Delhi Technological University

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Bawana Road, Delhi-110042

Machine Learning Lab

Subject Code: AI502

Lab File

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Submitted To:

Mr. Kavinder Singh

Assistant Professor

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 linear regression using gradient descent equations.		
4	To implement linear regression using normal equations.		
5	To implement logistic regression using gradient equations.		
6	To implement decision tree using ID3 algorithm.		
7	To implement decision tree using C4.5 algorithm.		
8	To implement decision tree using CART algorithm.		
9	To implement data classification using SVM.		
10	To implement data classification using KNN.		
11	To implement K Means Clustering Algorithm.		
12	To implement 2-layer Neural Network.		
13	To implement Convolution operation of CNN.		
14	To perform MNIST data classification using CNN.		

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:

jupyter Lab1_NumpyPandasMatplotlib Last Checkpoint: 10 minutes ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

Run Code

In [1]:
```
#=====================================================#
#     LAB 1 - NumPy, Pandas, and Matplotlib libraries  #
#     Author: SHIKHAR ASTHANA                          #
#     Roll No. : 2K22/AFI/24                           #
#     Subject: Machine Learning Lab (Kavinder Sir)     #
#=====================================================#
```

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

In [3]:
```
#Exploring NumPy Library

a = np.array([1,2,3,4,5])
b = np.array([[1,2,3],[4,5,6]])

print(f"a dimensions: {a.ndim}")
print(f"a shape: {a.shape}")
print(f"a length: {len(a)}")

print(f"b dimensions: {b.ndim}")
print(f"b shape: {b.shape}")
print(f"b length: {len(b)}")
```

```
a dimensions: 1
a shape: (5,)
a length: 5
b dimensions: 2
b shape: (2, 3)
b length: 2
```

In [4]:
```
c = np.ones((3,3))
d = np.zeros((3,5))
print(c)
print(d)
```

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

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")]
```

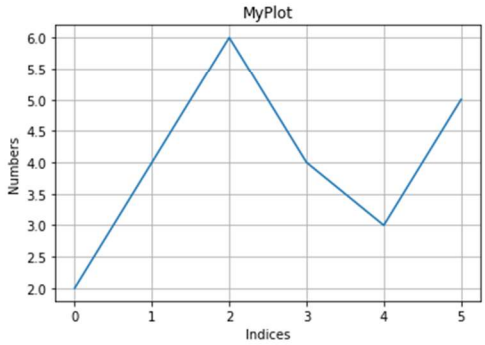
Out[15]:

	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

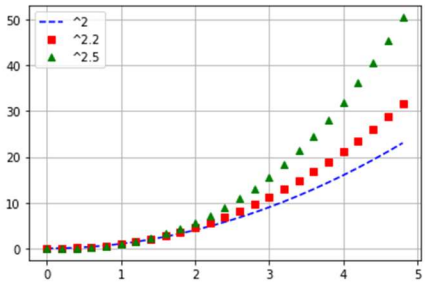
In [16]:

```
#Exploring Matplotlib

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



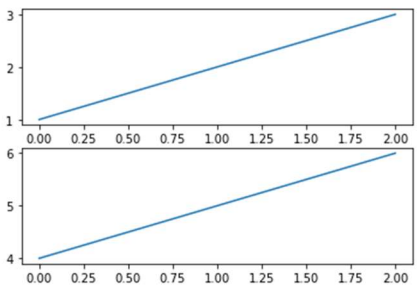
In [17]:

```
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()
```



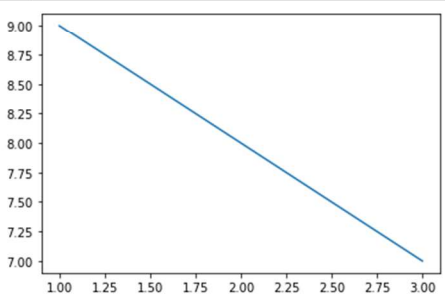
In [18]:

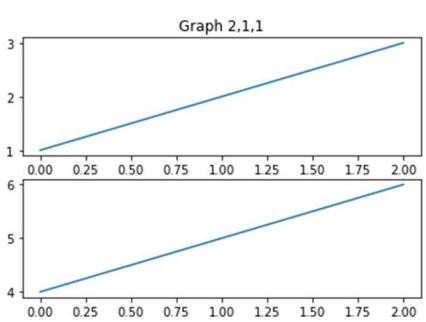
```
plt.figure(1)
plt.subplot(2,1,1)
plt.plot([1,2,3])
plt.subplot(2,1,2)
plt.plot([4,5,6])
plt.show()
```



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: 5 minutes ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

Run Code

In [1]:
```
#=====================================================#
#     LAB 2 - Iris Dataset EDA and Visualisation      #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: Machine Learning Lab (Kavinder 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]:

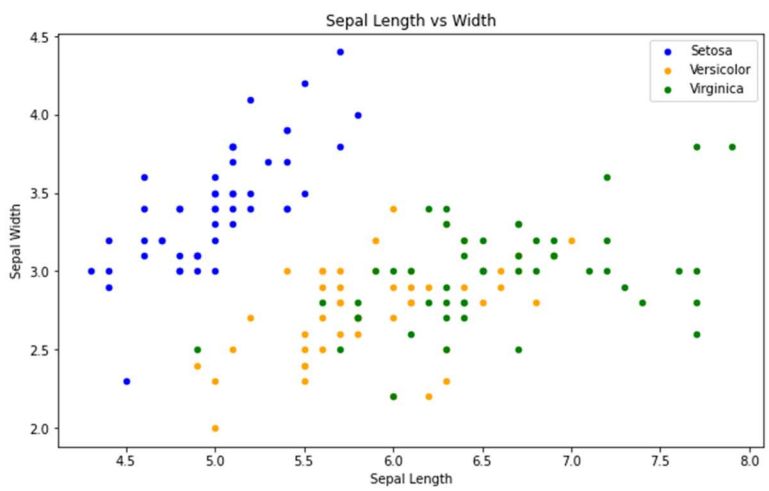
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

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

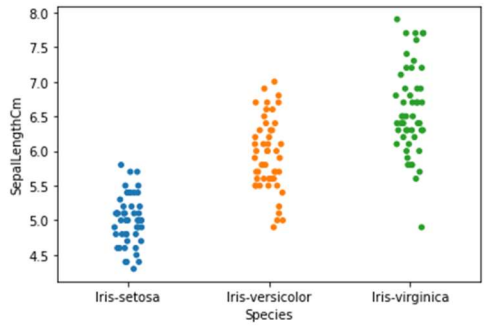
Out[5]:
```
Id               0
SepalLengthCm    0
SepalWidthCm     0
PetalLengthCm    0
PetalWidthCm     0
Species          0
dtype: int64
```

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.
```

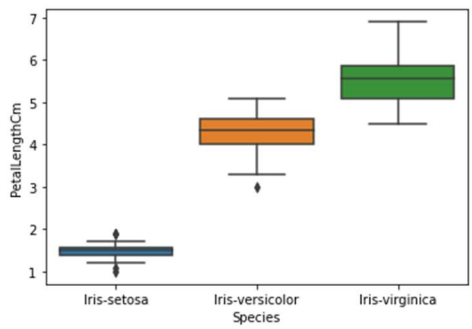
In [7]:
```
#Data Visualisation - Graphical visualisation.

#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()
```



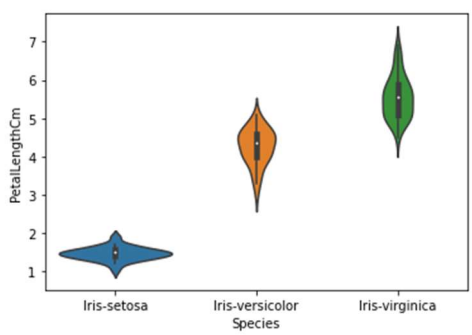
In [8]:
```
# Sepal Length using a Strip plot (Seaborn Library)
sns.stripplot(y ='SepalLengthCm', x = 'Species', data =iris)
plt.show()
```



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()
```

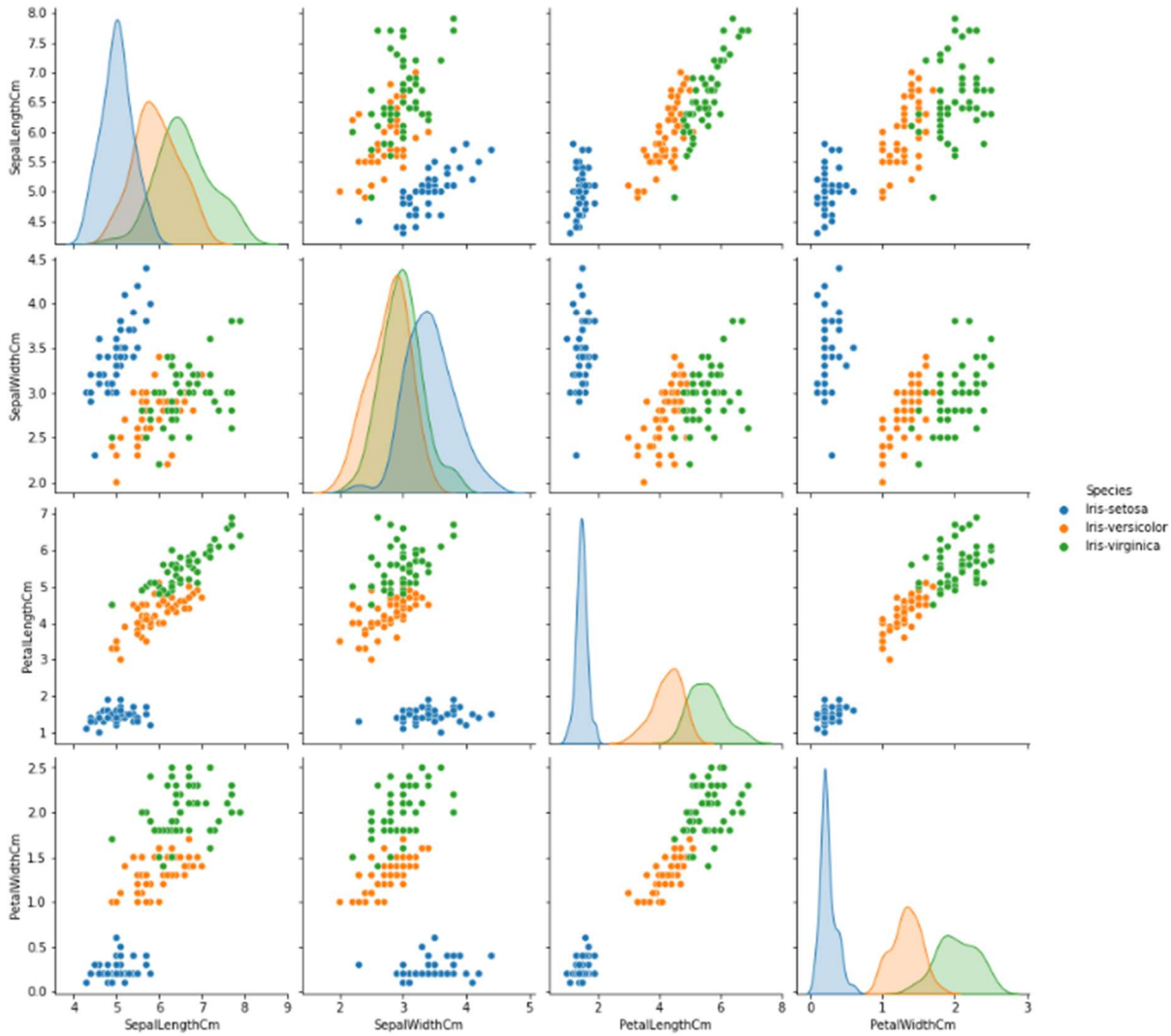


In [11]:
```
iris.columns
```

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 linear regression using gradient descent equations.

THEORY:

- Linear regression is a machine learning algorithm used for predicting a continuous output variable based on one or more input variables. It models the relationship between the input and output variables as a linear equation.
- Gradient descent is an optimization algorithm used for finding the values of the parameters of a model that minimize a cost function. In linear regression, the cost function is typically the mean squared error (MSE).
- 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:

jupyter Lab3_LinearRegression_GradientDescent Last Checkpoint: 8 minutes ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3 (ipykernel)

Run Code

In [1]:

```
#=====================================================#
#     LAB 3 - Linear Regression using Gradient Descent #
#     Author: SHIKHAR ASTHANA                          #
#     Roll No. : 2K22/AFI/24                           #
#     Subject: Machine Learning Lab (Kavinder Sir)     #
#=====================================================#
```

In [2]:

```
#importing the required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [3]:

```
#importing the dataset
dataset = pd.read_csv("advertising.csv")
dataset[["TV","Sales"]].head()
```

Out[3]:

	TV	Sales
0	230.1	22.1
1	44.5	10.4
2	17.2	12.0
3	151.5	16.5
4	180.8	17.9

In [4]:

```
dataset[["TV","Sales"]].shape
```

Out[4]: (200, 2)

In [5]:
```python
#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]:
```python
x[:5]
```

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

In [7]:
```python
#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]:
```python
theta
```

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

In [9]:
```python
#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
```

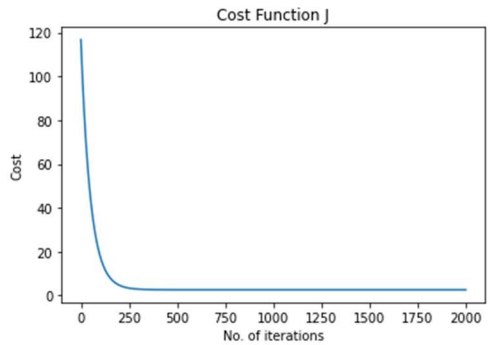
In [10]:
```python
#Actually running the above function on our dataset
past_thetas, past_costs = gradient_descent(x, y, theta, iterations, alpha)
theta = past_thetas[-1]
```

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

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

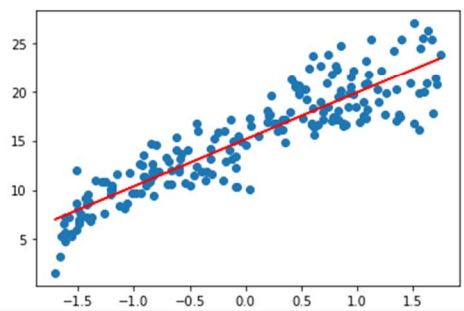
In [12]:

```
#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 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 linear regression using normal equations.

THEORY:

- Linear regression is a machine learning algorithm used for predicting a continuous output variable based on one or more input variables. It models the relationship between the input and output variables as a linear equation.
- Normal equations are a closed-form solution for finding the parameters of a linear regression model that minimize the mean squared error (MSE). Unlike gradient descent, normal equations directly solve for the optimal parameters.
- Normal equations work well when the number of features is not too large. However, they can become computationally expensive when dealing with a large number of features.

CODE & OUTPUT:

jupyter Lab4_LinearRegression_NormalEquation Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Not Trusted | Python 3 (ipykernel)

Run Code

In [1]:
```
#=====================================================#
#     LAB 4 - Linear Regression using Normal Equations #
#     Author: SHIKHAR ASTHANA                          #
#     Roll No. : 2K22/AFI/24                           #
#     Subject: Machine Learning Lab (Kavinder Sir)     #
#=====================================================#
```

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

In [3]:
```
#importing the dataset
dataset = pd.read_csv("advertising.csv")
dataset[["TV","Sales"]].head()
```

Out[3]:

	TV	Sales
0	230.1	22.1
1	44.5	10.4
2	17.2	12.0
3	151.5	16.5
4	180.8	17.9

In [4]:
```
dataset[["TV","Sales"]].shape
```

Out[4]: (200, 2)

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]
```

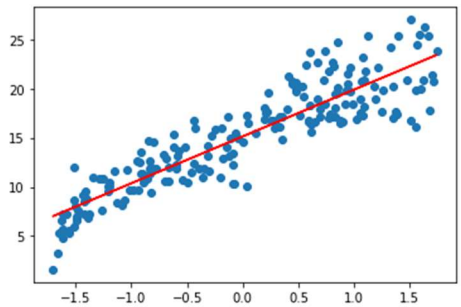
Out[6]:
```
array([[ 1.        ,  0.9674246 ],
       [ 1.        , -1.19437904],
       [ 1.        , -1.51235985],
       [ 1.        ,  0.05191939],
       [ 1.        ,  0.39319551]])
```

In [7]:
```
#Normal Equation is the Closed-form solution for the Linear Regression algorithm
#which means that we can obtain the optimal parameters by just using a formula.

theta_best = np.linalg.inv(x.T.dot(x)).dot(x.T).dot(y)
```

In [8]:
```
theta_best
```

Out[8]:
```
array([15.1305    ,  4.76188551])
```

In [10]:
```
#plot to show final linear regression line
prediction = np.dot(x, theta_best)
plt.scatter(x[:,1],y)
plt.plot(x[:,1],prediction,color="red")
plt.show()
```



LEARNING OUTCOMES:

- Learn how to implement linear regression using normal equations.
- Understand the advantages and disadvantages of normal equations compared to 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 equations.

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:

jupyter Lab5_LogisticRegression_GradientDescent Last Checkpoint: a minute ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

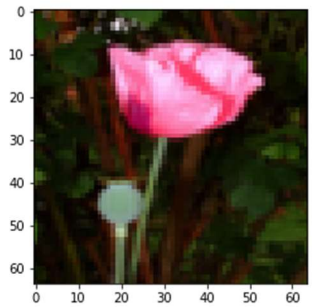
In [1]:
```
#=====================================================#
#     LAB 5 - Logistic Regression - Gradient Descent   #
#     Author: SHIKHAR ASTHANA                          #
#     Roll No. : 2K22/AFI/24                           #
#     Subject: Machine Learning Lab (Kavinder Sir)     #
#=====================================================#
```

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]))
```

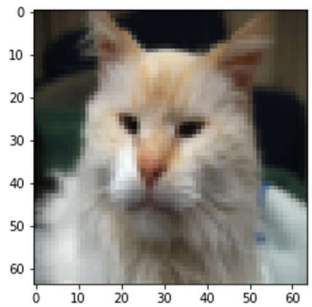
In [10]:
```
#checking a random image
plt.imshow(train_set_x_orig[30])
```

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]:

```python
#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]:

```python
#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]:

```python
#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]:

```python
#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
```

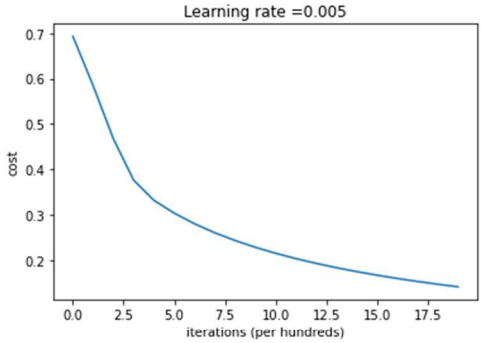
In [21]:

```python
#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]:

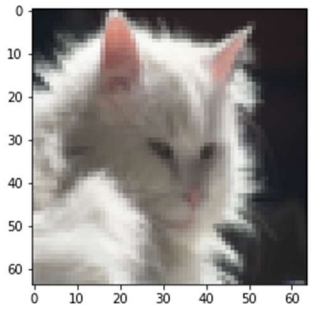
```
# Plot learning curve (with costs)
costs = np.squeeze(logistic_regression_model['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(logistic_regression_model["learning_rate"]))
plt.show()
```



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
```

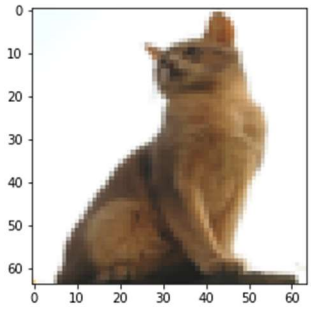
```
y = 1, Prediction: cat
```



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 decision tree using ID3 algorithm.

THEORY:

- Decision trees are a popular machine learning algorithm used for both classification and regression tasks. They consist of a tree-like structure where each internal node represents a decision based on one of the input features, and each leaf node represents a class or regression value.
- The ID3 (Iterative Dichotomiser 3) algorithm is a decision tree algorithm that works by selecting the feature that provides the most information gain (reduces the most uncertainty) at each node. Information gain is calculated using the entropy or the Gini impurity of the data at that node.
- The entropy is a measure of the uncertainty or randomness in the data, and it is calculated as the sum of the negative logarithm of the probabilities of each class.

CODE & OUTPUT:

jupyter Lab6_DecisionTree_ID3 Last Checkpoint: 3 minutes ago (autosaved)

Logout

File Edit View Insert Cell Kernel Help

Not Trusted | Python 3

Run Code

In [1]:

```
#=====================================================#
#     LAB 6 - Decision Tree - ID3                     #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: Machine Learning Lab (Kavinder Sir)    #
#=====================================================#
```

In [2]:

```
#importing the header files
import pandas as pd
import numpy as np
import matplotlib as plt
```

In [3]:

```
#importing the training data using pandas
data=pd.read_csv('PlayTennis.csv')
data.head()
```

Out[3]:

	Outlook	Temperature	Humidity	Wind	Play Tennis
0	Sunny	Hot	High	Weak	No
1	Sunny	Hot	High	Strong	No
2	Overcast	Hot	High	Weak	Yes
3	Rain	Mild	High	Weak	Yes
4	Rain	Cool	Normal	Weak	Yes

In [4]:

```
#importing the test data using pandas
testdata=pd.read_csv('test.csv')
testdata.head()
```

Out[4]:

	Outlook	Temperature	Humidity	Wind	Play Tennis
0	Overcast	Mild	Normal	Weak	Yes
1	Sunny	Mild	Normal	Weak	Yes
2	Rain	Mild	Normal	Weak	Yes
3	Rain	Cool	High	Strong	No

In [5]:

```python
#Calculating the entropy of the system
def calc_total_entropy(train_data, label, class_list):
    total_row = train_data.shape[0]
    total_entr = 0

    for c in class_list:
        total_class_count = train_data[train_data[label] == c].shape[0]
        total_class_entr = - (total_class_count/total_row)*np.log2(total_class_count/total_row)
        total_entr += total_class_entr

    return total_entr
```

In [6]:

```python
#Calculating the entropy of different features
def calc_entropy(feature_value_data, label, class_list):
    class_count = feature_value_data.shape[0]
    entropy = 0

    for c in class_list:
        label_class_count = feature_value_data[feature_value_data[label] == c].shape[0]

        entropy_class = 0
        if label_class_count != 0:
            probability_class = label_class_count/class_count
            entropy_class = - probability_class * np.log2(probability_class)

        entropy += entropy_class

    return entropy
```

In [7]:

```python
#Calculating the overall information gain
def calc_info_gain(feature_name, train_data, label, class_list):
    feature_value_list = train_data[feature_name].unique()
    total_row = train_data.shape[0]
    feature_info = 0.0

    for feature_value in feature_value_list:
        feature_value_data = train_data[train_data[feature_name] == feature_value]
        feature_value_count = feature_value_data.shape[0]
        feature_value_entropy = calc_entropy(feature_value_data, label, class_list)
        feature_value_probability = feature_value_count/total_row
        feature_info += feature_value_probability * feature_value_entropy

    return calc_total_entropy(train_data, label, class_list) - feature_info
```

In [8]:

```python
#Selecting max information gain feature
def find_most_informative_feature(train_data, label, class_list):
    feature_list = train_data.columns.drop(label)
    max_info_gain = -1
    max_info_feature = None

    for feature in feature_list:
        feature_info_gain = calc_info_gain(feature, train_data, label, class_list)
        if max_info_gain < feature_info_gain:
            max_info_gain = feature_info_gain
            max_info_feature = feature

    return max_info_feature
```

In [9]:
```python
#Subdivide the tree based on a specific feature
def generate_sub_tree(feature_name, train_data, label, class_list):
    feature_value_count_dict = train_data[feature_name].value_counts(sort=False)
    tree = {}

    for feature_value, count in feature_value_count_dict.iteritems():
        feature_value_data = train_data[train_data[feature_name] == feature_value]

        assigned_to_node = False
        for c in class_list:
            class_count = feature_value_data[feature_value_data[label] == c].shape[0]

            if class_count == count:
                tree[feature_value] = c
                train_data = train_data[train_data[feature_name] != feature_value]
                assigned_to_node = True
        if not assigned_to_node:
            tree[feature_value] = "?"

    return tree, train_data
```

In [10]:
```python
def make_tree(root, prev_feature_value, train_data, label, class_list):
    if train_data.shape[0] != 0:
        max_info_feature = find_most_informative_feature(train_data, label, class_list)
        tree, train_data = generate_sub_tree(max_info_feature, train_data, label, class_list)
        next_root = None

        if prev_feature_value != None:
            root[prev_feature_value] = dict()
            root[prev_feature_value][max_info_feature] = tree
            next_root = root[prev_feature_value][max_info_feature]
        else:
            root[max_info_feature] = tree
            next_root = root[max_info_feature]

        for node, branch in list(next_root.items()):
            if branch == "?":
                feature_value_data = train_data[train_data[max_info_feature] == node]
                make_tree(next_root, node, feature_value_data, label, class_list)
```

In [11]:
```python
def id3(data, label):
    train_data = data.copy()
    tree = {}
    class_list = train_data[label].unique()
    make_tree(tree, None, train_data, label, class_list)
    return tree
```

In [12]:
```python
def predict(tree, instance):
    if not isinstance(tree, dict):
        return tree
    else:
        root_node = next(iter(tree))
        feature_value = instance[root_node]
        if feature_value in tree[root_node]:
            return predict(tree[root_node][feature_value], instance)
        else:
            return None
```

In [13]:
```python
def evaluate(tree, testdata, label):
    correct_preditct = 0
    wrong_preditct = 0
    for index, row in testdata.iterrows():
        result = predict(tree, data.iloc[index])
        if result == testdata[label].iloc[index]:
            correct_preditct += 1
        else:
            wrong_preditct += 1
    accuracy = correct_preditct / (correct_preditct + wrong_preditct)

    return accuracy*100
```

In [14]:
```python
tree = id3(data, 'Play Tennis')
tree
```

Out[14]:
```
{'Outlook': {'Rain': {'Wind': {'Weak': 'Yes', 'Strong': 'No'}},
  'Overcast': 'Yes',
  'Sunny': {'Humidity': {'Normal': 'Yes', 'High': 'No'}}}}
```

In [15]:
```python
accuracy = evaluate(tree, data, 'Play Tennis')
print("accuracy:", accuracy)
```

```
accuracy: 100.0
```

LEARNING OUTCOMES:

- Understanding the basics of decision trees and how they work, including the role of internal nodes and leaf nodes in the tree structure.
- Familiarity with the ID3 algorithm and how it selects the best feature to split the data at each internal node.
- Ability to implement a decision tree using Python and the NumPy library, including the use of methods to represent the nodes and perform the splitting and prediction steps.

EXPERIMENT 7

AIM: To implement decision tree using C4.5 algorithm.

THEORY: The C4.5 algorithm is a decision tree learning algorithm that is an extension of the ID3 algorithm. The main difference is that C4.5 uses a statistical approach to handle continuous and discrete attributes, while ID3 only handles discrete attributes. The algorithm builds a decision tree from a set of training data using a top-down, greedy approach.

The main steps of the C4.5 algorithm are as follows:

- Calculate the entropy of the target variable (i.e., the class labels) in the dataset.
- For each attribute in the dataset, calculate the information gain ratio, which is a measure of how much the attribute reduces the entropy of the dataset when used to split it.
- Choose the attribute with the highest information gain ratio as the splitting attribute for the current node.
- Recursively apply the above steps to each subset of the data created by splitting on the chosen attribute, until all the data in a subset belongs to the same class or no further splitting is possible.

CODE & OUTPUT:

jupyter Lab7_DecisionTree_C4.5 Last Checkpoint: Last Monday at 10:10 AM (unsaved changes)

Logout

File Edit View Insert Cell Kernel Help Not Trusted Python 3

Run Code

In [1]:

```
#=====================================================#
#     LAB 7 - Decision Tree - C4.5                    #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: Machine Learning Lab (Kavinder Sir)    #
#=====================================================#
```

In [2]:

```
#importing the header files
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [3]:

```
#importing the dataset
df=pd.read_csv('PlayTennis.csv')
df
```

Out[3]:

	Outlook	Temperature	Humidity	Wind	Play Tennis
0	Sunny	Hot	High	Weak	No
1	Sunny	Hot	High	Strong	No
2	Overcast	Hot	High	Weak	Yes
3	Rain	Mild	High	Weak	Yes
4	Rain	Cool	Normal	Weak	Yes
5	Rain	Cool	Normal	Strong	No
6	Overcast	Cool	Normal	Strong	Yes
7	Sunny	Mild	High	Weak	No
8	Sunny	Cool	Normal	Weak	Yes
9	Rain	Mild	Normal	Weak	Yes
10	Sunny	Mild	Normal	Strong	Yes
11	Overcast	Mild	High	Strong	Yes
12	Overcast	Hot	Normal	Weak	Yes
13	Rain	Mild	High	Strong	No

In [4]:
```python
class DecisionTree:
    def __init__(self):
        self.value=""
        self.nodes={}
    def __str__(self):
        return self.value+f' {self.nodes}'
```

In [5]:
```python
def calc_entropy(column):
    total=len(column)
    category=np.array(column.value_counts())
    entropy=0
    for i in category:
        entropy-=(i/total)*np.log2(i/total)
    return entropy
```

In [6]:
```python
def calc_gain_ratio(dataframe,attribute,target_entropy,target):
    column=dataframe[attribute]
    n=len(column)
    values=column.value_counts().tolist()
    classes=column.value_counts().keys().tolist()
    entropy_of_class=[]
    for i in classes:
        col=dataframe[column==i][target]
        entropy_of_class.append(calc_entropy(col))
    ig=target_entropy
    category_entropy={}
    split_info=0
    for i in range(len(values)):
        ig-=(entropy_of_class[i]*(values[i]/n))
        split_info-=(values[i]/n)*np.log2(values[i]/n)
        category_entropy[classes[i]]=entropy_of_class[i]
    return ig/split_info,category_entropy
```

In [7]:
```python
def build_decisionTree(tree,dataframe,target,target_entropy):
    attributes=np.array(dataframe.columns[:-1])
    max_gr=0;
    child_node={}
    parent_node=""

    for attribute in attributes:
        gain_ratio,children=calc_gain_ratio(dataframe,attribute,target_entropy,target)
        if(gain_ratio>max_gr):
            max_gr=gain_ratio
            child_node=children
            parent_node=attribute
    tree.value=parent_node
    for i in list(child_node.keys()):
        child_tree=DecisionTree()
        temp_df=dataframe[dataframe[parent_node]==i]
        new_df=temp_df.drop([parent_node],axis=1)
        if(child_node[i]==0):
            tree.nodes[i]=temp_df[target].unique()[0]
        else:
            tree.nodes[i]=child_tree
            build_decisionTree(child_tree,new_df,target,child_node[i])
```

In [8]:
```python
tree=DecisionTree()
target='Play Tennis'
target_entropy=calc_entropy(df[target])
build_decisionTree(tree,df,target,target_entropy)
```

In [9]:
```python
print(tree)
```

```
Outlook {'Sunny': <__main__.DecisionTree object at 0x000001985343DAC8>, 'Rain': <__main__.DecisionTree object at 0x000001985345
F1D0>, 'Overcast': 'Yes'}
```

In [10]:
```python
print(tree.nodes['Sunny'])
```

```
Humidity {'High': 'No', 'Normal': 'Yes'}
```

In [11]:
```python
print(tree.nodes['Rain'])
```

```
Wind {'Weak': 'Yes', 'Strong': 'No'}
```

LEARNING OUTCOMES:

- Understanding the basics of decision trees and how they work, including the role of internal nodes and leaf nodes in the tree structure.
- Familiarity with the C4.5 algorithm and how it selects the best feature to split the data at each internal node.
- Ability to implement a decision tree using Python and the NumPy library, including the use of methods to represent the nodes and perform the splitting and prediction steps.

EXPERIMENT 8

AIM: To implement decision tree using CART algorithm.

THEORY: The CART (Classification and Regression Trees) algorithm is a decision tree learning algorithm that can be used for both classification and regression tasks. The algorithm builds a decision tree from a set of training data using a top-down, greedy approach.

The main steps of the CART algorithm are as follows:

- Select the best attribute to split the data based on a criterion such as the Gini index or entropy.
- Split the data into two subsets based on the value of the selected attribute.
- Recursively apply steps 1 and 2 to each subset until a stopping criterion is met, such as when all instances in a subset belong to the same class or when the depth of the tree exceeds a specified maximum depth.

For classification tasks, the CART algorithm uses the Gini index as the splitting criterion, which measures the impurity of a set of instances by calculating the probability of misclassifying a random instance from the set. For regression tasks, the CART algorithm uses the mean squared error (MSE) as the splitting criterion, which measures the variance of the target variable within each subset.

CODE & OUTPUT:

jupyter Lab8_DecisionTree_CART Last Checkpoint: Last Monday at 10:10 AM (unsaved changes)

Logout

File Edit View Insert Cell Kernel Help

Not Trusted | Python 3

Run Code

In [1]:

```
#=====================================================#
#     LAB 8 - Decision Tree - CART                    #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: Machine Learning Lab (Kavinder Sir)    #
#=====================================================#
```

In [2]:

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

In [3]:

```
df=pd.read_csv('PlayTennis.csv')
df
```

Out[3]:

	Outlook	Temperature	Humidity	Wind	Play Tennis
0	Sunny	Hot	High	Weak	No
1	Sunny	Hot	High	Strong	No
2	Overcast	Hot	High	Weak	Yes
3	Rain	Mild	High	Weak	Yes
4	Rain	Cool	Normal	Weak	Yes
5	Rain	Cool	Normal	Strong	No
6	Overcast	Cool	Normal	Strong	Yes
7	Sunny	Mild	High	Weak	No
8	Sunny	Cool	Normal	Weak	Yes
9	Rain	Mild	Normal	Weak	Yes
10	Sunny	Mild	Normal	Strong	Yes
11	Overcast	Mild	High	Strong	Yes
12	Overcast	Hot	Normal	Weak	Yes
13	Rain	Mild	High	Strong	No

In [4]:
```python
class DecisionTree:
    def __init__(self):
        self.value=""
        self.nodes={}
    def __str__(self):
        return self.value+f' {self.nodes}'
```

In [5]:
```python
def calc_impurity(column):
    total=len(column)
    category=np.array(column.value_counts())
    impurity=1
    for i in category:
        impurity-=(i/total)**2
    return impurity
```

In [6]:
```python
def calc_gini_impurity(dataframe,attribute,target):
    column=dataframe[attribute]
    n=len(column)
    values=column.value_counts().tolist()
    classes=column.value_counts().keys().tolist()
    impurity_of_class=[]
    for i in classes:
        col=dataframe[column==i][target]
        impurity_of_class.append(calc_impurity(col))
    gini=0
    category_impurity={}
    for i in range(len(values)):
        gini+=(impurity_of_class[i]*(values[i]/n))
        category_impurity[classes[i]]=impurity_of_class[i]
    return gini,category_impurity
```

In [7]:
```python
def build_decisionTree(tree,dataframe,target):
    attributes=np.array(dataframe.columns[:-1])
    min_gini=10000;
    child_node={}
    parent_node=""

    for attribute in attributes:
        gini,children=calc_gini_impurity(dataframe,attribute,target)
        if(gini<min_gini):
            min_gini=gini
            child_node=children
            parent_node=attribute
    tree.value=parent_node
    for i in list(child_node.keys()):
        child_tree=DecisionTree()
        temp_df=dataframe[dataframe[parent_node]==i]
        new_df=temp_df.drop([parent_node],axis=1)
        if(child_node[i]==0):
            tree.nodes[i]=temp_df[target].unique()[0]
        else:
            tree.nodes[i]=child_tree
            build_decisionTree(child_tree,new_df,target)
```

In [8]:
```python
tree=DecisionTree()
target='Play Tennis'
build_decisionTree(tree,df,target)
```

In [9]:
```python
print(tree)
```

```
Outlook {'Rain': <__main__.DecisionTree object at 0x0000028E1DA6E780>, 'Sunny': <__main__.DecisionTree object at 0x0000028E1DA6
E8D0>, 'Overcast': 'Yes'}
```

In [10]:
```python
print(tree.nodes['Sunny'])
```

```
Humidity {'High': 'No', 'Normal': 'Yes'}
```

In [11]:
```python
print(tree.nodes['Rain'])
```

```
Wind {'Weak': 'Yes', 'Strong': 'No'}
```

LEARNING OUTCOMES:

- Understanding the basics of decision trees and how they work, including the role of internal nodes and leaf nodes in the tree structure.
- Familiarity with the CART algorithm and how it selects the best feature to split the data at each internal node.
- Ability to implement a decision tree using Python and the NumPy library, including the use of methods to represent the nodes and perform the splitting and prediction steps.

EXPERIMENT 9

AIM: To implement data classification using SVM.

THEORY: Support Vector Machines (SVM) is a supervised machine learning algorithm that can be used for classification or regression tasks. SVM works by finding the hyperplane that best separates the different classes of data. The distance between the hyperplane and the closest data points from each class is known as the margin, and the goal of SVM is to maximize this margin.

The main steps of the SVM algorithm are as follows:

- Given a labeled training dataset, SVM finds the hyperplane that separates the different classes of data with the largest margin.
- If the data is not linearly separable, SVM uses a kernel function to transform the data into a higher-dimensional space where it is linearly separable.
- SVM selects the hyperplane that maximizes the margin and minimizes the training error.
- SVM can also handle non-binary classification tasks using multi-class SVM, which constructs multiple hyperplanes to separate different classes.

SVM has several advantages over other classification algorithms, including its ability to handle high-dimensional data and its effectiveness in dealing with noisy and overlapping data.

CODE & OUTPUT:

jupyter Lab9_SVM Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Help Not Trusted Python 3

Run Code

In [ ]:

```
#=====================================================#
#     LAB 9 - Support Vector Machine                  #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: Machine Learning Lab (Kavinder Sir)    #
#=====================================================#
```

In [ ]:

```
#importing the header files
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
```

In [1]:

```
#defining the SVM clas
class SVM:

    #initialising funciton - assign default values
    def __init__(self, learning_rate=0.001, lambda_param=0.01, n_iters=1000):
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    #apply gradient descent for SVM
    def fit(self, X, y):
        n_samples, n_features = X.shape

        y_ = np.where(y <= 0, -1, 1)

        # init weights
        self.w = np.zeros(n_features)
        self.b = 0

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y_[idx] * (np.dot(x_i, self.w) - self.b) >= 1
                if condition:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (2 * self.lambda_param * self.w - np.dot(x_i, y_[idx]))
                    self.b -= self.lr * y_[idx]


    #perform prediciton
    def predict(self, X):
        approx = np.dot(X, self.w) - self.b
        return np.sign(approx)
```

In [2]:

```python
#load the iris dataset from predefined library
iris = datasets.load_iris()
X = iris.data[:100,:2]
y = iris.target[:100]
y[y==0] = -1 # Convert 0 labels to -1

# Shuffle iris dataset data
np.random.seed(123)
shuffle_idx = np.random.permutation(len(y))
X = X[shuffle_idx]
y = y[shuffle_idx]

# Split iris dataset into training and test sets
train_size = 0.8
split_idx = int(len(y) * train_size)
X_train = X[:split_idx]
X_test = X[split_idx:]
y_train = y[:split_idx]
y_test = y[split_idx:]

#actually initialising the SVM class we created
clf = SVM()
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)
```

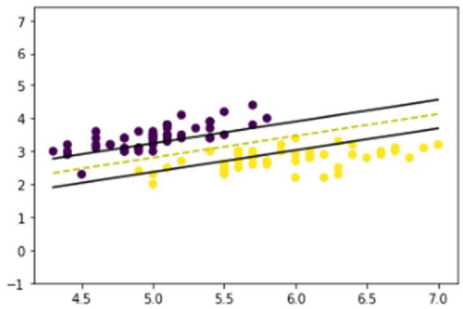
In [3]:

```python
#defining function to calculate the accuracy
#Number of correctly predicted/total
def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return accuracy

print("SVM classification accuracy", accuracy(y_test, predictions))
```

```
SVM classification accuracy 1.0
```

In [4]:

```python
#function to create a graph to visualise the SVM
def visualize_svm():
        def get_hyperplane_value(x, w, b, offset):
            return (-w[0] * x + b + offset) / w[1]

        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        plt.scatter(X[:, 0], X[:, 1], marker="o", c=y)

        x0_1 = np.amin(X[:, 0])
        x0_2 = np.amax(X[:, 0])

        x1_1 = get_hyperplane_value(x0_1, clf.w, clf.b, 0)
        x1_2 = get_hyperplane_value(x0_2, clf.w, clf.b, 0)

        x1_1_m = get_hyperplane_value(x0_1, clf.w, clf.b, -1)
        x1_2_m = get_hyperplane_value(x0_2, clf.w, clf.b, -1)

        x1_1_p = get_hyperplane_value(x0_1, clf.w, clf.b, 1)
        x1_2_p = get_hyperplane_value(x0_2, clf.w, clf.b, 1)

        ax.plot([x0_1, x0_2], [x1_1, x1_2], "y--")
        ax.plot([x0_1, x0_2], [x1_1_m, x1_2_m], "k")
        ax.plot([x0_1, x0_2], [x1_1_p, x1_2_p], "k")

        x1_min = np.amin(X[:, 1])
        x1_max = np.amax(X[:, 1])
        ax.set_ylim([x1_min - 3, x1_max + 3])

        plt.show()

    visualize_svm()
```



LEARNING OUTCOMES:

- Understanding the basic concepts of supervised machine learning and classification tasks.
- Familiarity with the SVM algorithm and its use for binary and multi-class classification tasks.
- Ability to implement SVM using Python and the scikit-learn library, including how to use SVM classes and methods to train and evaluate models.
- Understanding of how to evaluate the performance of an SVM model using metrics such as accuracy, precision, recall, and F1 score.
- Familiarity with techniques for improving the performance of SVM models, such as kernel selection, hyperparameter tuning, and regularization.

EXPERIMENT 10

AIM: To implement data classification using KNN.

THEORY: K-Nearest Neighbors (KNN) is a supervised machine learning algorithm that can be used for classification or regression tasks. KNN works by finding the k nearest neighbors of a given data point in the training dataset, and classifying the data point based on the majority class of its neighbors.

The main steps of the KNN algorithm are as follows:

- Given a labeled training dataset, KNN stores all data points and their corresponding class labels.
- Given a new data point, KNN finds the k nearest neighbors of the data point in the training dataset based on a distance metric such as Euclidean distance or Manhattan distance.
- KNN classifies the new data point based on the majority class of its k nearest neighbors. In case of a tie, the class of the nearest neighbor is used as the prediction.
- KNN can also handle regression tasks by predicting the average or median value of the target variable for the k nearest neighbors.

KNN has several advantages over other classification algorithms, including its simplicity and effectiveness in handling non-linear data.

CODE & OUTPUT:

jupyter Lab10_KNN Last Checkpoint: Last Monday at 10:14 AM (unsaved changes)

File Edit View Insert Cell Kernel Help

In [2]:
```
#=====================================================#
#     LAB 10 - K Nearest Neighbours                   #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: Machine Learning Lab (Kavinder Sir)    #
#=====================================================#
```

In [3]:
```
#Import header files
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from collections import Counter
```

In [4]:
```
#Importing the dataset
df=pd.read_csv('iris.csv')

df.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]:
```
df.shape
```

Out[5]: (150, 6)

In [6]:
```python
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 [7]:
```python
df.describe()
```

Out[7]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	75.500000	5.843333	3.054000	3.758667	1.198667
std	43.445368	0.828066	0.433594	1.764420	0.763161
min	1.000000	4.300000	2.000000	1.000000	0.100000
25%	38.250000	5.100000	2.800000	1.600000	0.300000
50%	75.500000	5.800000	3.000000	4.350000	1.300000
75%	112.750000	6.400000	3.300000	5.100000	1.800000
max	150.000000	7.900000	4.400000	6.900000	2.500000

In [8]:
```python
df['Species'].value_counts()
```

Out[8]:
```
Iris-versicolor    50
Iris-virginica     50
Iris-setosa        50
Name: Species, dtype: int64
```

In [9]:
```python
#Jumbling the dataset
shuffle_df = df.sample(frac=1)

#dividing the dataset into test and train sets
train_size = int(0.7 * len(df))

train_set = shuffle_df[:train_size]
test_set = shuffle_df[train_size:]
```

In [10]:
```python
rows, col = train_set.shape
rows ,col
```

Out[10]: (105, 6)

In [11]:
```python
#Creating the map for output variable
mapping = {
    'Iris-setosa' : 1,
    'Iris-versicolor' : 2,
    'Iris-virginica' : 3
}

#Removing the ID column and applying the output variable transformation
X_train = train_set.drop(['Id', 'Species'], axis=1).values # Input Feature Values
Y_train = train_set.Species.replace(mapping).values.reshape(rows,1) # Output values

#test data
X_test = test_set.drop(['Id', 'Species'], axis=1).values # Input Feature Values
Y_test = test_set.Species.replace(mapping).values.reshape(45,1) # Output values
```

In [12]:
```python
#colour map for visualisation in graph
cmap = ListedColormap(['#FF0000','#00FF00','#0000FF'])
```

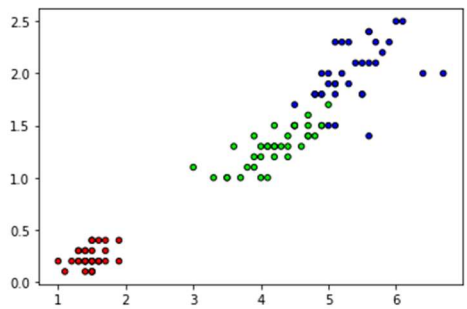
In [13]:
```python
#Funtion to calculate the euclidean distance
def euclidean_distance(x1, x2):
    distance = np.sqrt(np.sum((x1-x2)**2))
    return distance
```

In [14]:

```python
#Class for KNN
class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        predictions = [self._predict(x) for x in X]
        return predictions

    def _predict(self, x):
        # compute the distance
        distances = [euclidean_distance(x, x_train) for x_train in self.X_train]

        # get the closest k
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]

        # majority vote
        most_common = Counter(tuple(label) for label in k_nearest_labels).most_common()
        return most_common[0][0]
```

In [15]:

```python
#Visualising the dataset
plt.figure()
plt.scatter(X_train[:,2],X_train[:,3], c=Y_train, cmap=cmap, edgecolor='k', s=20)
plt.show()
```



In [19]:

```python
#Actually initialising the KNN class
clf = KNN(k=5)
clf.fit(X_train, Y_train)
predictions = clf.predict(X_test)

#Printing the predictions
print(predictions)

#printing the actuals
#print(Y_test)
```

```
[(3,), (2,), (1,), (3,), (3,), (3,), (1,), (3,), (2,), (2,), (2,), (2,), (1,), (1,), (1,), (1,), (1,), (1,), (3,), (2,), (3,),
(2,), (1,), (3,), (1,), (1,), (1,), (2,), (3,), (3,), (1,), (2,), (1,), (3,), (3,), (2,), (3,), (3,), (3,), (3,), (3,), (1,),
(2,), (1,), (1,)]
```

In [20]:

```python
#Calculating the accuracy
acc = np.sum(predictions == Y_test) / len(Y_test)
print(acc)
```

```
0.9777777777777777
```

LEARNING OUTCOMES:

- Understanding the basic concepts of supervised machine learning and classification tasks.
- Familiarity with the KNN algorithm and its use for classification and regression tasks.
- Ability to implement KNN using Python and the scikit-learn library, including how to use KNN classes and methods to train and evaluate models.
- Understanding of how to evaluate the performance of a KNN model using metrics such as accuracy, precision, recall, and F1 score (for classification tasks) or MSE and R-squared (for regression tasks).
- Familiarity with techniques for improving the performance of KNN models, such as selecting the optimal value of k, choosing an appropriate distance metric, and handling imbalanced data.

EXPERIMENT 11

AIM: To implement K Means Clustering Algorithm.

THEORY: K-Means is an unsupervised machine learning algorithm used to cluster data points into groups (called clusters) based on their similarity. The algorithm works by iteratively assigning each data point to the nearest cluster centroid and then updating the centroid based on the mean of the data points in the cluster. The process continues until the cluster centroids no longer change significantly.

To implement K-Means using Numpy and Pandas, we can follow these steps:

- Load the data into a Pandas DataFrame.
- Normalize the data by subtracting the mean and dividing by the standard deviation.
- Randomly initialize K cluster centroids.
- Assign each data point to the nearest centroid using the Euclidean distance metric.
- Update each centroid based on the mean of the data points in the cluster.
- Repeat steps 4 and 5 until the cluster centroids no longer change significantly or until a maximum number of iterations is reached.

CODE & OUTPUT:

jupyter Lab11_KMeans_Mall_Customers Last Checkpoint: an hour ago (autosaved)

Logout

File Edit View Insert Cell Kernel Help Trusted Python 3

Run Code

In [1]:

```
#=====================================================#
#     LAB 11 - K Means                                #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: Machine Learning Lab (Kavinder Sir)    #
#=====================================================#
```

In [2]:

```
#Import header files
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [3]:

```
data = pd.read_csv("Mall_Customers.csv")
data.head()
```

Out[3]:

	CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

In [4]:

```python
#Data Preprocessing
gender_map = {
    "Male":0,
    "Female":1
}
data["Gender"] = data["Genre"].map(gender_map)

#Dropping unecessary columns
data = data.drop(["CustomerID","Genre"], axis = 1)

data.head()
```

Out[4]:

	Age	Annual Income (k$)	Spending Score (1-100)	Gender
0	19	15	39	0
1	21	15	81	0
2	20	16	6	1
3	23	16	77	1
4	31	17	40	1

In [5]:

```python
def k_means(data, k, max_iter=100):
    # Normalize the data
    data_norm = (data - data.mean()) / data.std()

    # Initialize centroids randomly
    centroids = data_norm.sample(n=k)

    # Iterate until convergence or max number of iterations
    for i in range(max_iter):
        # Assign each data point to the nearest centroid
        distances = pd.DataFrame()
        for m in range(centroids.shape[0]):
            distances[f"{m}"] = np.sqrt(((data_norm - centroids.iloc[m])**2).sum(axis=1))
        distances = distances.iloc[:,:].values
        labels = np.argmin(distances, axis=1)

        # Update each centroid based on the mean of the data points in the cluster
        for j in range(k):
            centroids.iloc[j] = data_norm[labels == j].mean(axis=0)

    return labels, centroids
```

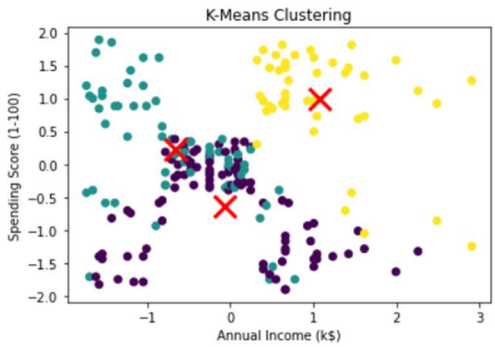
In [6]:

```python
# Run K-Means clustering with 3 clusters
labels, centroids = k_means(data, 3)

# Print the labels and centroids
print(labels)
print(centroids)
```

```
[1 1 1 1 1 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 1 1 0 0 0 0
 0 1 0 0 1 0 0 1 0 0 1 0 0 1 1 0 0 1 0 0 1 1 0 1 0 1 1 0 0 1 0 1 0 0 0 0 0
 1 0 1 1 1 0 0 0 0 1 0 2 2 1 2 0 2 0 2 0 2 1 2 1 2 0 2 1 2 0 2 1 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 1 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 2 2 0 2 2
 2 0 2 0 2 2 2 0 2 0 2 2 2 2 2]
          Age  Annual Income (k$)  Spending Score (1-100)    Gender
160  0.873716           -0.070274               -0.644782 -0.065364
1   -0.969110           -0.662033                0.225812  0.067821
159 -0.388559            1.083667                0.982739  0.035725
```

In [7]:

```
# Visualize the data points and centroids
data_norm = (data - data.mean()) / data.std()



fig, ax = plt.subplots()
ax.scatter(data_norm['Annual Income (k$)'], data_norm['Spending Score (1-100)'], c=labels)
ax.scatter(centroids['Annual Income (k$)'], centroids['Spending Score (1-100)'], marker='x', s=300, linewidths=3, color='r')
ax.set_xlabel('Annual Income (k$)')
ax.set_ylabel('Spending Score (1-100)')
ax.set_title('K-Means Clustering')
plt.show()
```



LEARNING OUTCOMES:

- Understanding of the K-Means clustering algorithm and its use in unsupervised machine learning.
- Knowledge of the steps involved in implementing K-Means using Numpy and Pandas.
- Ability to preprocess data for clustering by normalizing it using Pandas.
- Ability to implement the K-Means algorithm using Numpy and Pandas in Python.
- Understanding of how to determine the optimal number of clusters for a dataset using techniques such as the elbow method or silhouette analysis.

EXPERIMENT 12

AIM: To implement 2-layer Neural Network.

THEORY: A 2-layer Neural Network is a type of neural network architecture that consists of an input layer, one or more hidden layers, and an output layer. In this aim, we will be implementing a 2-layer neural network using Numpy and pandas to classify the MNIST Digit dataset. The MNIST Digit dataset is a dataset of handwritten digits, where each image is a 28x28 grayscale image of a digit between 0 and 9.

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:

- Preprocess the data: normalize the pixel values, and split the data into training and testing sets.
- Define the neural network architecture: specify the number and size of the layers, the activation functions, and the loss function.
- 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.
- Evaluate the network: feed the testing data through the network and calculate the accuracy of the predictions.

CODE & OUTPUT:

jupyter Lab12_2LayerNN_MNIST_Digit Last Checkpoint: a few seconds ago (autosaved)

File Edit View Insert Cell Kernel Help

In [1]:
```
#=====================================================#
#     LAB 12 - 2-layer Neural Network                 #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: Machine Learning Lab (Kavinder Sir)    #
#=====================================================#
```

In [2]:
```
#importing headers
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd
```

In [3]:
```
# Load the dataset
digits = load_digits()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size=0.2, random_state=42)

# Normalize the data
X_train = X_train / 255.0
X_test = X_test / 255.0
```

In [4]:

```python
# Convert the target variable to one-hot encoding
y_train = pd.get_dummies(y_train).values
y_test = pd.get_dummies(y_test).values

# Define the sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

In [5]:

```python
# Initialize the weights and biases randomly
np.random.seed(42)
weights1 = np.random.randn(X_train.shape[1], 128)
bias1 = np.zeros((1, 128))
weights2 = np.random.randn(128, 10)
bias2 = np.zeros((1, 10))

# Set the learning rate and number of epochs
learning_rate = 0.5
epochs = 1000
```

In [6]:

```python
# Train the model
for i in range(epochs):
    # Forward propagation
    layer1 = sigmoid(np.dot(X_train, weights1) + bias1)
    layer2 = sigmoid(np.dot(layer1, weights2) + bias2)

    # Calculate the error and delta for layer 2
    error = y_train - layer2
    delta_layer2 = error * layer2 * (1 - layer2)

    # Calculate the delta for layer 1
    delta_layer1 = np.dot(delta_layer2, weights2.T) * layer1 * (1 - layer1)

    # Update the weights and biases
    weights2 += learning_rate * np.dot(layer1.T, delta_layer2)
    bias2 += learning_rate * np.sum(delta_layer2, axis=0, keepdims=True)
    weights1 += learning_rate * np.dot(X_train.T, delta_layer1)
    bias1 += learning_rate * np.sum(delta_layer1, axis=0, keepdims=True)
    if i%100 == 0:
        print(f"Epoch {i} completed")
```

```
Epoch 0 completed
Epoch 100 completed
Epoch 200 completed
Epoch 300 completed
Epoch 400 completed
Epoch 500 completed
Epoch 600 completed
Epoch 700 completed
Epoch 800 completed
Epoch 900 completed
```

In [7]:

```python
# Predict on the test set
layer1 = sigmoid(np.dot(X_test, weights1) + bias1)
predictions = np.argmax(sigmoid(np.dot(layer1, weights2) + bias2), axis=1)

# Calculate the accuracy
accuracy = np.mean(predictions == np.argmax(y_test, axis=1))
print("Accuracy:", accuracy)
```

```
Accuracy: 0.09444444444444444
```

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.
- Implementing a 2-layer Neural Network using Numpy and pandas
- Preprocessing a dataset for a Neural Network
- Training a Neural Network using backpropagation
- Evaluating the accuracy.

EXPERIMENT 13

AIM: To implement Convolution operation of CNN.

THEORY: A convolutional neural network (CNN) is a type of deep learning model that is particularly effective at processing and analyzing images. One of the key components of a CNN is the convolutional operation, which involves taking a small matrix of weights called a kernel or filter and sliding it over the input image to extract features.

The aim of this experiment is to implement the convolution operation of a CNN. Numpy is a Python library that provides support for large, multi-dimensional arrays and matrices, making it well-suited for performing mathematical operations on image data.

The convolution operation involves taking a small kernel or filter, such as a 3x3 or 5x5 matrix, and sliding it over the input image, computing the dot product of the kernel and the corresponding input pixel values at each location. This produces a new output feature map that summarizes the presence of certain features in the input image.

To implement the convolution operation using numpy, we will need to perform the following steps:

- Load an input image and a kernel/filter.
- Flip the kernel horizontally and vertically.
- Pad the input image with zeros to ensure that the output feature map has the same dimensions as the input image.
- Slide the kernel over the padded input image and compute the dot product at each location to generate the output feature map.

CODE & OUTPUT:

jupyter Lab13_ConvolutionLayer Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Help Trusted Python 3

Run Code

In [1]:
```
#=====================================================#
#     LAB 13 - Implement Covolution Layer             #
#     Author: SHIKHAR ASTHANA                         #
#     Roll No. : 2K22/AFI/24                          #
#     Subject: Machine Learning Lab (Kavinder Sir)    #
#=====================================================#
```

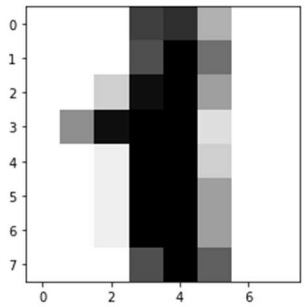
In [2]:
```
#importing headers
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
```

In [3]:
```
# Load the dataset
digits = load_digits()
```

In [4]:
```
#Selecting any random image to perform Convolution on.
input_image = digits["images"][1]
```

In [5]:
```
plt.imshow(input_image, cmap=plt.cm.binary)
```

Out[5]: <matplotlib.image.AxesImage at 0x1f32d4b5630>



In [6]:
```
#To apply convolution layer, we must first define the following:
# 1 - Kernel shape and values.
# 2 - Stride
# 3 - Padding

#1 - Let us define the kernel shape and size
kernel = np.array([[1, 0, -1],
                   [2, 0, -2],
                   [1, 0, -1]])
#Above is a vertical edge detector kernel.
#It can be changed to different types of kernels to see different fmaps.

#2 - Let us keep the stride = 1
stride = 1

#3 - Let us define the padding we want to keep
padding_size = 1
```

In [7]:
```
input_image
```

Out[7]:
```
array([[ 0.,  0.,  0., 12., 13.,  5.,  0.,  0.],
       [ 0.,  0.,  0., 11., 16.,  9.,  0.,  0.],
       [ 0.,  0.,  3., 15., 16.,  6.,  0.,  0.],
       [ 0.,  7., 15., 16., 16.,  2.,  0.,  0.],
       [ 0.,  0.,  1., 16., 16.,  3.,  0.,  0.],
       [ 0.,  0.,  1., 16., 16.,  6.,  0.,  0.],
       [ 0.,  0.,  1., 16., 16.,  6.,  0.,  0.],
       [ 0.,  0.,  0., 11., 16., 10.,  0.,  0.]])
```

In [8]:
```
pad_image = np.pad(input_image, padding_size, mode='constant')
pad_image
```

Out[8]:
```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0., 12., 13.,  5.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0., 11., 16.,  9.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  3., 15., 16.,  6.,  0.,  0.,  0.],
       [ 0.,  0.,  7., 15., 16., 16.,  2.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1., 16., 16.,  3.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1., 16., 16.,  6.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1., 16., 16.,  6.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0., 11., 16., 10.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

In [9]:
```
#Before applying convolution, we need to initialise the output feature map.
# Calculating the feature map size -> size = ((input - kernel + 2(padding))/stride) + 1
out_h = ((input_image.shape[0] - kernel.shape[0] + 2*padding_size) // stride) + 1
out_w = ((input_image.shape[1] - kernel.shape[1] + 2*padding_size) // stride) + 1
```

In [10]:
```
print("Input_Image", input_image.shape)
print("Pad_Image", pad_image.shape)
print("Output FMap", tuple([out_h,out_w]))
```

```
Input_Image (8, 8)
Pad_Image (10, 10)
Output FMap (8, 8)
```

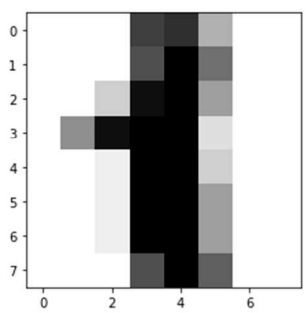
In [11]:
```
#Initialising the output featuremap
f_map = np.zeros((out_h, out_w))
f_map
```

Out[11]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.]])
```

In [12]:
```
#For convolution, we need to put the kernel on top of the padded image starting from the top left corner
#Multiple element wise the pixel values of the padded image with the kernel pixel values
#Then, take sum of each element of the matrix and this will be the value of one cell of the output fmap

rows = pad_image.shape[0]
columns = pad_image.shape[1]

k_rows = kernel.shape[0]
k_columns = kernel.shape[1]


#Here, kernel might need to be flipped in order to keep it mathematically in sync with the 'Convolution' definition.
#If we do not flip the kernel by 180 degree, we are actually applying 'cross-corelation'.
#According to what I could find, some implementations of neural networks flip it while some do not.
#Eg in Theano, some implementations use flipping the kernel while some do not.

#For this implementation, I am not including the flipping of the kernel as it is more intuitive to understand.

for i in range(0, rows-k_rows+1, stride):
    for j in range(0, columns-k_columns+1, stride):
        #print(i,j, " Done")
        f_map[i][j] = np.sum(pad_image[i:i+k_rows, j:j+k_columns] * kernel)

```

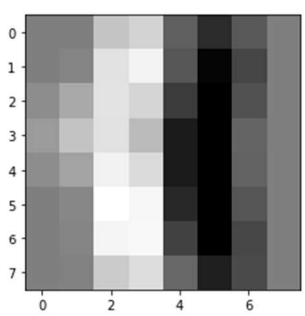
In [13]:
```
plt.imshow(input_image, cmap=plt.cm.binary)
```

Out[13]: <matplotlib.image.AxesImage at 0x1f32d443470>



In [14]:
```
plt.imshow(f_map, cmap=plt.cm.binary)
```

Out[14]: <matplotlib.image.AxesImage at 0x1f32d4f1ef0>



In [15]:
```
#After this, the convolution operation is completed.
#The next steps are -
# 1. Applying the activation function - Eg. ReLU
# 2. Applying Pooling - Eg. Max Pooling

#This will complete 1st convolution layer.
```

LEARNING OUTCOMES:

- Understanding the convolution operation in CNNs.
- Learning how to implement the convolution operation using numpy.
- Gaining practical experience with image processing and manipulation in Python.

EXPERIMENT 14

AIM: To perform MNIST data classification using CNN.

THEORY: A CNN is a type of neural network that is commonly used in image recognition tasks. It consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers. In a convolutional layer, a set of filters is applied to the input image to extract features, while in a pooling layer, the size of the input image is reduced by down-sampling. The fully connected layers are used for classification.

The Digit Recognizer dataset is a popular dataset used in computer vision and machine learning for handwritten digit recognition with its inspiration coming from the MNIST Dataset. It contains 70,000 images of handwritten digits from 0 to 9, with each image being a 28x28 grayscale image. The dataset is divided into a training set of 60,000 images and a test set of 10,000 images.

The dataset is widely used for image classification tasks, where the objective is to train a model to correctly identify the digit in the image. The task is challenging because the images are handwritten and can have variations in writing style, size, and orientation. The Digit Recognizer dataset is often used as a benchmark for testing new computer vision models and algorithms. Many researchers and developers have developed deep learning models using this dataset, achieving high accuracy rates in digit recognition tasks.

TensorFlow is an open-source machine learning framework developed by Google. It provides a collection of tools for building and training machine learning models, including neural networks. TensorFlow allows us to define and train our CNN model efficiently using high-level APIs, such as the Keras API. We can also leverage the power of GPUs to accelerate the training process.

The following steps can provide a high-level overview of the process involved in implementing MNIST data classification using CNN with TensorFlow:

- Preprocess the data: normalize the pixel values, and split the data into training and testing sets.
- Define the neural network architecture: specify the number and size of the layers, the activation functions, and the loss function.
- 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.
- Evaluate the network: feed the testing data through the network and calculate the accuracy of the predictions.

CODE & OUTPUT:

jupyter Lab14_Convolutional_Neural_Network_Final Last Checkpoint: a few seconds ago (autosaved)

File Edit View Insert Cell Kernel Help Not Trusted Python 3

In [1]:
```
#=====================================================#
#     LAB 14 - Implement Covolution Neural Network     #
#     Author: SHIKHAR ASTHANA                          #
#     Roll No. : 2K22/AFI/24                           #
#     Subject: Machine Learning Lab (Kavinder Sir)     #
#=====================================================#
```

In [2]:
```
import pandas as pd
import numpy as np

import keras
from keras.models import Sequential
from keras.layers import Conv2D, Lambda, MaxPooling2D # convolution layers
from keras.layers import Dense, Dropout, Flatten # core layers
from keras.utils.np_utils import to_categorical
from tensorflow.keras.layers import BatchNormalization
```

In [3]:
```
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
```

In [4]:
```
print(f"Training data size is {train.shape}\nTesting data size is {test.shape}")
```

```
Training data size is (42000, 785)
Testing data size is (28000, 784)
```

In [5]:
```
train = train.sample(frac=1).reset_index(drop=True)
```

In [6]:
```
train_size = int(0.85 * len(train))
train_size
```

Out[6]:
```
35700
```

In [7]:
```
train_X = train[:train_size]
test_X = train[train_size:]
```

In [8]:
```
#Train Test Split without using train_test_split()

#Training Data
X_train = train_X.drop(['label'], 1).values
X_train = X_train / 255.0
# Reshape image in 3 dimensions (height = 28px, width = 28px , channel = 1)
# channel = 1 => For gray scale
X_train = X_train.reshape(-1,28,28,1)

y_train = train_X['label'].values
# Encode labels to one hot vectors (ex : 2 -> [0,0,1,0,0,0,0,0,0,0])
y_train = to_categorical(y_train)



#Testing data
X_test = test_X.drop(['label'], 1).values
X_test = X_test / 255.0
# Reshape image in 3 dimensions (height = 28px, width = 28px , channel = 1)
# channel = 1 => For gray scale
X_test = X_test.reshape(-1,28,28,1)

y_test = test_X['label'].values
# Encode labels to one hot vectors (ex : 2 -> [0,0,1,0,0,0,0,0,0,0])
y_test = to_categorical(y_test)
```

In [9]:
```
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[9]:
```
((35700, 28, 28, 1), (6300, 28, 28, 1), (35700, 10), (6300, 10))
```

In [10]:
```
#Normalization
mean = np.mean(X_train)
std = np.std(X_train)

def standardize(x):
    return (x-mean)/std
```

In [11]:

```python
model=Sequential()

#model.add(Lambda(standardize,input_shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Flatten())
model.add(Dense(512,activation="relu"))

model.add(Dense(10,activation="softmax"))

model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
```

In [12]:

```python
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 64)        640

 conv2d_1 (Conv2D)           (None, 24, 24, 64)        36928

 max_pooling2d (MaxPooling2D  (None, 12, 12, 64)       0
 )

 batch_normalization (BatchN  (None, 12, 12, 64)       256
 ormalization)

 conv2d_2 (Conv2D)           (None, 10, 10, 128)       73856

 conv2d_3 (Conv2D)           (None, 8, 8, 128)         147584

 max_pooling2d_1 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 batch_normalization_1 (Batc  (None, 4, 4, 128)        512
 hNormalization)

 conv2d_4 (Conv2D)           (None, 2, 2, 256)         295168

 max_pooling2d_2 (MaxPooling  (None, 1, 1, 256)        0
 2D)

 batch_normalization_2 (Batc  (None, 1, 1, 256)        1024
 hNormalization)

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 512)               131584

 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 692,682
Trainable params: 691,786
Non-trainable params: 896
_________________________________________________________________
```

```
In [13]: history = model.fit(X_train, y_train, epochs=5,batch_size = 128)

Epoch 1/5
279/279 [==============================] - 273s 970ms/step - loss: 0.1068 - accuracy: 0.9668
Epoch 2/5
279/279 [==============================] - 273s 978ms/step - loss: 0.0390 - accuracy: 0.9879
Epoch 3/5
279/279 [==============================] - 270s 967ms/step - loss: 0.0262 - accuracy: 0.9920
Epoch 4/5
279/279 [==============================] - 270s 967ms/step - loss: 0.0163 - accuracy: 0.9946
Epoch 5/5
279/279 [==============================] - 269s 964ms/step - loss: 0.0190 - accuracy: 0.9939

In [14]: model.evaluate(X_test, y_test)

197/197 [==============================] - 12s 57ms/step - loss: 0.0874 - accuracy: 0.9779

Out[14]: [0.0874178558588028, 0.9779365062713623]
```

LEARNING OUTCOMES:

- Understanding of the MNIST dataset, and Digit Recognizer 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.
- Preprocessing a dataset for a Neural Network.
- Training a Neural Network.
- Evaluating the accuracy.