**Lab Practical File**

**Of**

**Fundamentals of Data Analytics (AIML301)**

Logo

Description automatically generated

**Submitted By:** Shaurya Guliani **Submitted To:** Mr. Roshan Lal

**Enrollment Number:** A2305219086 **Designation:** Assistant Professor III

**Class:** 6CSE-2X **Department:** CSE

**Department of Computer Science and Engineering**

**Amity School of Engineering and Technology**

**Amity University, Noida, Uttar Pradesh**

**Session 2021-2022**

**PRACTICAL - 1**

**AIM: Introduction to python environment and python installation.**

Python is a widely used interpreted, high level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.

Python is a dynamic, high level, free open source and interpreted programming language. It supports object-oriented programming as well as procedural oriented programming.

Python provides many useful features which make it popular and valuable from the other programming languages. It supports object-oriented programming, procedural programming approaches and provides dynamic memory allocation. We have listed below a few essential features.

**FEATURES OF PYTHON**

**1) Easy to Learn and Use**

Python is easy to learn as compared to other programming languages. Its syntax is straightforward and much the same as the English language. There is no use of the semicolon or curly-bracket, the indentation defines the code block. It is the recommended programming language for beginners.

**2) Expressive Language**

Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type **print("Hello World")**. It will take only one line to execute, while Java or C takes multiple lines.

**3) Interpreted Language**

Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable.

38.3M

632

Hello Java Program for Beginners

**4) Cross-platform Language**

Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, we can say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.

**5) Free and Open Source**

Python is freely available for everyone. It is freely available on its official website [www.python.org](https://www.python.org/). It has a large community across the world that is dedicatedly working towards make new python modules and functions. Anyone can contribute to the Python community. The open-source means, "Anyone can download its source code without paying any penny."

**6) Object-Oriented Language**

Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation, etc. The object-oriented procedure helps to programmer to write reusable code and develop applications in less code.

**7) Extensible**

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our Python code. It converts the program into byte code, and any platform can use that byte code.

**8) Large Standard Library**

It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting. There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids are the popular framework for Python web development.

**9) GUI Programming Support**

Graphical User Interface is used for the developing Desktop application. PyQT5, Tkinter, Kivy are the libraries which are used for developing the web application.

**10) Integrated**

It can be easily integrated with languages like C, C++, and JAVA, etc. Python runs code line by line like C,C++ Java. It makes easy to debug the code.

**11) Embeddable**

The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

**12) Dynamic Memory Allocation**

In Python, we don't need to specify the data-type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time. Suppose we are assigned integer value 15 to **x,** then we don't need to write **int x = 15.** Just write x = 15

**HARDWARE AND SOFTWARE REQUIREMENTS:**

* Modern Operating System:
  + Windows 7 or 10
  + Mac OS X 10.11 or higher, 64-bit
  + Linux: RHEL 6/7, 64-bit (almost all libraries also work in Ubuntu)
* x86 64-bit CPU (Intel / AMD architecture)
* 4 GB RAM
* 5 GB free disk space

**ADVANTAGES OF PYTHON:**

#### **1) Less Coding**

Almost all of the tasks done in Python requires less coding when the same task is done in other languages.

Python also has awesome standard library support, so you don’t have to search for any third-party libraries to get your job done.

This is the reason that many people suggest learning Python to beginners.

#### **2) Affordable**

Python is free therefore individuals, small companies or big organizations can leverage the free available resources to build applications.

Python is popular and widely used so it gives you better community support.

#### **3) Python is for Everyone**

Python code can run on any machine whether it is Linux, Mac or Windows.

Programmers need to learn different languages for different jobs but with Python, you can professionally build web apps, perform data analysis and **machine learning**, automate things, do web scraping and also build games and powerful visualizations.

It is an all-rounder programming language.

**DISADVANTAGES OF PYTHON**:

#### **1)** **Speed Limitations**

We have seen that Python code is executed line by line. But since Python is interpreted, it often results in **slow execution**.

This, however, isn’t a problem unless speed is a focal point for the project.

#### **2)** **Weak in Mobile Computing and Browsers**

While it serves as an excellent server-side language, Python is much rarely seen on the **client-side.**

Besides that, it is rarely ever used to implement smartphone-based applications.

The reason it is not so famous despite the existence of python is that it isn’t that secure.

#### **3)** **Design Restrictions**

As you know, Python is **dynamically-typed**. This means that you don’t need to declare the type of variable while writing the code.

While this is easy on the programmers during coding, it can**raise run-time errors**.

#### **4**) **Underdeveloped Database Access Layers**

Compared to more widely used technologies like **JDBC (Java DataBase Connectivity)** and **ODBC (Open DataBase Connectivity),** Python’s database access layers are a bit underdeveloped.

Consequently, it is less often applied in huge enterprises.

**SETTING UP PYTHON ON WINDOWS 10**

### Step 1: Select Version of Python to Install

The installation procedure involves downloading the official Python .exe installer and running it on your system.

The version you need depends on what you want to do in Python. For example, if you are working on a project coded in Python version 2.6, you probably need that version. If you are starting a project from scratch, you have the freedom to choose.

If you are learning to code in Python, we recommend you download both the latest version of Python 2 and 3. Working with Python 2 enables you to work on older projects or test new projects for backward compatibility.

### Step 2: Download Python Executable Installer

1. Open your web browser and navigate to the [Downloads for Windows section](https://www.python.org/downloads/windows/) of the [official Python website](https://www.python.org/).
2. Search for your desired version of Python. At the time of publishing this article, the latest Python 3 release is version 3.7.3, while the latest Python 2 release is version 2.7.16.
3. Select a link to download either the **Windows x86-64 executable installer** or **Windows x86 executable installer**. The download is approximately 25MB.

### Step 3: Run Executable Installer

1. Run the **Python Installer** once downloaded. (In this example, we have downloaded Python 3.7.3.)
2. Make sure you select the **Install launcher for all users** and **Add Python 3.7 to PATH** checkboxes. The latter places the interpreter in the execution path. For older versions of Python that do not support the **Add Python to Path** checkbox, see [Step 6](https://phoenixnap.com/kb/how-to-install-python-3-windows#htoc-step-5-add-python-path-to-environment-variables-optional).
3. Select **Install Now** – the recommended installation options.For all recent versions of Python, the recommended installation options include **Pip** and **IDLE**. Older versions might not include such additional features.
4. The next dialog will prompt you to select whether to **Disable path length limit**. Choosing this option will allow Python to bypass the 260-character MAX\_PATH limit. Effectively, it will enable Python to use long path names.

The **Disable path length** **limit** option will not affect any other system settings. Turning it on will resolve potential name length issues that may arise with Python projects developed in Linux

### Step 4: Verify Python Was Installed on Windows

1. Navigate to the directory in which Python was installed on the system.
2. Double-click **python.exe**.

**Step 5: Verify Pip Was Installed**

If you opted to install an older version of Python, it is possible that it did not come with Pip preinstalled. Pip is a powerful package management system for Python software packages. Thus, make sure that you have it installed.

We recommend using Pip for most Python packages, especially when working in virtual environments.

To verify whether Pip was installed:

1. Open the Start menu and type "cmd."
2. Select the Command Prompt application.
3. Enter pip -V in the console.

**PRACTICAL - 2**

**AIM: To implement Panda Library Functions**

**TOOL USED: VS Code**

**DATASET USED:** Used a sample Loan Default Prediction dataset from Kaggle which contains 500 rows and 5 columns with headers like Index, employment, bank balance, annual sales, defaulted.

**THEORY:**

Pandas is an open-source library that is made mainly for working with relational or labeled data both easily and intuitively. It provides various data structures and operations for manipulating numerical data and time series. This library is built on top of the NumPy library. Pandas is fast and it has high performance & productivity for users.

The following are few important Panda Functions:

1. **Read():**

* read\_csv() function helps read a comma-separated values (csv) file into a Pandas DataFrame. All you need to do is mention the path of the file you want it to read. It can also read files separated by delimiters other than comma, like | or tab.
* The read\_excel() function returns a new DataFrame with the data and labels from the file data.excel, which is specified with the first argument. This string can be any valid path, including URLs.

1. **Groupby():**

* Pandas groupby is used for grouping the data according to the categories and apply a function to the categories. It also helps to aggregate data efficiently. Pandas dataframe. groupby() function is used to split the data into groups based on some criteria.

1. **Describe():**

* The Describe function returns the statistical summary of the dataframe or series. This includes count, mean, median (or 50th percentile) standard variation, min-max, and percentile values of columns. To perform this function, chain . describe() to the dataframe or series.

1. **Print():**

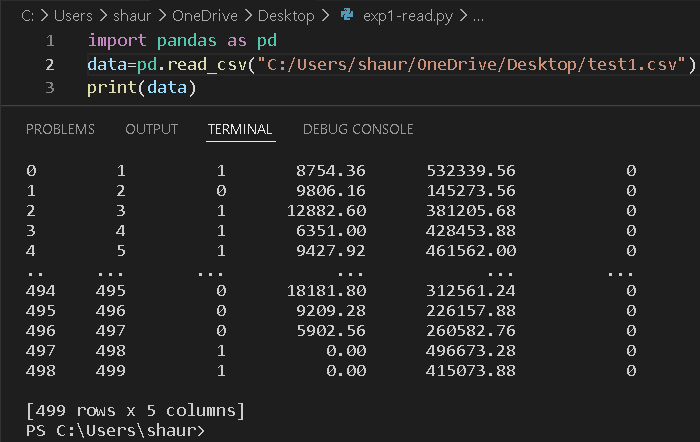
* The print() function prints the specified message to the screen, or other standard output device.

1. **DataFrame():**

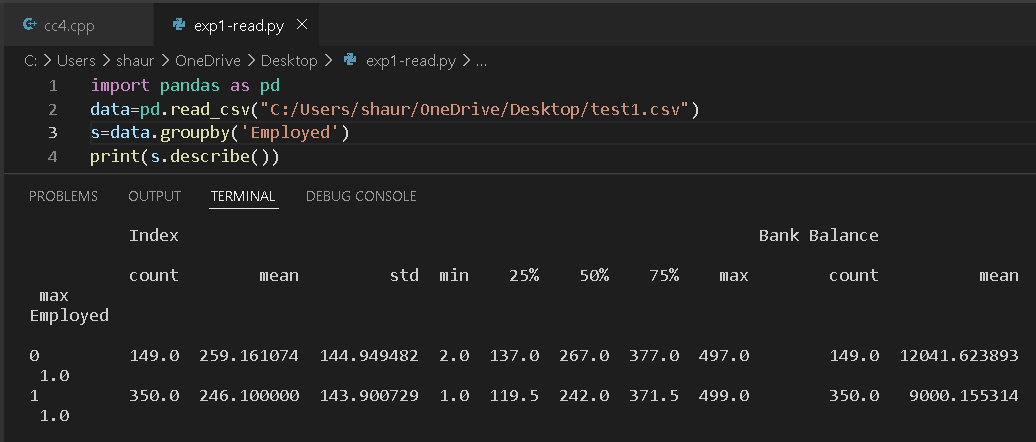
* a Pandas DataFrame will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, and Excel file. Pandas DataFrame can be created from the lists, dictionary, and from a list of dictionary etc.

**IMPLEMENTATION:**

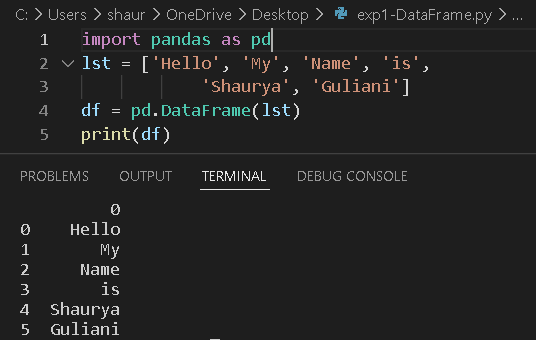
1. **Read\_csv() and Print():**



1. **Groupby() and describe()**



1. **DataFrame():**



**PRACTICAL - 3**

**AIM: To implement Simple Linear Regression**

**TOOL USED: VS Code**

**THEORY:**

Simple linear regression is an approach for predicting a response using a single feature.

It is assumed that the two variables are linearly related. Hence, we try to find a linear function that predicts the response value(y) as accurately as possible as a function of the feature or independent variable(x).

Let us consider a dataset where we have a value of response y for every feature x:  
For generality, we define:

x as feature vector, i.e x = [x\_1, x\_2, …., x\_n],

y as response vector, i.e y = [y\_1, y\_2, …., y\_n]

Simple linear regression is a [parametric test](https://www.scribbr.com/statistics/statistical-tests/#parametric), meaning that it makes certain assumptions about the data. These assumptions are:

1. Homogeneity of variance (homoscedasticity): the size of the error in our prediction doesn’t change significantly across the values of the independent variable.
2. Independence of observations: the observations in the dataset were collected using [statistically valid sampling methods](https://www.scribbr.com/methodology/sampling-methods/), and there are no hidden relationships among observations.
3. Normality: The data follows a [normal distribution](https://www.scribbr.com/statistics/normal-distribution/).

Linear regression makes one additional assumption:

1. The relationship between the independent and dependent variable is linear: the line of best fit through the data points is a straight line (rather than a curve or some sort of grouping factor).

**IMPLEMENTATION:**

import numpy as np

import matplotlib.pyplot as plt

def estimate\_coef(x, y):

    n = np.size(x)

    m\_x = np.mean(x)

    m\_y = np.mean(y)

    SS\_xy = np.sum(y\*x) - n\*m\_y\*m\_x

    SS\_xx = np.sum(x\*x) - n\*m\_x\*m\_x

    b\_1 = SS\_xy / SS\_xx

    b\_0 = m\_y - b\_1\*m\_x

    return (b\_0, b\_1)

def plot\_regression\_line(x, y, b):

    plt.scatter(x, y, color = "m",

               marker = "o", s = 30)

    y\_pred = b[0] + b[1]\*x

    plt.plot(x, y\_pred, color = "g")

    plt.xlabel('x')

    plt.ylabel('y')

    plt.show()

def main():

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

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

    b = estimate\_coef(x, y)

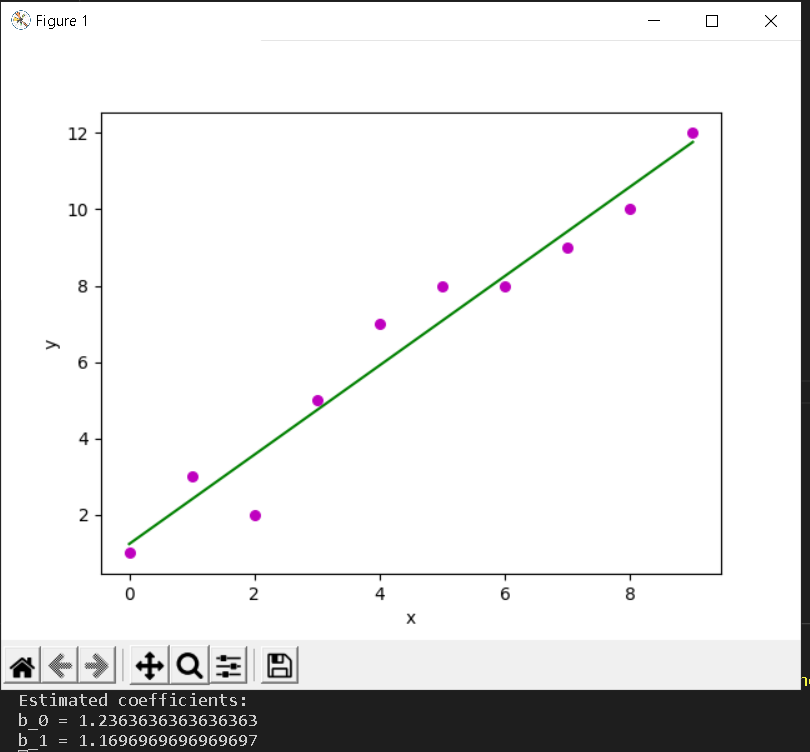
    print("Estimated coefficients:\nb\_0 = {}  \

          \nb\_1 = {}".format(b[0], b[1]))

    plot\_regression\_line(x, y, b)

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

    main()



**PRACTICAL - 4**

**AIM: To implement Logistic Regression**

**TOOL USED: VS Code**

**DATASET USED:** Used a sample User.csv dataset from Kaggle for Logistic Regression.

**THEORY:**

**It** is a Machine Learning classification algorithm that is used to predict the probability of a categorical dependent variable. In logistic regression, the dependent variable is a binary variable that contains data coded as 1 (yes, success, etc.) or 0 (no, failure, etc.). In other words, the logistic regression model predicts P(Y=1) as a function of X.

# Logistic Regression Assumptions

* Binary logistic regression requires the dependent variable to be binary.
* For a binary regression, the factor level 1 of the dependent variable should represent the desired outcome.
* Only the meaningful variables should be included.
* The independent variables should be independent of each other. That is, the model should have little or no multicollinearity.
* The independent variables are linearly related to the log odds.
* Logistic regression requires quite large sample sizes.

**IMPLEMENTATION:**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

print("Shaurya Guliani")

print("A2305219086")

dataset = pd.read\_csv('User\_Data.csv')

# input

x = dataset.iloc[:, [2, 3]].values

# output

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

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

xtrain, xtest, ytrain, ytest = train\_test\_split(

        x, y, test\_size = 0.25, random\_state = 0)

from sklearn.preprocessing import StandardScaler

sc\_x = StandardScaler()

xtrain = sc\_x.fit\_transform(xtrain)

xtest = sc\_x.transform(xtest)

print (xtrain[0:10, :])

from sklearn.linear\_model import LogisticRegression

classifier = LogisticRegression(random\_state = 0)

classifier.fit(xtrain, ytrain)

y\_pred = classifier.predict(xtest)

from sklearn.metrics import confusion\_matrix

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

print ("Confusion Matrix : \n", cm)

from sklearn.metrics import accuracy\_score

print ("Accuracy : ", accuracy\_score(ytest, y\_pred))

from matplotlib.colors import ListedColormap

X\_set, y\_set = xtest, ytest

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

                               stop = X\_set[:, 0].max() + 1, step = 0.01),

                     np.arange(start = X\_set[:, 1].min() - 1,

                               stop = X\_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(

             np.array([X1.ravel(), X2.ravel()]).T).reshape(

             X1.shape), alpha = 0.75, cmap = ListedColormap(('red', 'green')))

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

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

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

    plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],

                c = ListedColormap(('red', 'green'))(i), label = j)

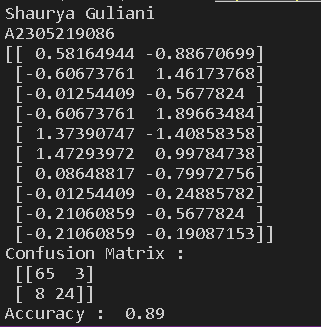
plt.title('Classifier (Test set)')

plt.xlabel('Age')

plt.ylabel('Estimated Salary')

plt.legend()

plt.show()





**PRACTICAL - 5**

**AIM: To implement Gradient Descent**

**TOOL USED: VS Code**

**THEORY:**

Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost).

Gradient descent is best used when the parameters cannot be calculated analytically (e.g. using linear algebra) and must be searched for by an optimization algorithm.

The procedure starts off with initial values for the coefficient or coefficients for the function. These could be 0.0 or a small random value.

coefficient = 0.0

The cost of the coefficients is evaluated by plugging them into the function and calculating the cost.

cost = f(coefficient)

or

cost = evaluate(f(coefficient))

The derivative of the cost is calculated. The derivative is a concept from calculus and refers to the slope of the function at a given point. We need to know the slope so that we know the direction (sign) to move the coefficient values in order to get a lower cost on the next iteration.

delta = derivative(cost)

Now that we know from the derivative which direction is downhill, we can now update the coefficient values. A [learning rate parameter](https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/) (alpha) must be specified that controls how much the coefficients can change on each update.

coefficient = coefficient – (alpha \* delta)

This process is repeated until the cost of the coefficients (cost) is 0.0 or close enough to zero to be good enough.

You can see how simple gradient descent is. It does require you to know the gradient of your cost function or the function you are optimizing, but besides that, it’s very straightforward. Next we will see how we can use this in machine learning algorithms.

**IMPLEMENTATION**

# Importing Libraries

import numpy as np

import matplotlib.pyplot as plt

def mean\_squared\_error(y\_true, y\_predicted):

    # Calculating the loss or cost

    cost = np.sum((y\_true-y\_predicted)\*\*2) / len(y\_true)

    return cost

# Gradient Descent Function

# Here iterations, learning\_rate, stopping\_threshold

# are hyperparameters that can be tuned

def gradient\_descent(x, y, iterations = 1000, learning\_rate = 0.0001,

                     stopping\_threshold = 1e-6):

    # Initializing weight, bias, learning rate and iterations

    current\_weight = 0.1

    current\_bias = 0.01

    iterations = iterations

    learning\_rate = learning\_rate

    n = float(len(x))

    costs = []

    weights = []

    previous\_cost = None

    # Estimation of optimal parameters

    for i in range(iterations):

        # Making predictions

        y\_predicted = (current\_weight \* x) + current\_bias

        # Calculationg the current cost

        current\_cost = mean\_squared\_error(y, y\_predicted)

        # If the change in cost is less than or equal to

        # stopping\_threshold we stop the gradient descent

        if previous\_cost and abs(previous\_cost-current\_cost)<=stopping\_threshold:

            break

        previous\_cost = current\_cost

        costs.append(current\_cost)

        weights.append(current\_weight)

        # Calculating the gradients

        weight\_derivative = -(2/n) \* sum(x \* (y-y\_predicted))

        bias\_derivative = -(2/n) \* sum(y-y\_predicted)

        # Updating weights and bias

        current\_weight = current\_weight - (learning\_rate \* weight\_derivative)

        current\_bias = current\_bias - (learning\_rate \* bias\_derivative)

        # Printing the parameters for each 1000th iteration

        print(f"Iteration {i+1}: Cost {current\_cost}, Weight \

        {current\_weight}, Bias {current\_bias}")

    # Visualizing the weights and cost at for all iterations

    plt.figure(figsize = (8,6))

    plt.plot(weights, costs)

    plt.scatter(weights, costs, marker='o', color='red')

    plt.title("Cost vs Weights")

    plt.ylabel("Cost")

    plt.xlabel("Weight")

    plt.show()

    return current\_weight, current\_bias

def main():

    print("Shaurya Guliani")

    print("A2305219086")

    # Data

    X = np.array([32.50234527, 53.42680403, 61.53035803, 47.47563963, 59.81320787,

           55.14218841, 52.21179669, 39.29956669, 48.10504169, 52.55001444,

           45.41973014, 54.35163488, 44.1640495 , 58.16847072, 56.72720806,

           48.95588857, 44.68719623, 60.29732685, 45.61864377, 38.81681754])

    Y = np.array([31.70700585, 68.77759598, 62.5623823 , 71.54663223, 87.23092513,

           78.21151827, 79.64197305, 59.17148932, 75.3312423 , 71.30087989,

           55.16567715, 82.47884676, 62.00892325, 75.39287043, 81.43619216,

           60.72360244, 82.89250373, 97.37989686, 48.84715332, 56.87721319])

    # Estimating weight and bias using gradient descent

    estimated\_weight, eatimated\_bias = gradient\_descent(X, Y, iterations=2000)

    print(f"Estimated Weight: {estimated\_weight}\nEstimated Bias: {eatimated\_bias}")

    # Making predictions using estimated parameters

    Y\_pred = estimated\_weight\*X + eatimated\_bias

    # Plotting the regression line

    plt.figure(figsize = (8,6))

    plt.scatter(X, Y, marker='o', color='red')

    plt.plot([min(X), max(X)], [min(Y\_pred), max(Y\_pred)], color='blue',markerfacecolor='red',

             markersize=10,linestyle='dashed')

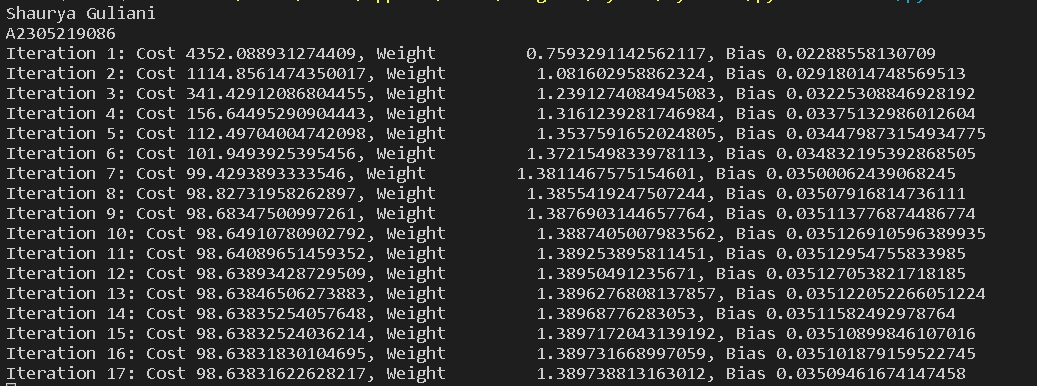
    plt.xlabel("X")

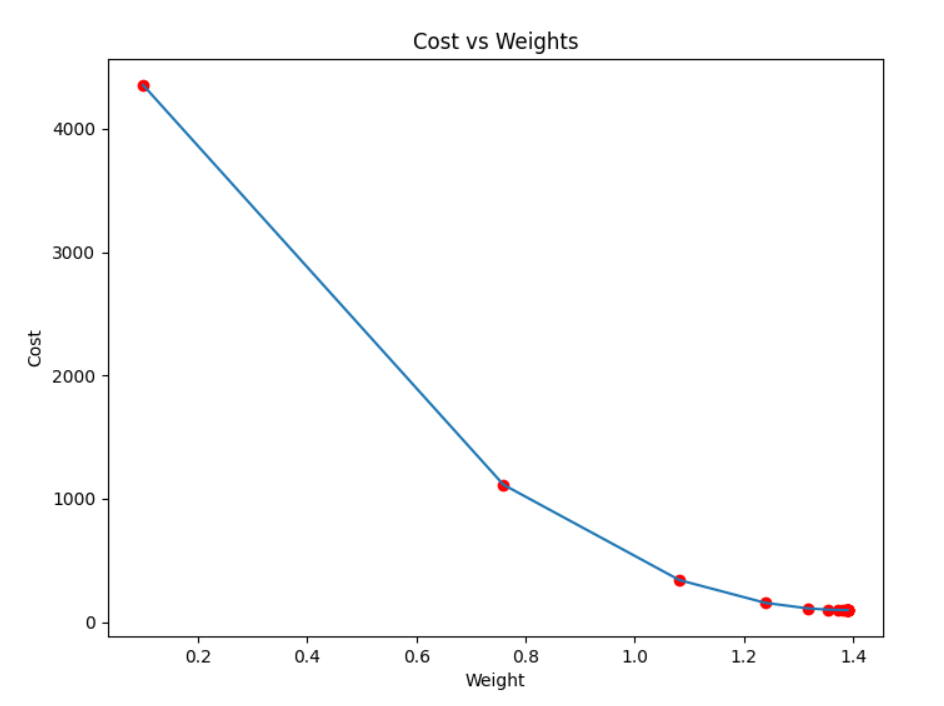
    plt.ylabel("Y")

    plt.show()

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

    main()





**PRACTICAL - 6**

**AIM: To Handle Missing Values in a dataset**

**TOOL USED: VS Code**

**DATASET USED:** Used a sample Employee.csv dataset from Kaggle consisting of Null values in the ‘Gender’ column.

**THEORY:**

The missing values in a dataset can be handled in the following ways:

### 1. Deleting Rows

This method commonly used to handle the null values. Here, we either delete a particular row if it has a null value for a particular feature and a particular column if it has more than 70-75% of missing values. This method is advised only when there are enough samples in the data set. One has to make sure that after we have deleted the data, there is no addition of bias. Removing the data will lead to loss of information which will not give the expected results while predicting the output.

### Pros:

* Complete removal of data with missing values results in robust and highly accurate model
* Deleting a particular row or a column with no specific information is better, since it does not have a high weightage

### Cons:

* Loss of information and data
* Works poorly if the percentage of missing values is high (say 30%), compared to the whole dataset

## **2. Replacing With Mean/Median/Mode**

This strategy can be applied on a feature which has numeric data like the age of a person or the ticket fare. We can calculate the mean, median or mode of the feature and replace it with the missing values. This is an approximation which can add variance to the data set. But the loss of the data can be negated by this method which yields better results compared to removal of rows and columns. Replacing with the above three approximations are a statistical approach of handling the missing values. This method is also called as leaking the data while training. Another way is to approximate it with the deviation of neighbouring values. This works better if the data is linear.

To replace it with median and mode we can use the following to calculate the same:

### Pros:

* This is a better approach when the data size is small
* It can prevent data loss which results in removal of the rows and columns

### Cons:

* Imputing the approximations add variance and bias
* Works poorly compared to other multiple-imputations method

## **3. Assigning An Unique Category**

A categorical feature will have a definite number of possibilities, such as gender, for example. Since they have a definite number of classes, we can assign another class for the missing values. Here, the features Cabin and Embarked have missing values which can be replaced with a new category, say, U for ‘unknown’. This strategy will add more information into the dataset which will result in the change of variance. Since they are categorical, we need to find one hot encoding to convert it to a numeric form for the algorithm to understand it. Let us look at how it can be done in Python:

### Pros:

* Less possibilities with one extra category, resulting in low variance after one hot encoding — since it is categorical
* Negates the loss of data by adding an unique category

### Cons:

* Adds less variance
* Adds another feature to the model while encoding, which may result in poor performance

## **4. Predicting The Missing Values**

Using the features which do not have missing values, we can predict the nulls with the help of a machine learning algorithm. This method may result in better accuracy, unless a missing value is expected to have a very high variance. We will be using linear [regression](https://analyticsindiamag.com/top-6-regression-algorithms-used-data-mining-applications-industry/) to replace the nulls in the feature ‘age’, using other available features. One can experiment with different algorithms and check which gives the best accuracy instead of sticking to a single algorithm.

### Pros:

* Imputing the missing variable is an improvement as long as the bias from the same is smaller than the omitted variable bias
* Yields unbiased estimates of the model parameters

### Cons:

* Bias also arises when an incomplete conditioning set is used for a categorical variable
* Considered only as a proxy for the true values

## **5. Using Algorithms Which Support Missing Values**

KNN is a machine learning algorithm which works on the principle of distance measure. This algorithm can be used when there are nulls present in the dataset. While the algorithm is applied, KNN considers the missing values by taking the majority of the K nearest values. In this particular dataset, taking into account the person’s age, sex, class etc, we will assume that people having same data for the above mentioned features will have the same kind of fare.

Unfortunately, the SciKit Learn library for the K – Nearest Neighbour algorithm in Python does not support the presence of the missing values.

Another algorithm which can be used here is RandomForest. This model produces a robust result because it works well on non-linear and the categorical data. It adapts to the data structure taking into consideration of the high variance or the bias, producing better results on large datasets.

### Pros:

* Does not require creation of a predictive model for each attribute with missing data in the dataset
* Correlation of the data is neglected

### Cons:

* Is a very time consuming process and it can be critical in data mining where large databases are being extracted
* Choice of distance functions can be Euclidean, Manhattan etc. which is do not yield a robust result

**IMPLEMENTATION:**

**Method Used:** Assigning a unique category

# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read\_csv("employees.csv")

# creating bool series True for NaN values

bool\_series = pd.isnull(data["Gender"])

# filtering data

# displaying data only with Gender = NaN

data[bool\_series]

# importing pandas package

import pandas as pd

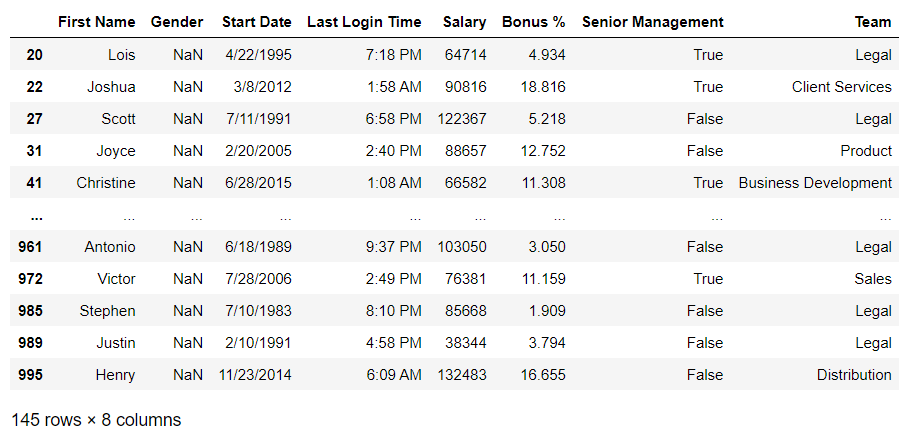
# making data frame from csv file

data = pd.read\_csv("employees.csv")

# filling a null values using fillna()

data["Gender"].fillna("No Gender", inplace = True)

data





**PRACTICAL - 7**

**AIM: To implement Underfitting and Overfitting**

**TOOL USED: VS Code**

**THEORY:**

**Underfitting:**   
A statistical model or a machine learning algorithm is said to have underfitting when it cannot capture the underlying trend of the data. Underfitting destroys the accuracy of our machine learning model. Its occurrence simply means that our model or the algorithm does not fit the data well enough. It usually happens when we have fewer data to build an accurate model and also when we try to build a linear model with fewer non-linear data. In such cases, the rules of the machine learning model are too easy and flexible to be applied on such minimal data and therefore the model will probably make a lot of wrong predictions. Underfitting can be avoided by using more data and also reducing the features by feature selection

Techniques to reduce underfitting:

1. Increase model complexity
2. Increase the number of features, performing feature engineering
3. Remove noise from the data.
4. Increase the number of epochs or increase the duration of training to get better results.

**Overfitting:**   
A statistical model is said to be overfitted when we train it with a lot of data. When a model gets trained with so much data, it starts learning from the noise and inaccurate data entries in our data set. Then the model does not categorize the data correctly, because of too many details and noise. The causes of overfitting are the non-parametric and non-linear methods because these types of machine learning algorithms have more freedom in building the model based on the dataset and therefore, they can really build unrealistic models. A solution to avoid overfitting is using a linear algorithm if we have linear data or using the parameters like the maximal depth if we are using decision trees.

Techniques to reduce overfitting:

1. Increase training data.
2. Reduce model complexity.
3. Early stopping during the training phase (have an eye over the loss over the training period as soon as loss begins to increase stop training).
4. Ridge Regularization and Lasso Regularization
5. Use dropout for neural networks to tackle overfitting.

**IMPLEMENTATION**

print(\_\_doc\_\_)

import numpy as np

import matplotlib.pyplot as plt

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import PolynomialFeatures

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import cross\_val\_score

def true\_fun(X):

    return np.cos(1.5 \* np.pi \* X)

np.random.seed(0)

n\_samples = 30

degrees = [1, 4, 15]

X = np.sort(np.random.rand(n\_samples))

y = true\_fun(X) + np.random.randn(n\_samples) \* 0.1

plt.figure(figsize=(14, 5))

for i in range(len(degrees)):

    ax = plt.subplot(1, len(degrees), i + 1)

    plt.setp(ax, xticks=(), yticks=())

    polynomial\_features = PolynomialFeatures(degree=degrees[i],

                                             include\_bias=False)

    linear\_regression = LinearRegression()

    pipeline = Pipeline([("polynomial\_features", polynomial\_features),

                         ("linear\_regression", linear\_regression)])

    pipeline.fit(X[:, np.newaxis], y)

    # Evaluate the models using crossvalidation

    scores = cross\_val\_score(pipeline, X[:, np.newaxis], y,

                             scoring="neg\_mean\_squared\_error", cv=10)

    X\_test = np.linspace(0, 1, 100)

    plt.plot(X\_test, pipeline.predict(X\_test[:, np.newaxis]), label="Model")

    plt.plot(X\_test, true\_fun(X\_test), label="True function")

    plt.scatter(X, y, edgecolor='b', s=20, label="Samples")

    plt.xlabel("x")

    plt.ylabel("y")

    plt.xlim((0, 1))

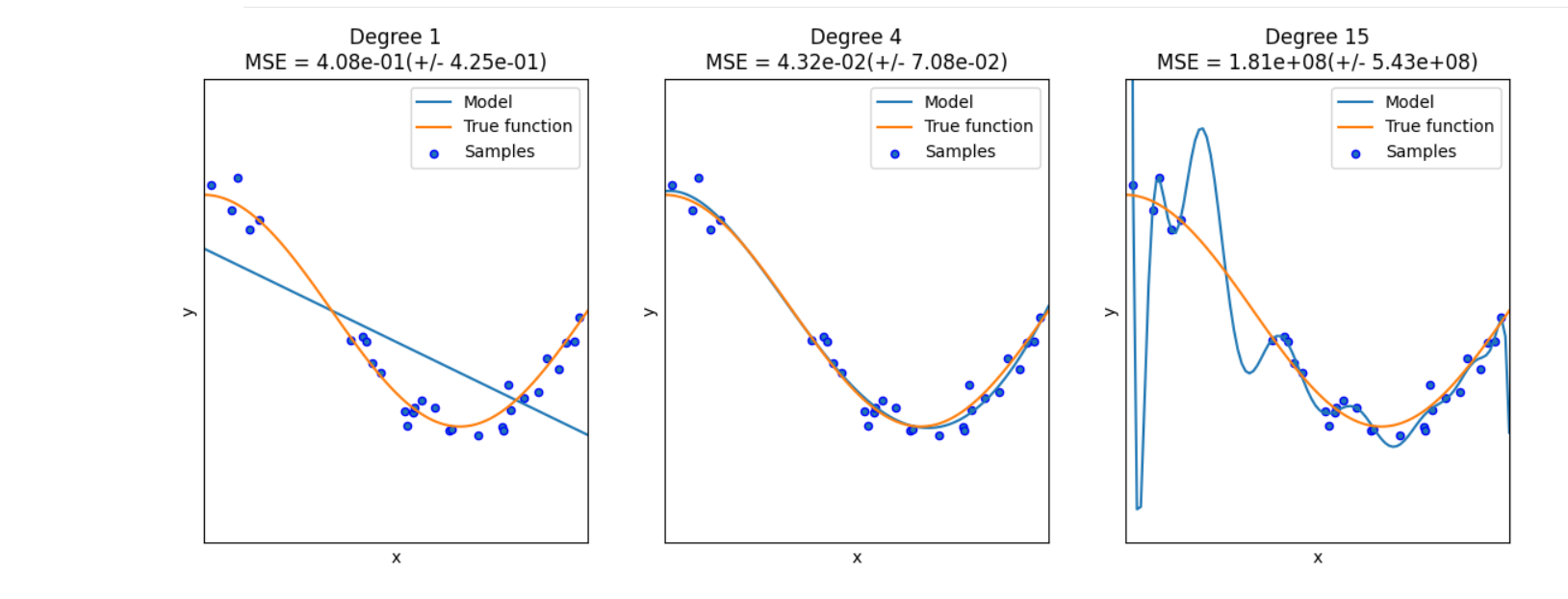
    plt.ylim((-2, 2))

    plt.legend(loc="best")

    plt.title("Degree {}\nMSE = {:.2e}(+/- {:.2e})".format(

        degrees[i], -scores.mean(), scores.std()))

plt.show()



**PRACTICAL - 8**

**AIM: To implement SVM Hyperplane algorithm for classification for the given dataset like to classify cancerous or non-cancerous cells in PYTHON using all necessary libraries like NUMPY and PANDAS.**

**TOOL USED: VS Code**

**THEORY:**

SVM is a supervised learning algorithm, that can be used for both classification as well as regression problems. However, mostly it is used for classification problems. It is a highly efficient and preferred algorithm due to significant accuracy with less computation power

In the SVM algorithm, we plot each observation as a point in an n-dimensional space (where n is the number of features in the dataset). Our task is to find an optimal hyperplane that successfully classifies the data points into their respective classes.

A hyperplane is a decision boundary that differentiates the two classes in SVM. A data point falling on either side of the hyperplane can be attributed to different classes. The dimension of the hyperplane depends on the number of input features in the dataset. If we have 2 input features the hyper-plane will be a line. likewise, if the number of features is 3, it will become a two-dimensional plane.

Support vectors are the data points that are nearest to the hyper-plane and affect the position and orientation of the hyper-plane. We have to select a hyperplane, for which the margin, i.e the distance between support vectors and hyper-plane is maximum. Even a little interference in the position of these support vectors can change the hyper-plane.

For example, we have a classification problem where we have to separate the red data points from the blue ones. Since it is a two-dimensional problem, our decision boundary will be a line, for the 3-dimensional problem we have to use a plane, and similarly, the complexity of the solution will increase with the rising number of features. we have multiple lines separating the data points successfully. But our objective is to look for the best solution. There are few rules that can help us to identify the best line:

1. **Maximum classification**, i.e. the selected line must be able to successfully segregate all the data points into the respective classes.
2. Second rule is**Best Separation**, which means, we must choose a line such that it is perfectly able to separate the points.

**IMPLEMENTATION**

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

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

from sklearn.svm import SVC

from sklearn.metrics import classification\_report

from sklearn.metrics import confusion\_matrix

import os

df = pd.read\_csv("C:/Users/shaur/OneDrive/Desktop/AIML301-DATA-ANALYTICS/Lab/programs/data.csv")

df.shape

df.dtypes

df.head()

df.tail()

X = df.iloc[:,2:32]

print(X.shape)

X.head()

y = df.diagnosis

print(y.shape)

y.head()

y\_num = pd.get\_dummies(y)

y\_num.tail()

y\_num = pd.get\_dummies(y)

y\_num.tail()

y = y\_num.M

print(y.shape)

y.tail()

X.corr()

plt.figure(figsize=(18, 12))

sns.heatmap(X.corr(), vmin=0.85, vmax=1, annot=True, cmap='YlGnBu', linewidths=.5)

scaler = StandardScaler()

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

X\_scaled = pd.DataFrame(X\_scaled)

X\_scaled\_drop = X\_scaled.drop(X\_scaled.columns[[2, 3, 12, 13, 22, 23]], axis=1)

pca = PCA(n\_components=0.95)

x\_pca = pca.fit\_transform(X\_scaled\_drop)

x\_pca = pd.DataFrame(x\_pca)

print("Before PCA, X dataframe shape = ",X.shape,"\nAfter PCA, x\_pca dataframe shape = ",x\_pca.shape)

print(pca.explained\_variance\_ratio\_)

print(pca.explained\_variance\_ratio\_.sum())

colnames = ['PC1','PC2','PC3','PC4','PC5','PC6','PC7','PC8','PC9','PC10','PC11','diagnosis']

diag = df.iloc[:,1:2]

Xy = pd.DataFrame(np.hstack([x\_pca,diag.values]),columns=colnames)

Xy.head()

sns.lmplot("PC1", "PC2", hue="diagnosis", data=Xy, fit\_reg=False, markers=["o", "x"])

plt.show()

X=(Xy.iloc[:,0:11]).values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=0)

print("X\_train shape ",X\_train.shape)

print("y\_train shape ",y\_train.shape)

print("X\_test shape ",X\_test.shape)

print("y\_test shape ",y\_test.shape)

svc = SVC()

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

y\_pred\_svc =svc.predict(X\_test)

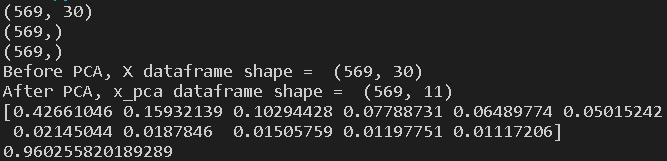
y\_pred\_svc.shape

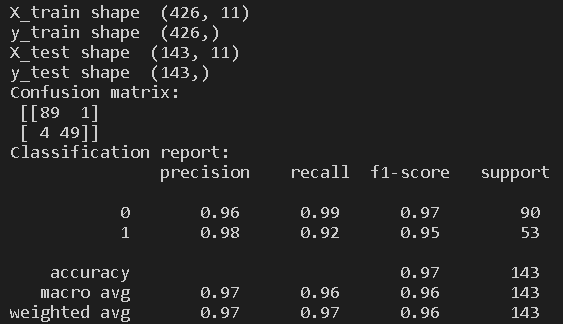
cm = confusion\_matrix(y\_test, y\_pred\_svc)

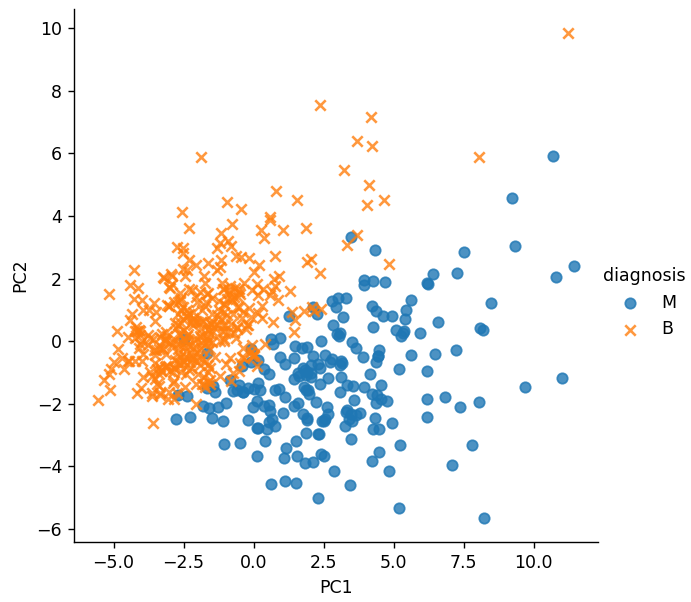
print("Confusion matrix:\n",cm)

creport = classification\_report(y\_test, y\_pred\_svc)

print("Classification report:\n",creport)







**Open Ended Experiment**

**AIM:** Implement the random forest model for regression and classification using the given dataset

**TOOL USED: VS Code**

**THEORY:**

Random forest is a [supervised learning algorithm](https://builtin.com/data-science/supervised-learning-python). The "forest" it builds, is an ensemble of decision trees, usually trained with the “bagging” method. The general idea of the [bagging method](https://builtin.com/data-science/tour-top-10-algorithms-machine-learning-newbies) is that a combination of learning models increases the overall result.

One big advantage of random forest is that it can be used for both classification and regression problems, which form the majority of current machine learning systems. Random forest has nearly the same hyperparameters as a decision tree or a bagging classifier. Fortunately, there's no need to combine a decision tree with a bagging classifier because you can easily use the classifier-class of random forest. With random forest, you can also deal with regression tasks by using the algorithm's regressor

Random forest adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

**IMPLEMENTATION**

import numpy as nm

import matplotlib.pyplot as mtp

import pandas as pd

#importing datasets

data\_set= pd.read\_csv('/content/open ended.csv')

#Extracting Independent and dependent Variable

x= data\_set.iloc[:, [2,3]].values

y= data\_set.iloc[:, 4].values

# Splitting the dataset into training and test set.

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

x\_train, x\_test, y\_train, y\_test= train\_test\_split(x, y, test\_size= 0.25, random\_state=0)

#feature Scaling

from sklearn.preprocessing import StandardScaler

st\_x= StandardScaler()

x\_train= st\_x.fit\_transform(x\_train)

#Fitting Decision Tree classifier to the training set

from sklearn.ensemble import RandomForestClassifier

classifier= RandomForestClassifier(n\_estimators= 10, criterion="entropy")

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

#Predicting the test set result

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

#Creating the Confusion matrix

from sklearn.metrics import confusion\_matrix

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

from matplotlib.colors import ListedColormap

x\_set, y\_set = x\_train, y\_train

x1, x2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step  =0.01),

nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))

mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),

alpha = 0.75, cmap = ListedColormap(('purple','green' )))

mtp.xlim(x1.min(), x1.max())

mtp.ylim(x2.min(), x2.max())

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

  mtp.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1], c = ListedColormap(('purple', 'green'))(i), label = j)

  mtp.title('Random Forest Algorithm (Training set)')

  mtp.xlabel('Age')

  mtp.ylabel('Estimated Salary')

  mtp.legend()

  mtp.show()

#Visulaizing the test set result

from matplotlib.colors import ListedColormap

x\_set, y\_set = x\_test, y\_test

x1, x2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step  =0.01),

nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))

mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),

alpha = 0.75, cmap = ListedColormap(('purple','green' )))

mtp.xlim(x1.min(), x1.max())

mtp.ylim(x2.min(), x2.max())

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

  mtp.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1], c = ListedColormap(('purple', 'green'))(i), label = j)

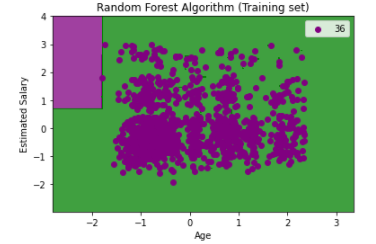
  mtp.title('Random Forest Algorithm(Test set)')

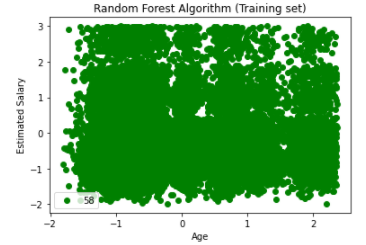
  mtp.xlabel('Age')

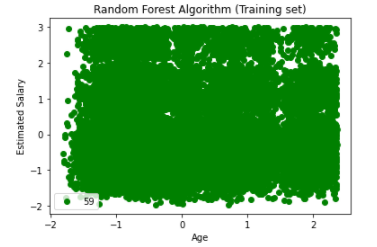
  mtp.ylabel('Estimated Salary')

  mtp.legend()

  mtp.show()

****

****

****

