D.Y.Patil International University, Akurdi, Pune

School of Computer Science Engineering and Applications (SCSEA)

Third Year Engineering (B.Tech)

Deep Neural Network (CSE3101)

Lab Manual

D Y PATIL
INTERNATIONAL
UNIVERSITY
AKURDI PUNE

Vision of the University:

"To Create a vibrant learning environment – fostering innovation and creativity, experiential learning, which is inspired by research, and focuses on regionally, nationally and globally relevant areas."

Mission of the School:

To provide a diverse, vibrant and inspirational learning environment.

To establish the university as a leading experiential learning and research oriented center.

To become a responsive university serving the needs of industry and society.

To embed internationalization, employability and value thinking

Deep Neural Network

Course Objectives:

Understand the Mathematics for Neural Networks.

Understanding Functioning of Neural Networks and Kernel Methods.

Understand the Linear Models for Regression and Classification

Understanding Mixture Models.

Course Outcomes:

On completion of the course, learner will be able to–

CO1: Design various types of Neural Networks.

CO2: Construct and Select Kernels for dataset

CO3: Implement Linear models for classification and regression

CO4: Implement Bayesian for different applications and design mixture models.

Program Outcomes:

PO1	**Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7	**Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Rules and Regulations for Laboratory:

- Students should be regular and punctual to all the Lab practical
- Lab assignments and practical should be submitted within given time.
- Mobile phones are strictly prohibited in the Lab.
- Please shut down the Computers before leaving the Lab.
- Please switch off the fans and lights and keep the chair in proper position before leaving the Lab
- Maintain proper discipline in Lab

D.Y.Patil International University, Akurdi, Pune

School of Computer Science Engineering and Applications

Index

Sr. No.	Name of the Practical	Date of Conduction	Page No.		Sign of Teacher	Remarks*
			From	To		
1.	Implement Polynomial Curve fitting using the polyfit function					
2.	Implement a basic single perceptron for regression and classification on XOR and XNOR truth table					
3.	Build a Basic neural network from scratch using a high level library like tensorflow or Pytorch. Use appropriate dataset					
4.	Optimize hyperparameters for a neural network model. Implement a hyperparameter optimization strategy and compare the performance with different hyperparameter configurations for both classification and regression task					
5.	Develop a Python function(Multivariate Optimization) to compute the Hessian matrix for a given scalar-valued function of multiple variables.					
6.	Implement Bayesian methods for classification					
7.	Implement Principal component analysis for dimensionality reduction of datapoints					
8.	Implement hidden Markov Model for sequence prediction and evaluate model performance					

***Absent/Attended/Late/Partially Completed/Completed**

CERTIFICATE

This is to certify that Mr. /Miss _ PRN: _ _ _ _ _ _ _ _ _ _ _ _ _ of class: _ _ _ _ _ _ _ _ _ _ _ _ _ has completed practical/term work in the course of Deep Neural Network of Third Year Engineering (B.Tech) within SCSEA, as prescribed by D.Y.Patil International University, Pune during the academic year 2023 - 2024.

Date: **Teaching Assistant** **Faculty I/C** **Director** (SCSEA)

Practical 01

Student Name: utkarsh chauhan
Date of Experiment: 12/01/24
Date of Submission:
PRN No:20210802019

Aim: Implement Polynomial Curve fitting using the polyfit function

Objectives:

1. Generate a synthetic dataset with a polynomial relationship.
2. Add noise to the dataset.
3. Use the polyfit function to fit a polynomial curve to the data.
4. Visualize the original data and the fitted curve.

Software/Tool: Python/Jupyter/Anaconda/Colab

Theory:

Polynomial curve fitting:

Polynomial curve fitting is a mathematical technique used to approximate a relationship between two variables using a polynomial function. It has various applications in fields such as statistics, engineering, physics, economics, and more. It is a very valuable tool in data analysis.

some common applications of polynomial curve fitting:

Data Modeling, Interpolation and Extrapolation, Function Approximation Signal Processing, Regression Analysis, Scientific Research.

A Simple Regression Problem

• We observe a real-valued input variable x and we wish to use this observation to predict the value of a real-valued target variable t.

• We use synthetically generated data from the function $\sin(2\pi x)$ with random noise included in the target values. – A small level of random noise having a Gaussian distribution

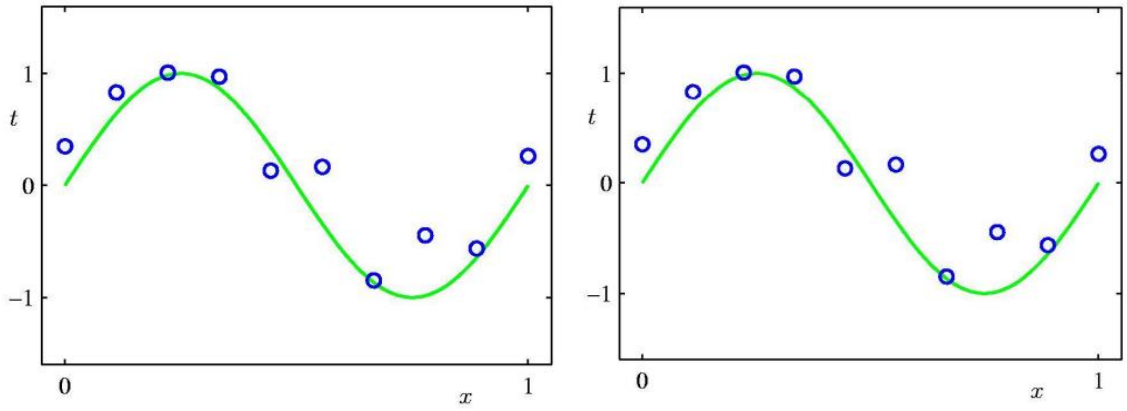
• We have a training set comprising N observations of x, written $x \equiv (x1 , \ldots , xN)^T$, together with corresponding observations of the values of t, denoted $t \equiv (t1 , \ldots , tN)^T$.

• Our goal is to predict the value of t for some new value of x,

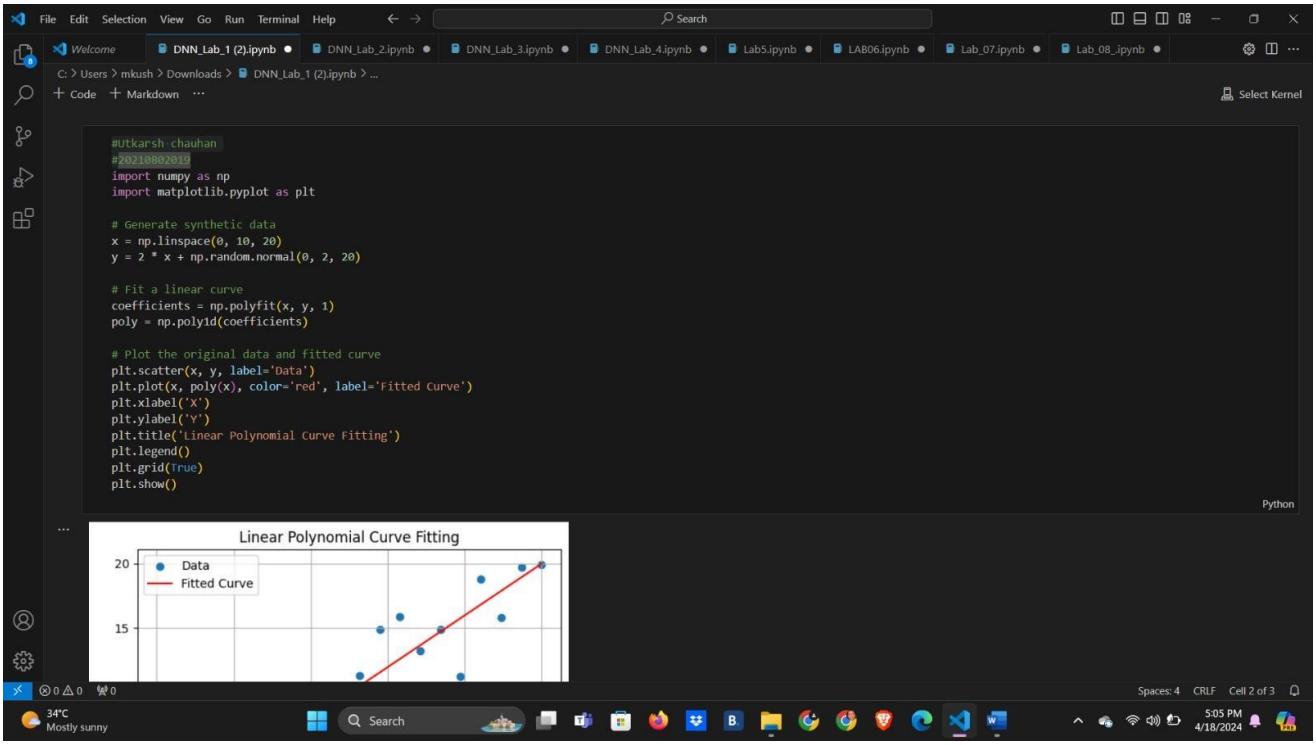
A training data set of N = 10 points, (blue circles),

• The green curve shows the actual function $\sin(2\pi x)$ used to generate the data.

• Our goal is to predict the value of t for some new value of x, without knowledge of the green curve



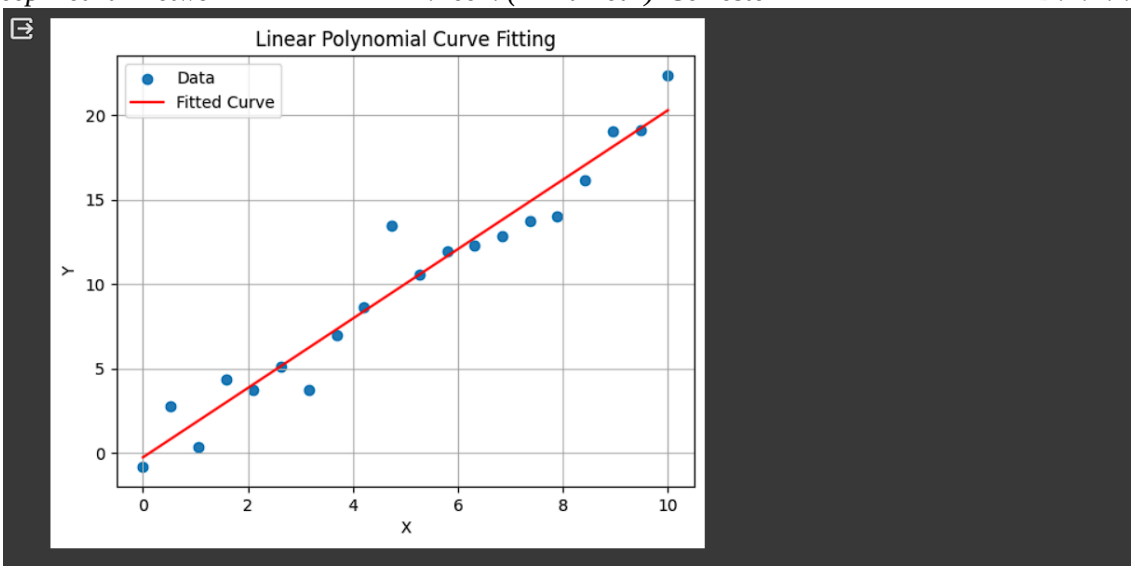
We try to fit the data using a polynomial function of the form



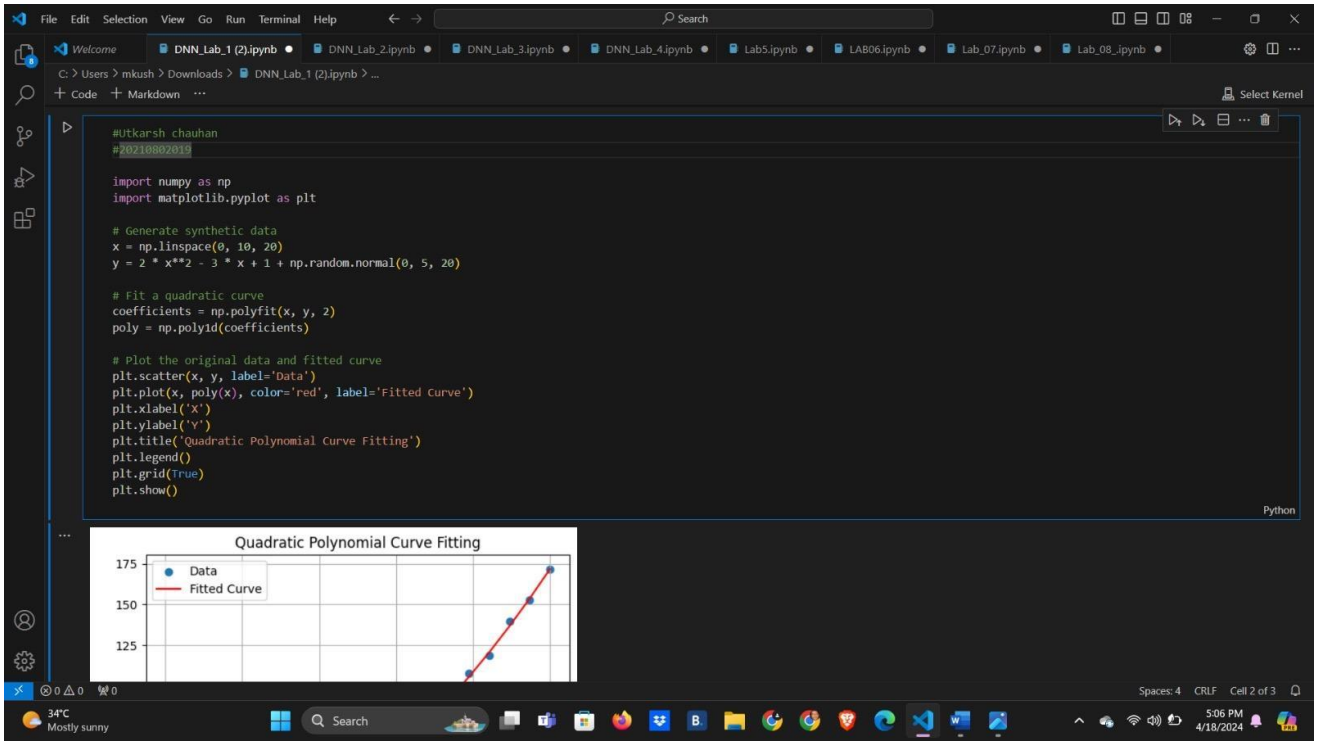
$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

Attachment: Algorithm, Program code, Results and output
Linear Polynomial Curve Fitting:
Input:

Output:

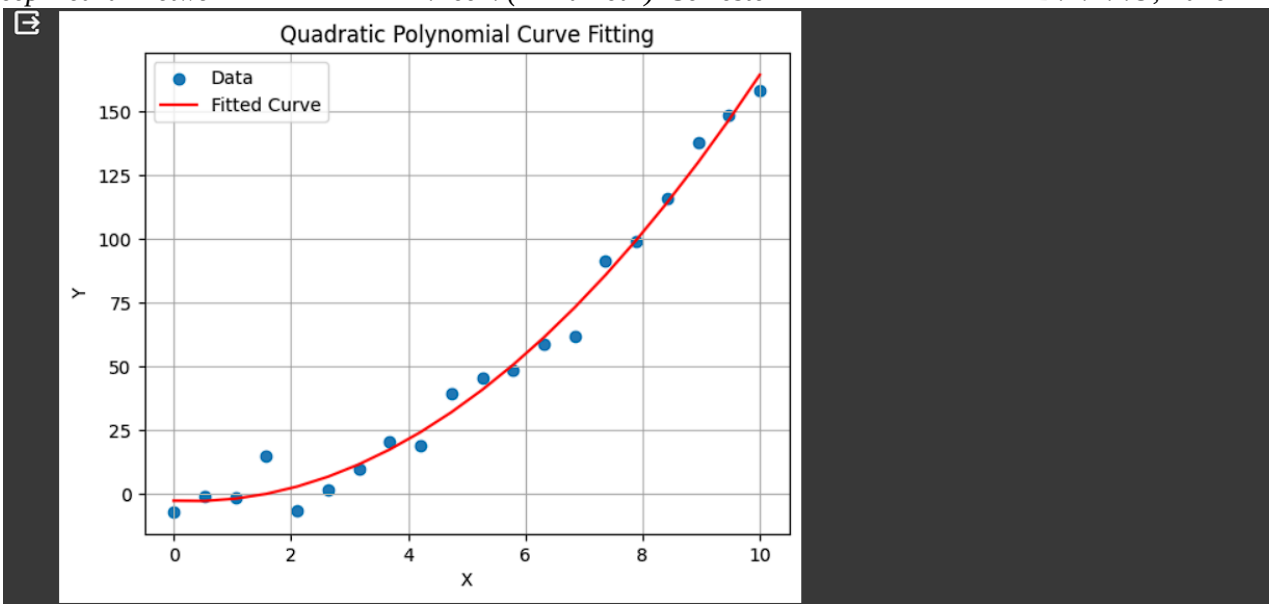


Quadratic Polynomial Curve Fitting
Input:

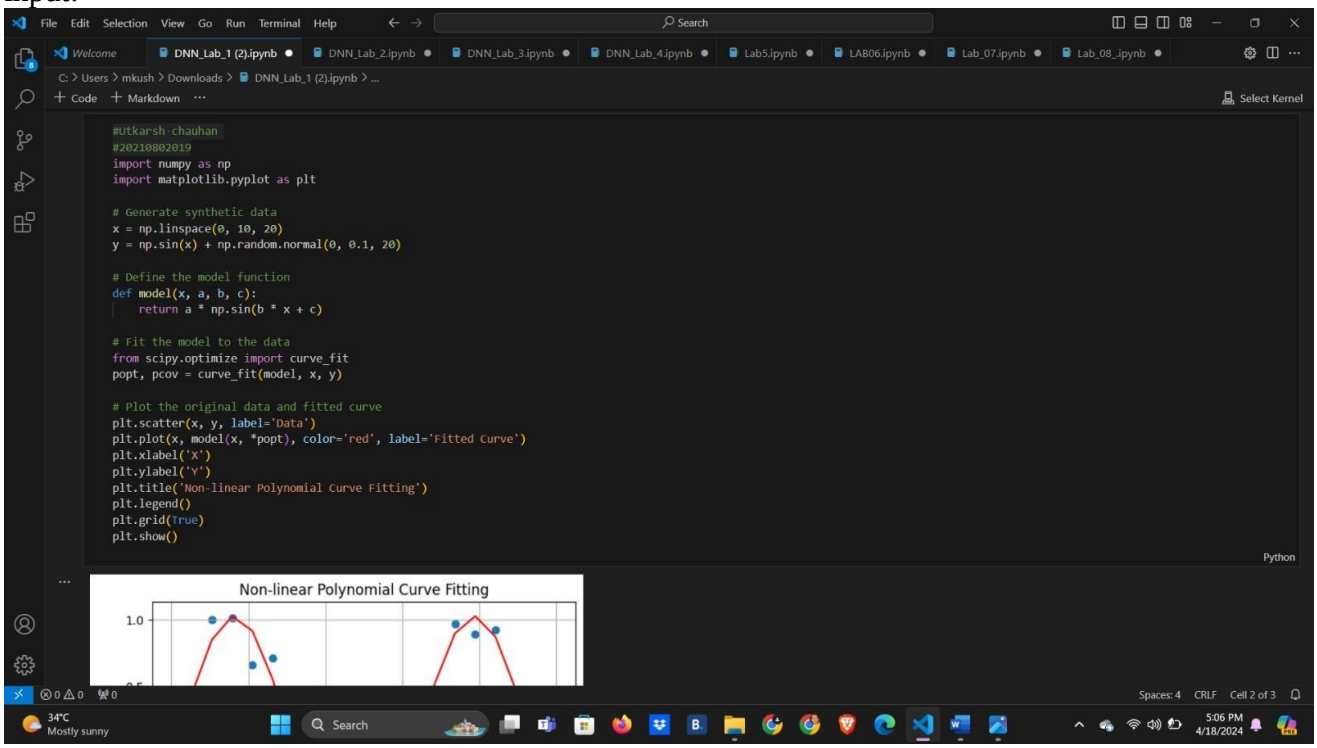


```
#Utkarsh chauhan
#20210802019

import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic data
x = np.linspace(0, 10, 20)
y = 2 * x**2 - 3 * x + 1 + np.random.normal(0, 5, 20)

# Fit a quadratic curve
coefficients = np.polyfit(x, y, 2)
poly = np.poly1d(coefficients)

# Plot the original data and fitted curve
plt.scatter(x, y, label='Data')
plt.plot(x, poly(x), color='red', label='Fitted Curve')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Quadratic Polynomial Curve Fitting')
plt.legend()
plt.grid(True)
plt.show()
```

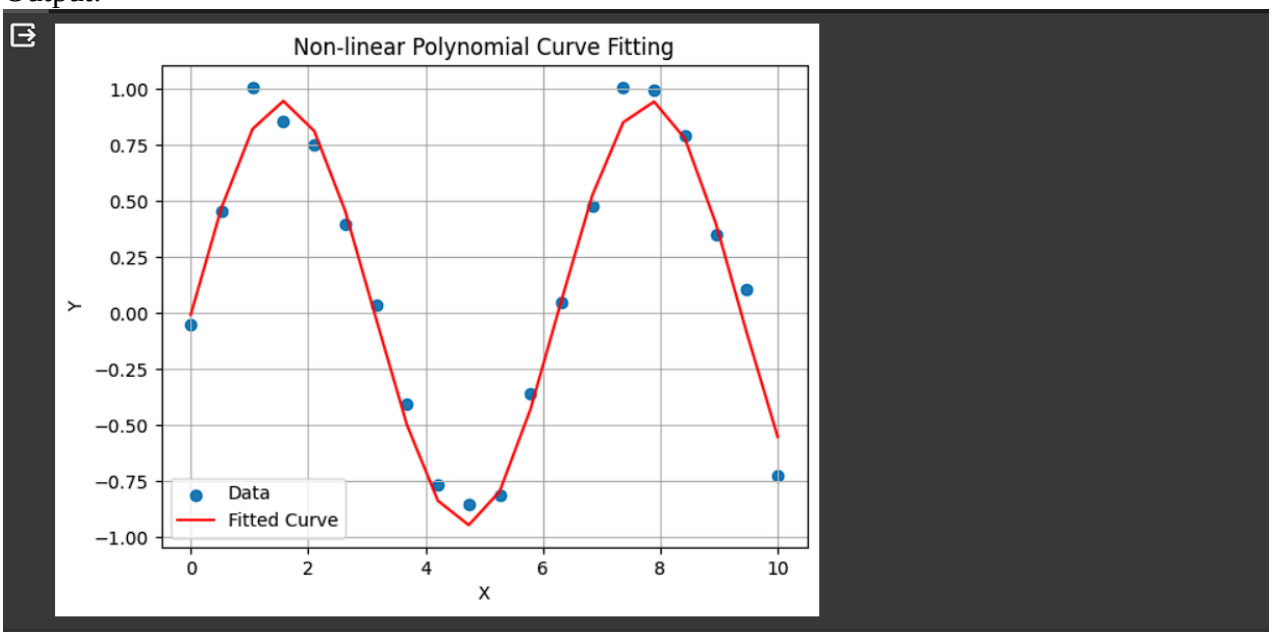

Output:



Non-Linear Polynomial Curve Fitting
input:



Output:



Conclusion: Polynomial curve fitting using the polyfit function successfully models synthetic data with added noise, visualizing the original data and the fitted curve.

Practical 02

Student Name: Utkarsh chauhan
Date of Experiment: 19/01/2024
Date of Submission:
PRN No:20210802019

Aim: Implement a basic single perceptron for regression and classification on XOR and XNOR truth table

Objectives:

Single Layer Perceptron

1. Define a simple dataset with linearly separable classes.
2. Implement a perceptron with a step activation function.
3. Train the perceptron using the perceptron learning algorithm.
4. Evaluate the model on the dataset.

Multi-Layer Perceptron

1. Define a simple dataset (e.g., XOR problem).
2. Implement a multilayer perceptron architecture.
3. Train the network using backpropagation.
4. Evaluate the model on the dataset.

Software/Tool: Python/Jupyter/Anaconda/Colab

Theory :

A single layer perceptron (SLP) is a feed-forward network based on a threshold transfer function. SLP is the simplest type of artificial neural networks and can only classify linearly separable cases with a binary target (1 , 0). The single layer perceptron does not have a priori knowledge, so the initial weights are assigned randomly. SLP sums all the weighted inputs and if the sum is above the threshold (some predetermined value), SLP is said to be activated (output=1).

Output

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n > \theta \Rightarrow 1$$

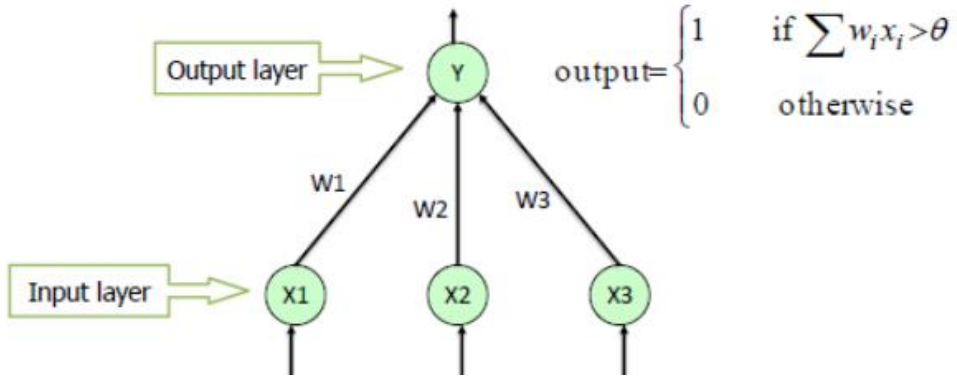
$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n \leq \theta \Rightarrow 0$$

$$\Delta w = \eta \times d \times x$$

$d \rightarrow$ Predicted output - Desired output

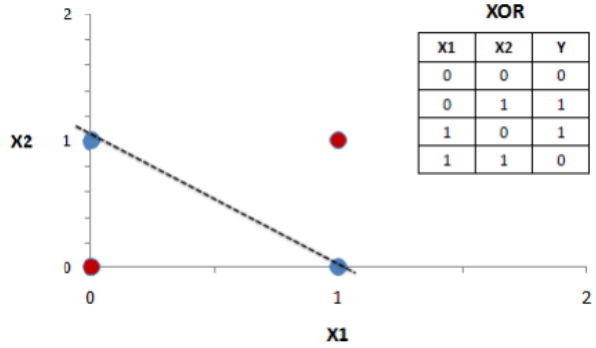
$\eta \rightarrow$ Learning rate, usually less than 1

$x \rightarrow$ Input data



The input values are presented to the perceptron, and if the predicted output is the same as the desired output, then the performance is considered satisfactory and no changes to the weights are made. However, if the output does not match the desired output, then the weights need to be changed to reduce the error.

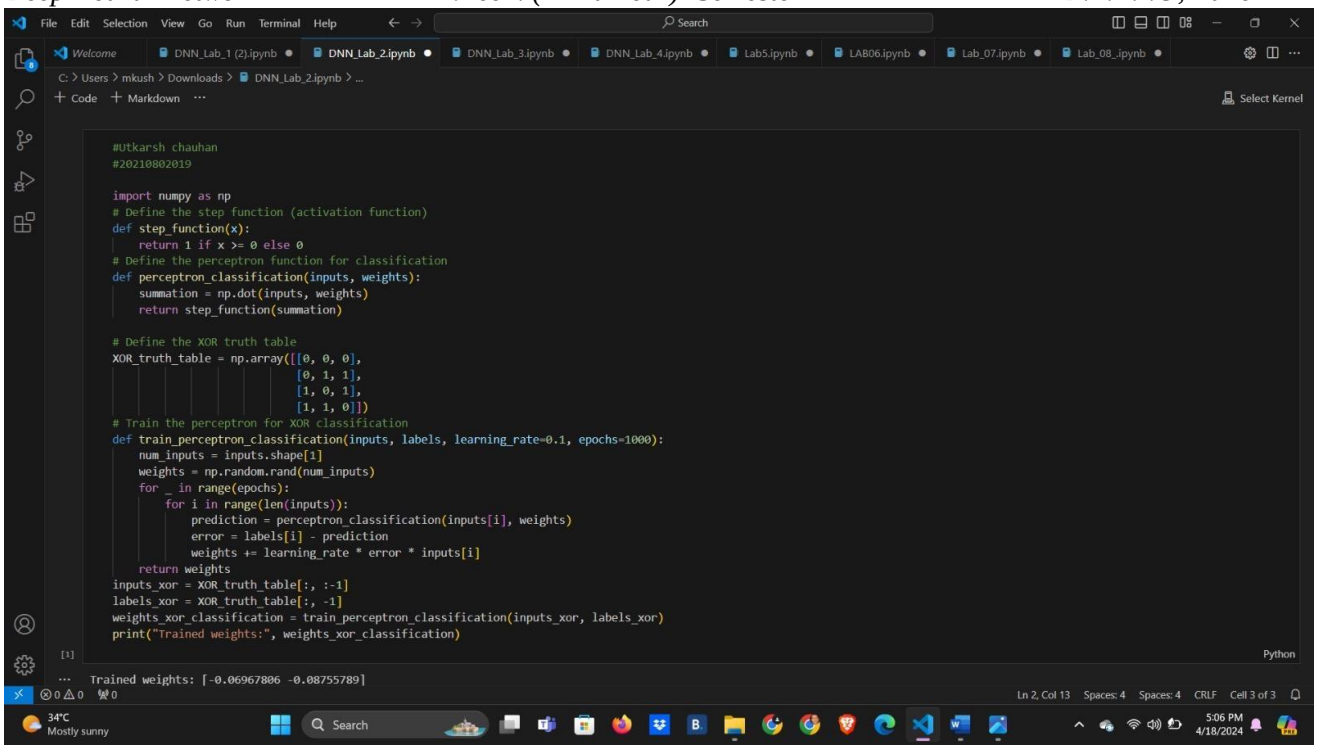
The most famous example of the inability of perceptron to solve problems with linearly non-separable cases is the XOR problem. However, a multi-layer perceptron using the backpropagation algorithm can successfully classify the XOR data. A multi-layer perceptron (MLP) has the same structure of a single layer perceptron with one or more hidden layers.



The backpropagation algorithm consists of two phases:
a) the forward phase where the activations are propagated from the input to the output layer, and b) the backward phase, where the error between the observed actual and the requested nominal value in the output layer is propagated backwards in order to modify the weights and bias values.

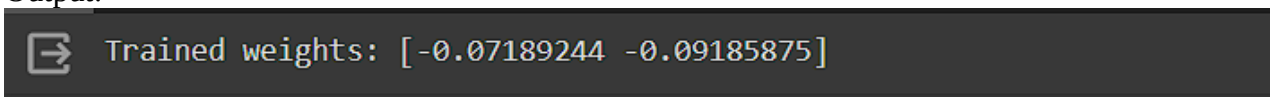
Attachment: Algorithm, Program code, Results and output

Input:



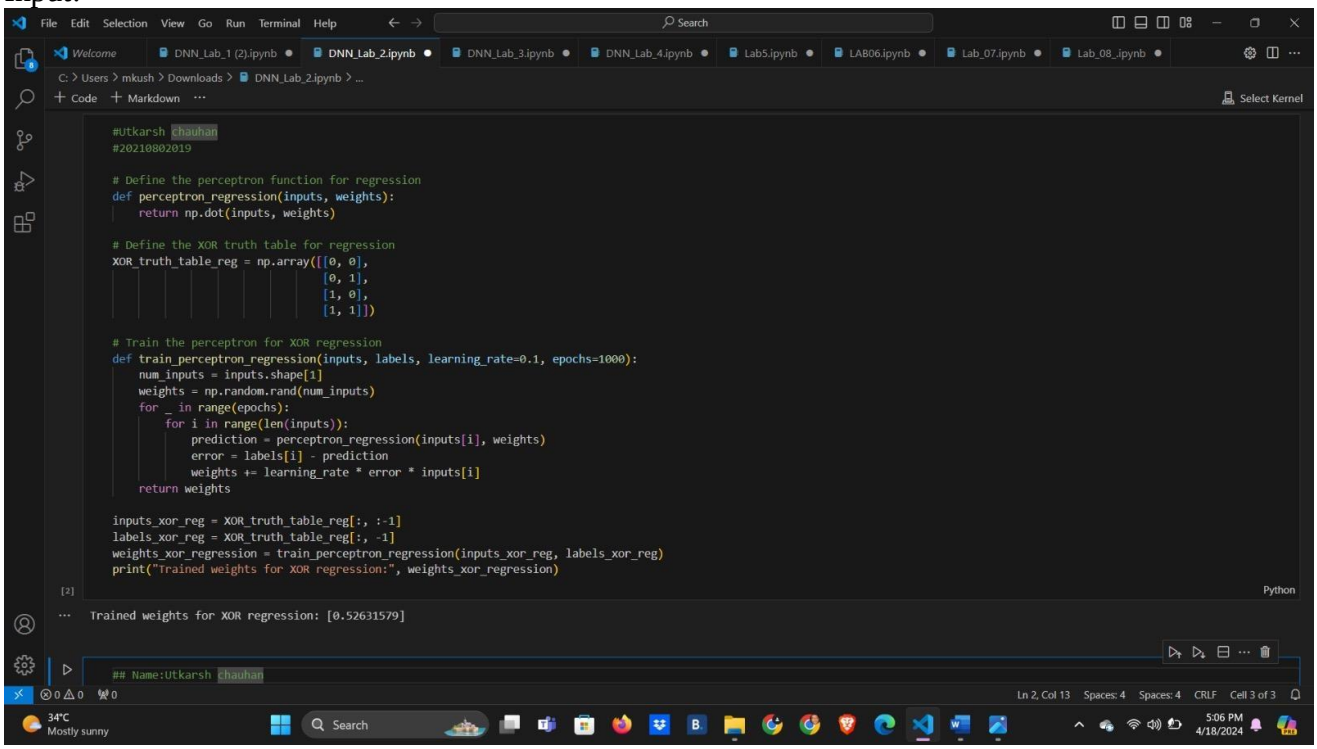
```python
#Utkarsh chauhan
#20210802019

import numpy as np
# Define the step function (activation function)
def step_function(x):
    return 1 if x >= 0 else 0
# Define the perceptron function for classification
def perceptron_classification(inputs, weights):
    summation = np.dot(inputs, weights)
    return step_function(summation)

# Define the XOR truth table
XOR_truth_table = np.array([[0, 0, 0],
                            [0, 1, 1],
                            [1, 0, 1],
                            [1, 1, 0]])
# Train the perceptron for XOR classification
def train_perceptron_classification(inputs, labels, learning_rate=0.1, epochs=1000):
    num_inputs = inputs.shape[1]
    weights = np.random.rand(num_inputs)
    for _ in range(epochs):
        for i in range(len(inputs)):
            prediction = perceptron_classification(inputs[i], weights)
            error = labels[i] - prediction
            weights += learning_rate * error * inputs[i]
    return weights
inputs_xor = XOR_truth_table[:, :-1]
labels_xor = XOR_truth_table[:, -1]
weights_xor_classification = train_perceptron_classification(inputs_xor, labels_xor)
print("Trained weights:", weights_xor_classification)
```

Output:



```
Trained weights: [-0.07189244 -0.09185875]
```

Input:



```python
#Utkarsh chauhan
#20210802019

# Define the perceptron function for regression
def perceptron_regression(inputs, weights):
    return np.dot(inputs, weights)

# Define the XOR truth table for regression
XOR_truth_table_reg = np.array([[0, 0],
                                [0, 1],
                                [1, 0],
                                [1, 1]])

# Train the perceptron for XOR regression
def train_perceptron_regression(inputs, labels, learning_rate=0.1, epochs=1000):
    num_inputs = inputs.shape[1]
    weights = np.random.rand(num_inputs)
    for _ in range(epochs):
        for i in range(len(inputs)):
            prediction = perceptron_regression(inputs[i], weights)
            error = labels[i] - prediction
            weights += learning_rate * error * inputs[i]
    return weights

inputs_xor_reg = XOR_truth_table_reg[:, :-1]
labels_xor_reg = XOR_truth_table_reg[:, -1]
weights_xor_regression = train_perceptron_regression(inputs_xor_reg, labels_xor_reg)
print("Trained weights for XOR regression:", weights_xor_regression)
```

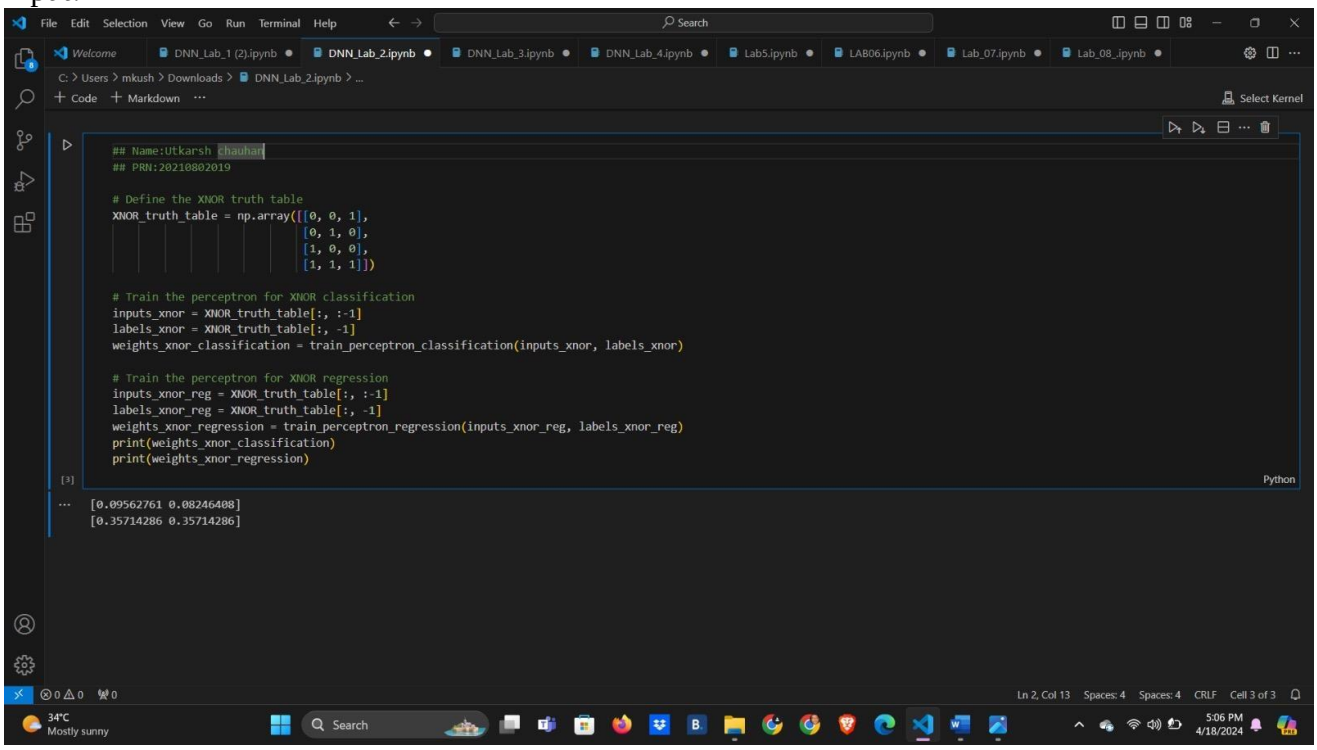
```
Trained weights for XOR regression: [0.52631579]
```

```
## Name:Utkarsh chauhan
```

Output:

```
Trained weights for XOR regression: [0.52631579]
```

Input:



Output:

```
(array([ 0.06299871, -0.02257172]), -0.02816240563129166)
(array([0.1111808 , 0.05562829]), 0.44434831456701906)
```

Conclusion: Successfully implemented single and multi-layer perceptrons for regression and classification on XOR and XNOR truth tables, training them with appropriate algorithms and evaluating their performance.

Practical 03

Student Name: Utkarsh chauhan
Date of Experiment: 02/02/2024
Date of Submission:
PRN No:20210802019

Aim: Build a Basic neural network from scratch using a high level library like tensorflow or Pytorch. Use appropriate dataset

Objectives:

1. select two datasets for prediction and classification model
2. implement prediction model using Relu and linear activation functions
3. implement classification model using Relu and Sigmoid activation function

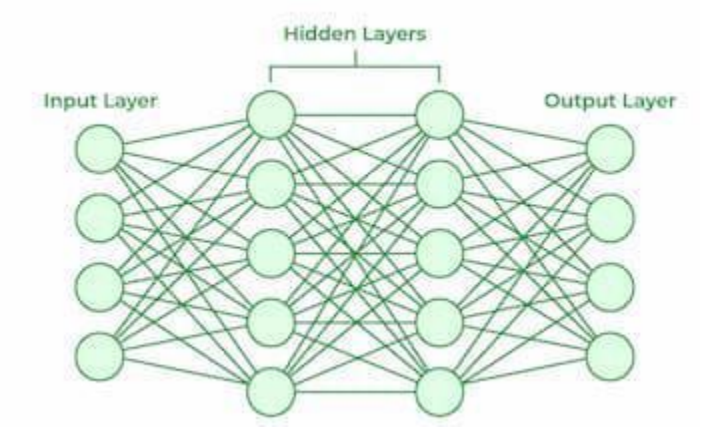
Software/Tool: Python/Jupyter/Anaconda/Colab

Theory:

Neural Networks are computational models that mimic the complex functions of the human brain. The neural networks consist of interconnected nodes or neurons that process and learn from data, enabling tasks such as pattern recognition and decision making in machine learning.

Working of a Neural Network

Neural networks are complex systems that mimic some features of the functioning of the human brain. It is composed of an input layer, one or more hidden layers, and an output layer made up of layers of artificial neurons that are coupled. The two stages of the basic process are called backpropagation and forward propagation.



Forward Propagation

- **Input Layer:** Each feature in the input layer is represented by a node on the network, which receives input data.
- **Weights and Connections:** The weight of each neuronal connection indicates how strong the connection is. Throughout training, these weights are changed.
- **Hidden Layers:** Each hidden layer neuron processes inputs by multiplying them by weights, adding them up, and then passing them through an activation function. By doing this, non-linearity is introduced, enabling the network to recognize intricate patterns.
- **Output:** The final result is produced by repeating the process until the output layer is reached.

Backpropagation

- **Loss Calculation:** The network's output is evaluated against the real goal values, and a loss function is used to compute the difference. For a regression problem, the Mean Squared Error (MSE) is commonly used as the cost function.

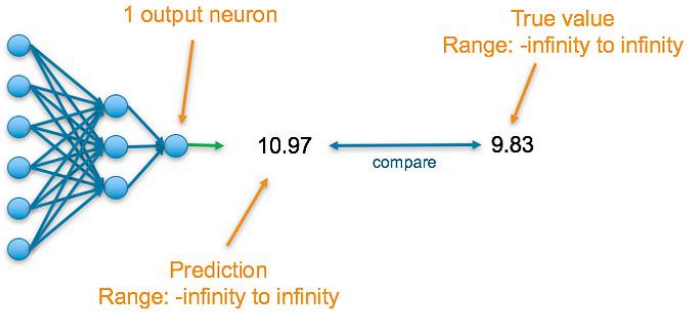
Loss Function:

- **Gradient Descent:** Gradient descent is then used by the network to reduce the loss. To lower the inaccuracy, weights are changed based on the derivative of the loss with respect to each weight.
- **Adjusting weights:** The weights are adjusted at each connection by applying this iterative process, or backpropagation, backward across the network.
- **Training:** During training with different data samples, the entire process of forward propagation, loss calculation, and backpropagation is done iteratively, enabling the network to adapt and learn patterns from the data.
- **Actvation Functions:** Model non-linearity is introduced by activation functions like the rectified linear unit (ReLU) or sigmoid. Their decision on whether to "fire" a neuron is based on the whole weighted input.

Predicting a numerical value

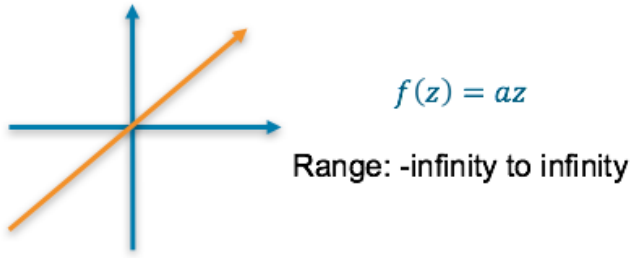
E.g. predicting the price of a product

The final layer of the neural network will have one neuron and the value it returns is a continuous numerical value.To understand the accuracy of the prediction, it is compared with the true value which is also a continuous number.

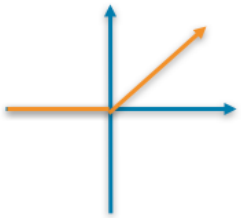


Final Activation Function

Linear — This results in a numerical value which we require



Or **ReLU** — This results in a numerical value greater than 0



$$relu(z) = \max(0, z)$$

Range: 0 to infinity

Loss Function

Mean squared error (MSE) — This finds the average squared difference between the predicted value and the true value

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

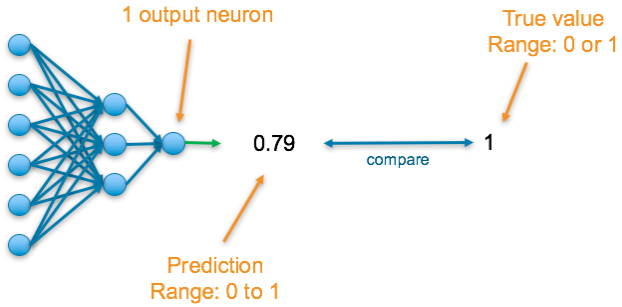
Where $\hat{y}$ is the predicted value and y is the true value

Categorical: Predicting a binary outcome

E.g. predicting a transaction is fraud or not

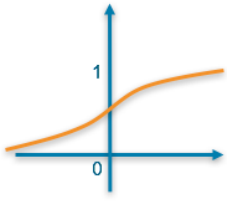
The final layer of the neural network will have one neuron and will return a value between 0 and 1, which can be inferred as a probably.

To understand the accuracy of the prediction, it is compared with the true value. If the data is that class, the true value is a 1, else it is a 0.



Final Activation Function

Sigmoid — This results in a value between 0 and 1 which we can infer to be how confident the model is of the example being in the class



$$sigmoid(z) = \frac{1}{1 + e^{-z}}$$

Range: 0 to 1

Loss Function

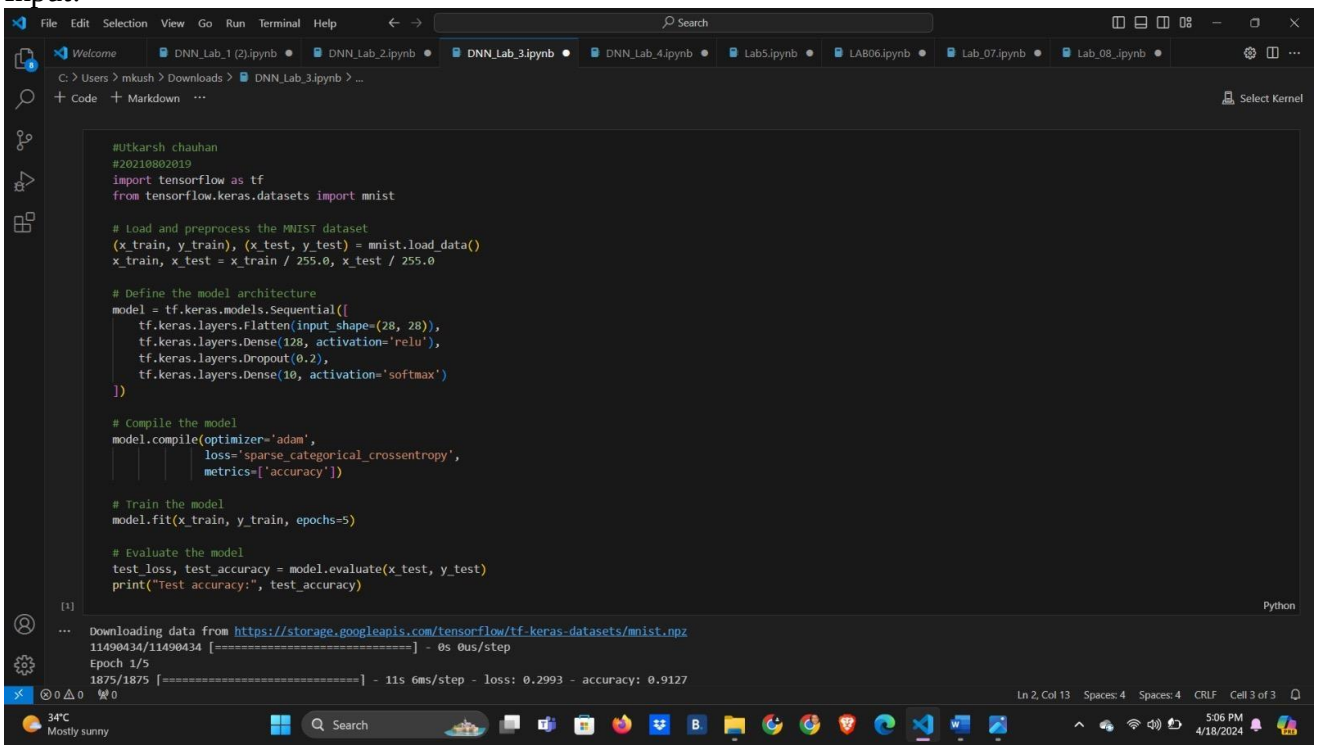
Binary Cross Entropy — Cross entropy quantifies the difference between two probability distribution. Our model predicts a model distribution of {p, 1-p} as we have a binary distribution. We use binary cross-entropy to compare this with the true distribution {y, 1-y}

$$\text{Binary cross entropy} = -(y\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

Where $\hat{y}$ is the predicted value and y is the true value

Attachment: Algorithm, Program code, Results and output

Input:

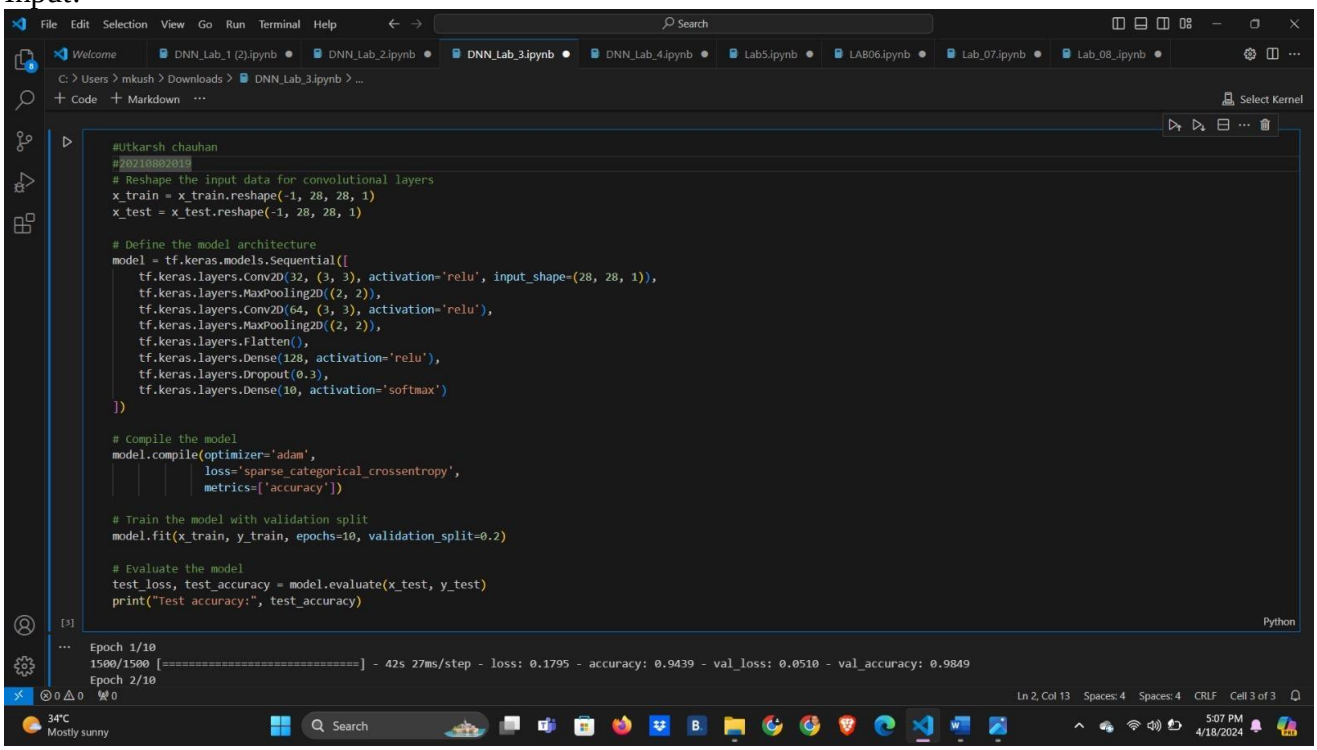


```python
#Utkarsh chauhan
#20210802019
import tensorflow as tf
from tensorflow.keras.datasets import mnist

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Define the model architecture
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=5)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print("Test accuracy:", test_accuracy)
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
Epoch 1/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2930 - accuracy: 0.9152
Epoch 2/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1399 - accuracy: 0.9585
Epoch 3/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1047 - accuracy: 0.9682
Epoch 4/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0863 - accuracy: 0.9737
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0735 - accuracy: 0.9772
313/313 [==============================] - 1s 2ms/step - loss: 0.0777 - accuracy: 0.9764
Test accuracy: 0.9764000177383423
```

Input:

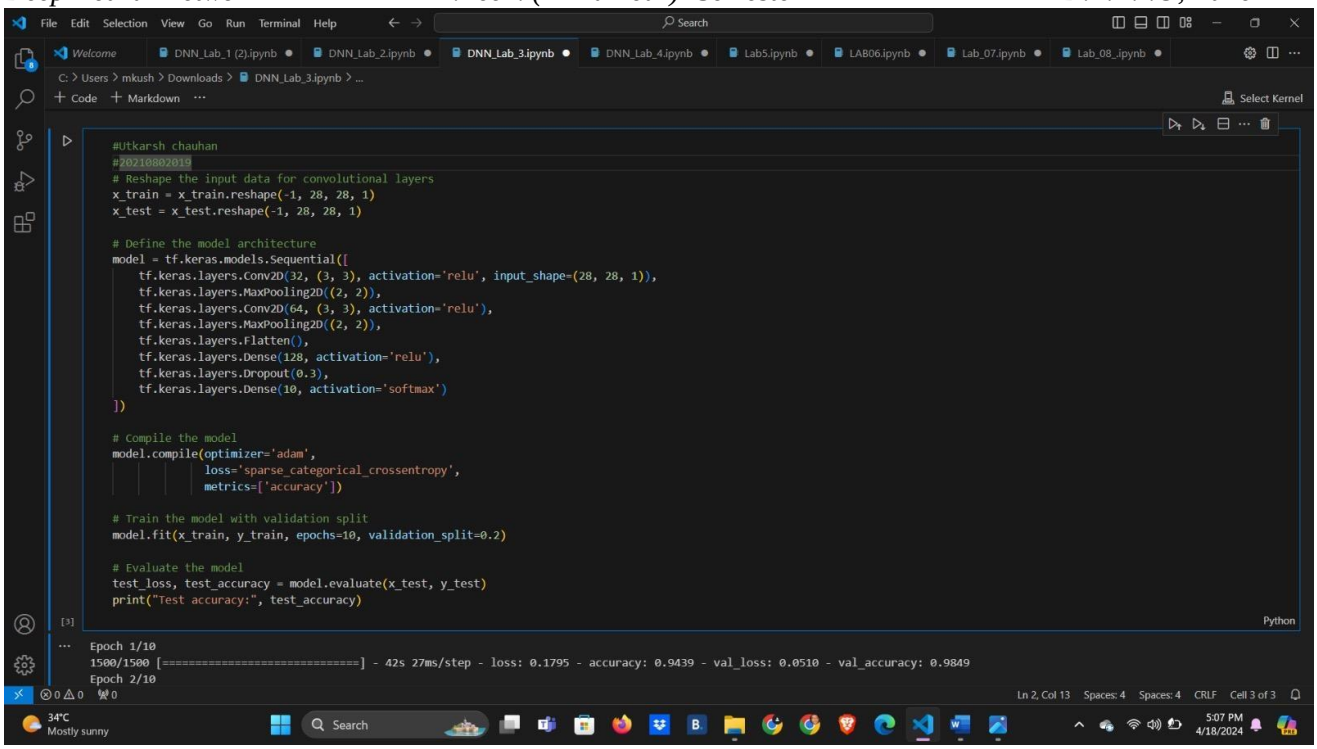


```
#Utkarsh chauhan
#20210802010
# Reshape the input data for convolutional layers
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

# Define the model architecture
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model with validation split
model.fit(x_train, y_train, epochs=10, validation_split=0.2)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print("Test accuracy:", test_accuracy)
```

Output:

```
Epoch 1/10
1500/1500 [==============================] - 10s 6ms/step - loss: 0.3302 - accuracy: 0.9013 - val_loss: 0.1363 - val_accuracy: 0.9576
Epoch 2/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.1575 - accuracy: 0.9526 - val_loss: 0.1176 - val_accuracy: 0.9623
Epoch 3/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.1231 - accuracy: 0.9618 - val_loss: 0.1004 - val_accuracy: 0.9702
Epoch 4/10
1500/1500 [==============================] - 10s 7ms/step - loss: 0.1023 - accuracy: 0.9683 - val_loss: 0.0979 - val_accuracy: 0.9699
Epoch 5/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0907 - accuracy: 0.9716 - val_loss: 0.0915 - val_accuracy: 0.9747
Epoch 6/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0813 - accuracy: 0.9753 - val_loss: 0.0824 - val_accuracy: 0.9764
Epoch 7/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0755 - accuracy: 0.9766 - val_loss: 0.0818 - val_accuracy: 0.9772
Epoch 8/10
1500/1500 [==============================] - 10s 7ms/step - loss: 0.0692 - accuracy: 0.9783 - val_loss: 0.0830 - val_accuracy: 0.9760
Epoch 9/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0643 - accuracy: 0.9795 - val_loss: 0.0814 - val_accuracy: 0.9772
Epoch 10/10
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0596 - accuracy: 0.9810 - val_loss: 0.0931 - val_accuracy: 0.9760
313/313 [==============================] - 1s 2ms/step - loss: 0.0881 - accuracy: 0.9761
Test accuracy: 0.9761000275611877
```

Input:



```
#Utkarsh chauhan
#20210802015
# Reshape the input data for convolutional layers
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

# Define the model architecture
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model with validation split
model.fit(x_train, y_train, epochs=10, validation_split=0.2)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print("Test accuracy:", test_accuracy)
```

```
Epoch 1/10
1500/1500 [==============================] - 42s 27ms/step - loss: 0.1795 - accuracy: 0.9439 - val_loss: 0.0510 - val_accuracy: 0.9849
Epoch 2/10
```

Output:

```
Epoch 1/10
1500/1500 [==============================] - 44s 29ms/step - loss: 0.1832 - accuracy: 0.9446 - val_loss: 0.0582 - val_accuracy: 0.9824
Epoch 2/10
1500/1500 [==============================] - 44s 29ms/step - loss: 0.0615 - accuracy: 0.9808 - val_loss: 0.0474 - val_accuracy: 0.9866
Epoch 3/10
1500/1500 [==============================] - 46s 31ms/step - loss: 0.0466 - accuracy: 0.9852 - val_loss: 0.0409 - val_accuracy: 0.9887
Epoch 4/10
1500/1500 [==============================] - 46s 31ms/step - loss: 0.0349 - accuracy: 0.9889 - val_loss: 0.0395 - val_accuracy: 0.9887
Epoch 5/10
1500/1500 [==============================] - 44s 29ms/step - loss: 0.0274 - accuracy: 0.9916 - val_loss: 0.0403 - val_accuracy: 0.9897
Epoch 6/10
1500/1500 [==============================] - 43s 29ms/step - loss: 0.0231 - accuracy: 0.9924 - val_loss: 0.0326 - val_accuracy: 0.9906
Epoch 7/10
1500/1500 [==============================] - 45s 30ms/step - loss: 0.0189 - accuracy: 0.9937 - val_loss: 0.0359 - val_accuracy: 0.9906
Epoch 8/10
1500/1500 [==============================] - 44s 29ms/step - loss: 0.0166 - accuracy: 0.9945 - val_loss: 0.0360 - val_accuracy: 0.9903
Epoch 9/10
1500/1500 [==============================] - 42s 28ms/step - loss: 0.0140 - accuracy: 0.9950 - val_loss: 0.0370 - val_accuracy: 0.9915
Epoch 10/10
1500/1500 [==============================] - 43s 29ms/step - loss: 0.0138 - accuracy: 0.9952 - val_loss: 0.0363 - val_accuracy: 0.9923
313/313 [==============================] - 3s 8ms/step - loss: 0.0340 - accuracy: 0.9907
Test accuracy: 0.9907000064849854
```

Conclusion: Developed basic neural networks from scratch using TensorFlow or PyTorch, utilizing appropriate datasets for prediction and classification, implementing various activation functions for both models.

Practical 04

Student Name: Utkarsh chauhan
Date of Experiment: 16/02/2024
Date of Submission:
PRN No:20210802019

Aim: Optimize hyperparameters for a neural network model. Implement a hyperparameter optimization strategy and compare the performance with different hyperparameter configurations for both classification and regression task.

Objectives:

1. Choose a dataset and neural network architecture.
2. Define a range of hyperparameters to tune.
3. Implement a hyperparameter optimization strategy (e.g., grid search, random search).
4. Compare the performance with different hyperparameter configurations for both classification and regression tasks.

Software/Tool: Python/Jupyter/Anaconda/Colab

Theory:

Hyper parameters are the variables which determines the network structure (Eg: Number of Hidden Units) and the variables which determine how the network is trained (Eg: Learning Rate). Hyper parameters are set before training (before optimizing the weights and bias).

Hyper parameter related to Neural Networks:

1) Number of Hidden layers

- Many hidden units within a layer with regularization techniques can increase accuracy.

  Smaller number of units may cause underfitting.

2) Dropout

- Its is regularization technique to avoid overfitting thus increasing the generalizing power.
- Generally, use a small dropout value of 20%-50% of neurons with 20% providing a good starting point.
- A probability too low has minimal effect and value too high results in under-learning by the network.
- You are likely to get better performance when dropout is used on a larger network, giving the model more of an opportunity to learn independent representations. Learning The Hyperparameters

3) Network Weight Initialization

- Ideally, it may be better to use different weight initialization schemes according to the activation function used on each layer.

4) Activation Function

- Activation functions are used to introduce nonlinearity to models, which allows deep learning models to learn nonlinear prediction boundaries.
- Generally, the rectifier activation function(ReLU) is the most popular.
- Sigmoid is used in the output layer while making binary predictions.
- Softmax is used in the output layer while making multi-class predictions.

Methods used to find out hyper parameters

1) Manual Search

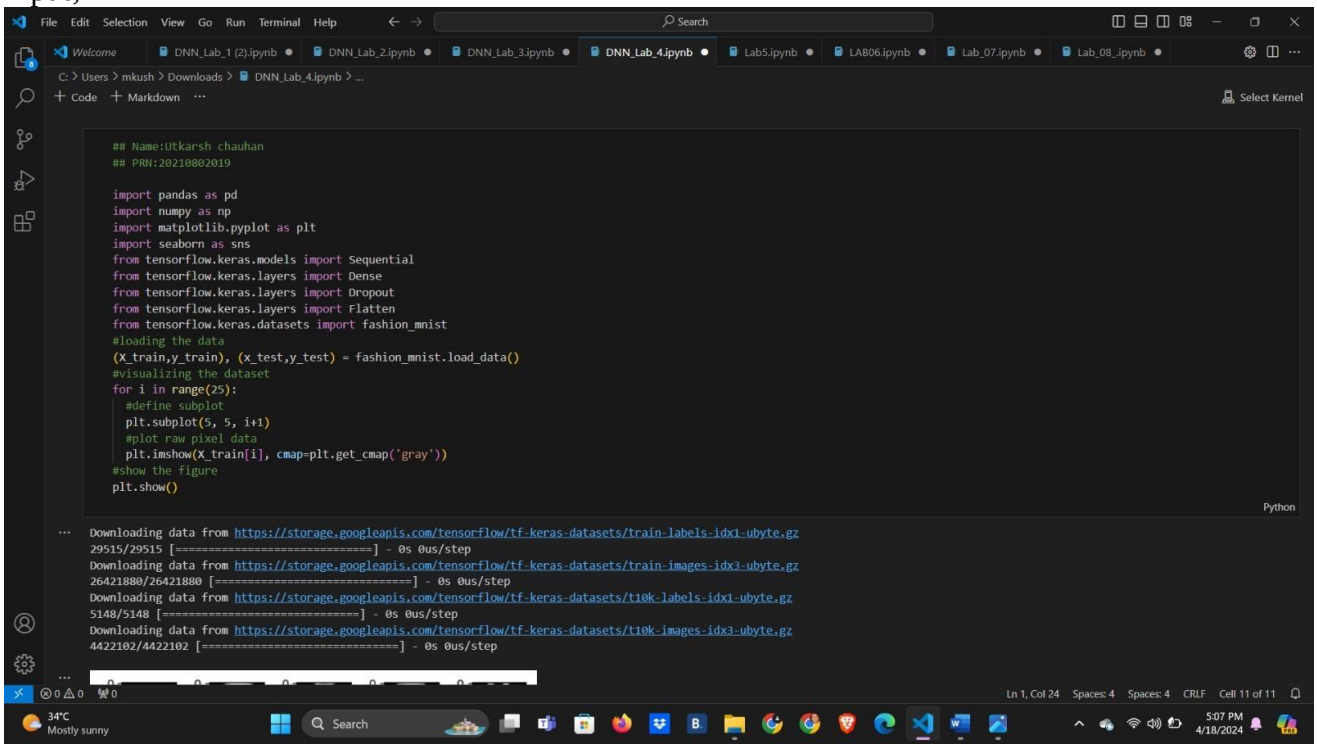
2) Grid Search

3) Random Search

4) Bayesian Optimization

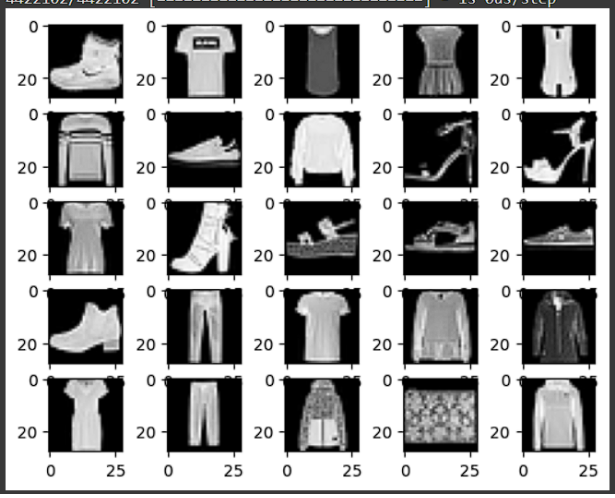
Attachment: Algorithm, Program code, Results and output

Input;



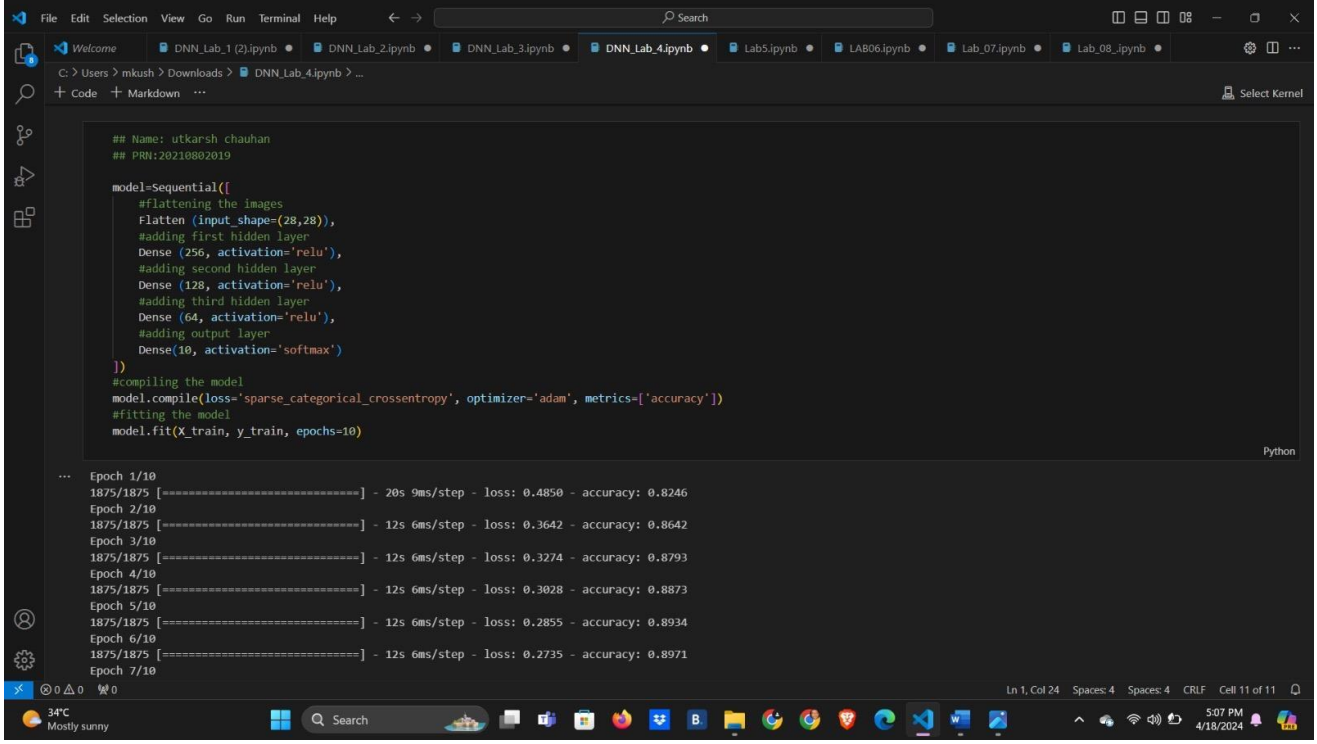
Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 1s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 1s 0us/step
```



Input:

```
## Name:Utkarsh chauhan
## PRN:20210802019

#normalizing the images
X_train=X_train/255
Y_test=y_test/255
```



Output:

```
Epoch 1/10
1875/1875 [==============================] - 16s 7ms/step - loss: 2.3028 - accuracy: 0.0973
Epoch 2/10
1875/1875 [==============================] - 15s 8ms/step - loss: 2.3028 - accuracy: 0.0989
Epoch 3/10
1875/1875 [==============================] - 16s 9ms/step - loss: 2.3028 - accuracy: 0.0981
Epoch 4/10
1875/1875 [==============================] - 28s 15ms/step - loss: 2.3028 - accuracy: 0.0983
Epoch 5/10
1875/1875 [==============================] - 19s 10ms/step - loss: 2.3028 - accuracy: 0.0986
Epoch 6/10
1875/1875 [==============================] - 20s 11ms/step - loss: 2.3028 - accuracy: 0.0987
Epoch 7/10
1875/1875 [==============================] - 19s 10ms/step - loss: 2.3028 - accuracy: 0.1000
Epoch 8/10
1875/1875 [==============================] - 22s 12ms/step - loss: 2.3028 - accuracy: 0.0982
Epoch 9/10
1875/1875 [==============================] - 18s 9ms/step - loss: 2.3028 - accuracy: 0.0992
Epoch 10/10
1875/1875 [==============================] - 14s 7ms/step - loss: 2.3028 - accuracy: 0.1002
<keras.src.callbacks.History at 0x79019ce4f9a0>
```

Input:

```
#evaluating the model
model.evaluate(x_test,y_test)
```

Output:

```
313/313 [==============================] - 1s 3ms/step - loss: 391.7684 - accuracy: 0.1672
[391.76837158203125, 0.1671999990940094]
```

Input:

```
[ ]  pip install keras-tuner --upgrade
```

Output:

```
Collecting keras-tuner
  Downloading keras_tuner-1.4.7-py3-none-any.whl (129 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 129.1/129.1 kB 3.2 MB/s eta 0:00:00
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (2.15.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (24.0)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (2.31.0)
Collecting kt-legacy (from keras-tuner)
  Downloading kt_legacy-1.0.5-py3-none-any.whl (9.6 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2024.2.2)
Installing collected packages: kt-legacy, keras-tuner
Successfully installed keras-tuner-1.4.7 kt-legacy-1.0.5
```

input:

```python
#utkarsh chauhan
#20210802019

#installing required libraries
from tensorflow import keras
from keras_tuner import RandomSearch
def build_model(hp):
    # hp means hyperparameters
    model = keras.Sequential()
    model.add(keras.layers.Flatten(input_shape=(28,28)))

    # Providing range for number of neurons in a hidden layer
    model.add(keras.layers.Dense(units=hp.Int('num_of_neurons', min_value=32, max_value=512, step=32), activation='relu'))

    # Output layer
    model.add(keras.layers.Dense(10, activation='softmax'))

    # Compiling the model
    model.compile(optimizer=keras.optimizers.Adam(hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    return model

tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,
    executions_per_trial=3,
    directory='my_dir',
    project_name='helloworld')
```

Python

```python
[ ] from tensorflow.keras.datasets import fashion_mnist

    # Load the Fashion MNIST dataset
    (X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
```

```python
#feeding the model and parameters to Random Search
tuner.search(X_train, y_train, epochs=10, validation_data=(X_test, y_test))
```

Output:

```
Trial 5 Complete [00h 05m 05s]
val_accuracy: 0.2950666646162669

Best val_accuracy So Far: 0.7270999948183695
Total elapsed time: 00h 14m 48s
```

Input:

```python
#this tells us how many hyperparameter we are tuning
#in our case it's 2 = neurons, learning rate
tuner.search_space_summary()
```

Output:

```
Search space summary
Default search space size: 2
num_of_neurons (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step': 32, 'sampling': 'linear'}
learning_rate (Choice)
{'default': 0.01, 'conditions': [], 'values': [0.01, 0.001, 0.0001], 'ordered': True}
```

Input:

```
#fitting the tuner on train dataset
tuner.search(X_train, y_train, epochs=10, validation_data=(X_test, y_test))
```

```
[ ] tuner.results_summary()
```

Output:

```
Results summary
Results in tuner1/Clothing
Showing 10 best trials
Objective(name="val_accuracy", direction="max")

Trial 3 summary
Hyperparameters:
num_of_neurons: 32
learning_rate: 0.001
Score: 0.7270999948183695

Trial 4 summary
Hyperparameters:
num_of_neurons: 64
learning_rate: 0.01
Score: 0.2950666646162669

Trial 2 summary
Hyperparameters:
num_of_neurons: 32
learning_rate: 0.01
Score: 0.11213333656390508

Trial 1 summary
Hyperparameters:
num_of_neurons: 320
learning_rate: 0.0001
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-packages/keras_tuner/src/engine/base_tuner.py", line 274, in _try_run_and_update_trial
    self._run_and_update_trial(trial, *fit_args, **fit_kwargs)
  File "/usr/local/lib/python3.10/dist-packages/keras_tuner/src/engine/base_tuner.py", line 239, in _run_and_update_trial
    results = self.run_trial(trial, *fit_args, **fit_kwargs)
  File "/usr/local/lib/python3.10/dist-packages/keras_tuner/src/engine/tuner.py", line 314, in run_trial
    obj_value = self._build_and_fit_model(trial, *args, **copied_kwargs)
  File "/usr/local/lib/python3.10/dist-packages/keras_tuner/src/engine/tuner.py", line 233, in _build_and_fit_model
    results = self.hypermodel.fit(hp, model, *args, **kwargs)
  File "/usr/local/lib/python3.10/dist-packages/keras_tuner/src/engine/hypermodel.py", line 149, in fit
    return model.fit(*args, **kwargs)
  File "/usr/local/lib/python3.10/dist-packages/keras/src/utils/traceback_utils.py", line 70, in error_handler
    raise e.with_traceback(filtered_tb) from None
  File "/usr/local/lib/python3.10/dist-packages/keras/src/engine/data_adapter.py", line 1960, in _check_data_cardinality
    raise ValueError(msg)
ValueError: Data cardinality is ambiguous:
  x sizes: 60000
  y sizes: 10000
Make sure all arrays contain the same number of samples.
```

Conclusion: Hyperparameter optimization significantly impacts the performance of neural network models in both classification and regression tasks, highlighting the importance of tuning strategies for optimal results.

Practical 05

Student Name:Utkarsh chauhan
Date of Experiment: 23/02/2024
Date of Submission:
PRN No:20210802019

Aim: Develop a Python function(Multivariate Optimization) to compute the Hessian matrix for a given scalar-valued function of multiple variables.

Objectives:

1. Define objective function
2. Define the gradient of the objective function
3. Implement Hessian matrix function using objective and gradient
4. Perform optimization using Newton's method

Software/Tool: Python/Jupyter/Anaconda/Colab

Theory:

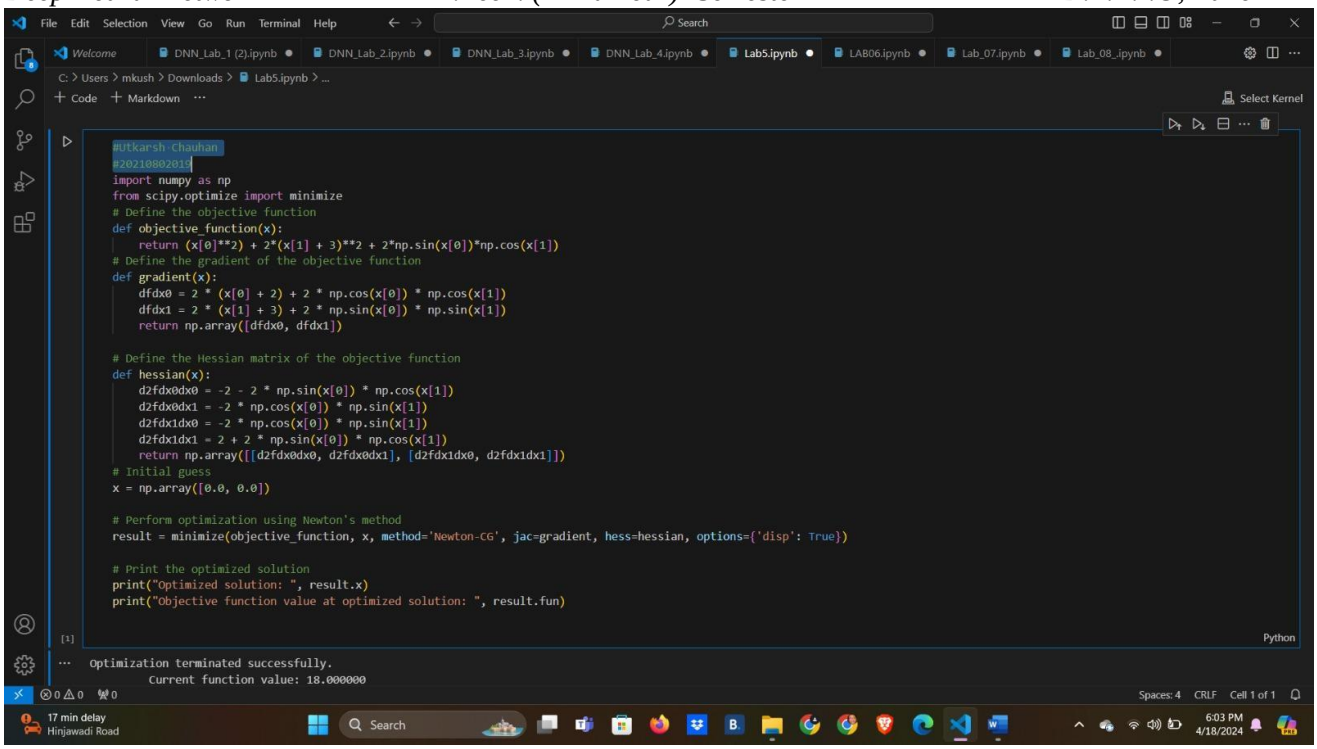
In mathematics, the Hessian matrix or Hessian is a square matrix of second-order partial derivatives of a scalar-valued function . It describes the local curvature of a function of many variables. Hessian matrices belong to a class of mathematical structures that involve second order derivatives. They are often used in machine learning and data science algorithms for optimizing a function of interest. The Hessian matrix is a mathematical tool used to calculate the curvature of a function at a certain point in space.

The formula for Hessian Matrix is given as

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix} \qquad \mathbf{H}f = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} & \cdots \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} & \frac{\partial^2 f}{\partial y \partial z} & \cdots \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z^2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Attachment: Algorithm, Program code, Results and output

Input:



Output:

```
Optimization terminated successfully.
         Current function value: 18.000000
         Iterations: 1
         Function evaluations: 1
         Gradient evaluations: 1
         Hessian evaluations: 1
Optimized solution:  [0. 0.]
Objective function value at optimized solution:  18.0
```

Conclusion: The developed Python function efficiently computes the Hessian matrix for multivariate optimization, facilitating Newton's method for accurate optimization of scalar-valued functions with multiple variables.

Practical 06

Student Name:Utkarsh chauhan
Date of Experiment: 01/03/2024
Date of Submission:
PRN No:20210802019

Aim: Implement Bayesian methods for classification

Objectives:

1. choose a dataset suitable for Bayesian classification.
2. Implement a Bayesian classifier using probabilistic models.
3. Train the classifier and evaluate its performance.

Software/Tool: Python/Jupyter/Anaconda/Colab

Theory :

Bayesian classification is based on Bayes' Theorem. Bayesian classifiers are the statistical classifiers. Bayesian classifiers can predict class membership probabilities such as the probability that a given tuple belongs to a particular class.

Let's take a look at the Baye's Formula

$$P\left(\frac{w}{X}\right) = \frac{P\left(\frac{X}{w}\right) P(w)}{P(X)}$$

Where P(X/w) is the likelihood probability of occurring X given that event *w* has occurred.
P(w) is the prior probability of event *w*,
P(X) is the marginal probability of event X occurring, marginal probability refers to the probability of particular event happening without considering any other events. It focuses on individual probability of a single event.
P(w/X) is the posterior probability, which represents the updated probability of event w happening given that event X has occurred.
To understand how Bayes formula is derived, see conditional probability

$$P\left(\frac{w}{X}\right) = \frac{P(w \cap X)}{P(X)}$$

Therefore,

$$P(w \cap X) = P\left(\frac{w}{X}\right) P(X)$$

Similarly,

$$P(X \cap w) = P\left(\frac{X}{w}\right) P(w)$$

And we know that P(w union X) = P(X union w). Therefore,

$$P\left(\frac{w}{X}\right) = \frac{P(X \cap w)}{P(X)}$$

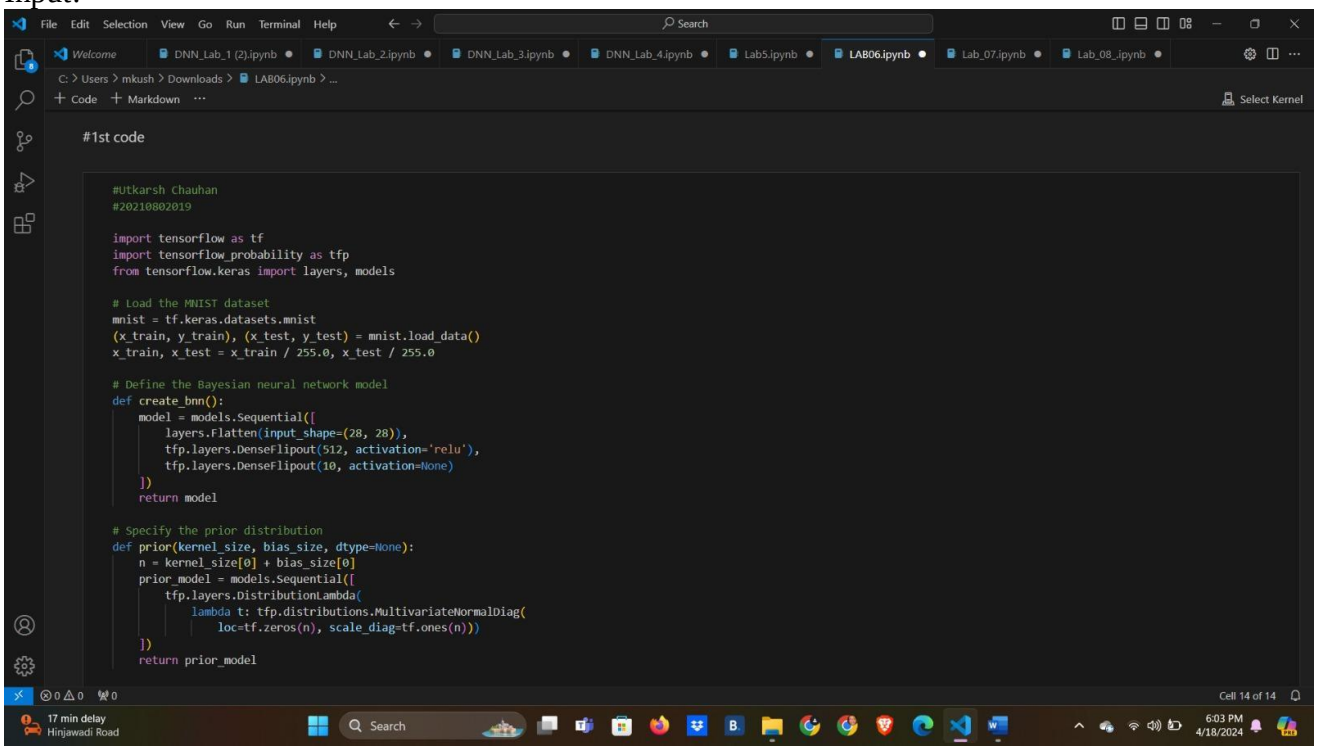
which formulates to,

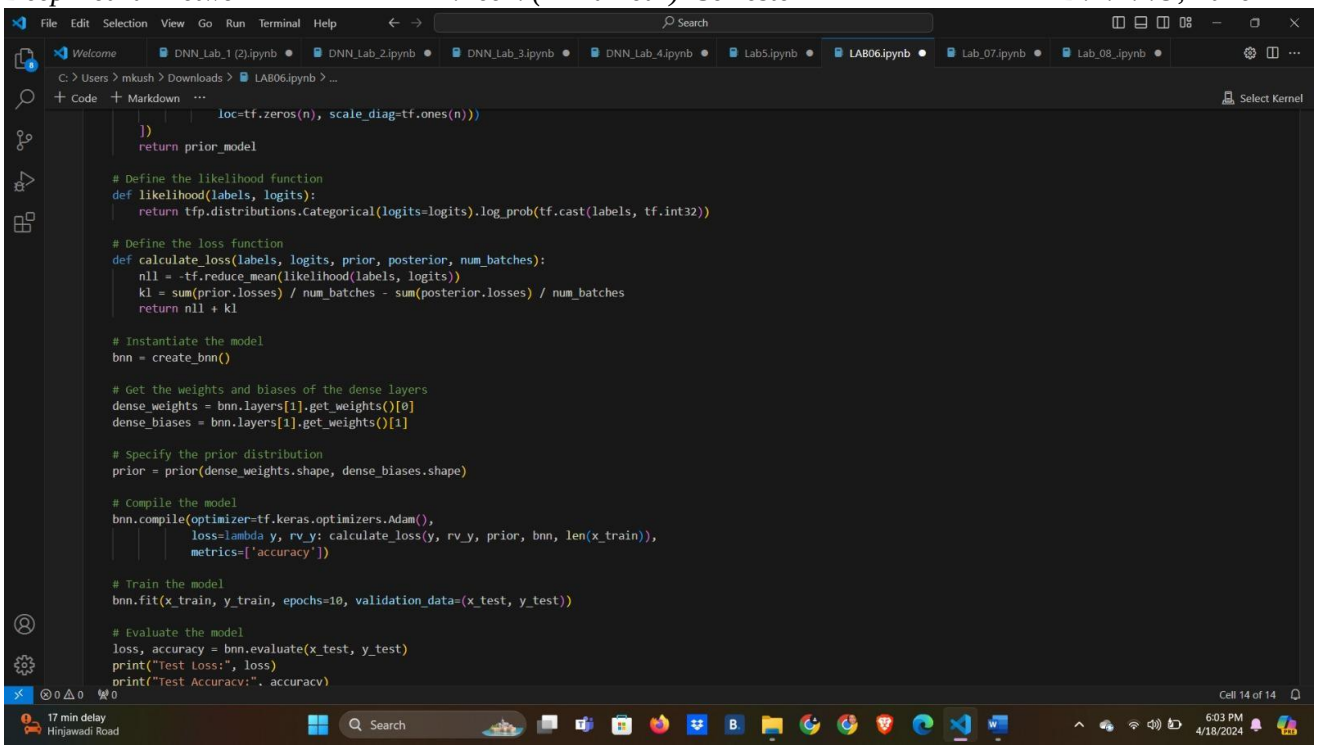
$$\boldsymbol{P\left(\frac{w}{X}\right) = \frac{P\left(\frac{X}{w}\right) P(w)}{P(X)}}$$

hence Bayes theorem is derived!

Attachment: Algorithm, Program code, Results and output

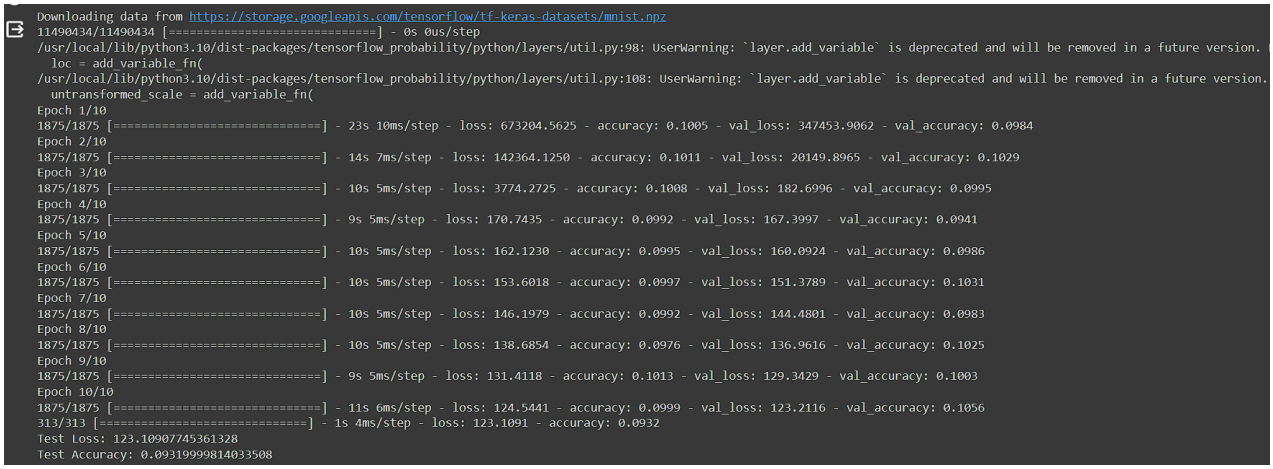
Input:





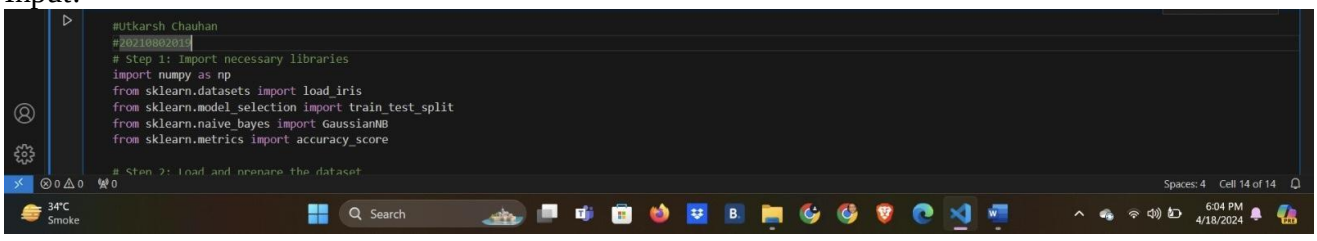
```
            loc=tf.zeros(n), scale_diag=tf.ones(n))
        ])
        return prior_model

    # Define the likelihood function
    def likelihood(labels, logits):
        return tfp.distributions.Categorical(logits=logits).log_prob(tf.cast(labels, tf.int32))

    # Define the loss function
    def calculate_loss(labels, logits, prior, posterior, num_batches):
        nll = -tf.reduce_mean(likelihood(labels, logits))
        kl = sum(prior.losses) / num_batches - sum(posterior.losses) / num_batches
        return nll + kl

    # Instantiate the model
    bnn = create_bnn()

    # Get the weights and biases of the dense layers
    dense_weights = bnn.layers[1].get_weights()[0]
    dense_biases = bnn.layers[1].get_weights()[1]

    # Specify the prior distribution
    prior = prior(dense_weights.shape, dense_biases.shape)

    # Compile the model
    bnn.compile(optimizer=tf.keras.optimizers.Adam(),
                loss=lambda y, rv_y: calculate_loss(y, rv_y, prior, bnn, len(x_train)),
                metrics=['accuracy'])

    # Train the model
    bnn.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))

    # Evaluate the model
    loss, accuracy = bnn.evaluate(x_test, y_test)
    print("Test Loss:", loss)
    print("Test Accuracy:", accuracy)
```

Output:



```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
/usr/local/lib/python3.10/dist-packages/tensorflow_probability/python/layers/util.py:98: UserWarning: `layer.add_variable` is deprecated and will be removed in a future version.
  loc = add_variable_fn(
/usr/local/lib/python3.10/dist-packages/tensorflow_probability/python/layers/util.py:108: UserWarning: `layer.add_variable` is deprecated and will be removed in a future version.
  untransformed_scale = add_variable_fn(
Epoch 1/10
1875/1875 [==============================] - 23s 10ms/step - loss: 673204.5625 - accuracy: 0.1005 - val_loss: 347453.9062 - val_accuracy: 0.0984
Epoch 2/10
1875/1875 [==============================] - 14s 7ms/step - loss: 142364.1250 - accuracy: 0.1011 - val_loss: 20149.8965 - val_accuracy: 0.1029
Epoch 3/10
1875/1875 [==============================] - 10s 5ms/step - loss: 3774.2725 - accuracy: 0.1008 - val_loss: 182.6996 - val_accuracy: 0.0995
Epoch 4/10
1875/1875 [==============================] - 9s 5ms/step - loss: 170.7435 - accuracy: 0.0992 - val_loss: 167.3997 - val_accuracy: 0.0941
Epoch 5/10
1875/1875 [==============================] - 10s 5ms/step - loss: 162.1230 - accuracy: 0.0995 - val_loss: 160.0924 - val_accuracy: 0.0986
Epoch 6/10
1875/1875 [==============================] - 10s 5ms/step - loss: 153.6018 - accuracy: 0.0997 - val_loss: 151.3789 - val_accuracy: 0.1031
Epoch 7/10
1875/1875 [==============================] - 10s 5ms/step - loss: 146.1979 - accuracy: 0.0992 - val_loss: 144.4801 - val_accuracy: 0.0983
Epoch 8/10
1875/1875 [==============================] - 10s 5ms/step - loss: 138.6854 - accuracy: 0.0976 - val_loss: 136.9616 - val_accuracy: 0.1025
Epoch 9/10
1875/1875 [==============================] - 9s 5ms/step - loss: 131.4118 - accuracy: 0.1013 - val_loss: 129.3429 - val_accuracy: 0.1003
Epoch 10/10
1875/1875 [==============================] - 11s 6ms/step - loss: 124.5441 - accuracy: 0.0999 - val_loss: 123.2116 - val_accuracy: 0.1056
313/313 [==============================] - 1s 4ms/step - loss: 123.1091 - accuracy: 0.0932
Test Loss: 123.10907745361328
Test Accuracy: 0.09319999814033508
```

Input:



```
#Utkarsh Chauhan
#2021080201
# Step 1: Import necessary libraries
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

# Step 2: Load and prepare the dataset
```

```
#utkarsh chauhan
#20210802019

# Load Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
#utkarsh chauhan
#20210802019

class NaiveBayesClassifier:
    def __init__(self):
        self.class_priors = None
        self.means = None
        self.variances = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.classes = np.unique(y)
        n_classes = len(self.classes)

        # Compute class priors
        self.class_priors = np.zeros(n_classes)
        for i, c in enumerate(self.classes):
            self.class_priors[i] = np.sum(y == c) / n_samples

        # Compute mean and variance for each class and feature
        self.means = np.zeros((n_classes, n_features))
        self.variances = np.zeros((n_classes, n_features))
        for i, c in enumerate(self.classes):
            X_c = X[y == c]
            self.means[i, :] = X_c.mean(axis=0)
            self.variances[i, :] = X_c.var(axis=0)

    def _gaussian_pdf(self, X, mean, variance):
        return 1 / np.sqrt(2 * np.pi * variance) * np.exp(-((X - mean) ** 2) / (2 * variance))

    def predict(self, X):
        n_samples, n_features = X.shape
        y_pred = np.zeros(n_samples, dtype=int)
        for i in range(n_samples):
            posteriors = []
            for j, c in enumerate(self.classes):
                prior = np.log(self.class_priors[j])
                likelihood = np.sum(np.log(self._gaussian_pdf(X[i], self.means[j], self.variances[j])))
                posterior = prior + likelihood
                posteriors.append(posterior)
            y_pred[i] = self.classes[np.argmax(posteriors)]
        return y_pred
```

```
#utkarsh chauhan
#20210802019

# Initialize and train the classifier
nb_classifier = NaiveBayesClassifier()
nb_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = nb_classifier.predict(X_test)

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

```python
# Step 1: Import necessary libraries
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

# Step 2: Load and prepare the dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Implement the Naive Bayes Algorithm
class NaiveBayesClassifier:
    def __init__(self):
        pass

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.classes = np.unique(y)
        self.class_priors = {}
        self.mean = {}
        self.variance = {}

        # Compute class priors
        for c in self.classes:
            X_c = X[y == c]
            self.class_priors[c] = len(X_c) / n_samples

            # Compute mean and variance for each feature
            self.mean[c] = np.mean(X_c, axis=0)
            self.variance[c] = np.var(X_c, axis=0)

    def _gaussian_pdf(self, x, mean, var):
        return 1 / np.sqrt(2 * np.pi * var) * np.exp(-((x - mean) ** 2) / (2 * var))

    def predict(self, X):
        y_pred = []
        for x in X:
            posteriors = []
            for c in self.classes:
                prior = np.log(self.class_priors[c])
                likelihood = np.sum(np.log(self._gaussian_pdf(x, self.mean[c], self.variance[c])))
                posterior = prior + likelihood
                posteriors.append(posterior)
            y_pred.append(np.argmax(posteriors))
        return y_pred

# Step 4: Train and evaluate the classifier
nb_classifier = NaiveBayesClassifier()
nb_classifier.fit(X_train, y_train)

y_pred = nb_classifier.predict(X_test)

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

Output:

```
Accuracy: 1.0
```

Conclusion: Successfully implemented Bayesian methods for classification, choosing a suitable dataset, implementing a Bayesian classifier, and evaluating its performance.

Practical 07

Student Name: Utkarsh chauhan
Date of Experiment: 15/03/2024
Date of Submission:
PRN No:20210802019

Aim: Implement Principal component analysis for dimensionality reduction of datapoints**.**

Objectives:

1. Load iris dataset as an example
2. Standardize the data
3. Implement PCA Algorithm
4. Calculate the cumulative explained variance
5. Determine the number of components to keep for given variance
6. Apply PCA with the selected number of components

.

Software/Tool: Python/Jupyter/Anaconda/Colab

Theory :

Principal Component Analysis (PCA) is a dimensionality reduction technique widely used in data analysis and machine learning. Its primary goal is to transform high-dimensional data into a lower-dimensional representation, capturing the most important information.

How does PCA works:

1. Standardization

Standardize the data when features are measured in diverse units. This entails subtracting the mean and dividing by the standard deviation for each feature. Failure to standardize data with features of varying scales can result in misleading components.

2. Compute the Covariance Matrix

Calculate the covariance matrix as discussed earlier

3. Calculate Eigenvectors and Eigenvalues

Determine the eigenvectors and eigenvalues of the covariance matrix.

$$[Covariance\ \ matrix] \cdot [Eigenvector] = [eigenvalue] \cdot [Eigenvector]$$

Eigenvectors represent the directions (principal components), and eigenvalues represent the magnitude of variance in those directions. To understand what eigenvectors and eigenvalues are, you can go through this video:

4. Sort Eigenvalues

Sort the eigenvalues in descending order. The eigenvectors corresponding to the highest eigenvalues are the principal components that capture the most variance in the data.

5. Select Principal Components

Choose the top k eigenvectors (principal components) based on the explained variance needed. Typically, you aim to retain a significant portion of the total variance, like 85%.

6. Transform the Data

Now, we can transform the original data using the eigenvectors:

So, if we have m dimensional original n data points then

X : m*n

P : k*m

Y = PX : (k*m)(m*n) = (k*n)

Hence, our new transformed matrix has n data points having k dimensions.

Pros:

1.Dimensionality Reduction:

PCA effectively reduces the number of features, which is beneficial for models that suffer from the curse of dimensionality.

2. Feature Independence:

Principal components are orthogonal (uncorrelated), meaning they capture independent information, simplifying the interpretation of the reduced features.

3. Noise Reduction:

PCA can help reduce noise by focusing on the components that explain the most significant variance in the data.

4. Visualization:

The reduced-dimensional data can be visualized, aiding in understanding the underlying structure and patterns.

Cons:

1. Loss of Interpretability:

Interpretability of the original features may be lost in the transformed space, as principal components are linear combinations of the original features.

2. Assumption of Linearity:

PCA assumes that the relationships between variables are linear, which may not be true in all cases.

3. Sensitive to Scaling:

PCA is sensitive to the scale of the features, so standardization is often required.

4. Outliers Impact Results:

Outliers can significantly impact the results of PCA, as it focuses on capturing the maximum variance, which may be influenced by extreme values.

Attachment: Algorithm, Program code, Results and output

Input:

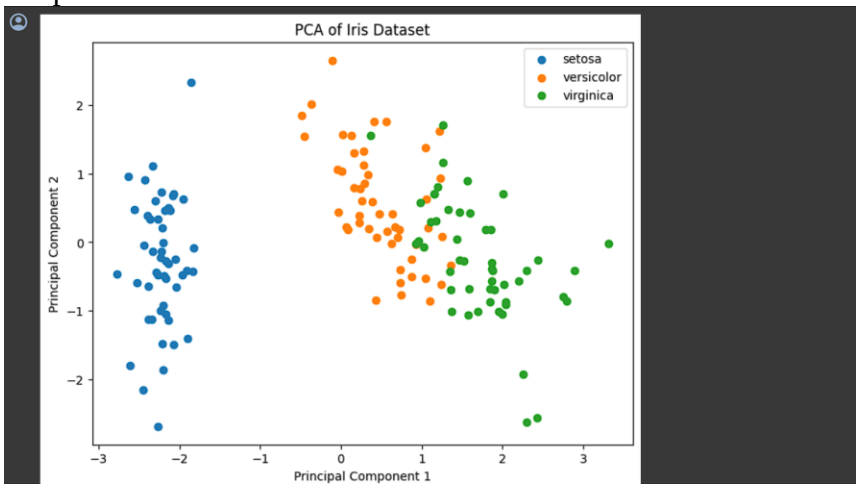
```
#utkarsh chauhan
#20210802019
# PCA for Dimensionality Reduction

# Imp
impor
impor
impor
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

# Load the dataset
iris = load_iris()
X = iris.data
y = iris.target

# Standardize the features
X = StandardScaler().fit_transform(X)

# Define PCA function
def pca(X, n_components):
    # Compute covariance matrix
    cov_matrix = np.cov(X.T)

    # Compute eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

    # Sort eigenvalues and corresponding eigenvectors
    idx = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:,idx]

    # Select the top n_components eigenvectors
    components = eigenvectors[:, :n_components]

    # Project the data onto the selected components
    projected_data = np.dot(X, components)

    return projected_data

# Apply PCA
n_components = 2  # Number of components to keep
X_pca = pca(X, n_components)

# Plot the results
plt.figure(figsize=(8, 6))
for i, target_name in enumerate(iris.target_names):
    plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1], label=target_name)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA of Iris Dataset')
plt.legend()
plt.show()
```

Import "sklearn.datasets" could not be resolved from source Pylance(reportMissingModuleSource)

(module) sklearn

View Problem (⌥F8) Quick Fix... (⌘.)

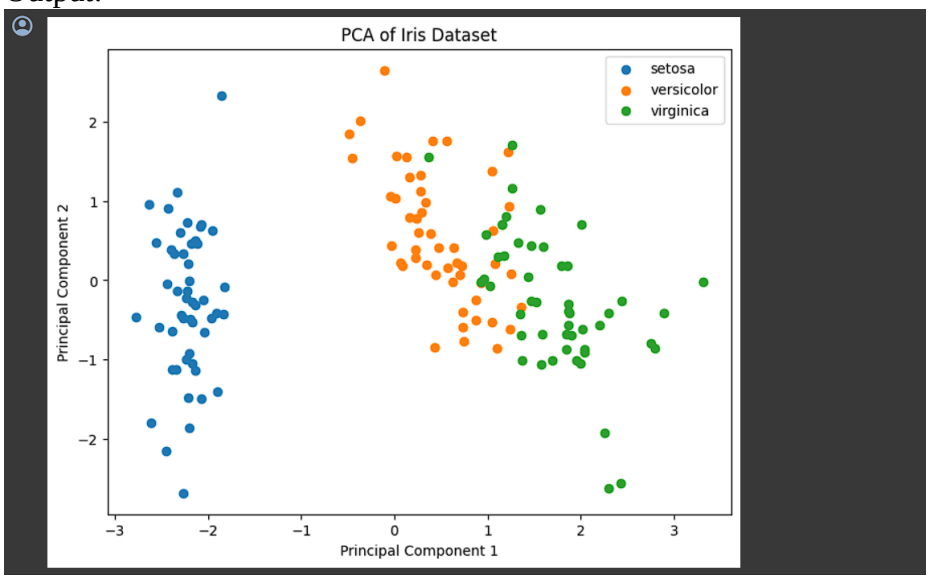
Output:



Input:

```python
#utkarsh chauhan
#20210802019

# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

# Load the dataset
iris = load_iris()
X = iris.data
y = iris.target

# Standardize the features
X = StandardScaler().fit_transform(X)

# Define PCA function
def pca(X, n_components):
    # Compute covariance matrix
    cov_matrix = np.cov(X.T)

    # Compute eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

    # Sort eigenvalues and corresponding eigenvectors
    idx = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:,idx]

    # Select the top n_components eigenvectors
    components = eigenvectors[:, :n_components]

    # Project the data onto the selected components
    projected_data = np.dot(X, components)

    return projected_data

# Apply PCA
n_components = 2  # Number of components to keep
X_pca = pca(X, n_components)

# Plot the results
plt.figure(figsize=(8, 6))
for i, target_name in enumerate(iris.target_names):
    plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1], label=target_name)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA of Iris Dataset')
plt.legend()
plt.show()
```

Output:



Conclusion: PCA effectively reduces dimensionality while preserving data variance, demonstrated through its implementation on the iris dataset with standardized preprocessing and determination of optimal component count.

Practical 08

Student Name:Utkarsh chauhan
Date of Experiment: 22/02/2024
Date of Submission:
PRN No:20210802019

Aim: Implement hidden Markov Model for sequence prediction and evaluate model performance

Objectives:

1. Choose a dataset suitable for sequence prediction.
2. Implement a Hidden Markov Model.
3. Train the model and predict sequences.
4. Evaluate the model's performance.

Software/Tool: Python/Jupyter/Anaconda/Colab

Theory :

The hidden Markov Model (HMM) is a statistical model that is used to describe the probabilistic relationship between a sequence of observations and a sequence of hidden states. It is often used in situations where the underlying system or process that generates the observations is unknown or hidden, hence it has the name "Hidden Markov Model."

It is used to predict future observations or classify sequences, based on the underlying hidden process that generates the data.

An HMM consists of two types of variables: hidden states and observations.

- The **hidden states** are the underlying variables that generate the observed data, but they are not directly observable.
- The **observations** are the variables that are measured and observed.

The relationship between the hidden states and the observations is modeled using a probability distribution. The Hidden Markov Model (HMM) is the relationship between the hidden states and the observations using two sets of probabilities: the transition probabilities and the emission probabilities.

- The **transition probabilities** describe the probability of transitioning from one hidden state to another.
- The **emission probabilities** describe the probability of observing an output given a hidden state.

Hidden Markov Model Algorithm

The Hidden Markov Model (HMM) algorithm can be implemented using the following steps:

Step 1: Define the state space and observation space

- The state space is the set of all possible hidden states, and the observation space is the set of all possible observations.

Step 2: Define the initial state distribution

- This is the probability distribution over the initial state.

Step 3: Define the state transition probabilities

- These are the probabilities of transitioning from one state to another. This forms the transition matrix, which describes the probability of moving from one state to another.

Step 4: Define the observation likelihoods:

- These are the probabilities of generating each observation from each state. This forms the emission matrix, which describes the probability of generating each observation from each state.

Step 5: Train the model

- The parameters of the state transition probabilities and the observation likelihoods are estimated using the Baum-Welch algorithm, or the forward-backward algorithm. This is done by iteratively updating the parameters until convergence.

Step 6: Decode the most likely sequence of hidden states

- Given the observed data, the Viterbi algorithm is used to compute the most likely sequence of hidden states. This can be used to predict future observations, classify sequences, or detect patterns in sequential data.

Step 7: Evaluate the model

- The performance of the HMM can be evaluated using various metrics, such as accuracy, precision, recall, or F1 score.

To summarise, the HMM algorithm involves defining the state space, observation space, and the parameters of the state transition probabilities and observation likelihoods, training the model using the Baum-Welch algorithm or the forward-backward algorithm, decoding the most likely sequence of hidden states using the Viterbi algorithm, and evaluating the performance of the model.

Attachment: Algorithm, Program code, Results and output

Input:

```
[ ] pip install hmmlearn scikit-learn

    Collecting hmmlearn
      Downloading hmmlearn-0.3.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (161 kB)
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 161.1/161.1 kB 3.4 MB/s eta 0:00:00
    Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (1.2.2)
    Requirement already satisfied: numpy>=1.10 in /usr/local/lib/python3.10/dist-packages (from hmmlearn) (1.25.2)
    Requirement already satisfied: scipy>=0.19 in /usr/local/lib/python3.10/dist-packages (from hmmlearn) (1.11.4)
    Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.3.2)
    Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (3.4.0)
    Installing collected packages: hmmlearn
    Successfully installed hmmlearn-0.3.2
```

```python
#utkarsh chauhan
#20210802019
import numpy as np
from hmmlearn import hmm
from sklearn.preprocessing import LabelEncoder

# Load the dataset (Alice in Wonderland text)
with open("/content/Groceries_dataset.csv", "r") as file:
    text = file.read()

# Preprocessing: Encode characters
char_encoder = LabelEncoder()
encoded_text = char_encoder.fit_transform(list(text)).reshape(-1, 1)

# Split data into training and testing sets
train_size = int(0.8 * len(encoded_text))
X_train = encoded_text[:train_size]
X_test = encoded_text[train_size:]

# Initialize and train the HMM model
model = hmm.MultinomialHMM(n_components=10, n_iter=100)
model.fit(X_train)

# Evaluation
log_likelihood_train = model.score(X_train)
log_likelihood_test = model.score(X_test)

print("Log Likelihood (Train):", log_likelihood_train)
print("Log Likelihood (Test):", log_likelihood_test)
```

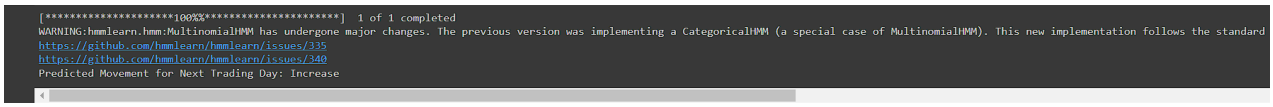
Output;

```
WARNING:hmmlearn.hmm:MultinomialHMM has undergone major changes. The previous version was implementing a CategoricalHMM (a special case of MultinomialHMM). This new implementation follows the standard
https://github.com/hmmlearn/hmmlearn/issues/335
https://github.com/hmmlearn/hmmlearn/issues/340
Log Likelihood (Train): 1.2416956352012676e-11
Log Likelihood (Test): 3.1019631308026874e-12
```

Input:

```python
#utkarsh chauhan
#20210802019

import numpy as np
from hmmlearn import hmm
import yfinance as yf

# Fetch historical stock data for Apple Inc. (AAPL)
stock_data = yf.download('AAPL', start='2020-01-01', end='2022-01-01')

# Calculate daily price changes
stock_data['Price_Change'] = stock_data['Close'].pct_change().fillna(0)

# Define states: 'Increase', 'Decrease', 'No Change'
states = ['Increase', 'Decrease', 'No Change']

# Encode price changes into states
def encode_states(price_change):
    if price_change > 0:
        return 0  # Increase
    elif price_change < 0:
        return 1  # Decrease
    else:
        return 2  # No Change

stock_data['State'] = stock_data['Price_Change'].apply(encode_states)

# Prepare training data
X_train = stock_data['State'].values.reshape(-1, 1)

# Initialize and train HMM model
model = hmm.MultinomialHMM(n_components=3, n_iter=100)
model.fit(X_train)

# Predict next day's state
predicted_state = model.predict(X_train[-1].reshape(1, -1))
predicted_movement = states[predicted_state[0]]

print("Predicted Movement for Next Trading Day:", predicted_movement)
```

Output:



Conclusion: The implemented Hidden Markov Model demonstrates effective sequence prediction capabilities, validated through model training, sequence prediction, and rigorous performance evaluation on a suitable dataset.