Delhi Technological University

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Bawana Road, Delhi-110042

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Artificial Neural Networks

Subject Code: AI504

Lab File

Submitted To:

Anil Singh Parihar Sir

Department of Computer Science

And Engineering

Submitted By:

SHIKHAR ASTHANA

M.Tech AFI

2K22/AFI/24

INDEX

S. No.	Objective	Date	Sign
1	To study about NumPy, Pandas, and Matplotlib libraries in python.		
2	To perform data preprocessing and data visualization on iris dataset.		
3	To implement uni-variate linear regression using gradient descent.		
4	To implement multi-variate linear regression using gradient descent.		
5	To implement logistic regression using gradient descent.		
6	To implement basic perceptron model.		
7	To implement multi-layer neural network.		
8	To perform classification on MNIST Digit dataset using Neural Network.		
9	To perform classification on MNIST Fashion dataset using Neural Network.		

EXPERIMENT 1

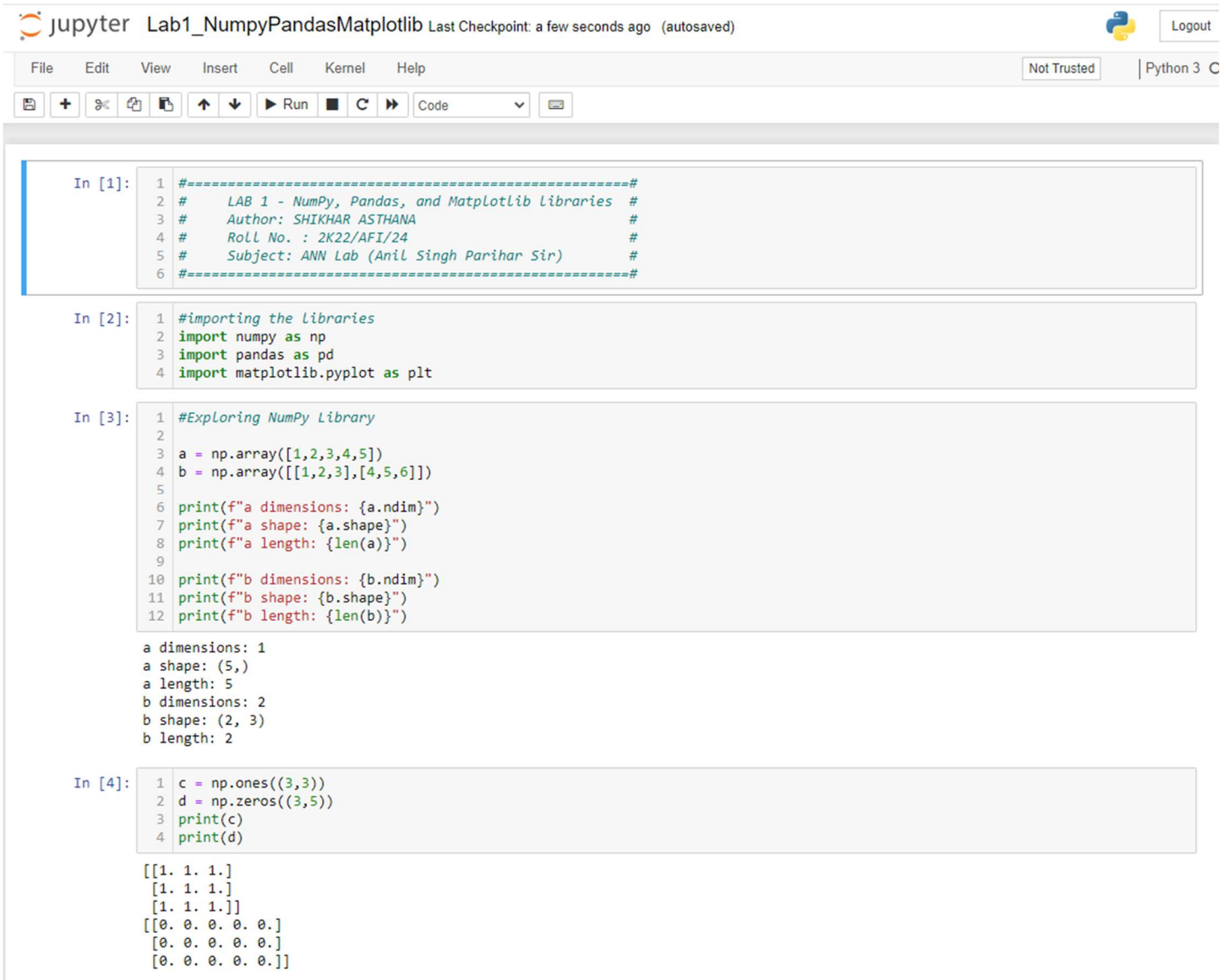
AIM: To study about NumPy, Pandas, and Matplotlib libraries in python.

THEORY: NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. NumPy is widely used for scientific computing and data analysis tasks.

Pandas is a Python library for data manipulation and analysis. It provides data structures for efficiently storing and manipulating large datasets and tools for working with structured data, such as SQL tables or Excel spreadsheets.

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK.

CODE & OUTPUT:



In [5]:
```
e = np.eye(3)
print(e)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

In [6]:
```
#operations in numpy
f = np.dot(c,d)
print(f)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

In [7]:
```
print(np.logical_or(np.array([1,0,0,0]),np.array([1,1,0,0])))
print(np.logical_and(np.array([1,0,0,0]),np.array([1,1,0,0])))
```

```
[ True  True False False]
[ True False False False]
```

In [8]:
```
#Exploring Pandas library

#imporitng the iris dataset CSV file using pandas
iris_df = pd.read_csv("iris.csv")
```

In [9]:
```
#Exploratory commands of dataframe in pandas

iris_df.head()
```

Out[9]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

In [10]:
```
iris_df.tail()
```

Out[10]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
145	146	6.7	3.0	5.2	2.3	Iris-virginica
146	147	6.3	2.5	5.0	1.9	Iris-virginica
147	148	6.5	3.0	5.2	2.0	Iris-virginica
148	149	6.2	3.4	5.4	2.3	Iris-virginica
149	150	5.9	3.0	5.1	1.8	Iris-virginica

In [11]:
```
iris_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             150 non-null    int64
 1   SepalLengthCm  150 non-null    float64
 2   SepalWidthCm   150 non-null    float64
 3   PetalLengthCm  150 non-null    float64
 4   PetalWidthCm   150 non-null    float64
 5   Species        150 non-null    object
dtypes: float64(4), int64(1), object(1)
memory usage: 7.2+ KB
```

In [12]:
```
iris_df.columns
```

Out[12]:
```
Index(['Id', 'SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm',
       'Species'],
      dtype='object')
```

In [13]:
```
iris_df["Species"].unique()
```

Out[13]:
```
array(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'], dtype=object)
```

In [14]:
```
#Filtering commands of dataframes in pandas

iris_df[iris_df["Species"]=="Iris-setosa"].head()
```

Out[14]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

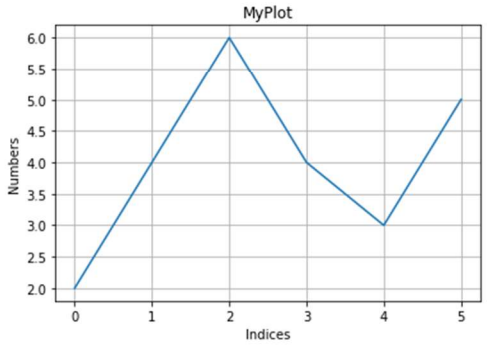
In [15]:
```
iris_df[(iris_df["Species"]=="Iris-setosa") | (iris_df["Species"]=="Iris-versicolor")]
```

Out[15]:

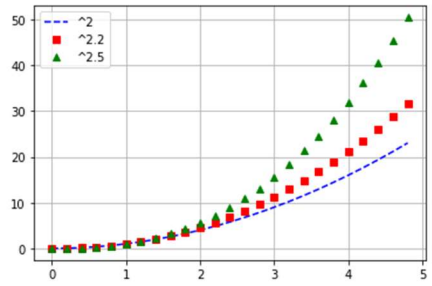
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...	...
95	96	5.7	3.0	4.2	1.2	Iris-versicolor
96	97	5.7	2.9	4.2	1.3	Iris-versicolor
97	98	6.2	2.9	4.3	1.3	Iris-versicolor
98	99	5.1	2.5	3.0	1.1	Iris-versicolor
99	100	5.7	2.8	4.1	1.3	Iris-versicolor

100 rows × 6 columns

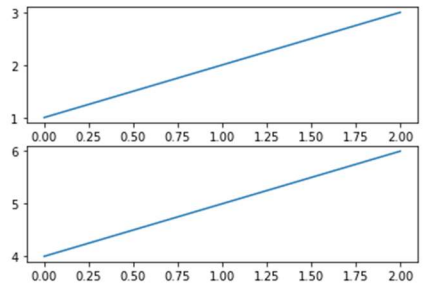
In [16]:
```
#Exploring Matplotlib

plt.plot([2,4,6,4,3,5])
plt.ylabel("Numbers")
plt.xlabel("Indices")
plt.title("MyPlot")
plt.grid()
plt.show()
```



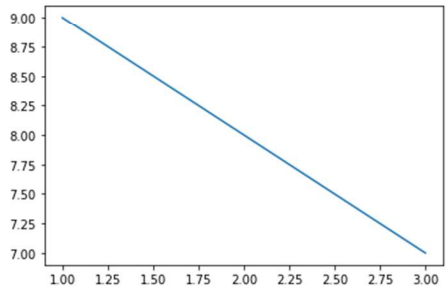
In [17]:
```
t = np.arange(0,5,0.2)

plt.plot(t,t**2,'b--',label="^2")
plt.plot(t,t**2.2,'rs',label="^2.2")
plt.plot(t,t**2.5,'g^',label="^2.5")
plt.grid()
plt.legend()
plt.show()
```

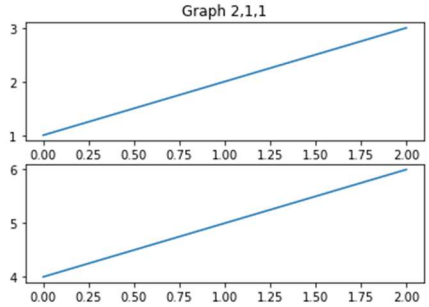


In [18]:
```
plt.figure(1)
plt.subplot(2,1,1)
plt.plot([1,2,3])
plt.subplot(2,1,2)
plt.plot([4,5,6])
plt.show()
```



In [19]:
```
plt.figure(1)
plt.subplot(2,1,1)
plt.plot([1,2,3])
plt.subplot(2,1,2)
plt.plot([4,5,6])

plt.figure(2)
plt.plot([1,2,3],[9,8,7])
plt.figure(1)
plt.subplot(2,1,1)
plt.title("Graph 2,1,1")
plt.show()
```





LEARNING OUTCOMES:

- Gain a better understanding of the NumPy, Pandas, and Matplotlib libraries.
- Learn how to use these libraries for data manipulation, analysis, and visualization.
- Understand how to use NumPy arrays, Pandas data frames, and Matplotlib plots to represent and manipulate data in Python.

EXPERIMENT 2

AIM: To perform data preprocessing and data visualization on iris dataset.

THEORY:

- Data preprocessing is the process of cleaning, transforming, and preparing raw data for analysis. It involves techniques such as data cleaning, feature scaling, feature extraction, and data integration.

- Data visualization is the representation of data in a graphical or pictorial format. It is used to help understand and communicate patterns, trends, and relationships in data.

- The iris dataset is a classic dataset in machine learning and is often used for testing classification algorithms. It consists of 150 samples of iris flowers, each with four features (sepal length, sepal width, petal length, and petal width) and a target variable (the species of the flower).

CODE & OUTPUT:

jupyter Lab2_DataPreprocessingandDataVisualisationIris Last Checkpoint: 10 minutes ago (autosaved) Logout

File Edit View Insert Cell Kernel Help Not Trusted Python 3

Run Code

In [1]:
```
#=======================================================#
#     LAB 2 - Iris Dataset EDA and Visualisation        #
#     Author: SHIKHAR ASTHANA                           #
#     Roll No. : 2K22/AFI/24                            #
#     Subject: ANN Lab (Anil Singh Parihar Sir)         #
#=======================================================#
```

In [2]:
```
# Importing the libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

#Additional libraries and commands
import warnings # current version of seaborn generates a lot of warnings that can be ignored
warnings.filterwarnings("ignore")
```

In [3]:
```
#Performing data preprocessing steps on Iris Dataset
#1. Importing the dataset
#2. Taking care of missing data, if any.
#3. Encoding Categorical data, if required.
#4. Train test splitting, if required.
#5. Feature Scaling, if required.

#Since in this lab, we are doing only EDA and visualisation, we can ignore steps 3, 4 and 5
```

In [4]:
```
#STEP 1 - Importing the Dataset
iris = pd.read_csv("iris.csv")
iris.head()
```

Out[4]:

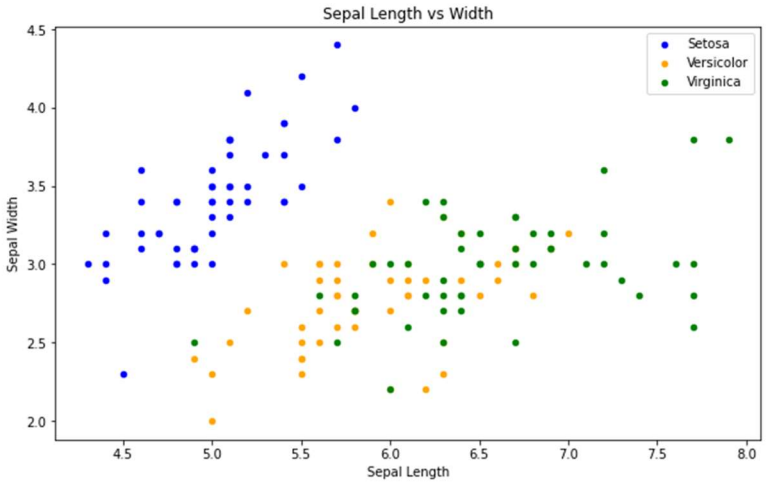
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

In [5]:
```
#STEP 2 - Taking care of missing data, if any.
iris.isna().sum()
```

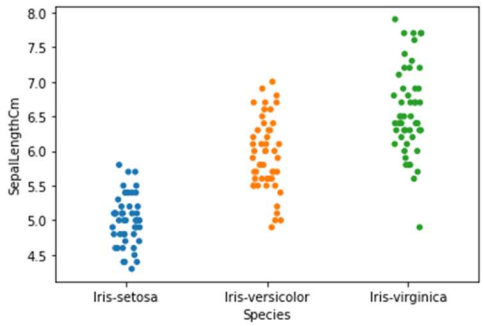
Out[5]:
```
Id              0
SepalLengthCm   0
SepalWidthCm    0
PetalLengthCm   0
PetalWidthCm    0
Species         0
dtype: int64
```

In [6]:
```
#As we can see, there are no missing values.
#Thus, we can see this dataset does not require much preprocessing. Which is to be expected considering
#Iris dataset is one of the most simplest and commonly used small dataset.
```

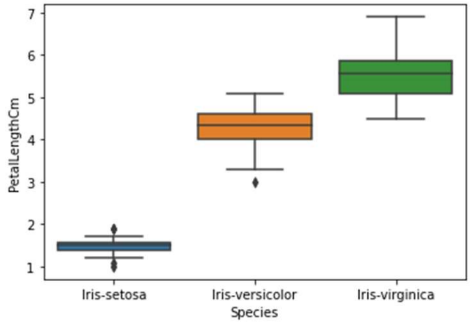
In [7]:
```
#Data Visualisation - Graphical visualisation.

#Using ScatterPlots of the Iris features
fig = iris[iris.Species=='Iris-setosa'].plot(kind='scatter',x='SepalLengthCm',y='SepalWidthCm',color='blue',
                                             label='Setosa',  figsize= (10,6))
iris[iris.Species=='Iris-versicolor'].plot(kind='scatter',x='SepalLengthCm',y='SepalWidthCm',color='orange',
                                           label='Versicolor',ax=fig)
iris[iris.Species=='Iris-virginica'].plot(kind='scatter',x='SepalLengthCm',y='SepalWidthCm',color='green',
                                          label='Virginica', ax=fig)
fig.set_xlabel("Sepal Length")
fig.set_ylabel("Sepal Width")
fig.set_title("Sepal Length vs Width")
plt.show()
```



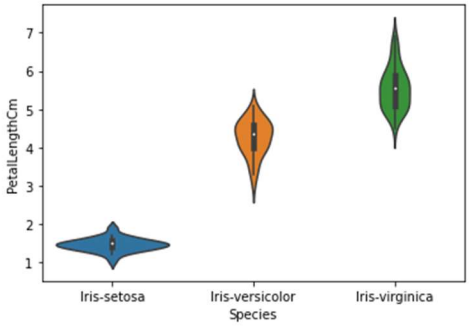
In [8]:
```
# Sepal Length using a Strip plot (Seaborn Library)
sns.stripplot(y ='SepalLengthCm', x = 'Species', data =iris)
plt.show()
```



In [9]:
```
#Individual feature in through a boxplot (Seaborn)
sns.boxplot(x="Species", y="PetalLengthCm", data=iris)
plt.show()
```



In [10]:
```
#Petal Length through Violin Plot
# A violin plot combines the benefits of the previous two plots (Strip Plot + Box Plot) and simplifies them
# Denser regions of the data are fatter, and sparser thiner in a violin plot

sns.violinplot(x="Species", y="PetalLengthCm", data=iris, size=6)
plt.show()
```

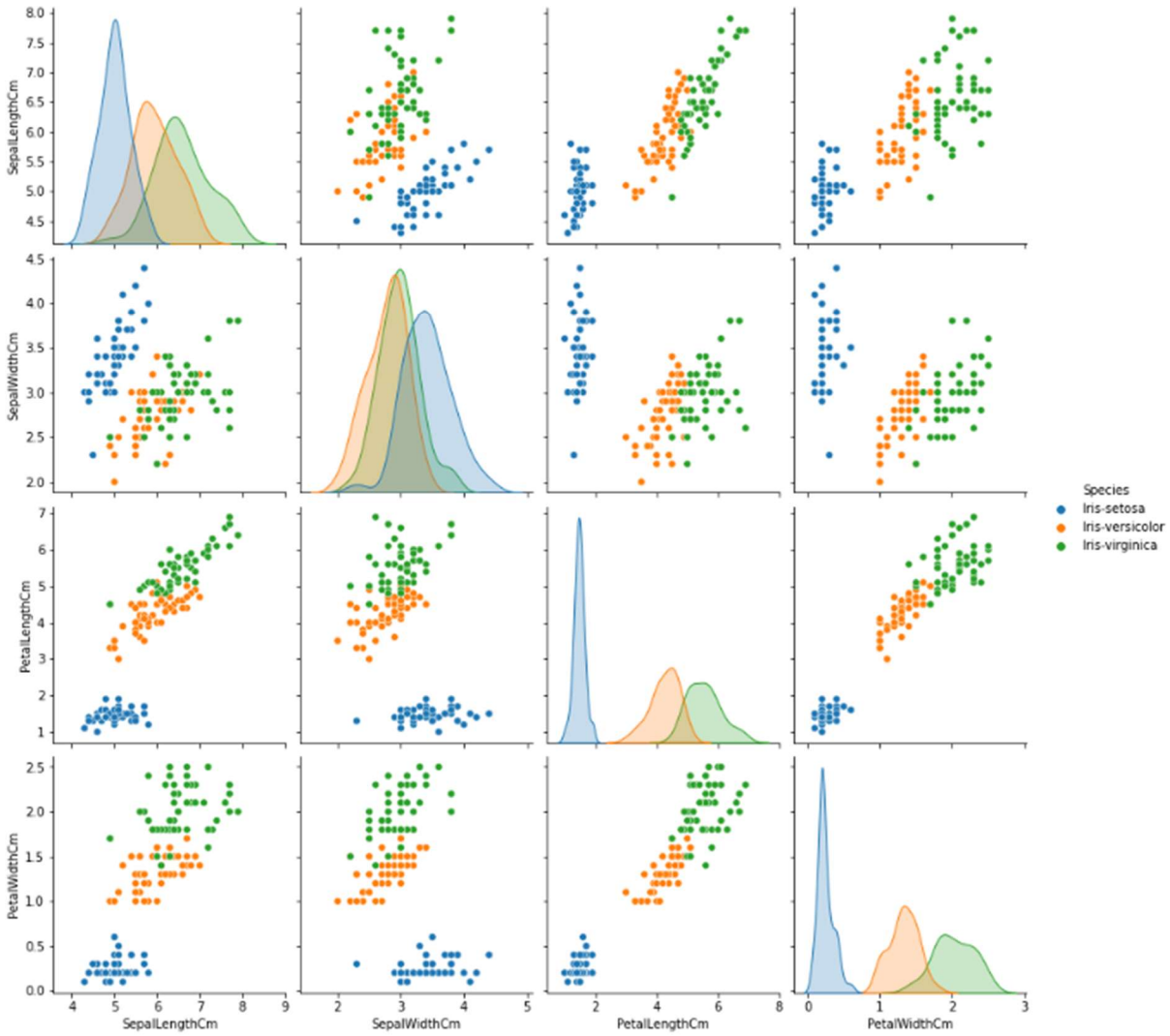


In [11]:
```
iris.columns
```

Out[11]:
```
Index(['Id', 'SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm',
       'Species'],
      dtype='object')
```

In [12]:

```
#pairplot, which shows the bivariate relation between each pair of features
sns.pairplot(data = iris[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm','Species']],
             hue="Species", size=3)
#ignoring ID column as it does not contribute anything to the dataset for classification or related purposes.
plt.show()
#From the pairplot, it is evident that Iris-setosa species is separataed from the other two across all feature combinations
```



LEARNING OUTCOMES:

- Learn how to perform data preprocessing techniques such as data cleaning, feature scaling, and feature extraction.
- Gain an understanding of data visualization techniques and how to use them to explore and communicate patterns in data.
- Understand how to use Python libraries such as Pandas, NumPy, Matplotlib, and Seaborn for data analysis and visualization.
- From the pair plot, it is evident that Iris-Setosa species is separated from the other two across all feature combinations.

EXPERIMENT 3

AIM: To implement uni-variate linear regression using gradient descent.

THEORY:

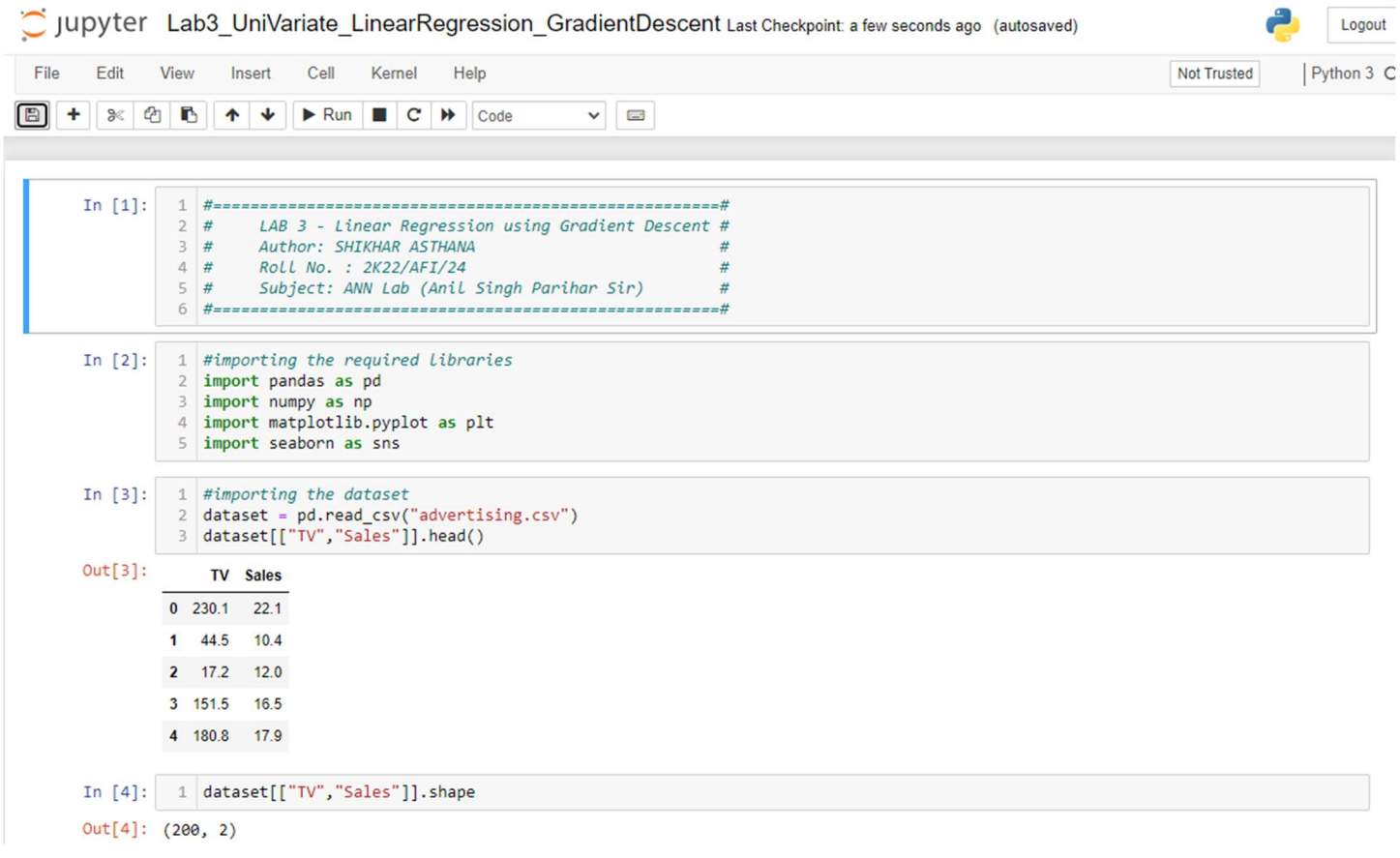
Linear regression is a supervised machine learning algorithm used for predicting a continuous output variable based on one or more input variables. Uni-variate linear regression is a specific type of linear regression that involves predicting a single output variable based on a single input variable.

The main steps of the uni-variate linear regression using gradient descent algorithm are as follows:

1. Given a training dataset of input-output pairs, initialize the model parameters (slope and intercept) to some random values.
2. Compute the predicted output for each input using the current model parameters.
3. Calculate the error between the predicted output and the actual output for each input.
4. Update the model parameters using the gradient of the cost function with respect to each parameter. The cost function is typically the mean squared error between the predicted output and the actual output.
5. Repeat steps 2-4 until the model converges or a maximum number of iterations is reached.

Gradient descent is an optimization algorithm that is used to minimize the cost function by iteratively adjusting the model parameters in the direction of steepest descent.

CODE & OUTPUT:



In [5]:
```
#keeping only the first column and target column

reg_data = dataset[["TV","Sales"]]

#Separating the columns based on independent and dependent variable.
x = reg_data['TV']
y = reg_data['Sales']

#Applying standardisation to the independent variable
x = (x - x.mean()) / x.std()

#adding a column of ones to signify the bias to be added when we do dot product with theta vector
x = np.c_[np.ones(x.shape[0]), x]
```

In [6]:
```
x[:5]
```

Out[6]:
```
array([[ 1.        ,  0.9674246 ],
       [ 1.        , -1.19437904],
       [ 1.        , -1.51235985],
       [ 1.        ,  0.05191939],
       [ 1.        ,  0.39319551]])
```

In [7]:
```
#Starting the gradient descent

#Setting the learning rate
alpha = 0.01

#Setting the number of iterations
iterations = 2000

#Finding number of points available in the dataset - 200
m = y.size

#Setting seed so that our values can be replicated
np.random.seed(123)
#Assigning random values as the start to our gradient descent
theta = np.random.rand(2)
```

In [8]:
```
theta
```

Out[8]:
```
array([0.69646919, 0.28613933])
```

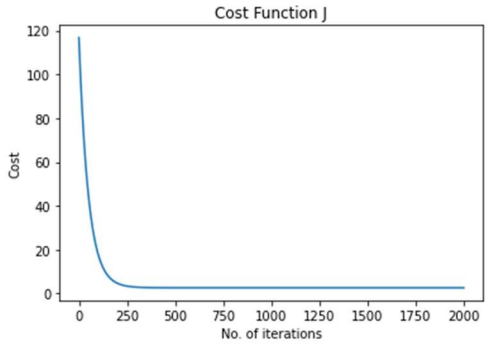
In [9]:
```
#Defining a function which apply gradient descent for the specified iterations and
#Calculate the final theta values
def gradient_descent(x, y, theta, iterations, alpha):

    #Creating past records to help us plot the graph later on and evaluate the performance of our algo
    past_costs = []
    past_thetas = [theta]

    #Actual loop to calculate predictions, then errors and cost, then gradients and finally update the thetas/weights
    for i in range(iterations):
        prediction = np.dot(x, theta)
        error = prediction - y
        cost = 1/(2*m) * np.dot(error.T, error)
        past_costs.append(cost)
        theta = theta - (alpha * (1/m) * np.dot(x.T, error))
        past_thetas.append(theta)

    return past_thetas, past_costs
```

In [10]:
```
#Actually running the above function on our dataset
past_thetas, past_costs = gradient_descent(x, y, theta, iterations, alpha)
theta = past_thetas[-1]
```

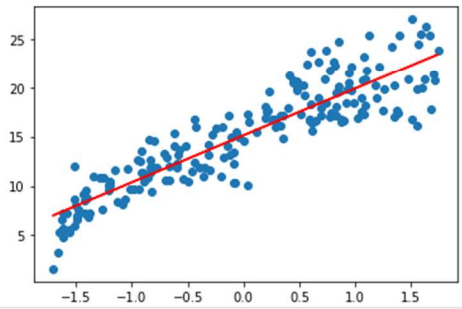
In [11]:
```
#Final values of our thetas
print(f" Gradient Descent: {round(theta[0],2)}, {round(theta[1],2)}")
```

```
Gradient Descent: 15.13, 4.76
```

In [12]:
```
#Plot the cost function...
plt.plot(past_costs)
plt.title('Cost Function J')
plt.xlabel('No. of iterations')
plt.ylabel('Cost')
plt.show()
```



In [13]:
```
#plot to show final linear regression line
prediction = np.dot(x, theta)
plt.scatter(x[:,1],y)
plt.plot(x[:,1],prediction,color="red")
plt.show()
```



LEARNING OUTCOMES:

- Learn how to implement uni-variate linear regression using gradient descent.
- Understand the concept of a cost function and how to minimize it using gradient descent.
- Learn how to use NumPy and Matplotlib to generate random data, manipulate arrays, and visualize the data and model.

EXPERIMENT 4

AIM: To implement multi-variate linear regression using gradient descent.

THEORY:

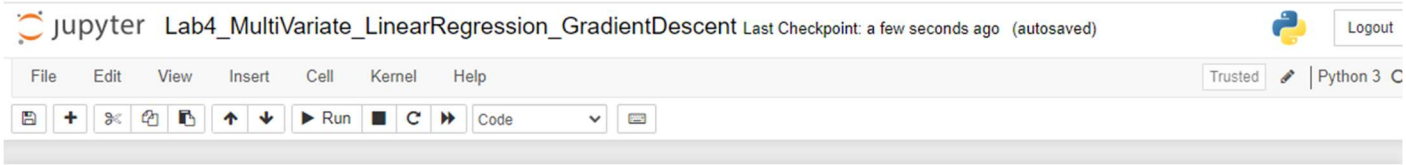
Multi-variate linear regression is a supervised machine learning algorithm used for predicting a continuous output variable based on multiple input variables. In contrast to uni-variate linear regression, which involves predicting a single output variable based on a single input variable, multi-variate linear regression involves predicting a single output variable based on multiple input variables.

The main steps of the multi-variate linear regression using gradient descent algorithm are as follows:

1. Given a training dataset of input-output pairs, initialize the model parameters (slopes and intercept) to some random values.
2. Compute the predicted output for each input using the current model parameters.
3. Calculate the error between the predicted output and the actual output for each input.
4. Update the model parameters using the gradient of the cost function with respect to each parameter. The cost function is typically the mean squared error between the predicted output and the actual output.
5. Repeat steps 2-4 until the model converges or a maximum number of iterations is reached.

Gradient descent is an optimization algorithm that is used to minimize the cost function by iteratively adjusting the model parameters in the direction of steepest descent.

CODE & OUTPUT:



In [1]:
```
#=====================================================#
#     LAB 4 - MultiVariate Linear Regression using GD  #
#     Author: SHIKHAR ASTHANA                          #
#     Roll No. : 2K22/AFI/24                           #
#     Subject: ANN Lab (Anil Singh Parihar Sir)        #
#=====================================================#
```

In [3]:
```
#importing the required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [4]:
```
#importing the dataset
dataset = pd.read_csv("advertising.csv")
dataset.head()
```

Out[4]:

	TV	Radio	Newspaper	Sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	12.0
3	151.5	41.3	58.5	16.5
4	180.8	10.8	58.4	17.9

In [5]:
```
dataset.shape
```

Out[5]:
```
(200, 4)
```

In [6]:
```
dataset.columns
```

Out[6]:
```
Index(['TV', 'Radio', 'Newspaper', 'Sales'], dtype='object')
```

In [11]:
```
#keeping only the first column and target column

reg_data = dataset.copy()

#Separating the columns based on independent and dependent variable.
x = reg_data[['TV', 'Radio', 'Newspaper']]
y = reg_data['Sales']
```

In [13]:
```
#Applying standardisation to the independent variable
x = (x - x.mean()) / x.std()

#adding a column of ones to signify the bias to be added when we do dot product with theta vector
x = np.c_[np.ones(x.shape[0]), x]
```

In [14]:
```
x[:5]
```

Out[14]:
```
array([[ 1.        ,  0.9674246 ,  0.97906559,  1.77449253],
       [ 1.        , -1.19437904,  1.0800974 ,  0.66790272],
       [ 1.        , -1.51235985,  1.52463736,  1.77908419],
       [ 1.        ,  0.05191939,  1.21480648,  1.28318502],
       [ 1.        ,  0.39319551, -0.83950698,  1.27859336]])
```

In [17]:
```
#Starting the gradient descent

#Setting the learning rate
alpha = 0.01

#Setting the number of iterations
iterations = 2000

#Finding number of points available in the dataset - 200
m = y.size

#Setting seed so that our values can be replicated
np.random.seed(123)
#Assigning random values as the start to our gradient descent
theta = np.random.rand(x.shape[1])
```

In [18]:
```
theta
```

Out[18]:
```
array([0.69646919, 0.28613933, 0.22685145, 0.55131477])
```

In [19]:
```
#Defining a function which apply gradient descent for the specified iterations and
#Calculate the final theta values
def gradient_descent(x, y, theta, iterations, alpha):

    #Creating past records to help us plot the graph later on and evaluate the performance of our algo
    past_costs = []
    past_thetas = [theta]

    #Actual loop to calculate predictions, then errors and cost, then gradients and finally update the thetas/weights
    for i in range(iterations):
        prediction = np.dot(x, theta)
        error = prediction - y
        cost = 1/(2*m) * np.dot(error.T, error)
        past_costs.append(cost)
        theta = theta - (alpha * (1/m) * np.dot(x.T, error))
        past_thetas.append(theta)

    return past_thetas, past_costs
```

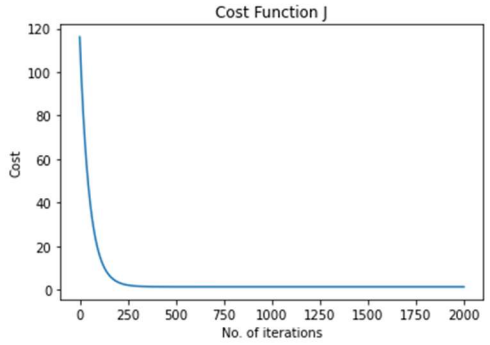
In [20]:
```
#Actually running the above function on our dataset
past_thetas, past_costs = gradient_descent(x, y, theta, iterations, alpha)
theta = past_thetas[-1]
```

In [26]:
```
#Final values of our thetas
print(f" Gradient Descent: {round(theta[0],2)}, {round(theta[1],2)}, {round(theta[2],2)}, {round(theta[3],2)}")
```

```
Gradient Descent: 15.13, 4.67, 1.59, 0.01
```

In [27]:

```
#Plot the cost function...
plt.plot(past_costs)
plt.title('Cost Function J')
plt.xlabel('No. of iterations')
plt.ylabel('Cost')
plt.show()
```



LEARNING OUTCOMES:

- Learn how to implement multi-variate linear regression using gradient descent.
- Understand the concept of a cost function and how to minimize it using gradient descent.
- Learn how to use NumPy and Matplotlib to generate random data, manipulate arrays, and visualize the data and model.

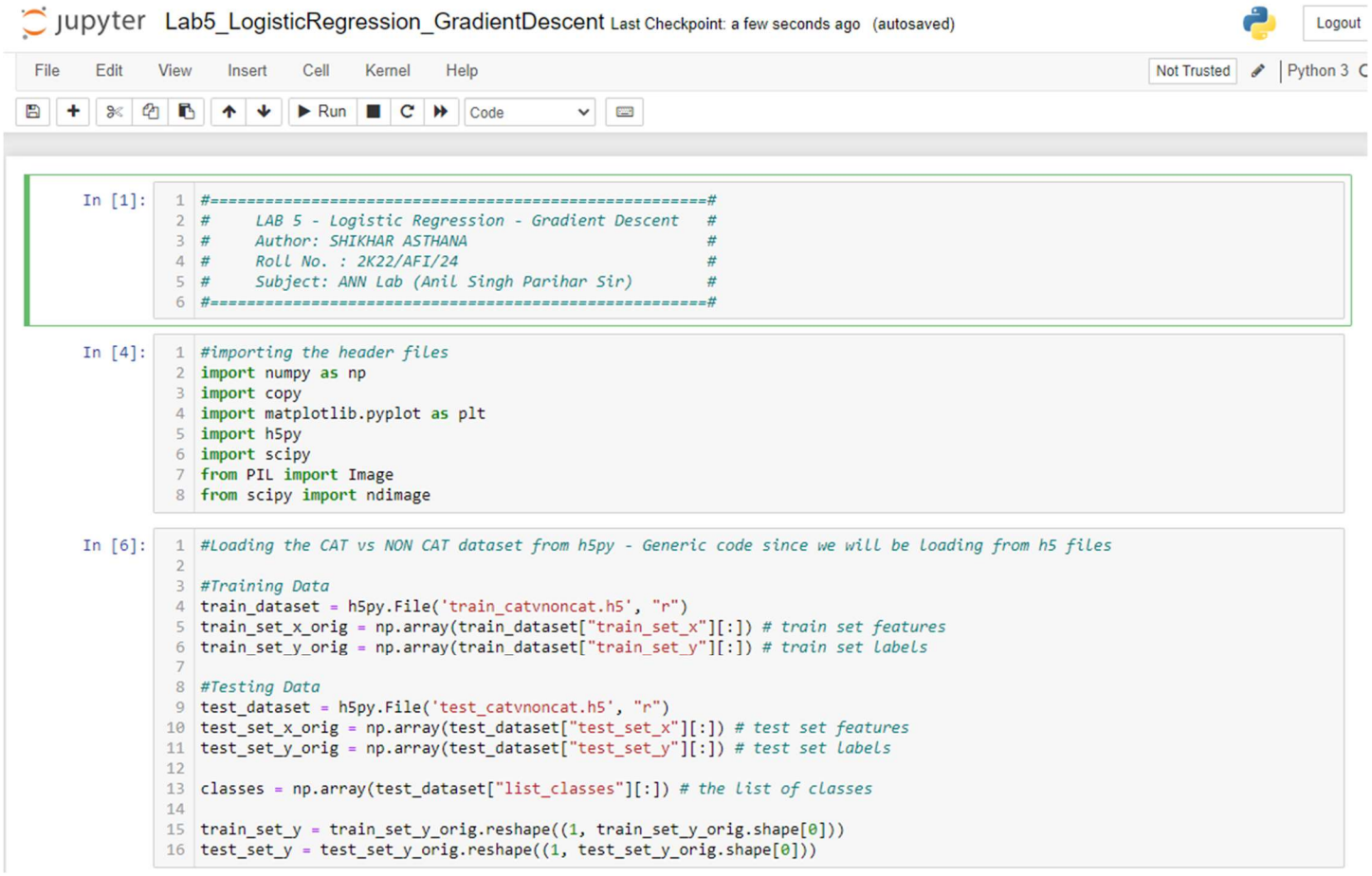
EXPERIMENT 5

AIM: To implement logistic regression using gradient descent.

THEORY:

- Logistic regression is a machine learning algorithm used for binary classification. It models the probability of a binary output variable (0 or 1) based on one or more input variables. It uses the sigmoid function to map any real-valued input to a value between 0 and 1.
- Gradient descent is an optimization algorithm used for finding the values of the parameters of a model that minimize a cost function. In logistic regression, the cost function is typically the log loss (also known as the cross-entropy loss).
- The gradient of a function is a vector that points in the direction of the steepest increase in the function. By taking steps in the opposite direction of the gradient, the parameters of a model can be updated to minimize the cost function.

CODE & OUTPUT:



In [1]:

```
#=====================================================#
#     LAB 5 - Logistic Regression - Gradient Descent  #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: ANN Lab (Anil Singh Parihar Sir)       #
#=====================================================#
```

In [4]:

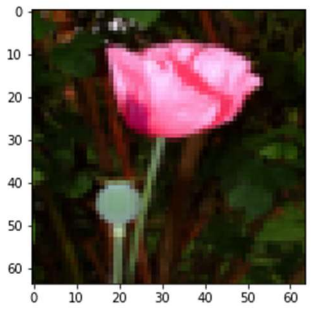
```
#importing the header files
import numpy as np
import copy
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
```

In [6]:

```
#Loading the CAT vs NON CAT dataset from h5py - Generic code since we will be loading from h5 files

#Training Data
train_dataset = h5py.File('train_catvnoncat.h5', "r")
train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # train set features
train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # train set labels

#Testing Data
test_dataset = h5py.File('test_catvnoncat.h5', "r")
test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # test set features
test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # test set labels

classes = np.array(test_dataset["list_classes"][:]) # the list of classes

train_set_y = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
test_set_y = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))
```

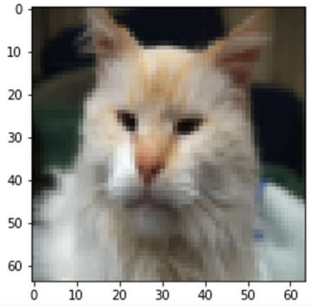
In [10]:
```
#checking a random image
plt.imshow(train_set_x_orig[30])
```

Out[10]: <matplotlib.image.AxesImage at 0x19f206e5130>



In [9]:
```
plt.imshow(train_set_x_orig[27])
```

Out[9]: <matplotlib.image.AxesImage at 0x19f2067b160>



In [12]:
```
#Finding information about the number of training samples, number of testing samples and image size

m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]

print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))

#Square images so image size is num_px * num_px * 3 (rgb)
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")

print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
train_set_x shape: (209, 64, 64, 3)
train_set_y shape: (1, 209)
test_set_x shape: (50, 64, 64, 3)
test_set_y shape: (1, 50)
```

In [15]:
```
#converting the images into a single flattened input vector
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0],-1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0],-1).T

print(train_set_x_flatten.shape)
print(test_set_x_flatten.shape)
```

```
(12288, 209)
(12288, 50)
```

In [16]:
```
#We need to standardise the images so that the value is 0-255
train_set_x = train_set_x_flatten / 255.
test_set_x = test_set_x_flatten / 255.
```

In [17]:

```
#Defining the sigmoid function which we will be using in our logistic regression
def sigmoid(z):
    s = 1/(1+np.exp(-z))
    return s

#Defining a function to initialise the weight vector with 0 values
def initialize_with_zeros(dim):
    w = np.zeros((dim,1))
    b = 0.0

    return w, b


#Defining a function which will calculate the forward pass and backward pass of the algorithm once
def propagate(w, b, X, Y):
    m = X.shape[1]

    #Forward propogation
    A = sigmoid(np.dot(w.T,X)+b)
    cost = (-1/m)*np.sum(np.dot(np.log(A),Y.T)+np.dot(np.log(1-A),1-Y.T))

    #backward pass
    dw = (1/m)*np.dot(X,(A-Y).T)
    db = (1/m)*np.sum(A-Y)

    #cost will be an array which will retain costs of previous cycles as well
    cost = np.squeeze(np.array(cost))

    #grads will only be for the current cycle
    grads = {"dw": dw,
             "db": db}

    return grads, cost
```

In [18]:

```
#Function to actually implement the gradient descent algorithm
def optimize(w, b, X, Y, num_iterations=100, learning_rate=0.009, print_cost=False):

    #creating deep copies so that we can update them simultaneously at the end
    w = copy.deepcopy(w)
    b = copy.deepcopy(b)

    costs = []

    for i in range(num_iterations):
        #Applying the forward and backward pass
        grads,cost = propagate(w,b,X,Y)

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        #Updating weights using gradient descent
        w = w - (learning_rate * dw)
        b = b - (learning_rate * db)

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)

            # Print the cost every 100 training iterations
            if print_cost:
                print ("Cost after iteration %i: %f" %(i, cost))


    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs
```

In [19]:

```
#Function to predict
def predict(w, b, X):

    m = X.shape[1]
    Y_prediction = np.zeros((1, m))
    w = w.reshape(X.shape[0], 1)

    #calculating the values using dot product
    A = sigmoid(np.dot(w.T,X)+b)

    #these values need to be assigned either 0 or 1 since logistic regression does not work on continuous values
    for i in range(A.shape[1]):
        if A[0,i] > 0.5:
            Y_prediction[0,i] = 1
        else:
            Y_prediction[0,i] = 0

    return Y_prediction
```

In [20]:

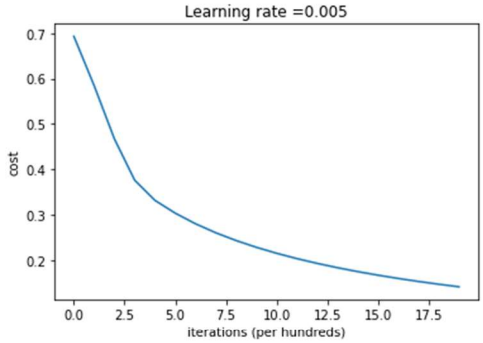
```
#Building a complete logistic regression model using the above defined functions
def model(X_train, Y_train, X_test, Y_test, num_iterations=2000, learning_rate=0.5, print_cost=False):

    #Initialise the weights
    w,b = initialize_with_zeros(X_train.shape[0])

    #Apply gradient descent
    params, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate, print_cost)
    w = params["w"]
    b = params["b"]

    #Get predictions for both train and test
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    # Print train/test Errors
    if print_cost:
        print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - Y_train)) * 100))
        print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100))


    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d
```

In [21]:

```
#actually running the aove defined model on our cats dataset
logistic_regression_model = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations=2000,
                                  learning_rate=0.005, print_cost=True)

```

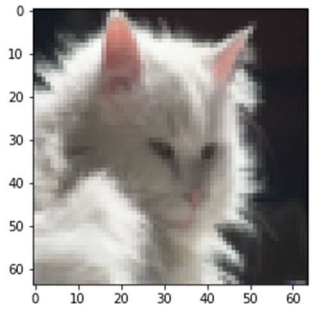
```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 800: 0.242941
Cost after iteration 900: 0.228004
Cost after iteration 1000: 0.214820
Cost after iteration 1100: 0.203078
Cost after iteration 1200: 0.192544
Cost after iteration 1300: 0.183033
Cost after iteration 1400: 0.174399
Cost after iteration 1500: 0.166521
Cost after iteration 1600: 0.159305
Cost after iteration 1700: 0.152667
Cost after iteration 1800: 0.146542
Cost after iteration 1900: 0.140872
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
```

In [22]:
```
# Plot learning curve (with costs)
costs = np.squeeze(logistic_regression_model['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(logistic_regression_model["learning_rate"]))
plt.show()
```



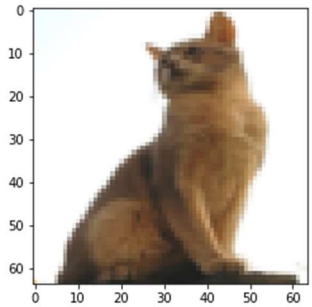
In [25]:
```
#checking random output
index = 8
plt.imshow(test_set_x[:, index].reshape((num_px, num_px, 3)))
print ("y = " + str(test_set_y[0,index]) + ", Prediction: " + classes[int(logistic_regression_model['Y_prediction_test'][0,i
```

```
y = 1, Prediction: cat
```



In [27]:
```
#checking random output
index = 6
plt.imshow(test_set_x[:, index].reshape((num_px, num_px, 3)))
print ("y = " + str(test_set_y[0,index]) + ", Prediction: " + classes[int(logistic_regression_model['Y_prediction_test'][0,i
```

```
y = 1, Prediction: non-cat
```



LEARNING OUTCOMES:

- Learn how to implement logistic regression using gradient descent.
- Understand the concept of a cost function and how to minimize it using gradient descent.
- Learn how to use NumPy and Matplotlib to generate random data, manipulate arrays, and visualize the data and model.

EXPERIMENT 6

AIM: To implement basic perceptron model.

THEORY:

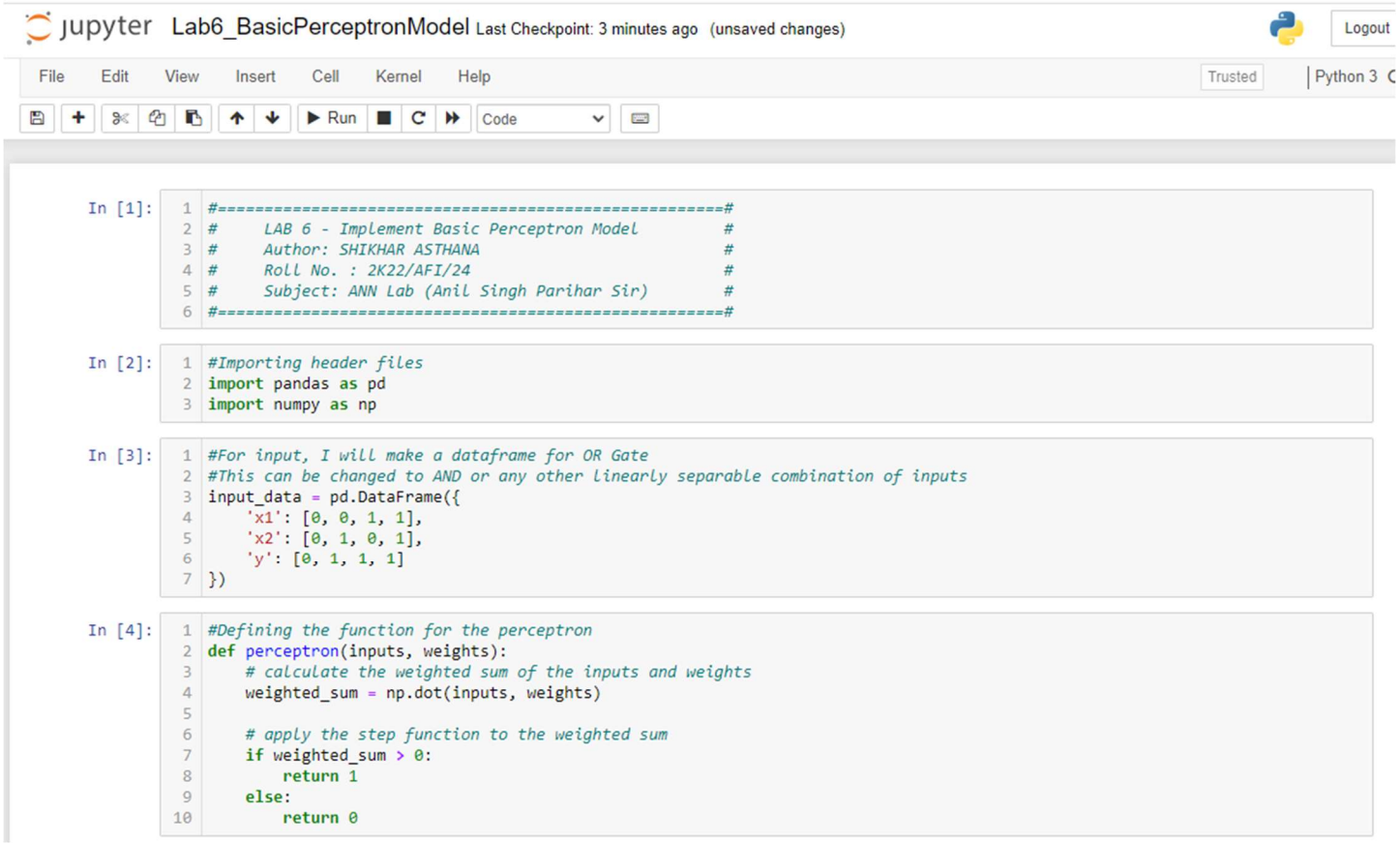
Perceptron is a simple supervised learning algorithm used for binary classification. It takes in input values and produces a binary output (0 or 1) based on a weighted sum of those input values. The weights are adjusted during training in order to minimize the classification error.

The basic steps of the perceptron algorithm are as follows:

1. Initialize the weights to small random values.
2. For each input-output pair in the training data:
    a. Calculate the weighted sum of the inputs.
    b. Apply an activation function to the weighted sum to obtain the predicted output.
    c. Update the weights based on the difference between the predicted output and the true output.
3. Repeat step 2 for a fixed number of epochs or until convergence.

The perceptron algorithm is a linear classification algorithm, which means it can only separate data that is linearly separable.

CODE & OUTPUT:



In [1]:
```
#=====================================================#
#     LAB 6 - Implement Basic Perceptron Model        #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: ANN Lab (Anil Singh Parihar Sir)       #
#=====================================================#
```

In [2]:
```
#Importing header files
import pandas as pd
import numpy as np
```

In [3]:
```
#For input, I will make a dataframe for OR Gate
#This can be changed to AND or any other linearly separable combination of inputs
input_data = pd.DataFrame({
    'x1': [0, 0, 1, 1],
    'x2': [0, 1, 0, 1],
    'y': [0, 1, 1, 1]
})
```

In [4]:
```
#Defining the function for the perceptron
def perceptron(inputs, weights):
    # calculate the weighted sum of the inputs and weights
    weighted_sum = np.dot(inputs, weights)

    # apply the step function to the weighted sum
    if weighted_sum > 0:
        return 1
    else:
        return 0
```

In [5]:
```python
# set the learning rate and maximum number of iterations
learning_rate = 0.1
max_iterations = 100

# randomly initialize the weights
weights = np.random.uniform(size=3)

# iterate through the data and update the weights
for i in range(max_iterations):
    # iterate through each row in the input data
    for index, row in input_data.iterrows():
        # extract the inputs and output for this row
        inputs = row[['x1', 'x2']]
        expected_output = row['y']

        # add a bias term to the inputs
        inputs_with_bias = np.append(inputs, 1)

        # calculate the predicted output using the perceptron function
        predicted_output = perceptron(inputs_with_bias, weights)

        # calculate the error and update the weights
        error = expected_output - predicted_output
        weights += learning_rate * error * inputs_with_bias

        # check if the model has converged
        if error == 0:
            break
```

In [6]:
```python
#Checking final weights
print(weights)
```

```
[ 0.23507456  0.53046904 -0.00477351]
```

In [7]:
```python
# create a new input with values [1, 1]
new_input = np.array([1, 1, 1])

# predict the output using the perceptron
predicted_output = perceptron(new_input, weights)

print(predicted_output)  # should output 1
```

```
1
```

LEARNING OUTCOMES:

- Understanding the basic concepts of supervised machine learning and binary classification tasks.
- Familiarity with the perceptron algorithm and its use for binary classification tasks.
- Ability to implement the perceptron algorithm using Python and the numpy library, including how to calculate the weighted sum, apply the activation function, and update the weights.
- Understanding of the limitations of the perceptron algorithm, including its inability to handle non-linearly separable data and its sensitivity to the initial weights.
- Familiarity with techniques for improving the performance of the model, such as using a different activation function or incorporating regularization.

EXPERIMENT 7

AIM: To implement multi-layer neural network.

THEORY:

A multi-layer neural network is a type of deep learning model that consists of multiple layers of interconnected nodes (neurons). Each layer performs a set of computations on its inputs and produces an output that serves as the input to the next layer. The first layer is called the input layer, the last layer is called the output layer, and all other layers are called hidden layers.

The basic steps of training a multi-layer neural network are as follows:

1. Initialize the weights and biases to small random values.
2. For each input-output pair in the training data:
    a. Feed the input forward through the network to obtain the predicted output.
    b. Calculate the error between the predicted output and the true output.
    c. Backpropagate the error through the network to update the weights and biases.
3. Repeat step 2 for a fixed number of epochs or until convergence.

During training, the weights and biases are updated using an optimization algorithm such as stochastic gradient descent. There are also many variations of the basic neural network architecture, including different activation functions, regularization techniques, and architectures such as convolutional neural networks and recurrent neural networks.

The Boston Housing Dataset is a derived from information collected by the U.S. Census Service concerning housing in the area of Boston MA. The following describes the dataset columns:

1. CRIM - per capita crime rate by town
2. ZN - proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS - proportion of non-retail business acres per town.
4. CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)
5. NOX - nitric oxides concentration (parts per 10 million)
6. RM - average number of rooms per dwelling
7. AGE - proportion of owner-occupied units built prior to 1940
8. DIS - weighted distances to five Boston employment centers
9. RAD - index of accessibility to radial highways
10. TAX - full-value property-tax rate per $10,000
11. PTRATIO - pupil-teacher ratio by town
12. B - 1000(Bk - 0.63) ^2 where Bk is the proportion of blacks by town
13. LSTAT - % lower status of the population
14. MEDV - Median value of owner-occupied homes in $1000's

Variables 1-13 will be our independent variables.
Variable 14 would be our dependent variable.

CODE & OUTPUT:

jupyter Lab7_MultiLayerNeuralNetwork_BostonHousing_Dataset Last Checkpoint: a few seconds ago (autosaved) Logout

File Edit View Insert Cell Kernel Help Trusted Python 3

Run Code

In [1]:
```
#=====================================================#
#     LAB 7 - MultiLayer Neural Network               #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: ANN Lab (Anil Singh Parihar Sir)       #
#=====================================================#
```

In [17]:
```
#Importing header files
import tensorflow as tf
from tensorflow import keras
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
```

In [4]:
```
#Using the shallow neural network for house price prediction
#Using the prebuild dataset - Boston Housing Dataset
boston = load_boston()
X = boston.data
y = boston.target
```

In [9]:
```
X[:1][0]
```

Out[9]:
```
array([6.320e-03, 1.800e+01, 2.310e+00, 0.000e+00, 5.380e-01, 6.575e+00,
       6.520e+01, 4.090e+00, 1.000e+00, 2.960e+02, 1.530e+01, 3.969e+02,
       4.980e+00])
```

In [10]:
```
y[:1][0]
```

Out[10]:
```
24.0
```

In [11]:
```
#Splitting the dataset into train test
#Using 80 - 20 split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [12]:
```
#Applying standardization for the datasets
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [13]:
```
#Building a sequential neural network model
#Using one hidden layer containing 32 neurons
#A ReLU activation function
#linear activation function in the output layer, as this is a regression problem
model = keras.Sequential([
    keras.layers.Dense(32, activation='relu', input_shape=(X_train.shape[1],)),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(1, activation='linear')
])
```

In [14]:
```
model.summary()
```

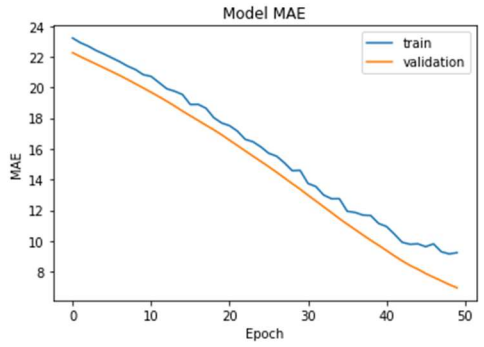
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 32)                448
_________________________________________________________________
dropout (Dropout)            (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 481
Trainable params: 481
Non-trainable params: 0
_________________________________________________________________
```

In [15]:
```
#Defining generic loss functions, optimizers and metrics to be used by our model in training
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mean_absolute_error'])
```

In [16]:
```
#Actually training the model
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2)

test_loss, test_mae = model.evaluate(X_test, y_test)
print('Test MAE:', test_mae)
```

```
323/323 [==============================] - 0s 95us/sample - loss: 144.6639 - mean_absolute_error: 9.8163 - val_loss: 99.3749
- val_mean_absolute_error: 8.1632
Epoch 46/50
323/323 [==============================] - 0s 142us/sample - loss: 138.1007 - mean_absolute_error: 9.6319 - val_loss: 94.1519
- val_mean_absolute_error: 7.8846
Epoch 47/50
323/323 [==============================] - 0s 88us/sample - loss: 146.8096 - mean_absolute_error: 9.8130 - val_loss: 89.8243
- val_mean_absolute_error: 7.6433

Epoch 48/50
323/323 [==============================] - 0s 94us/sample - loss: 136.8551 - mean_absolute_error: 9.2996 - val_loss: 85.7512
- val_mean_absolute_error: 7.4069
Epoch 49/50
323/323 [==============================] - 0s 91us/sample - loss: 124.8400 - mean_absolute_error: 9.1616 - val_loss: 81.7915
- val_mean_absolute_error: 7.1644
Epoch 50/50
323/323 [==============================] - 0s 88us/sample - loss: 129.9768 - mean_absolute_error: 9.2384 - val_loss: 78.3691
- val_mean_absolute_error: 6.9540
102/102 [==============================] - 0s 65us/sample - loss: 71.3065 - mean_absolute_error: 6.8688
Test MAE: 6.868761
```

In [18]:
```
# Plot training history
plt.plot(history.history['mean_absolute_error'])
plt.plot(history.history['val_mean_absolute_error'])
plt.title('Model MAE')
plt.ylabel('MAE')
plt.xlabel('Epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



LEARNING OUTCOMES:

- Understanding the basic concepts of deep learning and neural networks.
- Familiarity with the architecture of a multi-layer neural network, including input, output, and hidden layers.
- Understanding of the training process for a neural network, including forward propagation, backpropagation, and weight updates.
- Familiarity with techniques for improving the performance of a neural network, such as using different activation functions or regularization techniques.
- Knowledge of the various applications of neural networks, such as image classification, natural language processing, and time series analysis.

EXPERIMENT 8

AIM: To perform classification on MNIST Digit dataset using Neural Network.

THEORY:

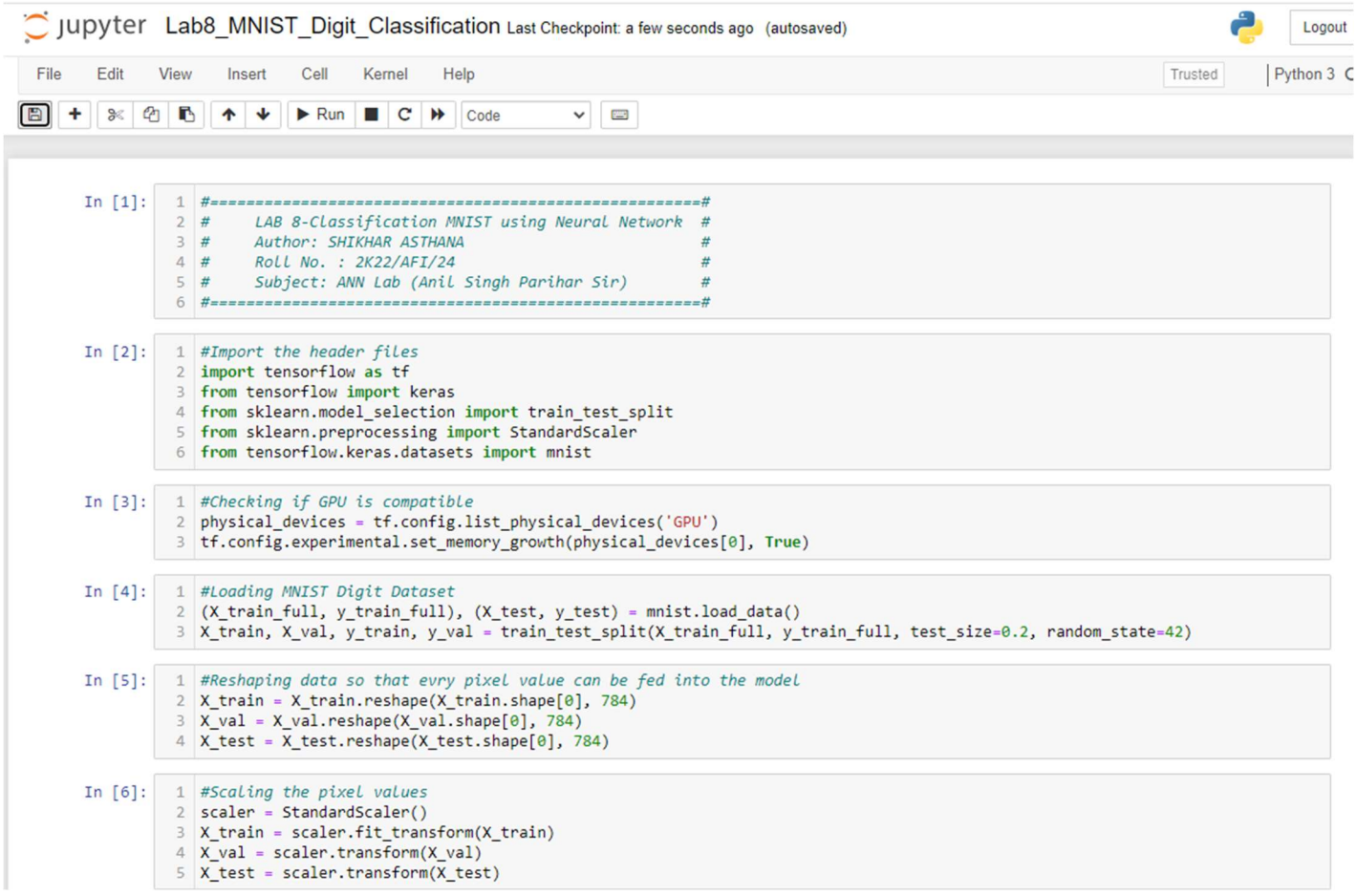
The MNIST dataset is a large collection of handwritten digit images that is commonly used as a benchmark dataset for image classification tasks. The dataset consists of 60,000 training images and 10,000 testing images, with each image being a grayscale image of size 28x28 pixels.

Neural networks are a class of machine learning algorithms that are designed to model the behavior of the human brain. They consist of interconnected nodes (neurons) that process information and produce an output. Neural networks can be used for a variety of tasks, including image classification.

To perform classification on the MNIST dataset using a neural network, we typically follow these steps:

1. Preprocess the data: normalize the pixel values, and split the data into training and testing sets.
2. Define the neural network architecture: specify the number and size of the layers, the activation functions, and the loss function.
3. Train the network: feed the training data through the network and adjust the weights and biases using an optimization algorithm such as stochastic gradient descent.
4. Evaluate the network: feed the testing data through the network and calculate the accuracy of the predictions.

CODE & OUTPUT:



In [1]:
```
#=====================================================#
#     LAB 8-Classification MNIST using Neural Network  #
#     Author: SHIKHAR ASTHANA                          #
#     Roll No. : 2K22/AFI/24                           #
#     Subject: ANN Lab (Anil Singh Parihar Sir)        #
#=====================================================#
```

In [2]:
```
#Import the header files
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.datasets import mnist
```

In [3]:
```
#Checking if GPU is compatible
physical_devices = tf.config.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

In [4]:
```
#Loading MNIST Digit Dataset
(X_train_full, y_train_full), (X_test, y_test) = mnist.load_data()
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full, test_size=0.2, random_state=42)
```

In [5]:
```
#Reshaping data so that evry pixel value can be fed into the model
X_train = X_train.reshape(X_train.shape[0], 784)
X_val = X_val.reshape(X_val.shape[0], 784)
X_test = X_test.reshape(X_test.shape[0], 784)
```

In [6]:
```
#Scaling the pixel values
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

In [7]:
```
#Building the model
#Currently only 1 hidden layer
model = keras.Sequential([
    keras.layers.Dense(128, activation='relu', input_shape=(784,)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation='softmax')
])
```

In [8]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

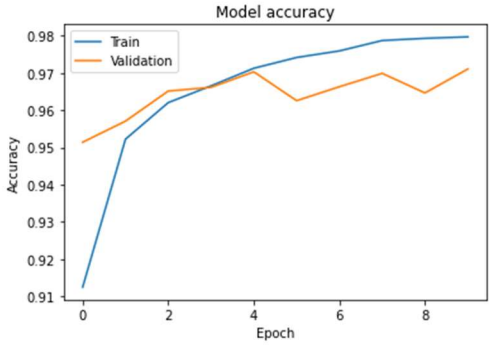
In [9]:
```
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In [10]:
```
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val, y_val))

test_loss, test_acc = model.evaluate(X_test, y_test)
print('Test accuracy:', test_acc)
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/10
48000/48000 [==============================] - 85s 2ms/sample - loss: 0.3116 - accuracy: 0.9125 - val_loss: 0.2300 - val_accura
cy: 0.9514
Epoch 2/10
48000/48000 [==============================] - 2s 50us/sample - loss: 0.1759 - accuracy: 0.9522 - val_loss: 0.2219 - val_accura
cy: 0.9571
Epoch 3/10
48000/48000 [==============================] - 2s 49us/sample - loss: 0.1314 - accuracy: 0.9621 - val_loss: 0.2143 - val_accura
cy: 0.9652
Epoch 4/10
48000/48000 [==============================] - 2s 49us/sample - loss: 0.1155 - accuracy: 0.9666 - val_loss: 0.2018 - val_accura
cy: 0.9662
Epoch 5/10
48000/48000 [==============================] - 2s 48us/sample - loss: 0.0947 - accuracy: 0.9713 - val_loss: 0.2050 - val_accura
cy: 0.9703
Epoch 6/10
48000/48000 [==============================] - 3s 52us/sample - loss: 0.0855 - accuracy: 0.9742 - val_loss: 0.2848 - val_accura
cy: 0.9626
Epoch 7/10
48000/48000 [==============================] - 2s 49us/sample - loss: 0.0760 - accuracy: 0.9760 - val_loss: 0.3622 - val_accura
cy: 0.9663
Epoch 8/10
48000/48000 [==============================] - 2s 48us/sample - loss: 0.0677 - accuracy: 0.9787 - val_loss: 0.3407 - val_accura
cy: 0.9699
Epoch 9/10
48000/48000 [==============================] - 2s 49us/sample - loss: 0.0791 - accuracy: 0.9793 - val_loss: 0.3061 - val_accura
cy: 0.9647
Epoch 10/10
48000/48000 [==============================] - 2s 51us/sample - loss: 0.0686 - accuracy: 0.9797 - val_loss: 0.3045 - val_accura
cy: 0.9711
10000/10000 [==============================] - 0s 43us/sample - loss: 0.1843 - accuracy: 0.9708
Test accuracy: 0.9708
```

In [11]:

```
# Plot training history
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



LEARNING OUTCOMES:

- Understanding of the MNIST dataset and its use as a benchmark dataset for image classification tasks.
- Familiarity with the architecture of a neural network for image classification, including the number and size of layers, activation functions, and loss function.
- Ability to implement a neural network using Python and a deep learning library such as TensorFlow or PyTorch for image classification on the MNIST dataset.

EXPERIMENT 9

AIM: To perform classification on MNIST Fashion dataset using Neural Network.

THEORY:

The MNIST Fashion dataset is a collection of grayscale images of clothing items, such as t-shirts, dresses, shoes, and bags. The dataset consists of 60,000 training images and 10,000 testing images, each of size 28x28 pixels. The goal of the classification task is to classify each image into one of 10 categories representing different clothing types.

Neural networks are a class of machine learning algorithms that are designed to model the behavior of the human brain. They consist of interconnected nodes (neurons) that process information and produce an output. Neural networks can be used for a variety of tasks, including image classification.

To perform classification on the MNIST dataset using a neural network, we typically follow these steps:

1. Preprocess the data: normalize the pixel values, and split the data into training and testing sets.
2. Define the neural network architecture: specify the number and size of the layers, the activation functions, and the loss function.
3. Train the network: feed the training data through the network and adjust the weights and biases using an optimization algorithm such as stochastic gradient descent.
4. Evaluate the network: feed the testing data through the network and calculate the accuracy of the predictions.

CODE & OUTPUT:

In [6]:
```
#Scaling the pixel values
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

In [7]:
```
#Building the model
#Currently only 3 hidden layer
model = keras.Sequential([
    keras.layers.Dense(128, activation='relu', input_shape=(784,)),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation='softmax')
])
```

In [8]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_2 (Dense)              (None, 32)                2080
_________________________________________________________________
dropout (Dropout)            (None, 32)                0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                330
=================================================================
Total params: 111,146
Trainable params: 111,146
Non-trainable params: 0
_________________________________________________________________
```

In [9]:
```
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

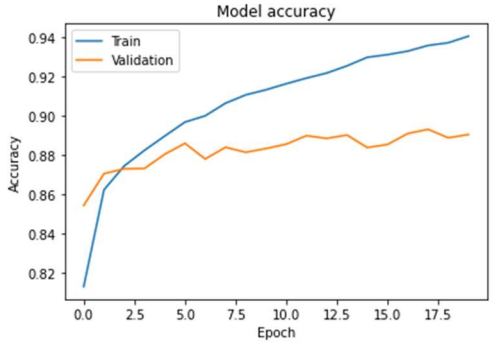
In [10]:
```
history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_val, y_val))

test_loss, test_acc = model.evaluate(X_test, y_test)
print('Test accuracy:', test_acc)
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [==============================] - 100s 2ms/sample - loss: 0.5328 - accuracy: 0.8128 - val_loss: 0.3998 - val_accur
acy: 0.8543
Epoch 2/20
48000/48000 [==============================] - 5s 112us/sample - loss: 0.3876 - accuracy: 0.8622 - val_loss: 0.3563 - val_accur
acy: 0.8705
Epoch 3/20
48000/48000 [==============================] - 5s 113us/sample - loss: 0.3480 - accuracy: 0.8744 - val_loss: 0.3567 - val_accur
acy: 0.8730
Epoch 4/20
48000/48000 [==============================] - 6s 115us/sample - loss: 0.3237 - accuracy: 0.8824 - val_loss: 0.3576 - val_accur
acy: 0.8732
Epoch 5/20
48000/48000 [==============================] - 6s 116us/sample - loss: 0.3001 - accuracy: 0.8897 - val_loss: 0.3425 - val_accur
acy: 0.8804
Epoch 6/20
48000/48000 [==============================] - 5s 113us/sample - loss: 0.2811 - accuracy: 0.8969 - val_loss: 0.3378 - val_accur
acy: 0.8860
Epoch 7/20
48000/48000 [==============================] - 6s 115us/sample - loss: 0.2713 - accuracy: 0.9001 - val_loss: 0.3554 - val_accur
acy: 0.8780
Epoch 8/20
48000/48000 [==============================] - 6s 116us/sample - loss: 0.2540 - accuracy: 0.9065 - val_loss: 0.3341 - val_accur
acy: 0.8840
Epoch 9/20
48000/48000 [==============================] - 5s 114us/sample - loss: 0.2433 - accuracy: 0.9107 - val_loss: 0.3656 - val_accur
acy: 0.8814
Epoch 10/20
48000/48000 [==============================] - 5s 111us/sample - loss: 0.2340 - accuracy: 0.9134 - val_loss: 0.3557 - val_accur
acy: 0.8833
Epoch 11/20
48000/48000 [==============================] - 6s 118us/sample - loss: 0.2229 - accuracy: 0.9165 - val_loss: 0.3575 - val_accur
acy: 0.8856
Epoch 12/20
48000/48000 [==============================] - 6s 122us/sample - loss: 0.2145 - accuracy: 0.9193 - val_loss: 0.3443 - val_accur
acy: 0.8899
Epoch 13/20
48000/48000 [==============================] - 6s 116us/sample - loss: 0.2079 - accuracy: 0.9219 - val_loss: 0.3876 - val_accur
acy: 0.8885
```

```
Epoch 14/20
48000/48000 [==============================] - 6s 116us/sample - loss: 0.2022 - accuracy: 0.9256 - val_loss: 0.3645 - val_accur
acy: 0.8903
Epoch 15/20
48000/48000 [==============================] - 6s 117us/sample - loss: 0.1889 - accuracy: 0.9300 - val_loss: 0.4088 - val_accur
acy: 0.8838
Epoch 16/20
48000/48000 [==============================] - 5s 114us/sample - loss: 0.1834 - accuracy: 0.9313 - val_loss: 0.3853 - val_accur
acy: 0.8854
Epoch 17/20
48000/48000 [==============================] - 5s 115us/sample - loss: 0.1789 - accuracy: 0.9331 - val_loss: 0.3819 - val_accur
acy: 0.8910
Epoch 18/20
48000/48000 [==============================] - 5s 115us/sample - loss: 0.1689 - accuracy: 0.9360 - val_loss: 0.3919 - val_accur
acy: 0.8932
Epoch 19/20
48000/48000 [==============================] - 5s 113us/sample - loss: 0.1680 - accuracy: 0.9374 - val_loss: 0.4258 - val_accur
acy: 0.8888
Epoch 20/20
48000/48000 [==============================] - 5s 115us/sample - loss: 0.1585 - accuracy: 0.9408 - val_loss: 0.4343 - val_accur
acy: 0.8905
10000/10000 [==============================] - 1s 86us/sample - loss: 0.4757 - accuracy: 0.8794
Test accuracy: 0.8794
```

In [11]:

```python
# Plot training history
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



LEARNING OUTCOMES:

- Understanding of the MNIST Fashion dataset and its use as a benchmark dataset for image classification tasks.
- Familiarity with the architecture of a neural network for image classification, including the number and size of layers, activation functions, and loss function.
- Ability to preprocess image data for use in a neural network, including normalization and splitting into training and testing sets.
- Ability to implement a neural network using Python and a deep learning library such as TensorFlow or PyTorch for image classification on the MNIST Fashion dataset.