 56 |

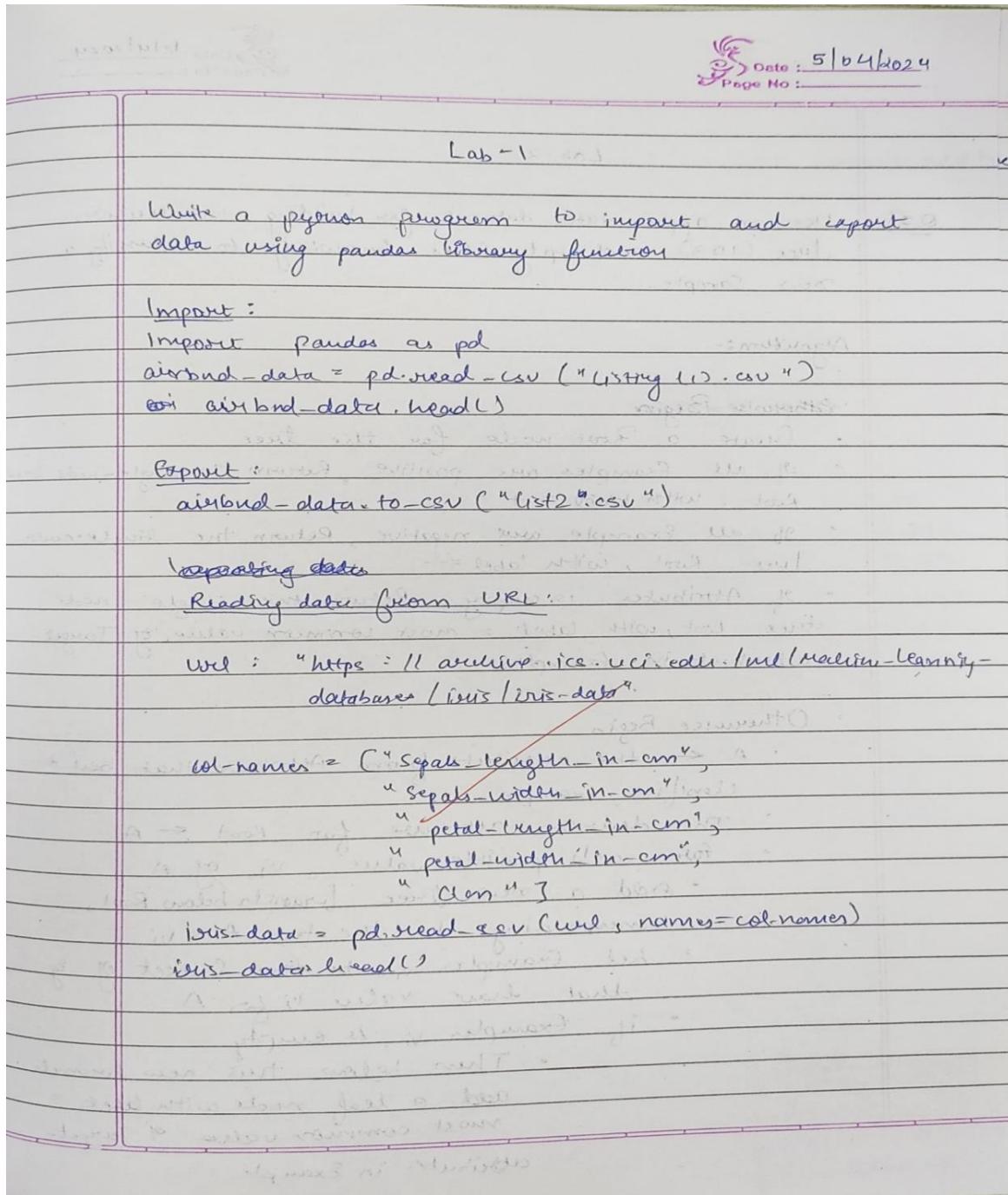
Course outcomes:

| | |
|-----|--|
| 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 |

04.04.2024

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

Screenshot



Code:

Import

```
import pandas as pd
```

```
airbnb_data = pd.read_csv("listings (1).csv")
```

```
airbnb_data.head()
```

| 0] | id | name | host_id | host_name | neighbourhood_group | neighbourhood | latitude | longitude | room_type | price | minimum_nights | number_of_reviews | last_review |
|----|--------|--|---------|-----------|---------------------|---------------|----------|-----------|-----------------|-------|----------------|-------------------|-------------|
| 0 | 329172 | Hillside designer home, 10 min.dwntn | 1680871 | Janet | NaN | 78746 | 30.30085 | -97.80794 | Entire home/apt | 495 | 3 | 7 | 2022- |
| 1 | 329306 | Urban Homestead, 5 minutes to downtown | 880571 | Angel | NaN | 78702 | 30.27232 | -97.72579 | Private room | 63 | 2 | 570 | 2022- |
| 2 | 331549 | One Room with Private Bathroom | 1690383 | Sandra | NaN | 78725 | 30.23911 | -97.58625 | Private room | 100 | 2 | 0 | |
| 3 | 333815 | Solar Sanctuary - Austin Room | 372962 | Kim | NaN | 78704 | 30.25381 | -97.75262 | Private room | 102 | 2 | 164 | 2022- |
| 4 | 333442 | Rare Secluded 1940s Estate | 1698318 | Virginia | NaN | 78703 | 30.31267 | -97.76641 | Entire home/apt | 286 | 3 | 163 | 2022- |

Export

```
airbnb_data.to_csv("list2.csv")
```

Reading the file from the URL:

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

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

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

```
iris_data.head()
```

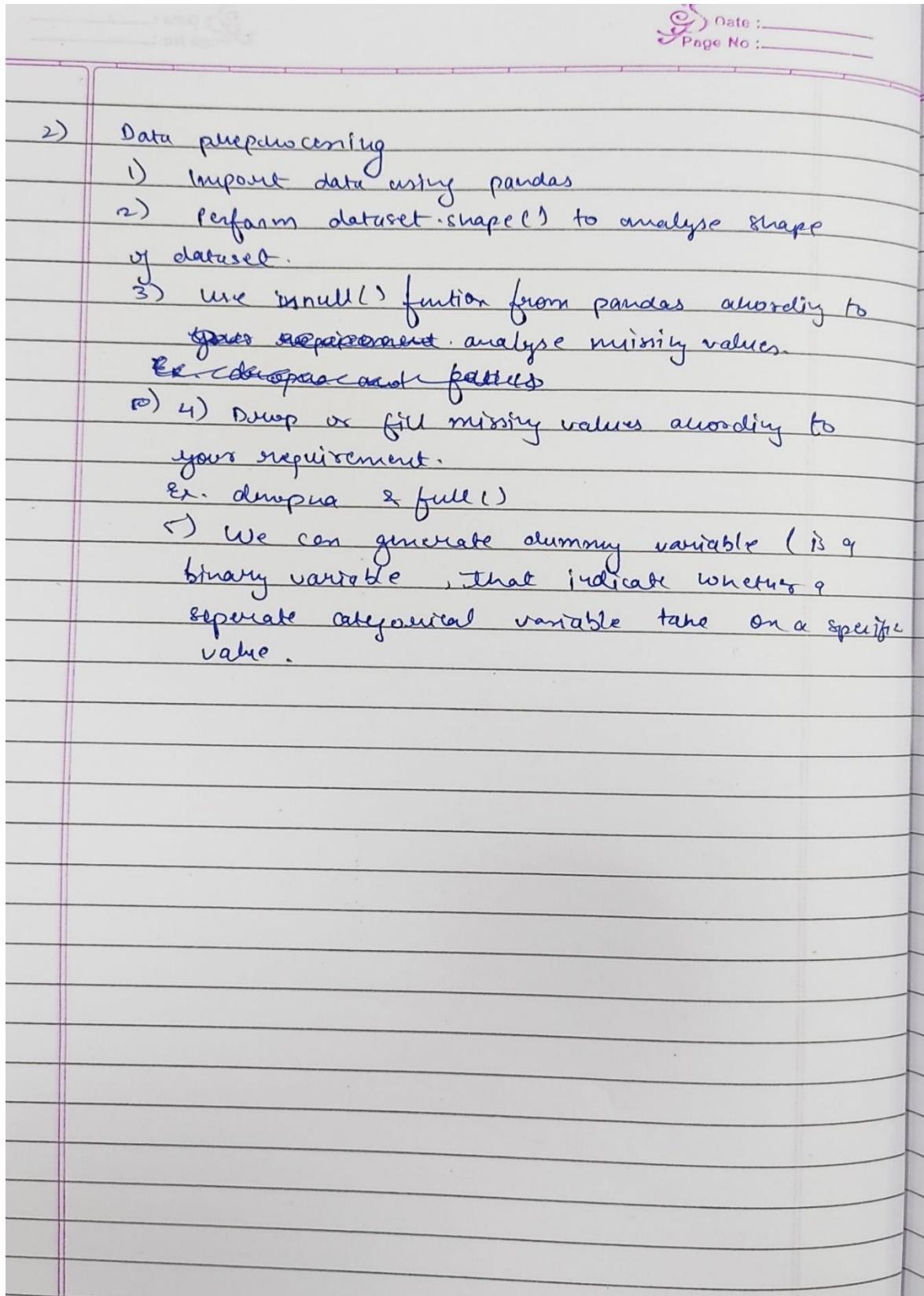
2]:

| | 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 |

04.04.2024

Demonstrate various data processing unit for a dataset.

Screenshot



Code:

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

dataset = pd.read_csv("Data.csv")
df = pd.DataFrame(dataset)
print(df.head())

   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

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

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]]]

print(y)

['No' 'Yes' 'No' 'No' 'Yes' 'Yes' 'No' 'Yes' 'No' 'Yes']
```

```

df.isnull().sum()

Country      0
Age          1
Salary       1
Purchased    0
dtype: int64

df1 = df.copy()

# 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)

```

: df2
:   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

```

```

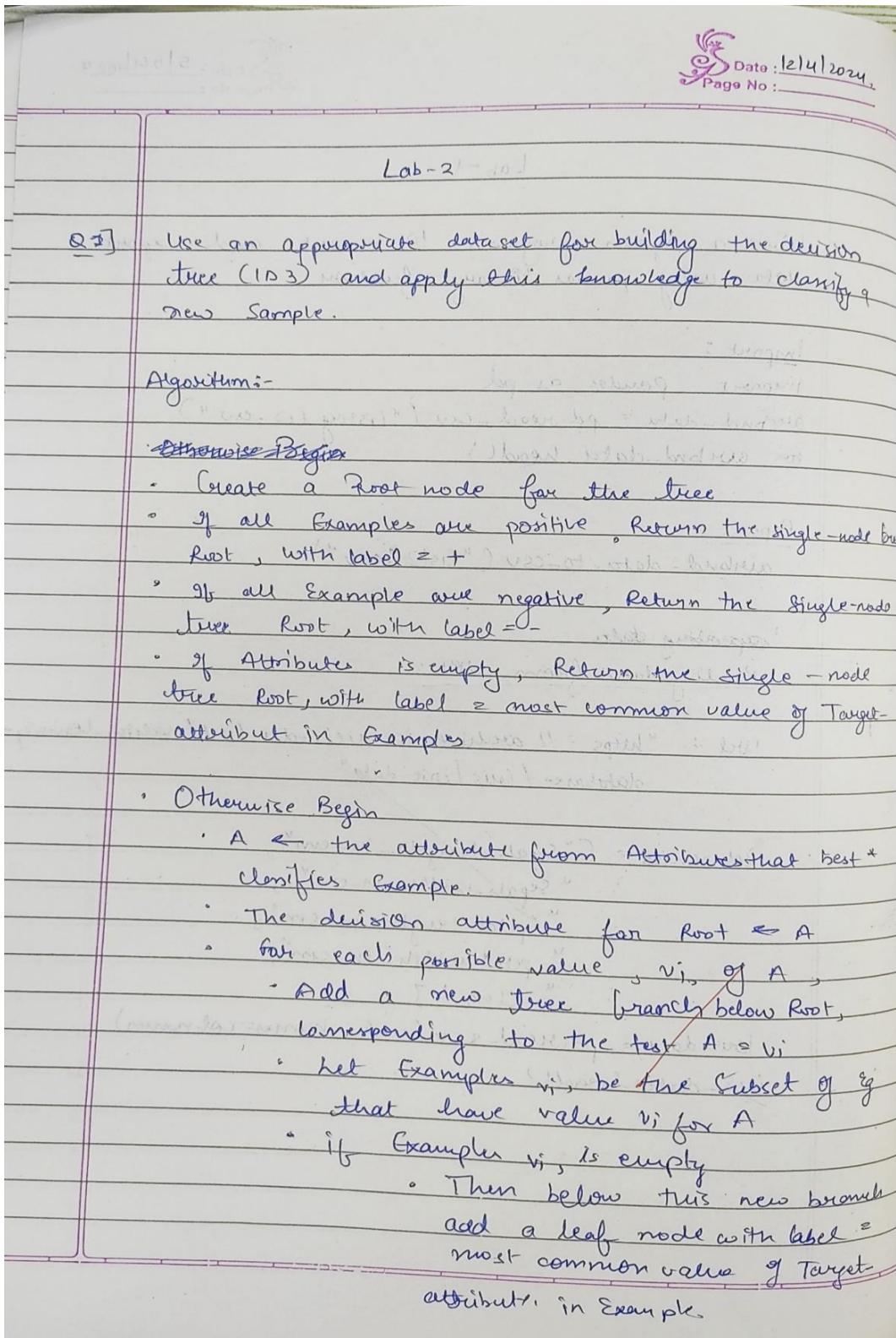
: pd.get_dummies(df2)
:   Age   Salary Country_France Country_Germany Country_Spain Purchased_No Purchased_Yes
: 0  44.0  72000.0        True        False       False      True      False
: 1  27.0  48000.0       False       False       True      False      True
: 2  30.0  54000.0       False       True       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

```

12.04.2024

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

4.snapshot



• Else below this new branch add the subtree ID3

- End
- Return Root.

Output :

(1) Entropy of the dataset : 0.93313

(2) Calculating Entropy & IG for each case :

Pregnancies - Entropy : 3.482 , Info-gain: 0.062

Glucose - Entropy : 6.751 , Info-gain: 0.304

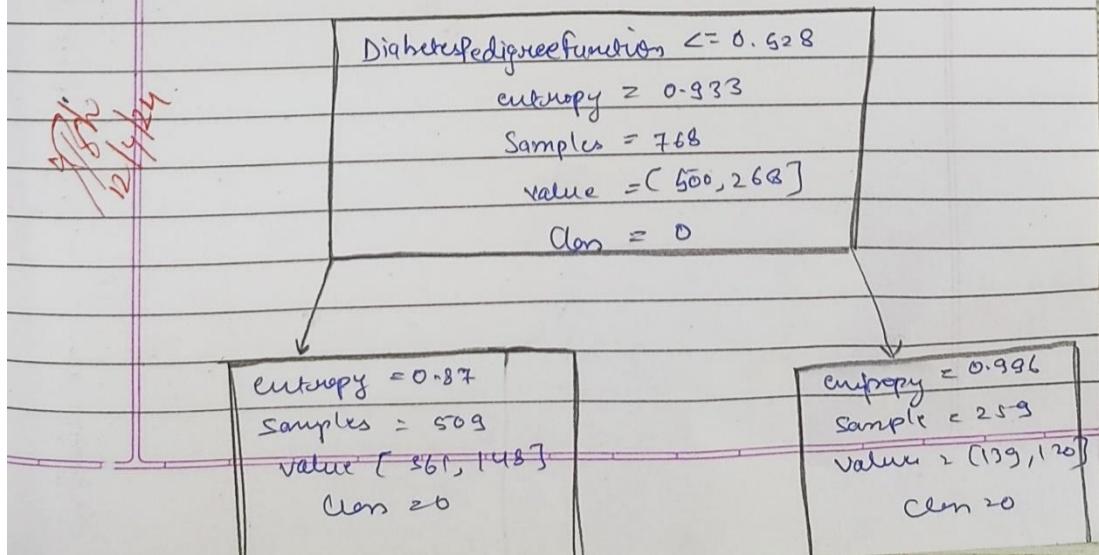
BP - Entropy : 4.732 , Info-gain : 0.059

Insulin - Entropy : 4.682 , Info-gain: 0.277

BMI - Entropy : 7.594 , Info-gain: 0.344.

Age - Entropy : 5.029 → Info-gain : 0.141

(3) Decision Tree :



1.importing database

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

[2] df = pd.read_csv('/content/diabetes.csv')
df.head()
```

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

Next steps: View recommended plots

2.Calculating entropy and information gain.

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

    entropy = 0
    for value in target_values:
        value_count = len(data[data[target_column] == value])
        proportion = value_count / total_rows
        entropy -= proportion * math.log2(proportion)

    return entropy

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

Entropy of the dataset: 0.9331343166407831

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

    entropy = 0
    for value in target_values:
        value_count = len(data[data[target_column] == value])
        proportion = value_count / total_rows
        entropy -= proportion * math.log2(proportion) if proportion != 0 else 0

    return entropy

def calculate_information_gain(data, feature, target_column):

    unique_values = data[feature].unique()
    weighted_entropy = 0

    for value in unique_values:
        subset = data[data[feature] == value]
```

3. Making Decision tree.

```

unique_values = data[feature].unique()
weighted_entropy = 0

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

information_gain = entropy_outcome - weighted_entropy

return information_gain

```

```

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

```

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

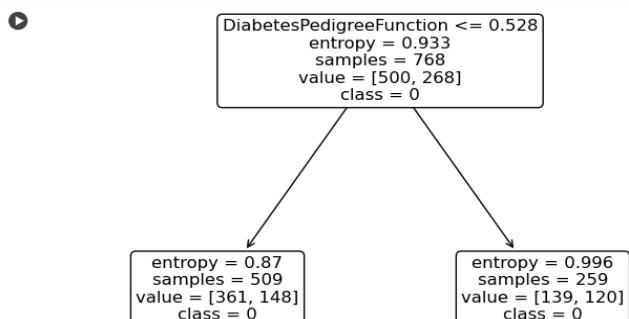
```

selected_feature = 'DiabetesPedigreeFunction'

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

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

```



```

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

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

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

    tree = {best_feature: {}}

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

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

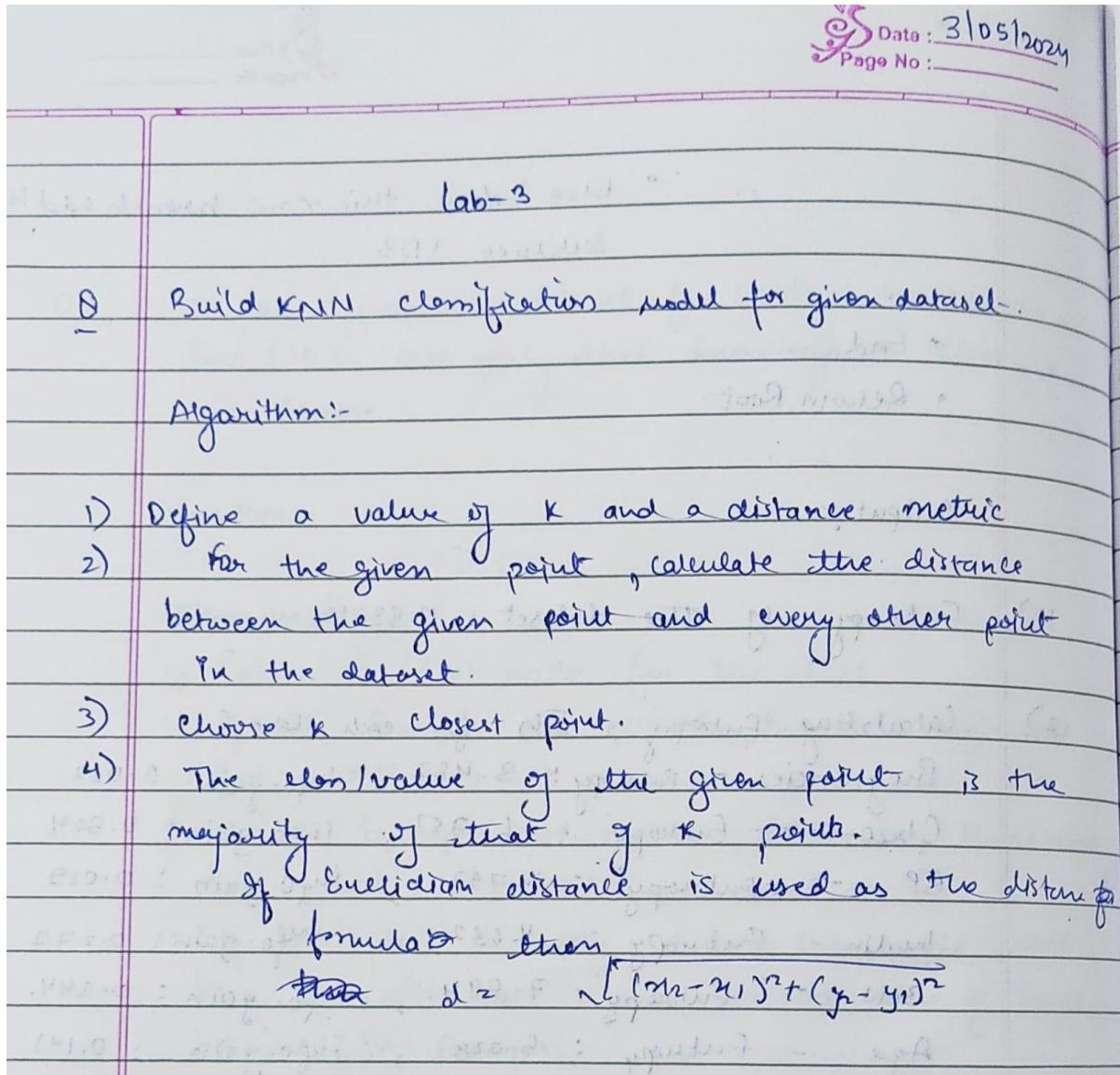
    return tree

```

19.04.2024

Build KNN Classification model for given dataset.

Snapshot:



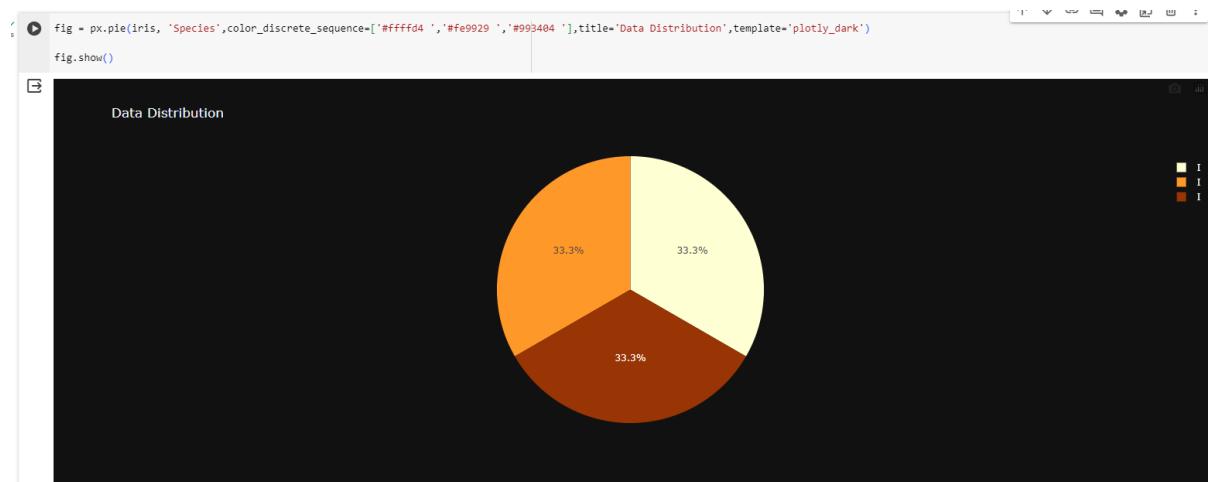
Code:

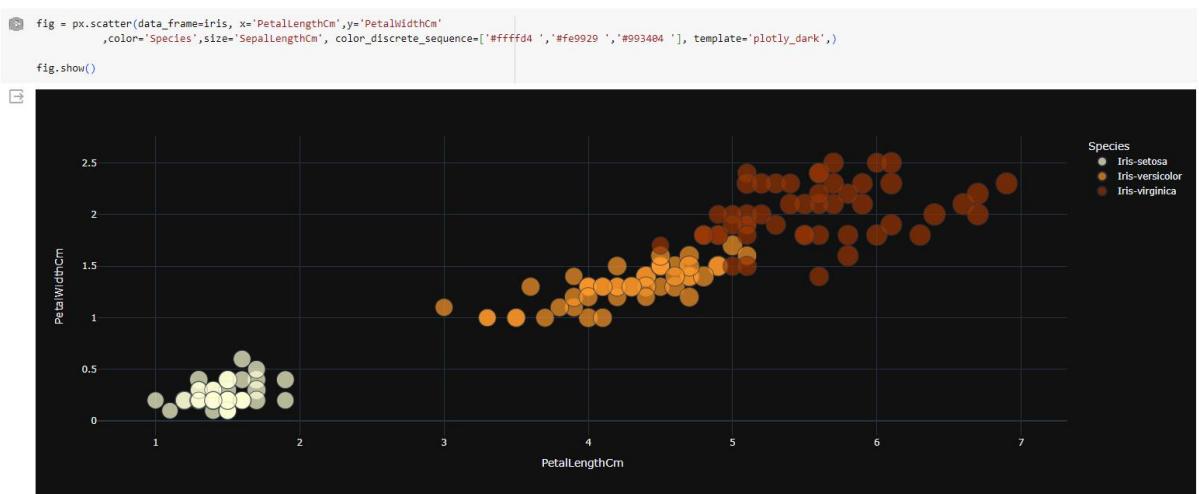
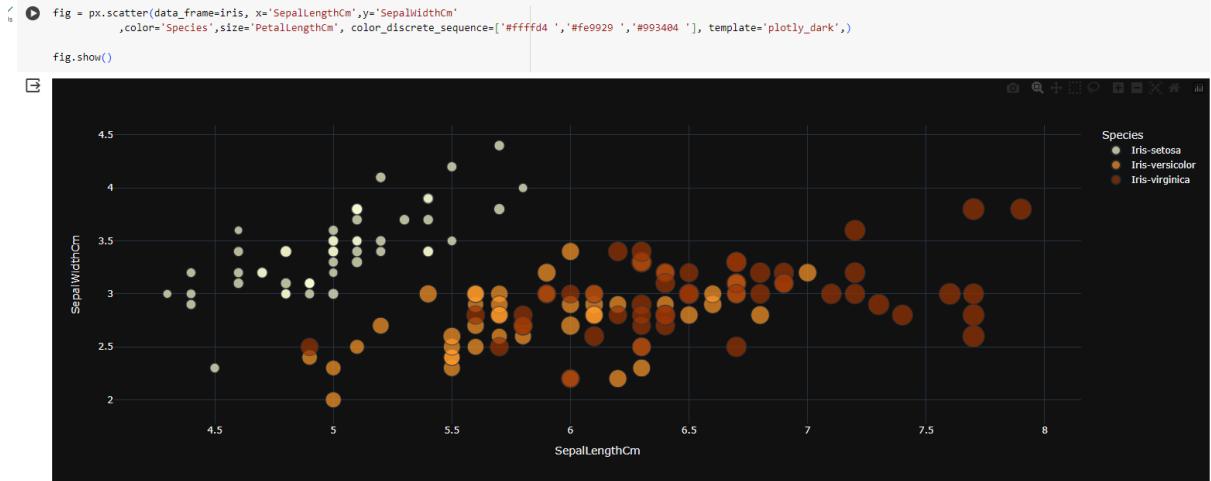
```
✓ [1] import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import plotly.express as px  
import seaborn as sns  
  
✓ [3] iris = pd.read_csv("Iris.csv") #Load Data  
iris.drop('Id',inplace=True, axis=1) #Drop Id column  
  
✓ [4] iris.head()
```

| | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm | Species | grid icon | more icon |
|---|---------------|--------------|---------------|--------------|-------------|-----------|-----------|
| 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 | | |

Next steps: [View recommended plots](#)

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





```
[9] ls
ls [9] 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 data point in the test dataset.

    Returns:
    -----
    most_occuring_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_occuring_value = max(labels, key=labels.count)
    return most_occuring_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) #
```

```
' [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

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

      KNeighborsClassifier(n_neighbors=7)

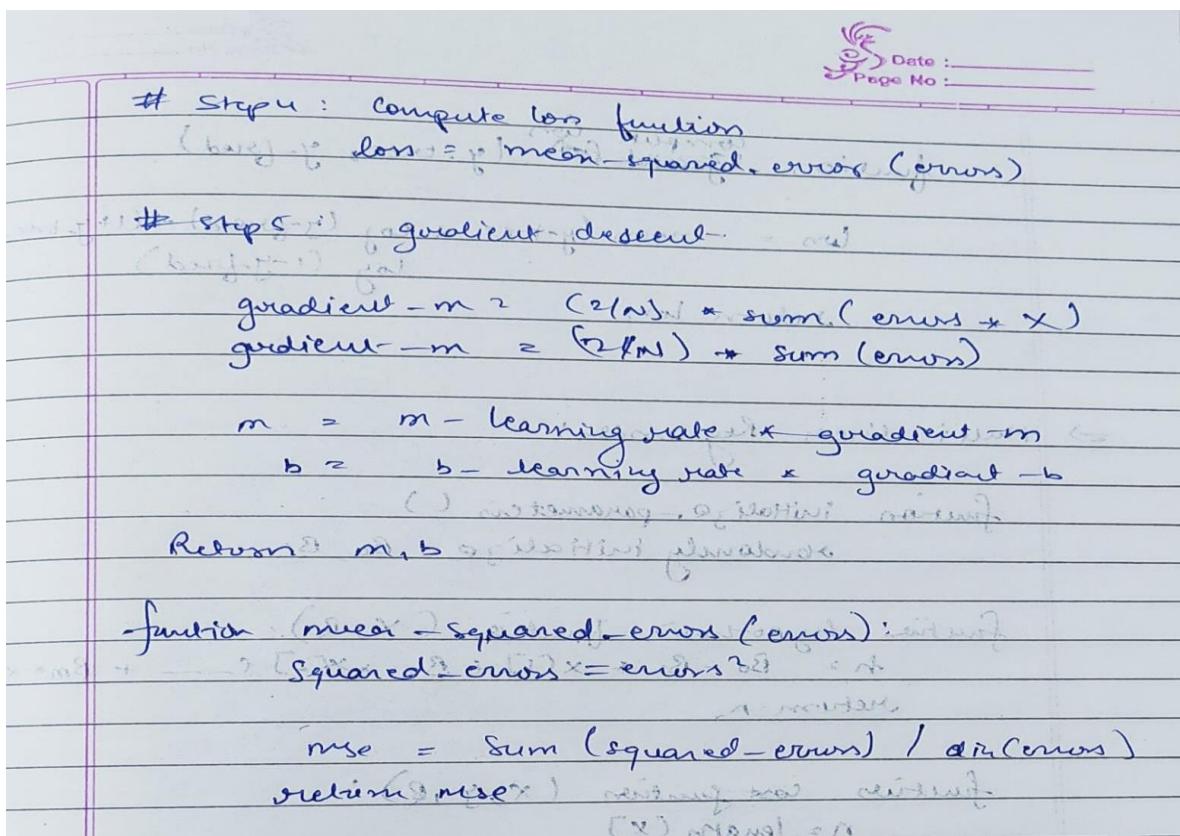
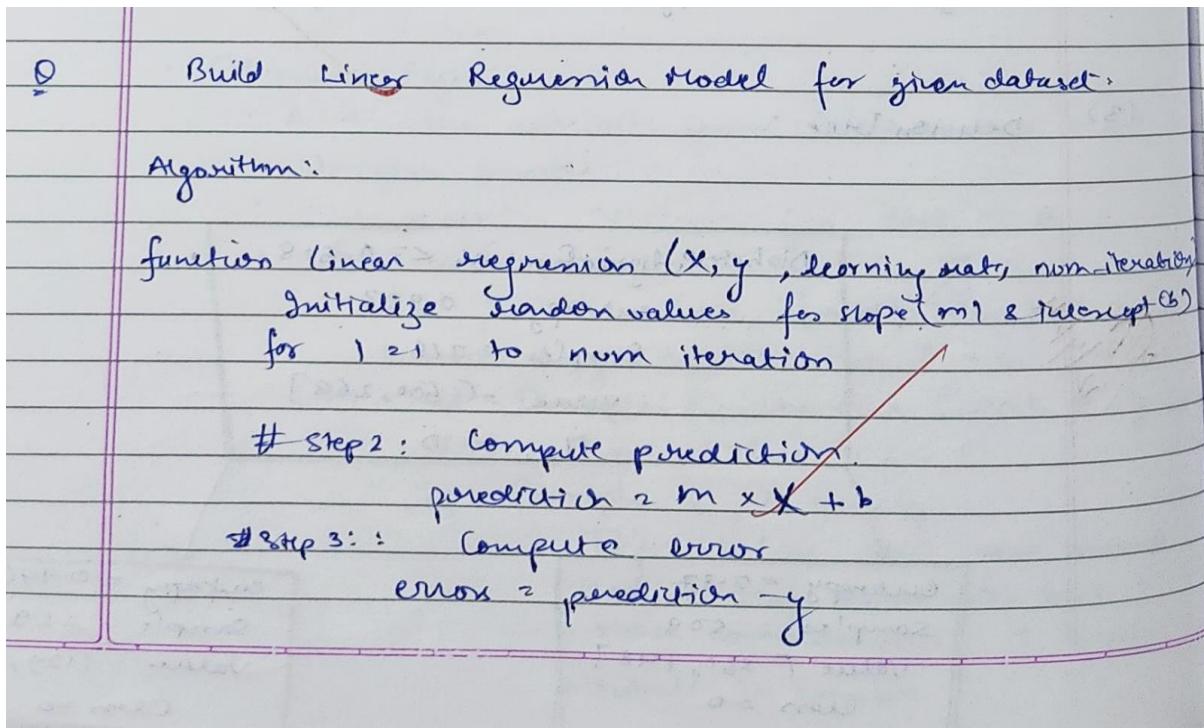
' [17] sk_predictions = skmodel.predict(X_test)
      sk_accuracy = compute_accuracy(y_test, sk_predictions)
      print(f" sklearn-model got accuracy score of : {sk_accuracy}")

      sklearn-model got accuracy score of : 0.9666666666666667
```

19.04.2024

Build linear regression model for the dataset:

Snapshot:



Code:

```
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('train.csv')
test_data = pd.read_csv('test.csv')

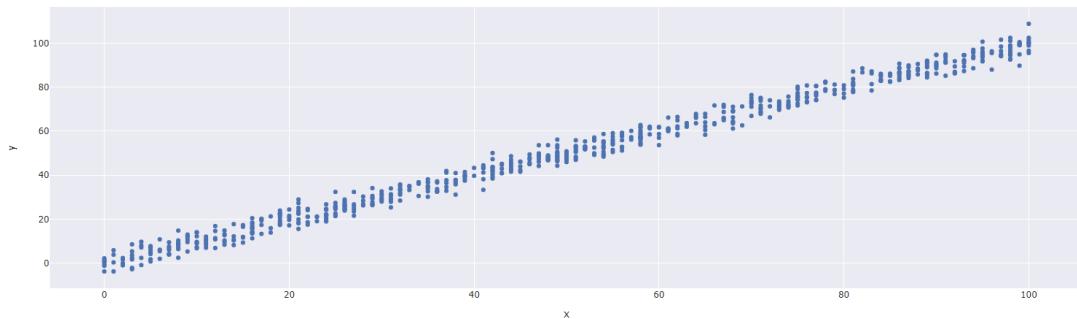
# Remove rows with missing values
train_data = train_data.dropna()
test_data = test_data.dropna()

[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 |

Next steps: [View recommended plots](#)

```
[4] 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] """
Standardizes the input data using mean and standard deviation.

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

Returns:
    Tuple of standardized training and testing data.
"""

# Calculate the mean and standard deviation using the training data
mean = np.mean(X_train, axis=0)
std = np.std(X_train, axis=0)

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

return X_train, X_test

X_train, X_test = standardize_data(X_train, X_test)

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

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

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

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

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

Attributes:
"""

[8] 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).
        """

```

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

        Parameters:
            filename (str): The name of the file to load the model from.
        """
        with open(filename, 'rb') as file:
            model_data = pickle.load(file)

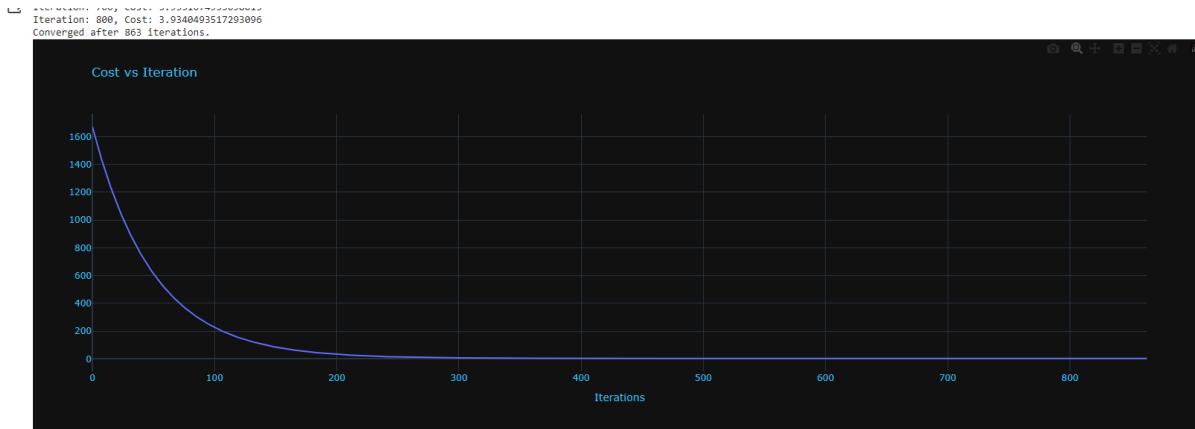
        # Create a new instance of the class and initialize it with the loaded parameters
        loaded_model = cls(model_data['learning_rate'], model_data['convergence_tol'])
        loaded_model.W = model_data['W']
        loaded_model.b = model_data['b']

        return loaded_model

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

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

```



```
[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:

```

```
[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])
```

19.04.2024

Implement logistic regression using appropriate dataset:

Snapshot:

(x) input set
Q Implement Logistic regression with appropriate dataset.
(num-itr):
(fun)(x,y, learning rate, num-itr):
for i = 1 to num-itr:
 logit = x * w + b
 pred = sigmoid(logit)
 loss = compute loss(y, pred)
 update weight & bias with gradient
Return weight & bias

fun sigmoid(x):
 return 1/(1+exp(-x))

Date : _____
Page No. : _____

function compute loss(y-tree, y-pred):
 loss = -mean(y-tree * log(y-pred) + (1-y-tree) * log(1-y-pred))
 return loss

Compute loss = (1/y) * mean(-y-tree * log(y-pred) + (1-y-tree) * log(1-y-pred))

Code:

```
4s  import math
      import numpy as np
      import pandas as pd
      import matplotlib.pyplot as plt
      import seaborn as sns
      import plotly.express as px
      import pprint
      import pickle

5s  [4] df = pd.read_csv('breast-cancer.csv')

6s  [5] 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 x 32 columns

6s  [6] df.drop('id', axis=1, inplace=True) #drop redundant columns

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

8s  [8] corr = df.corr()
```



```

✓ [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)

[> ['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']

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

[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] In [15]: X_train, X_test, y_train, y_test = train_test_split(X,y)

[16] In [16]: 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] In [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).

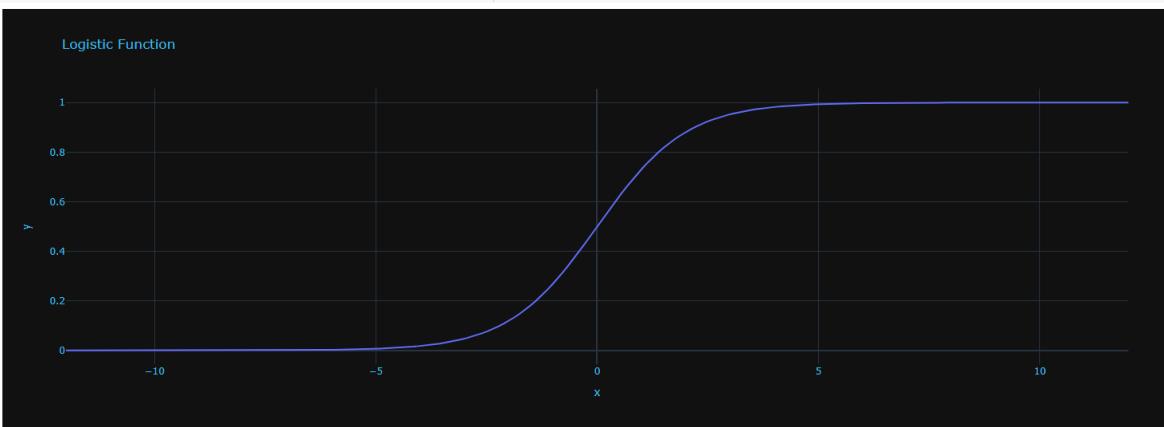
    Example:
        >>> sigmoid(0)
        0.5
    """
    # Compute the sigmoid function using the formula: 1 / (1 + e^(-z)).
    sigmoid_result = 1 / (1 + np.exp(-z))

    # Return the computed sigmoid value.
    return sigmoid_result

[18] In [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.
    """

[19]
    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

```

```
[19]     """
    # calculate 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
```

```
19]     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 % 1000 == 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.
         """

```

```
[19]     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.

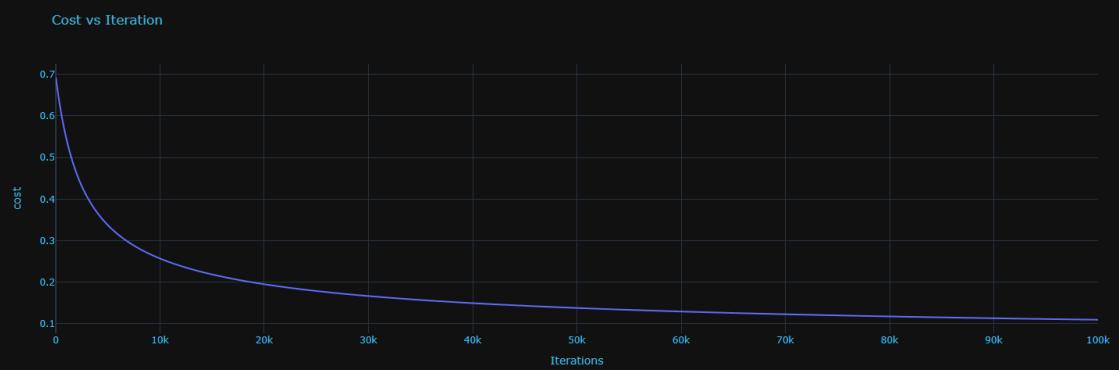
        Returns:
            LogisticRegression: An instance of the LogisticRegression class with loaded parameters.
        """
        with open(filename, 'rb') as file:
            model_data = pickle.load(file)

        # 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
```

```
[20] lg = LogisticRegression()
lg.fit(X_train, y_train, 100000)
```

Cost after iteration 0: 0.6931471605599454
Cost after iteration 10000: 0.2570778377358246
Cost after iteration 20000: 0.19529178673689726
Cost after iteration 30000: 0.16685820756163852
Cost after iteration 40000: 0.14978939548676498
Cost after iteration 50000: 0.1381876134031554
Cost after iteration 60000: 0.1296814121248933
Cost after iteration 70000: 0.1231144039988139
Cost after iteration 80000: 0.11785163708799082
Cost after iteration 90000: 0.1135137713860602



```
[22] lg.save_model("model.pkl")

[23] 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 predictions
        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)
```

```
[23]
    @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)

    @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)
```

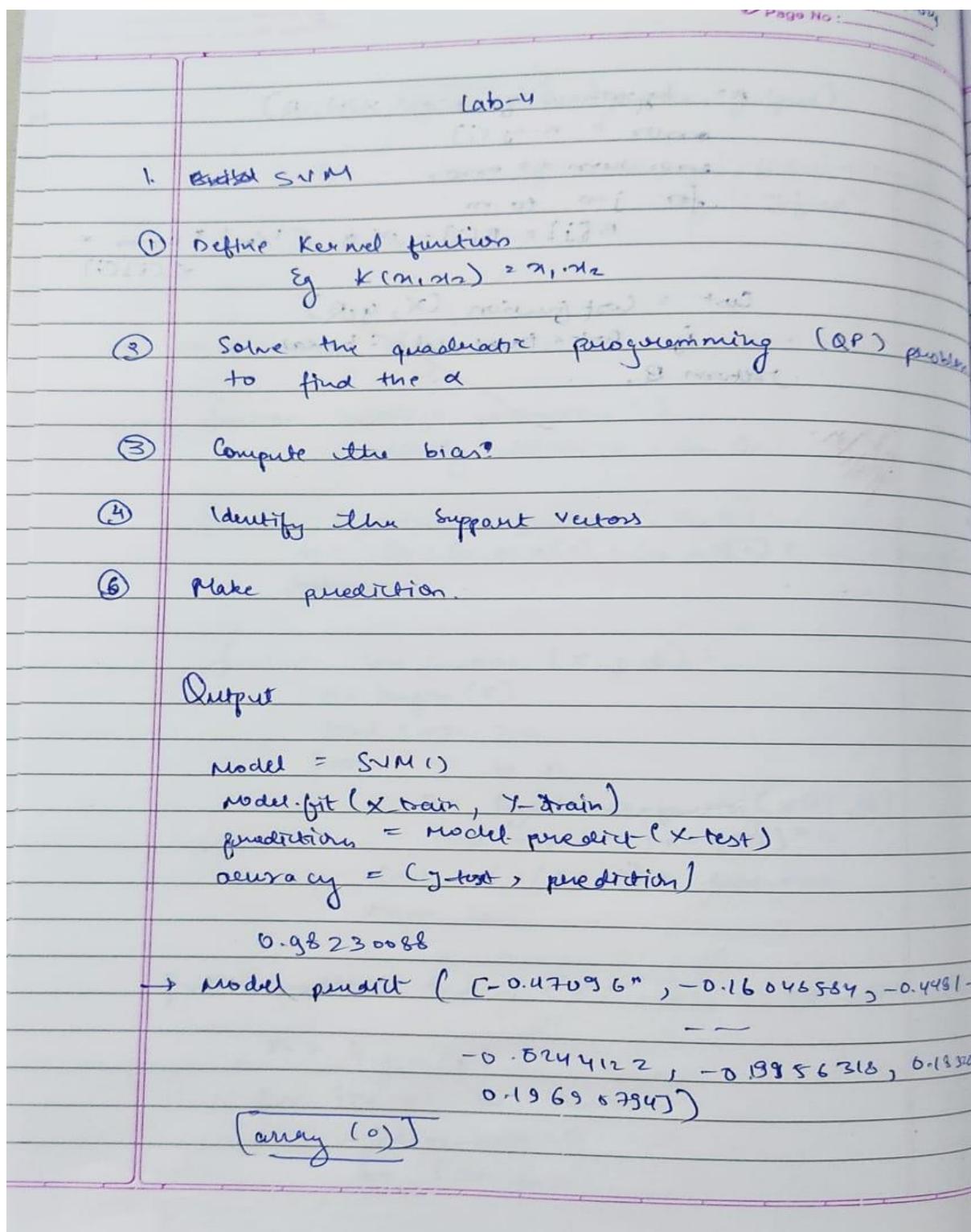
```
[24] model = LogisticRegression.load_model("model.pkl")
```

```
✓ [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%
```

24.05.2024

Build support vector machine for the dataset:

Snapshot:



Code:

```
[1]: from google.colab import drive
drive.mount('/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
```

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

```
[5]: 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 | 0.025470 | 0.017000 | 0.002262 | 0.012080 | 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 validatin

```

```
[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.lambaa * self.w
        db = 0
    else:
        dw = 2 * self.lambaa * 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:
```

```
✓ [18]      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
```

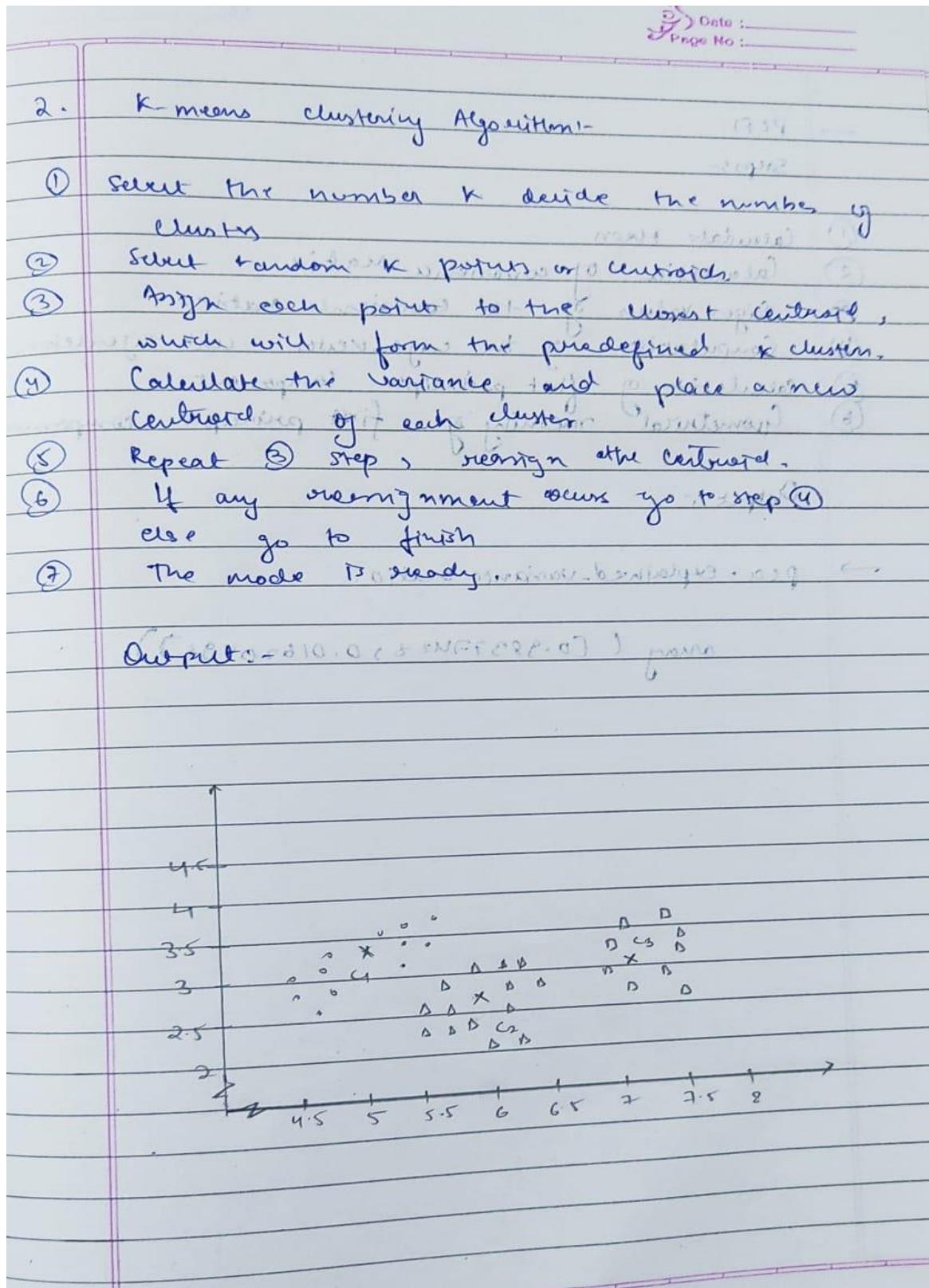
```
[28] model.predict([-0.47069438, -0.16048584, -0.44810956, -0.49199876,  0.23411429,
                  0.02765051, -0.10984741, -0.27623152,  0.41394897, -0.03274296,
                  -0.18269561, -0.22105292, -0.35591235, -0.16192949, -0.23133322,
                  -0.26903951, -0.16890536, -0.33393537,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

```
→ array(0)
```

24.05.2024

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

Snapshot:



Code:

```
[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 [5]
    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
```

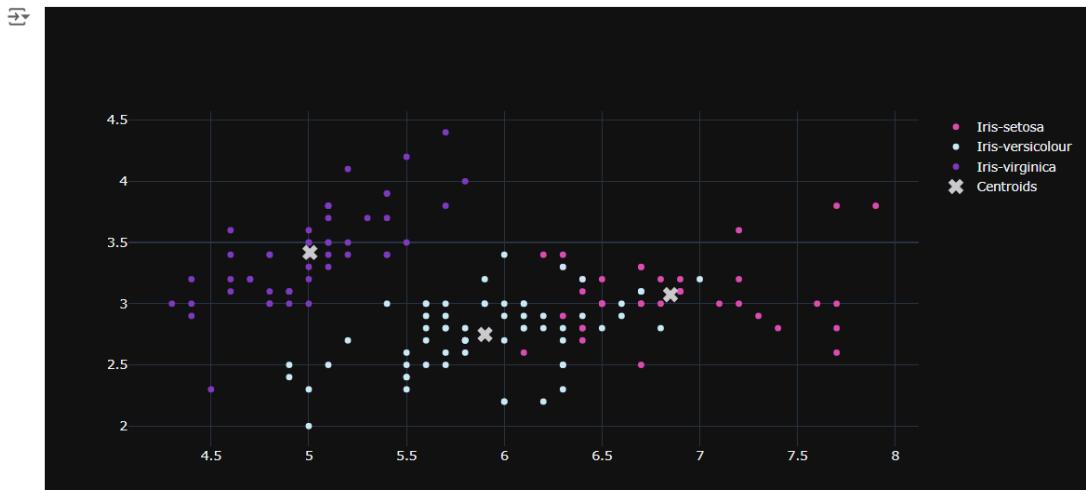
```
✓ 0s [6] X = X.values
```

```
✓ 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'
))
```

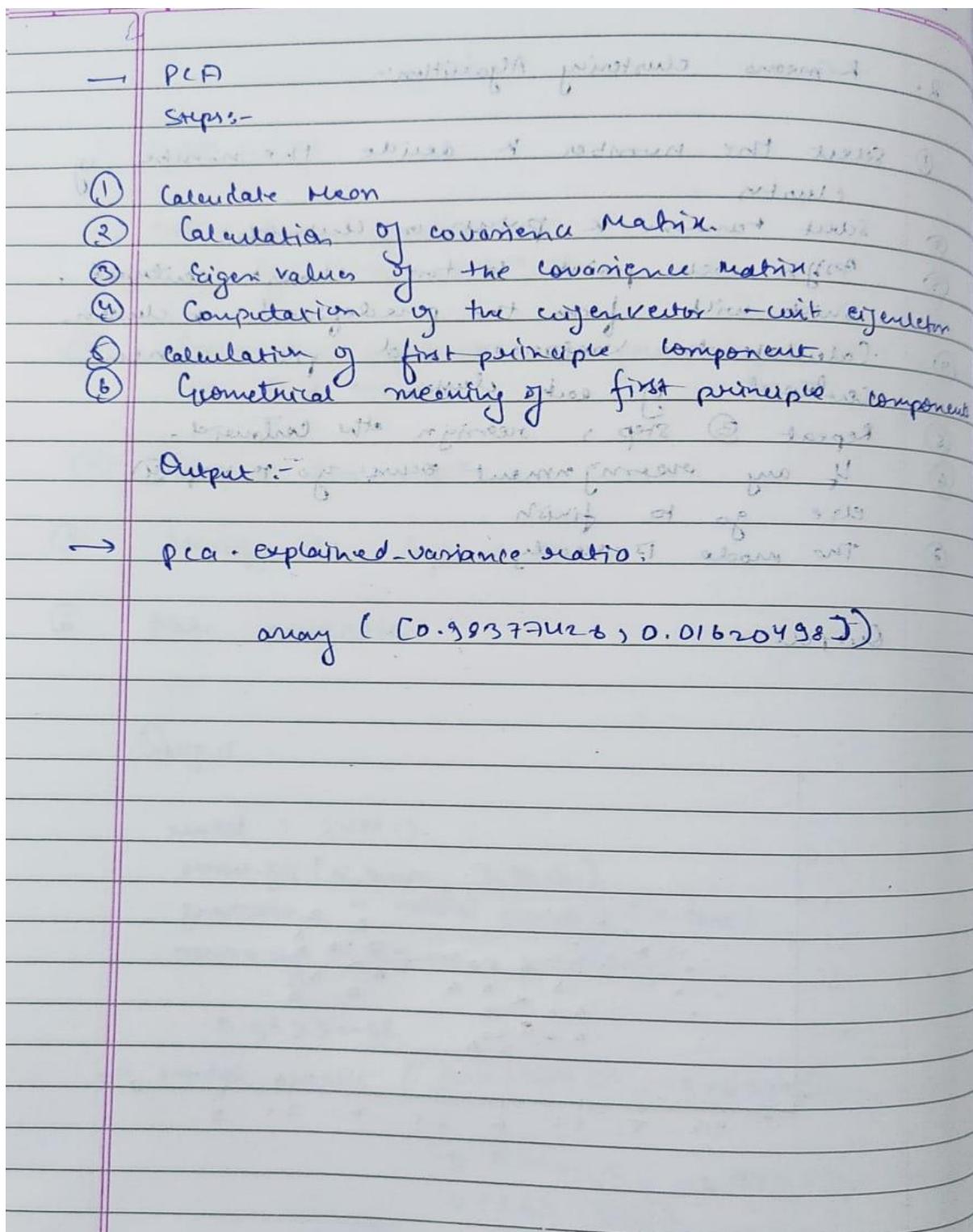
```
[8] ))
fig.add_trace(go.Scatter(
    x=X[points == 2, 0], y=X[points == 2, 1],
    mode='markers',marker_color='#7D3AC1',name='Iris-virginica'
))
fig.add_trace(go.Scatter(
    x=centroids[:, 0], y=centroids[:,1],
    mode='markers',marker_color="#CAC9CD",marker_symbol=4,marker_size=13,name='Centroids'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



24.05.2024

Implement dimensionality reduction using principle component analysis (PCA).

Snapshot:



Code:

```
+ Code + Text
```

✓ [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_p
↳

✓ [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] Inert code cell# 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_
```

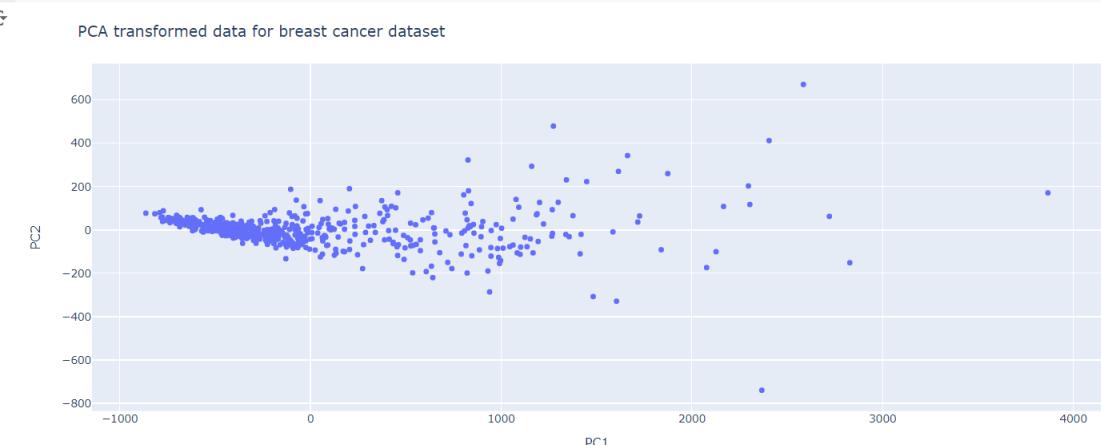
```
→ array([0.98377428, 0.01620498])
```

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

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

```
→ (569,)
```

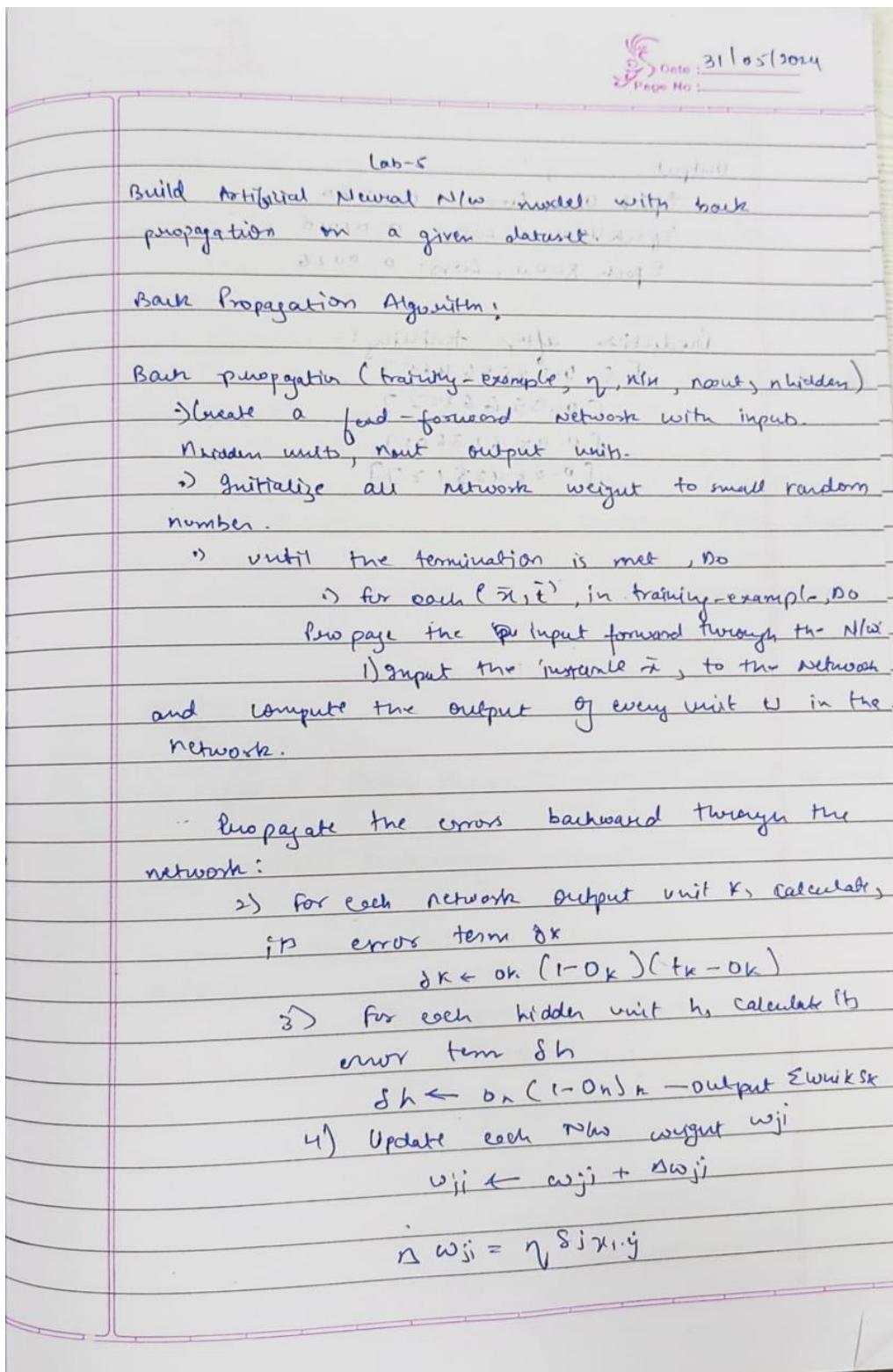
```
✓ [2] fig = px.scatter(x=X_transformed[:,0], y=X_transformed[:,1])  
fig.update_layout(  
    title="PCA transformed data for breast cancer dataset",  
    xaxis_title="PC1",  
    yaxis_title="PC2"  
)  
fig.show()
```



31.05.2024

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

Snapshot:



Output:

Epoch 0, loss: 0.2780
Epoch 4000, loss: 0.01056
Epoch 8000, loss: 0.0026.

Prediction after training:-

[0.02586485]

[0.9566322]

[0.95613605]

[0.05428127]

Code:

```
▶ import numpy as np

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)

        # Initialize the biases
        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.bias_output = np.zeros((1, self.output_size))

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

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def feedforward(self, X):
        # Input to hidden
        self.hidden_activation = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = self.sigmoid(self.hidden_activation)

        # Hidden to output
        self.output_activation = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
        self.predicted_output = self.sigmoid(self.output_activation)

        return self.predicted_output

    def backward(self, X, y, learning_rate):
        # Compute the output layer error
        output_error = y - self.predicted_output
        output_delta = output_error * self.sigmoid_derivative(self.predicted_output)

        # Compute the hidden layer error
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)
```

```

# Update weights and biases
self.weights_hidden_output += np.dot(self.hidden_output.T, output_delta) * learning_rate
self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate
self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate

def train(self, X, y, epochs, learning_rate):
    for epoch in range(epochs):
        output = self.feedforward(X)
        self.backward(X, y, learning_rate)
        if epoch % 4000 == 0:
            loss = np.mean(np.square(y - output))
            print(f"Epoch {epoch}, Loss:{loss}")

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

nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)
nn.train(X, y, epochs=10000, learning_rate=0.1)

# Test the trained model
output = nn.feedforward(X)
print("Predictions after training:")
print(output)

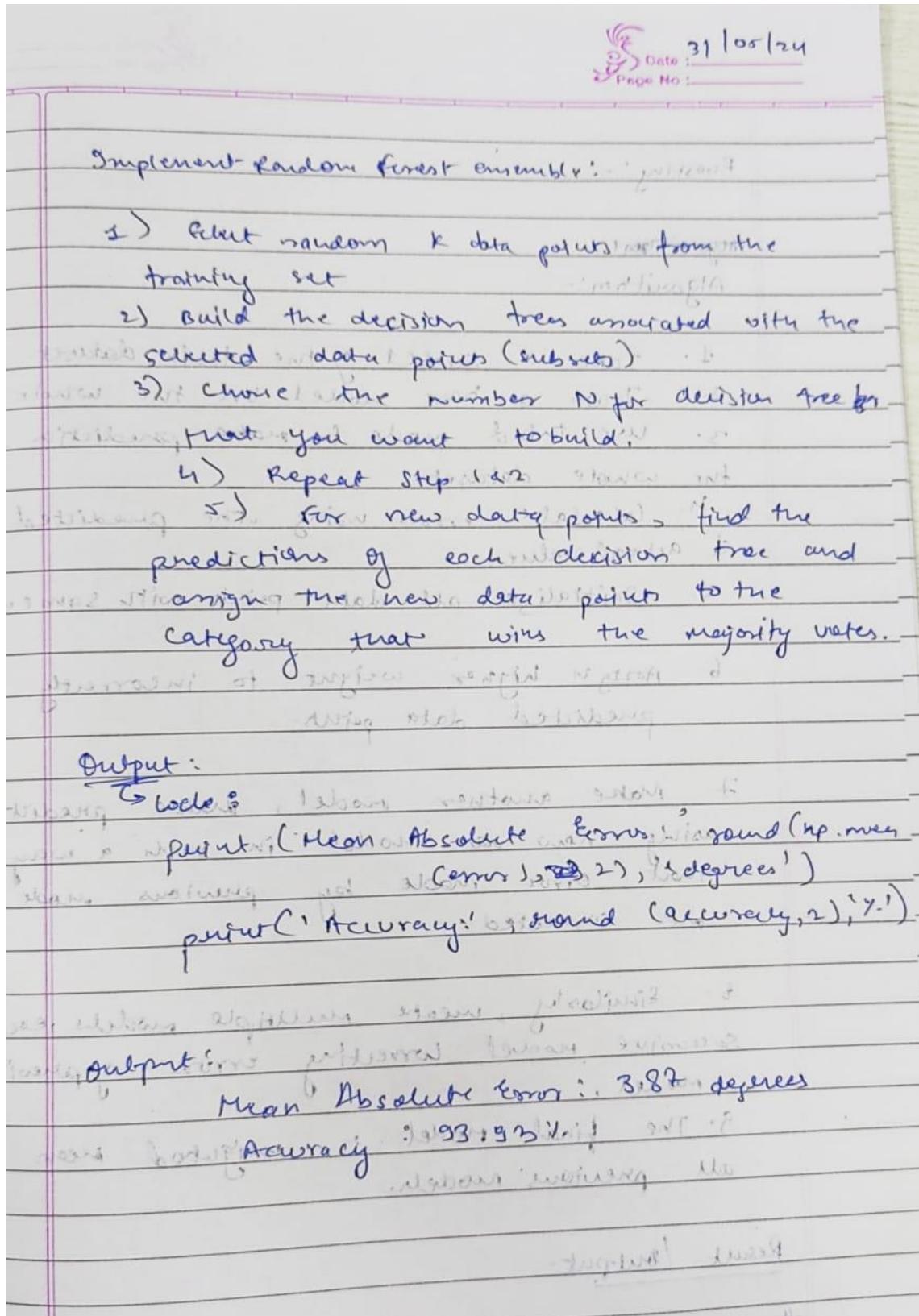
```

Epoch 0, Loss:0.2746467774699536
 Epoch 4000, Loss:0.007581868337942686
 Epoch 8000, Loss:0.002423001703124975
 Predictions after training:
 [[0.03880939]
 [0.95944268]
 [0.95465187]
 [0.04336319]]

31.05.2024

2.a) Implement Random Forest ensemble method on a given dataset.

Screenshot:



Code:

```
+ Code + Text
✓ 0s # Pandas is used for data manipulation
  import pandas as pd
  # Read in data and display first 5 rows
  features = pd.read_csv('temps.csv')
  features.head(5)

  year month day week temp_2 temp_1 average actual forecast_noaa forecast_acc forecast_under friend
  0 2016 1 1 Fri 45 45 45.6 45 43 50 44 29
  1 2016 1 2 Sat 44 45 45.7 44 41 50 44 61
  2 2016 1 3 Sun 45 44 45.8 41 43 46 47 56
  3 2016 1 4 Mon 44 41 45.9 40 44 48 46 53
  4 2016 1 5 Tues 41 40 46.0 44 46 46 46 41

Next steps: View recommended plots

✓ 0s [3] print('The shape of our features is:', features.shape)
  # The shape of our features is: (348, 9)

  The shape of our features is: (348, 12)

✓ 0s [4] # Descriptive statistics for each column
  features.describe()

  year month day temp_2 temp_1 average actual forecast_noaa forecast_acc forecast_under friend
  count 348.0 348.000000 348.000000 348.000000 348.000000 348.000000 348.000000 348.000000 348.000000 348.000000
  mean 2016.0 6.477011 15.514368 62.652299 62.701149 59.760632 62.543103 57.238506 62.373563 59.772989 60.034483
  std 0.0 3.498380 8.772982 12.165398 12.120542 10.527306 11.794146 10.605746 10.549381 10.705256 15.626179
  min 2016.0 1.000000 1.000000 35.000000 35.000000 45.100000 35.000000 41.000000 46.000000 44.000000 28.000000
  25% 2016.0 3.000000 8.000000 54.000000 54.000000 49.975000 54.000000 48.000000 53.000000 50.000000 47.750000
  50% 2016.0 6.000000 15.000000 62.500000 62.500000 58.200000 62.500000 56.000000 61.000000 58.000000 60.000000
  75% 2016.0 10.000000 23.000000 71.000000 71.000000 69.025000 71.000000 66.000000 72.000000 69.000000 71.000000
  max 2016.0 12.000000 31.000000 117.000000 117.000000 77.400000 92.000000 77.000000 82.000000 79.000000 95.000000
  (1s) completed at 2:56 PM

+ Code + Text
✓ 0s # One-hot encode the data using pandas get_dummies
  features = pd.get_dummies(features)
  # Display the first 5 rows of the last 12 columns
  features.iloc[:,5: ].head(5)

  average actual forecast_noaa forecast_acc forecast_under friend week_Fri week_Mon week_Sat week_Sun week_Thurs week_Tues week_Wed
  0 45.6 45 43 50 44 29 True False False False False False False
  1 45.7 44 41 50 44 61 False False True False False False False
  2 45.8 41 43 46 47 56 False False False True False False False
  3 45.9 40 44 48 46 53 False True False False False False False
  4 46.0 44 46 46 46 41 False False False False True False

✓ 0s [6] # Use numpy to convert to arrays
  import numpy as np
  # Labels are the values we want to predict
  labels = np.array(features['actual'])
  # Remove the labels from the features
  # axis 1 refers to the columns
  features = features.drop('actual', axis = 1)
  # Saving feature names for later use
  feature_list = list(features.columns)
  # Convert to numpy array
  features = np.array(features)

✓ 0s [7] # Using Skicit-learn to split data into training and testing sets
  from sklearn.model_selection import train_test_split
  # Split the data into training and testing sets
  train_features, test_features, train_labels, test_labels = train_test_split(features, labels, test_size = 0.25, random_state = 42)

✓ 0s [9] print('Training Features Shape:', train_features.shape)
  print('Training Labels Shape:', train_labels.shape)
  print('Testing Features Shape:', test_features.shape)
  print('Testing Labels Shape:', test_labels.shape)
```

+ Code + Text

```
[9] Training Features Shape: (261, 17)
    Training Labels Shape: (261,)
    Testing Features Shape: (87, 17)
    Testing Labels Shape: (87,)

[11] # The baseline predictions are the historical averages
    baseline_preds = test_features[:, feature_list.index('average')]
    # Baseline errors, and display average baseline error
    baseline_errors = abs(baseline_preds - test_labels)
    print('Average baseline error: ', round(np.mean(baseline_errors), 2))

    ↴ Average baseline error:  5.06

[12] # Import the model we are using
    from sklearn.ensemble import RandomForestRegressor
    # Instantiate model with 1000 decision trees
    rf = RandomForestRegressor(n_estimators = 1000, random_state = 42)
    # Train the model on training data
    rf.fit(train_features, train_labels);

[14] # Use the forest's predict method on the test data
    predictions = rf.predict(test_features)
    # Calculate the absolute errors
    errors = abs(predictions - test_labels)
    # Print out the mean absolute error (mae)
    print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')

    ↴ Mean Absolute Error: 3.87 degrees.

[15] # Calculate mean absolute percentage error (MAPE)
    mape = 100 * (errors / test_labels)
    # Calculate and display accuracy
    accuracy = 100 - np.mean(mape)
    print('Accuracy:', round(accuracy, 2), '%.')

    ↴ Accuracy: 93.93 %.
```

+ Code + Text

```
✓ [16] # Import tools needed for visualization
    from sklearn.tree import export_graphviz
    import pydot
    # Pull out one tree from the forest
    tree = rf.estimators_[5]
    # Import tools needed for visualization
    from sklearn.tree import export_graphviz
    import pydot
    # Pull out one tree from the forest
    tree = rf.estimators_[5]
    # Export the image to a dot file
    export_graphviz(tree, out_file = 'tree.dot', feature_names = feature_list, rounded = True, precision = 1)
    # Use dot file to create a graph
    (graph, ) = pydot.graph_from_dot_file('tree.dot')
    # Write graph to a png file
    graph.write_png('tree.png')

✓ [17] # Limit depth of tree to 3 levels
    rf_small = RandomForestRegressor(n_estimators=10, max_depth = 3)
    rf_small.fit(train_features, train_labels)
    # Extract the small tree
    tree_small = rf_small.estimators_[5]
    # Save the tree as a png image
    export_graphviz(tree_small, out_file = 'small_tree.dot', feature_names = feature_list, rounded = True, precision = 1)
    (graph, ) = pydot.graph_from_dot_file('small_tree.dot')
    graph.write_png('small_tree.png');

✓ [18] # Get numerical feature importances
    importances = list(rf.feature_importances_)
    # List of tuples with variable and importance
    feature_importances = [(feature, round(importance, 2)) for feature, importance in zip(feature_list, importances)]
    # Sort the feature importances by most important first
    feature_importances = sorted(feature_importances, key = lambda x: x[1], reverse = True)
    # Print out the feature and importances
    [print('Variable: {:20} Importance: {}'.format(*pair)) for pair in feature_importances];

    ↴ Variable: temp_1           Importance: 0.66
    Variable: average            Importance: 0.15
    Variable: forecast_noaa      Importance: 0.05
    Variable: forecast_acc       Importance: 0.03
    Variable: day                 Importance: 0.02
```

```

+ Code + Text
[18] Variable: temp_1           Importance: 0.66
     Variable: average          Importance: 0.15
     Variable: forecast_noaa    Importance: 0.05
     Variable: forecast_acc     Importance: 0.03
     Variable: day               Importance: 0.02
     Variable: temp_2             Importance: 0.02
     Variable: forecast_under   Importance: 0.02
     Variable: friend            Importance: 0.02
     Variable: month              Importance: 0.01
     Variable: year               Importance: 0.0
     Variable: week_Fri          Importance: 0.0
     Variable: week_Mon          Importance: 0.0
     Variable: week_Sat          Importance: 0.0
     Variable: week_Sun          Importance: 0.0
     Variable: week_Thurs         Importance: 0.0
     Variable: week_Tues         Importance: 0.0
     Variable: week_Wed          Importance: 0.0

5s  # New random forest with only the two most important variables
rf_most_important = RandomForestRegressor(n_estimators= 1000, random_state=42)
# Extract the two most important features
important_indices = [feature_list.index('temp_1'), feature_list.index('average')]
train_important = train_features[:, important_indices]
test_important = test_features[:, important_indices]
# Train the random forest
rf_most_important.fit(train_important, train_labels)
# Make predictions and determine the error
predictions = rf_most_important.predict(test_important)
errors = abs(predictions - test_labels)
# Display the performance metrics
print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')
mape = np.mean(100 * (errors / test_labels))
accuracy = 100 - mape
print('Accuracy:', round(accuracy, 2), '%.')

→ Mean Absolute Error: 3.92 degrees.
Accuracy: 93.76 %.

[20] # Import matplotlib for plotting and use magic command for Jupyter Notebooks
import matplotlib.pyplot as plt
%matplotlib inline
# Set the style
+ Code + Text

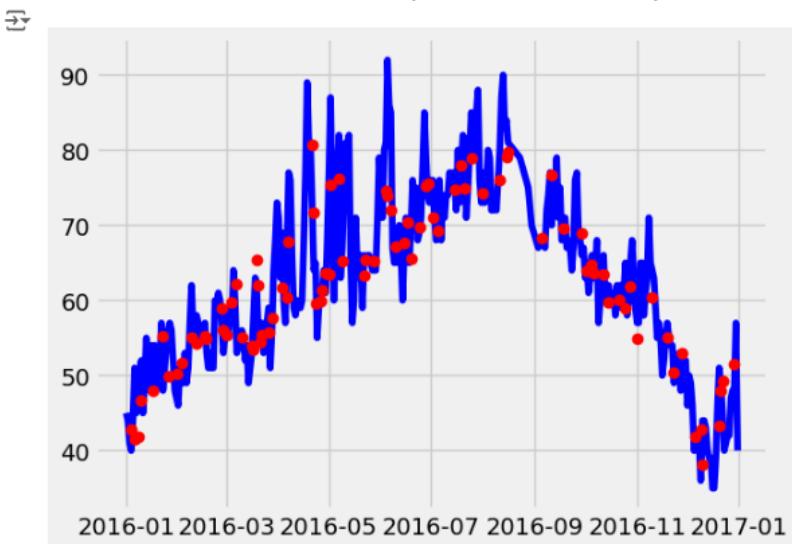
1s  # Import matplotlib for plotting and use magic command for Jupyter Notebooks
import matplotlib.pyplot as plt
%matplotlib inline
# Set the style
plt.style.use('fivethirtyeight')
# list of x locations for plotting
x_values = list(range(len(importances)))
# Make a bar chart
plt.bar(x_values, importances, orientation = 'vertical')
# Tick labels for x axis
plt.xticks(x_values, feature_list, rotation='vertical')
# Axis labels and title
plt.ylabel('Importance'); plt.xlabel('Variable'); plt.title('Variable Importances');

→


```

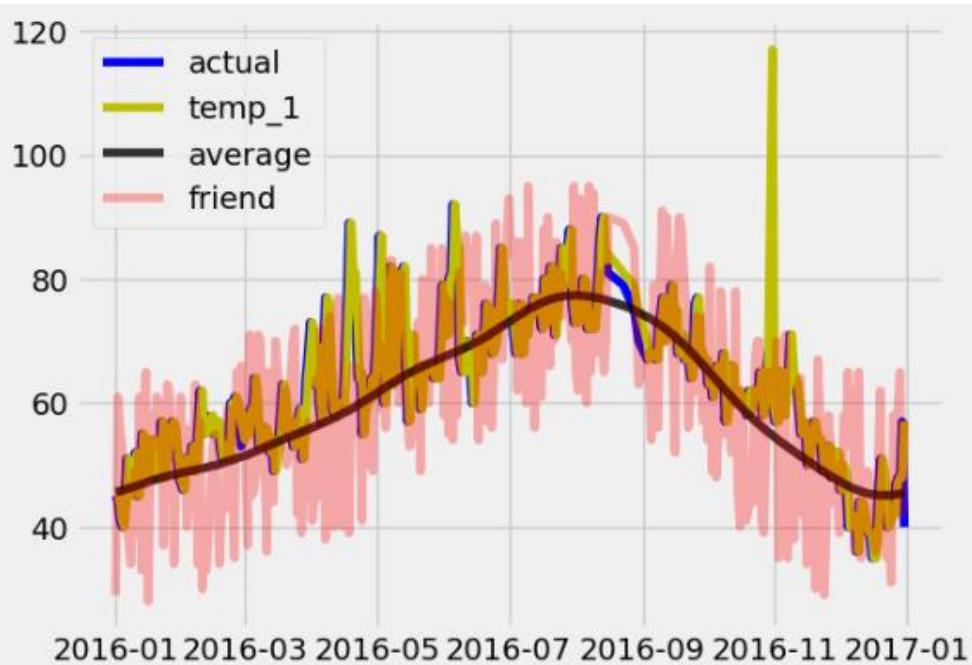
+ Code + Text

```
# Use datetime for creating date objects for plotting
import datetime
# Dates of training values
months = features[:, feature_list.index('month')]
days = features[:, feature_list.index('day')]
years = features[:, feature_list.index('year')]
# List and then convert to datetime object
dates = [str(int(year)) + '-' + str(int(month)) + '-' + str(int(day)) for year, month, day in zip(years, months, days)]
dates = [datetime.datetime.strptime(date, '%Y-%m-%d') for date in dates]
# Dataframe with true values and dates
true_data = pd.DataFrame(data = {'date': dates, 'actual': labels})
# Dates of predictions
months = test_features[:, feature_list.index('month')]
days = test_features[:, feature_list.index('day')]
years = test_features[:, feature_list.index('year')]
# Column of dates
test_dates = [str(int(year)) + '-' + str(int(month)) + '-' + str(int(day)) for year, month, day in zip(years, months, days)]
# Convert to datetime objects
test_dates = [datetime.datetime.strptime(date, '%Y-%m-%d') for date in test_dates]
# Dataframe with predictions and dates
predictions_data = pd.DataFrame(data = {'date': test_dates, 'prediction': predictions})
# Plot the actual values
plt.plot(true_data['date'], true_data['actual'], 'b-', label = 'actual')
# Plot the predicted values
plt.plot(predictions_data['date'], predictions_data['prediction'], 'ro', label = 'prediction')
plt.xticks(rotation = '60')
plt.legend()
# Graph labels
plt.xlabel('Date'); plt.ylabel('Maximum Temperature (F)'); plt.title('Actual and Predicted Values');
```



```
# Make the data accessible for plotting
true_data['temp_1'] = features[:, feature_list.index('temp_1')]
true_data['average'] = features[:, feature_list.index('average')]
true_data['friend'] = features[:, feature_list.index('friend')]
# Plot all the data as lines
plt.plot(true_data['date'], true_data['actual'], 'b-', label = 'actual', alpha = 1.0)
plt.plot(true_data['date'], true_data['temp_1'], 'y-', label = 'temp_1', alpha = 1.0)
plt.plot(true_data['date'], true_data['average'], 'k-', label = 'average', alpha = 0.8)
plt.plot(true_data['date'], true_data['friend'], 'r-', label = 'friend', alpha = 0.3)
# Formatting plot
plt.legend()
plt.xticks(rotation = '60')
# Labels and title
plt.xlabel('Date'); plt.ylabel('Maximum Temperature (F)'); plt.title('Actual Max Temp and Variables');
```

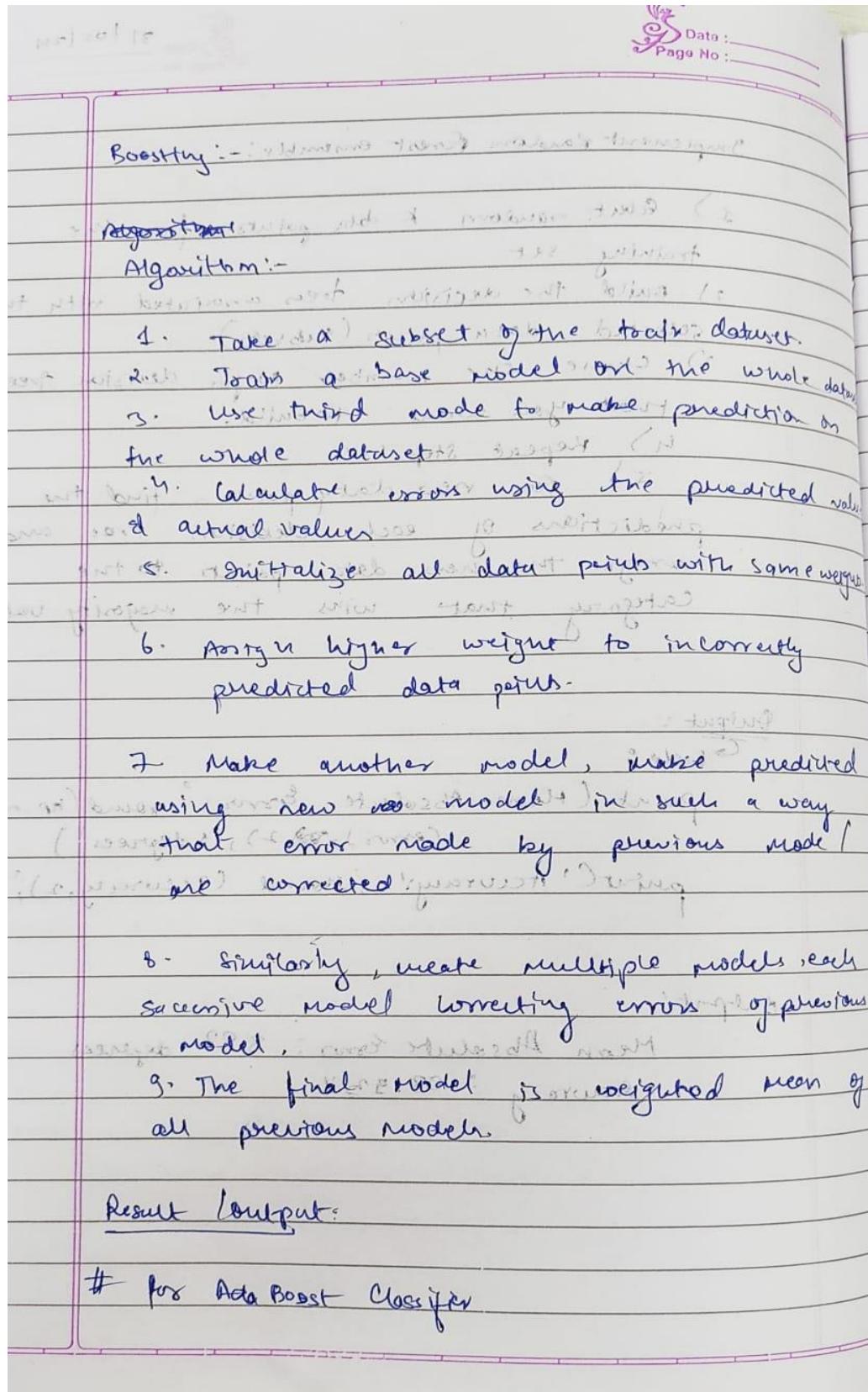
+ Code + Text



31.05.2024

b) Implement Boosting ensemble method on a given dataset

Screenshot:



Training Result

Confusion Matrix
 $\begin{bmatrix} 7310 & 39 \\ 51 & 137 \end{bmatrix}$

Accuracy Score : 0.8324

Testing results:

Confusion Matrix
 $\begin{bmatrix} 123 & 28 \\ 27 & 53 \end{bmatrix}$

Accuracy score : 0.7619

For Gradient Boosting Classifier :-

Training Result:

Confusion Matrix
 $\begin{bmatrix} 342 & 7 \\ 19 & 169 \end{bmatrix}$

Accuracy Score : 0.9516

Training Result:

Confusion Matrix
 $\begin{bmatrix} 116 & 35 \\ 26 & 54 \end{bmatrix}$

Accuracy Score : 0.7359

Code:

+ Code + Text

```
✓ ⓘ import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
sns.set_style("whitegrid")
plt.style.use("fivethirtyeight")
```

```
[3] df = pd.read_csv("diabetes.csv")
df.head()
```

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

Next steps: [View recommended plots](#)

```
✓ ⓘ [4] df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Pregnancies      768 non-null    int64  
 1   Glucose          768 non-null    int64  
 2   BloodPressure    768 non-null    int64  
 3   SkinThickness    768 non-null    int64  
 4   Insulin          768 non-null    int64  
 5   BMI              768 non-null    float64 
 6   DiabetesPedigreeFunction 768 non-null    float64 
 7   Age              768 non-null    int64  
 8   Outcome          768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

```
✓ ⓘ [5] df.isnull().sum()
```

```
✓ ⓘ [5] df.isnull().sum()
```

```
Pregnancies      0
Glucose          0
BloodPressure    0
SkinThickness    0
Insulin          0
BMI              0
DiabetesPedigreeFunction 0
Age              0
Outcome          0
dtype: int64
```

```
✓ ⓘ pd.set_option('display.float_format', '{:.2f}'.format)
df.describe()
```

| | Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin | BMI | DiabetesPedigreeFunction | Age | Outcome |
|-------|-------------|---------|---------------|---------------|---------|--------|--------------------------|--------|---------|
| count | 768.00 | 768.00 | 768.00 | 768.00 | 768.00 | 768.00 | 768.00 | 768.00 | 768.00 |
| mean | 3.85 | 120.89 | 69.11 | 20.54 | 79.80 | 31.99 | 0.47 | 33.24 | 0.35 |
| std | 3.37 | 31.97 | 19.36 | 15.95 | 115.24 | 7.88 | 0.33 | 11.76 | 0.48 |
| min | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.08 | 21.00 | 0.00 |
| 25% | 1.00 | 99.00 | 62.00 | 0.00 | 0.00 | 27.30 | 0.24 | 24.00 | 0.00 |
| 50% | 3.00 | 117.00 | 72.00 | 23.00 | 30.50 | 32.00 | 0.37 | 29.00 | 0.00 |
| 75% | 6.00 | 140.25 | 80.00 | 32.00 | 127.25 | 36.60 | 0.63 | 41.00 | 1.00 |
| max | 17.00 | 199.00 | 122.00 | 99.00 | 846.00 | 67.10 | 2.42 | 81.00 | 1.00 |

```
✓ ⓘ [7] categorical_val = []
continous_val = []
for column in df.columns:
    #   print('*****')
    #   print(f'{column} : {df[column].unique()}')
    if len(df[column].unique()) <= 10:
        categorical_val.append(column)
    else:
        continous_val.append(column)
```

```
✓ ⓘ [8] df.columns
```

```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

```
✓ [9] # How many missing zeros are missing in each feature
feature_columns = [
    'Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
    'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age'
]

for column in feature_columns:
    print("====")
    print(f"{column} ==> Missing zeros : {len(df.loc[df[column] == 0])}")

====  
Pregnancies ==> Missing zeros : 111  
=====  
Glucose ==> Missing zeros : 5  
=====  
BloodPressure ==> Missing zeros : 35  
=====  
SkinThickness ==> Missing zeros : 227  
=====  
Insulin ==> Missing zeros : 374  
=====  
BMI ==> Missing zeros : 11  
=====  
DiabetesPedigreeFunction ==> Missing zeros : 0  
=====  
Age ==> Missing zeros : 0

✓ [10] from sklearn.impute import SimpleImputer

fill_values = SimpleImputer(missing_values=0, strategy="mean", copy=False)
df[feature_columns] = fill_values.fit_transform(df[feature_columns])

for column in feature_columns:
    print("====")
    print(f"{column} ==> Missing zeros : {len(df.loc[df[column] == 0])}")

====  
Pregnancies ==> Missing zeros : 0  
=====  
Glucose ==> Missing zeros : 0  
=====  
BloodPressure ==> Missing zeros : 0  
=====  
SkinThickness ==> Missing zeros : 0  
=====  
Insulin ==> Missing zeros : 0  
=====  
BMI ==> Missing zeros : 0  
=====  
DiabetesPedigreeFunction ==> Missing zeros : 0  
=====  
Age ==> Missing zeros : 0
```

```
+ Code + Text
[11] from sklearn.model_selection import train_test_split

x = df[feature_columns]
y = df.Outcome

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=42)

[12] from sklearn.metrics import confusion_matrix, accuracy_score, classification_report

def evaluate(model, X_train, X_test, y_train, y_test):
    y_test_pred = model.predict(X_test)
    y_train_pred = model.predict(X_train)

    print("TRAINING RESULTS: \n====")
    clf_report = pd.DataFrame(classification_report(y_train, y_train_pred, output_dict=True))
    print(f"CONFUSION MATRIX:\n{confusion_matrix(y_train, y_train_pred)}")
    print(f"ACCURACY SCORE:\n{accuracy_score(y_train, y_train_pred):.4f}")
    print(f"CLASSIFICATION REPORT:\n{clf_report}")

    print("TESTING RESULTS: \n====")
    clf_report = pd.DataFrame(classification_report(y_test, y_test_pred, output_dict=True))
    print(f"CONFUSION MATRIX:\n{confusion_matrix(y_test, y_test_pred)}")
    print(f"ACCURACY SCORE:\n{accuracy_score(y_test, y_test_pred):.4f}")
    print(f"CLASSIFICATION REPORT:\n{clf_report}")

# AdaBoostClassifier
from sklearn.ensemble import AdaBoostClassifier

ada_boost_clf = AdaBoostClassifier(n_estimators=30)
ada_boost_clf.fit(X_train, y_train)
evaluate(ada_boost_clf, X_train, X_test, y_train, y_test)

TRAINING RESULTS:
=====
CONFUSION MATRIX:
[[310  39]
 [ 51 137]]
ACCURACY SCORE:
0.8324
CLASSIFICATION REPORT:
          0      1  accuracy  macro avg  weighted avg
precision  0.86   0.78     0.83     0.82      0.83
recall    0.89   0.73     0.83     0.81      0.83
f1-score   0.87   0.75     0.83     0.81      0.83
support   349.00 188.00    0.83    537.00    537.00
TESTING RESULTS:
=====
CONFUSION MATRIX:
[[123  28]
 [ 27  52]]
ACCURACY SCORE:
0.7619
CLASSIFICATION REPORT:
```

+ Code + Text

```
          0      1  accuracy  macro avg  weighted avg
precision  0.82  0.65      0.76      0.74      0.76
recall    0.81  0.66      0.76      0.74      0.76
f1-score   0.82  0.66      0.76      0.74      0.76
support   151.00 80.00      231.00      231.00

▶ # Gradient Boosting Classifier
from sklearn.ensemble import GradientBoostingClassifier

grad_boost_clf = GradientBoostingClassifier(n_estimators=100, random_state=42)
grad_boost_clf.fit(X_train, y_train)
evaluate(grad_boost_clf, X_train, X_test, y_train, y_test)

◀ TRAINING RESULTS:
=====
CONFUSION MATRIX:
[[342  7]
 [ 19 169]]
ACCURACY SCORE:
0.9516
CLASSIFICATION REPORT:
          0      1  accuracy  macro avg  weighted avg
precision  0.95  0.96      0.95      0.95      0.95
recall    0.98  0.90      0.95      0.94      0.95
f1-score   0.96  0.93      0.95      0.95      0.95
support   349.00 188.00      537.00      537.00
TESTING RESULTS:
=====
CONFUSION MATRIX:
[[116  35]
 [ 26  54]]
ACCURACY SCORE:
0.7359
CLASSIFICATION REPORT:
          0      1  accuracy  macro avg  weighted avg
precision  0.82  0.61      0.74      0.71      0.74
recall    0.77  0.68      0.74      0.72      0.74
f1-score   0.79  0.64      0.74      0.72      0.74
support   151.00 80.00      231.00      231.00
```