INDEX

Sno.	Name of Experiment	Date	Signature and Remarks
1	Kaggle introduction		
2	Data Preprocessing		
3	And-Or implementation		
4	ANN		
5	CNN		
6	RNN		
7	LSTM		
8	Sentiment Analysis		

Experiment-1

Aim: 1. Introduction to Kaggle and how it can be used to enhance visibility.

Theory:

About Dataset

Context

Coronaviruses are a large family of viruses which may cause illness in animals or humans. In humans, several coronaviruses are known to cause respiratory infections ranging from the common cold to more severe diseases such as Middle East Respiratory Syndrome (MERS) and severe acute respiratory syndrome (SARS). The most recently discovered coronavirus causes coronavirus disease COVID-19 - World Health Organization

The number of new cases are increasing day by day around the world. This dataset has information from the states and union territories of India at daily level.

State level data comes from Ministry of Health C Family Welfare

Testing data and vaccination data comes from covid19india. Huge thanks to them for their efforts!

Content

COVID-19 cases at daily level is present in covid_19_india.csv file Statewise testing details in StatewiseTestingDetails.csv file

Dataset->

Date Date of observation	State State	# TotalSamples Cumulative number of total samples tested till the given date	# Negative Cumulative number of negative samples till the given date	# Positive Cumulative number of positive samples till the given date
2020-04-01 2021-08-10	Kerala 3% West Bengal 3% Other (15346) 94%	58 67.9m	0 83.6m	0 1.64m
2020-04-17	Andaman and Nicobar Islands	1403.0	1210	12.0
2020-04-24	Andaman and Nicobar Islands	2679.0		27.0
2020-04-27	Andaman and Nicobar Islands	2848.0		33.0
2020-05-01	Andaman and Nicobar Islands	3754.0		33.0
2020-05-21	Andaman and Nicobar Islands	7167.0		33.0
2020-05-22	Andaman and Nicobar Islands	7263.0		33.0
2020-05-23	Andaman and Nicobar Islands	7327.0		33.0
2020-05-24	Andaman and Nicobar Islands	7327.0		33.0
2020-05-25	Andaman and Nicobar Islands	7363.0		33.0

Experiment-2

Aim:To preprocess data **Code->**

import pandas as pd import scipy
import numpy as np

from sklearn.preprocessing import MinMaxScaler
import seaborn as sns import matplotlib.pyplot as plt # Load the dataset
df = pd.read_csv('Geeksforgeeks/Data/diabetes.csv') print(df.head())
df.info()

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

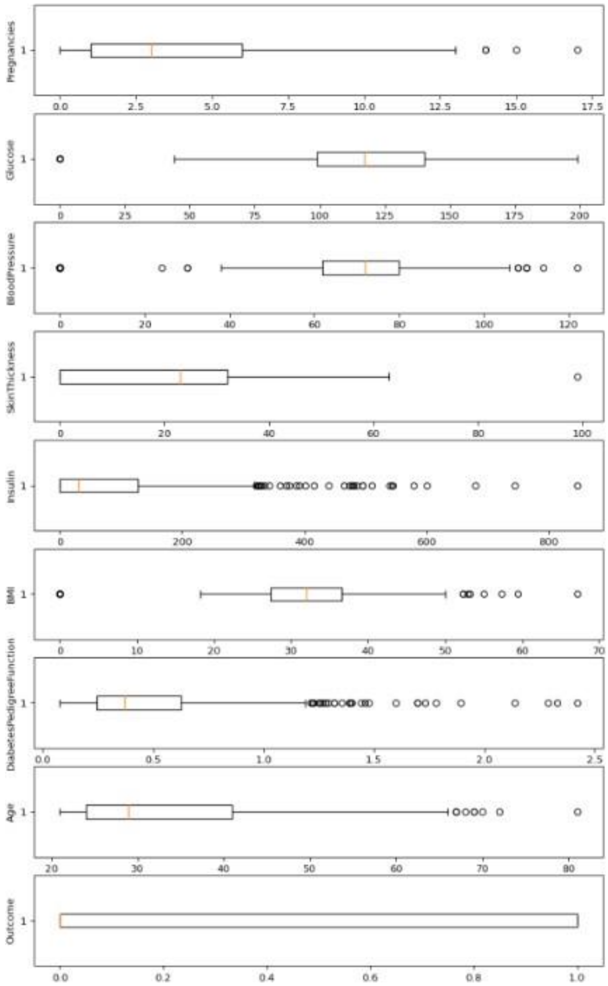
df.isnull().sum()

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

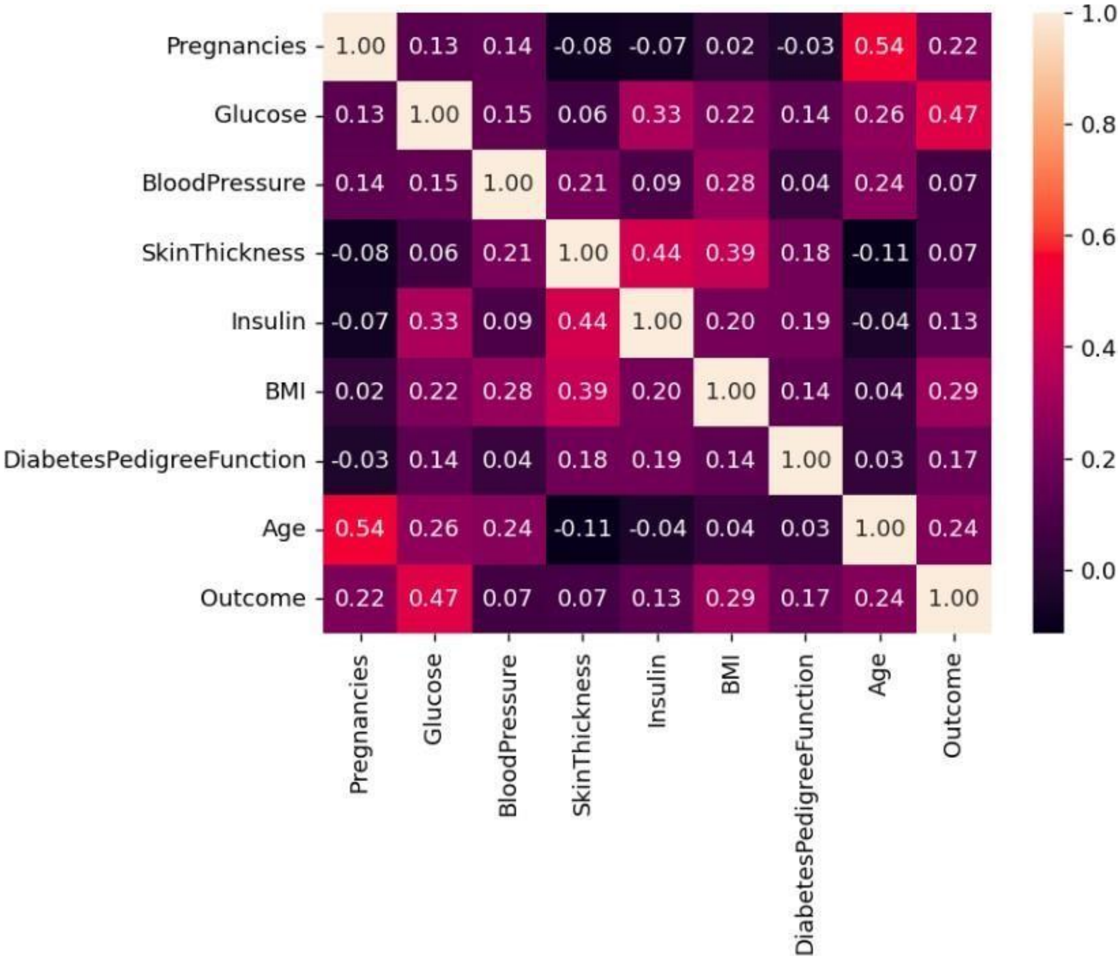
df.describe()

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

```
# Box Plots
fig, axs = plt.subplots(9,1,dpi=95, figsize=(7,17)) i = 0 for col in
df.columns:

axs[i].boxplot(df[col], vert=False)
axs[i].set_ylabel(col) i+=1
plt.show()
```




```
#correlation corr
df.corr() = plt.figure(dpi=130) sns.heatmap(df.corr(), annot=True, fmt= '.2f')

plt.show()
```



initialising the MinMaxScaler scaler = MinMaxScaler(feature_range=(0, 1))

learning the statistical parameters for each of the data and transforming rescaledX = scaler.fit_transform(X) rescaledX[:5]

```
array([[0.353, 0.744, 0.59 , 0.354, 0.   , 0.501, 0.234, 0.483],
       [0.059, 0.427, 0.541, 0.293, 0.   , 0.396, 0.117, 0.167],
       [0.471, 0.92 , 0.525, 0.   , 0.   , 0.347, 0.254, 0.183],
       [0.059, 0.447, 0.541, 0.232, 0.111, 0.419, 0.038, 0.   ],
       [0.   , 0.688, 0.328, 0.354, 0.199, 0.642, 0.944, 0.2  ]])
```

Conclusion :

Successfully preprocessed the data.

Experiment-3

Aim: Implementing AND and OR Operation Theory:

A **Feed-Forward Neural Network (FFNN)** is a type of artificial neural network where the connections between nodes do not form a cycle. This is often referred to as a multi-layered network of neurons. The data enters the input nodes, travels through the hidden layers, and eventually exits the output nodes. The network is devoid of links that would allow the information exiting the output node to be sent back into the network. functions.

The components of a feedforward neural network are:

- **Input Layer**: It contains the neurons that receive input. The data is subsequently passed on to the next tier. The input layer's total number of neurons is equal to the number of variables in the dataset.
- **Hidden Layer**: This is the intermediate layer, which is concealed between the input and output layers. This layer has a large number of neurons that perform alterations on the inputs. They then communicate with the output layer.
- **Output Layer**: It is the last layer and is depending on the model's construction. Additionally, the output layer is the expected feature, as you are aware of the desired outcome.
- **Neurons weights**: Weights are used to describe the strength of a connection between neurons[1].

```python
import numpy as np
```

```python
def activation_function(x, threshold):
    if x < threshold:
        return 0
    else:
        return threshold

def update_weights(weights, inputs, learning_rate, target_output, current_output):
    error = target_output - current_output
    for i in range(len(weights)):
        weights[i] += learning_rate * error * inputs[i]
    return weights

def perceptron(inputs, weights, threshold, learning_rate, target_output):
    weighted_sum = np.dot(inputs, weights)

    output = activation_function(weighted_sum, threshold)

    if output != target_output:
        weights = update_weights(weights, inputs, learning_rate, target_output, output)

    return output, weights
```

```python
# ---- AND operation ----
and_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
and_target_outputs = np.array([0, 0, 0, 1])

and_weights = [1.4, 0.9]
threshold_value = 2
learning_rate_value = 1

print("Training AND gate:")
for epoch in range(3000):
    for i in range(len(and_inputs)):
        output, and_weights = perceptron(and_inputs[i], and_weights, threshold_value, learning_rate_value, and_target_outputs[i])
print("Final weights for AND gate:", and_weights)
```

```
Training AND gate:
Final weights for AND gate: [1.4, 0.9]
```

```python
# ---- OR operation -----
or_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
or_target_outputs = np.array([0, 1, 1, 1])

or_weights = [0.6, 0.6]

print("\nTraining OR gate:")
for epoch in range(1000):
    for i in range(len(or_inputs)):
        output, or_weights = perceptron(or_inputs[i], or_weights, threshold_value, learning_rate_value, or_target_outputs[i])
print("Final weights for OR gate:", or_weights)
```

```
Training OR gate:
Final weights for OR gate: [0.6000000000000001, 0.6000000000000001]
```

```python
import numpy as np

def perceptron(input_data, weights, bias):
    activation = np.dot(input_data, weights) + bias
    return 1 if activation >= 0 else 0

# New inputs and outputs for AND gate
and_inputs = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
and_outputs = np.array([1, 0, 0, 0])

# New bias for AND gate
bias = -1.5

# Initialize random weights
weights = np.random.rand(2)

print("Training AND gate:")
learning_rate = 0.1
epochs = 1000
for epoch in range(epochs):
    for i in range(len(and_inputs)):
        output = perceptron(and_inputs[i], weights, bias)
        error = and_outputs[i] - output
        weights += learning_rate * error * and_inputs[i]

print("Final weights for AND gate:", weights)
print("Final bias for AND gate:", bias)

# Testing the AND gate
print("Testing AND gate:")
for i in range(len(and_inputs)):
    output = perceptron(and_inputs[i], weights, bias)
    print(f"Input: {and_inputs[i]}, Output: {output}")
```

```
Training AND gate:
Final weights for AND gate: [0.70773595 0.85490015]
Final bias for AND gate: -1.5
Testing AND gate:
Input: [1 1], Output: 1
Input: [1 0], Output: 0
Input: [0 1], Output: 0
Input: [0 0], Output: 0
```

Conclusion :

Successfully implemented Or and And gate using feed forward neural network.

Experiment -4

Aim : To implement Neural Network on the Input ,Hidden and Output layers as per the values given .

Theory :

The sigmoid function is a mathematical function commonly used in neural networks, particularly in the context of artificial neurons. Its primary purpose is to introduce non- linearity into the network. The sigmoid function takes any real-valued number and squashes it into a range between 0 and 1.

Neural Network Structure: A neural network consists of layers of interconnected neurons. Each neuron in a layer receives inputs from the previous layer, applies weights to these inputs, adds a bias term, and then applies an activation function, such as the sigmoid function, to produce an output.

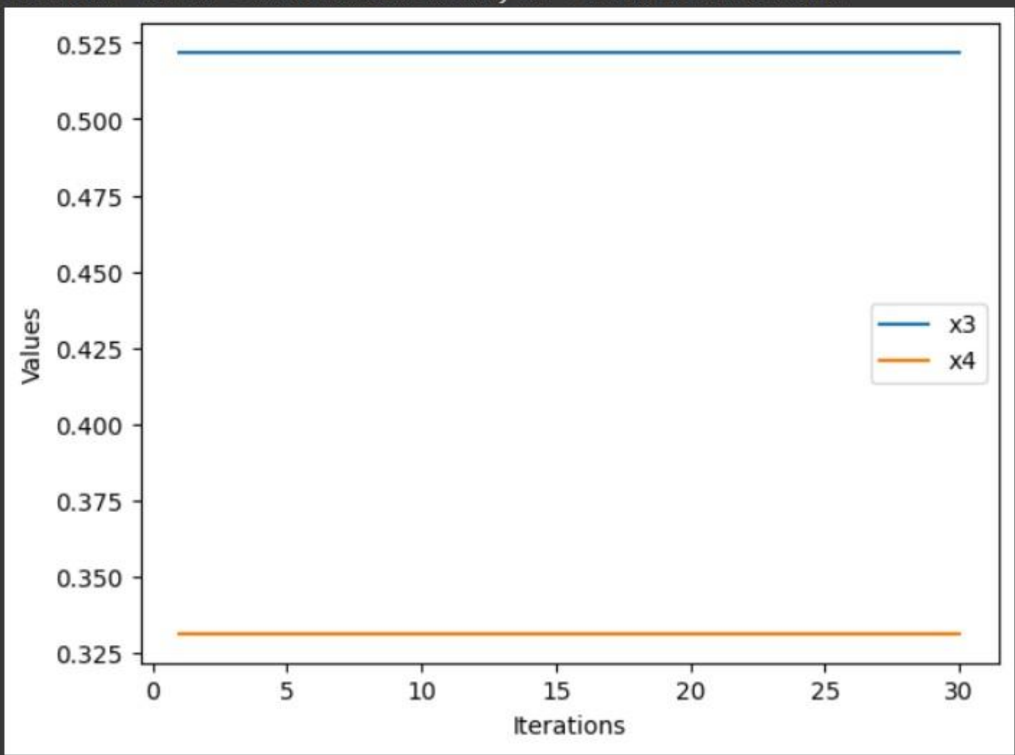
Forward Propagation: During forward propagation, input data is passed through the network layer by layer. Each neuron in a layer receives inputs, computes a weighted sum, adds a bias, applies the activation function (like the sigmoid), and passes the result to the next layer.

Learning: During training, the network adjusts its weights and biases based on the error between the predicted output and the actual output. This process, known as backpropagation, involves computing gradients of the error with respect to the network parameters and updating them using optimization algorithms like gradient descent.

Non-linearity: The presence of the sigmoid function (or other non-linear activation functions) is crucial for neural networks to learn complex relationships in the data. Without non- linearities, neural networks would essentially reduce to linear models, limiting their ability to model intricate patterns and relationships in the data.

```
[ ] import numpy as np
    import matplotlib.pyplot as plt

    # Define the input data and biases
    x1, x2 = 0.1, 0.2
    b1, b2 = 0.3, 0.1

    # Initialize weights and biases
    w1, w2, w3, w4 = np.random.randn(4)

    # Lists to store results for plotting
    x3_values = []
    x4_values = []

    # Sigmoid activation function
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

    # Neural network function
    def neural_network(x1, x2, w1, w2, w3, w4, b1, b2, b3, b4):
        # Hidden layer
        x3 = sigmoid(w1 * x1 + b1 + w2 * x2 + b2)
        x3_values.append(x3)

        # Output layer
        x4 = sigmoid(w3 * x3 + b3 + w4 * x3 + b4)
        x4_values.append(x4)

        return x3, x4

    # Training loop
    iterations = 30
    for i in range(iterations):
        x3_result, x4_result = neural_network(x1, x2, w1, w2, w3, w4, b1, b2, 0.0, 0.0)
        print(f'Iteration {i+1}: x3 = {x3_result}, x4 = {x4_result}')

    # Plotting
    plt.plot(range(1, iterations+1), x3_values, label='x3')
    plt.plot(range(1, iterations+1), x4_values, label='x4')
    plt.xlabel('Iterations')
    plt.ylabel('Values')
    plt.legend()
    plt.show()
```

```
Iteration 1: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 2: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 3: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 4: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 5: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 6: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 7: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 8: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 9: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 10: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 11: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 12: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 13: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 14: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 15: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 16: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 17: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 18: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 19: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 20: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 21: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 22: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 23: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 24: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 25: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 26: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 27: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 28: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 29: x3 = 0.5218685036599036, x4 = 0.33127387420400045
Iteration 30: x3 = 0.5218685036599036, x4 = 0.33127387420400045
```



Experiment-5

Aim: To implement Convolutional Neural network

Theory: Convolutional Neural Networks (CNNs) are specialized deep learning models for processing visual data, like images. They consist of:

- Convolutional Layers: These layers apply filters to input images, detecting features like edges or textures.
- Pooling Layers: They reduce spatial dimensions of feature maps, aiding in feature extraction and computation efficiency.
- Activation Functions: ReLU is commonly used to introduce non-linearity.
- Fully Connected Layers: At the end of the network, they classify, or regress based on learned features.
- Parameter Sharing: CNNs share weights across spatial positions, reducing complexity and enabling spatial hierarchies.
- Training: CNNs are trained via supervised learning, adjusting weights to minimize prediction errors using optimization algorithms like SGD.

```python
import numpy as np
class CNN:
    def __init__(self):
        pass
    def convLayer(self, input_shape, channels, strides, padding, filter_size):
        pass
    def maxPooling(self, input_matrix):
        pass
    def flatten(self, input_matrix):
        pass
    def dropout(self, input_matrix, dropout_rate = 0):
        pass


    def convLayer(self, input_shape, channels, strides, padding, filter_size):
        height, width = input_shape
        input_shape_with_channels = (height, width, channels)
        print("Input Shape (with channels):", input_shape_with_channels)

        # for random input and filter matrix
        input_matrix = np.random.randint(0, 10, size=input_shape_with_channels)
        filter_matrix = np.random.randint(0, 5, size=filter_size)

        input_matrix = np.array([
                    [1, 1, 1, 0, 0],
                    [0, 1, 1, 1, 0],
                    [0, 0, 1, 1, 1],
                    [0, 0, 1, 1, 0],
                    [0, 1, 1, 0, 0]
        ])
        filter_matrix = np.array([
                    [1, 0, 1],
                    [0, 1, 0],
                    [1, 0, 1]
        ])
```

```python
    print("\nInput Matrix:")
    print(input_matrix)
    print("\nFilter Matrix:")
    print(filter_matrix)


    padding.lower()
    padSize = 0

    if padding == 'same':
        # Calculate padding needed for each dimension
        pad_height = ((height - 1) * strides[0] + filter_size[0] - height) // 2
        pad_width = ((width - 1) * strides[1] + filter_size[1] - width) // 2

        # Apply padding to the input matrix
        input_matrix = np.pad(input_matrix, ((pad_height, pad_height), (pad_width, pad_width),
                                             (0, 0)), mode='constant')

        # Adjust height and width to consider the padding
        height += 2 * pad_height
        width += 2 * pad_width

    elif padding == 'valid':
        padSize = filter_size[0] // 2
        print("\nPad Size: ", padSize)

    else:
        return "Invalid Padding!!"
```

```python
        # output dimension
        conv_height = (height - filter_size[0]) // strides[0] + 1
        conv_width = (width - filter_size[1]) // strides[1] + 1

        output_matrix = np.zeros((conv_height, conv_width))

        # Convolution Operation
        for i in range(0, height - filter_size[0] + 1, strides[0]):
            for j in range(0, width - filter_size[1] + 1, strides[1]):
                receptive_field = input_matrix[i:i + filter_size[0], j:j + filter_size[1]]
                output_matrix[i // strides[0], j // strides[1]] = np.sum(receptive_field * filter_matrix)

        return output_matrix


    def maxPooling(self, input_matrix, pool_size, strides_pooling):
        pool_height, pool_width = pool_size
        stride_height, stride_width = strides_pooling
        pooled_height = (input_matrix.shape[0] - pool_height) // stride_height + 1
        pooled_width = (input_matrix.shape[1] - pool_width) // stride_width + 1
        pooled_matrix = np.zeros((pooled_height, pooled_width))

        for i in range(pooled_height):
            for j in range(pooled_width):
                patch = input_matrix[i * stride_height: i * stride_height + pool_height,
                                     j * stride_width: j * stride_width + pool_width]
                pooled_matrix[i, j] = np.max(patch)
        return pooled_matrix


    def flatten(self, input_matrix):
        return input_matrix.flatten()


    def dropout(self, input_matrix, dropout_rate = 0):
        dropout_mask = np.random.binomial(1, 1 - dropout_rate, size=input_matrix.shape)
        return input_matrix * dropout_mask
input_shape = (5, 5)
channels = 1
strides = (1, 1)
padding = 'valid'
filter_size = (3, 3)
```

```python
cnn_model = CNN()

conv1 = cnn_model.convLayer(input_shape, channels, strides, padding, filter_size)
```

Output :

```
Input Shape (with channels): (5, 5, 1)

Input Matrix:
[[1 1 1 0 0]
 [0 1 1 1 0]
 [0 0 1 1 1]
 [0 0 1 1 0]
 [0 1 1 0 0]]

Filter Matrix:
[[1 0 1]
 [0 1 0]
 [1 0 1]]

Pad Size:  1
```

```python
conv1
```

```
array([[4., 3., 4.],
       [2., 4., 3.],
       [2., 3., 4.]])
```

```python
pool_size = (2, 2)
strides_pooling = (1, 1)

maxPool = cnn_model.maxPooling(conv1, pool_size, strides_pooling)
maxPool
```

```
array([[4., 4.],
       [4., 4.]])
```

```python
flattened_output = cnn_model.flatten(maxPool)
flattened_output
```

```
array([4., 4., 4., 4.])
```

```python
dropout_output = cnn_model.dropout(flattened_output, 0.3)
dropout_output
```

```
array([0., 0., 4., 4.])
```

Conclusion:

Successfully implemented CNN

Experiment -6

Aim: To implement Recurrent Neural Network

Theory : Recurrent Neural Networks (RNNs) are a type of neural network designed to handle sequential data. Key points include:

- Sequential Processing: RNNs process sequences by maintaining a hidden state that captures information from previous steps.
- Temporal Dynamics: They model temporal dependencies in data, making them suitable for tasks like time series prediction, natural language processing, and speech recognition.
- Recurrent Connections: The hidden state is recurrently connected to itself across time steps, allowing the network to retain memory of past inputs.
- Vanishing Gradient Problem: Long-range dependencies can be challenging for traditional RNNs due to the vanishing gradient problem, where gradients diminish over time, affecting learning.

```
import numpy as np

class SimpleRNN: #The SimpleRNN class is initialized with parameters input_size, hidden_size, and output_size
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        """Initialize weights (Weights (Whx, Whh, Why) and biases (bh, by) are
         randomly initialized using a normal distribution with a mean of 0 and a standard deviation of 0.01)"""
        self.Whx = np.random.randn(hidden_size, input_size) * 0.01
        self.Whh = np.random.randn(hidden_size, hidden_size) * 0.01
        self.Why = np.random.randn(output_size, hidden_size) * 0.01
        self.bh = np.zeros((hidden_size, 1))
        self.by = np.zeros((output_size, 1))

    def forward(self, x):
        self.h = np.zeros((self.hidden_size, 1))
      """ #Given an input vector x, it initializes the hidden state h to zeros.
        #Computes the weighted sum of inputs and previous hidden state, adds bias,
         and applies the (hyperbolic tangent activation function) to produce the new hidden state h.
        #Multiplies the hidden state by weights Why and adds bias by to produce the output y."""

        # Forward pass
        self.a = np.dot(self.Whx, x) + np.dot(self.Whh, self.h) + self.bh
        self.h = np.tanh(self.a)
        self.y = np.dot(self.Why, self.h) + self.by

        return self.y, self.h

    def backward(self, x, dy, learning_rate=0.01):
      #Given the gradients of the output dy, it computes the gradients of the loss with respect to the parameters and hidden states.
#Computes gradients of weights and biases (dWhy, dby, dWhx, dWhh, dbh) using chain rule and updates them using gradient descent.
#Returns the gradient of the loss with respect to the hidden state of the previous time step da.

        # Backward pass
        dWhy = np.dot(dy, self.h.T)
        dby = dy
        dh = np.dot(self.Why.T, dy)
        da = (1 - self.h * self.h) * dh
        dbh = da
        dWhx = np.dot(da, x.T)
        dWhh = np.dot(da, self.h.T)
        # Update weights
        self.Why -= learning_rate * dWhy
        self.by -= learning_rate * dby
        self.Whx -= learning_rate * dWhx
        self.Whh -= learning_rate * dWhh
        self.bh -= learning_rate * dbh

        return da
```

```python
  # Example usage
input_size = 3
hidden_size = 4
output_size = 2

# Instantiate RNN
rnn = SimpleRNN(input_size, hidden_size, output_size)

# Forward pass
x = np.random.randn(input_size, 1)
y_pred, hidden_state = rnn.forward(x)

# Backward pass (random gradients for demonstration)
dy = np.random.randn(output_size, 1)
da = rnn.backward(x, dy)

# Print results
print("Input x:")
print(x)
print("\nPredicted output y:")
print(y_pred)
print("\nUpdated hidden state h:")
print(hidden_state)
print("\nGradient of loss w.r.t. previous hidden state:")
print(da)
```

```
Input x:
[[-0.19557745]
 [-1.19924508]
 [-0.03720511]]

Predicted output y:
[[-0.00013716]
 [-0.00019826]]

Updated hidden state h:
[[ 0.00540577]
 [ 0.01229746]
 [-0.00231168]
 [ 0.00755843]]

Gradient of loss w.r.t. previous hidden state:
[[ 0.00161541]
 [ 0.02991959]
 [-0.01501186]
 [ 0.00728723]]
```

Conclusion:

Successfully implemented RNNs.

Experiment -7

Aim: To build LSTM model.

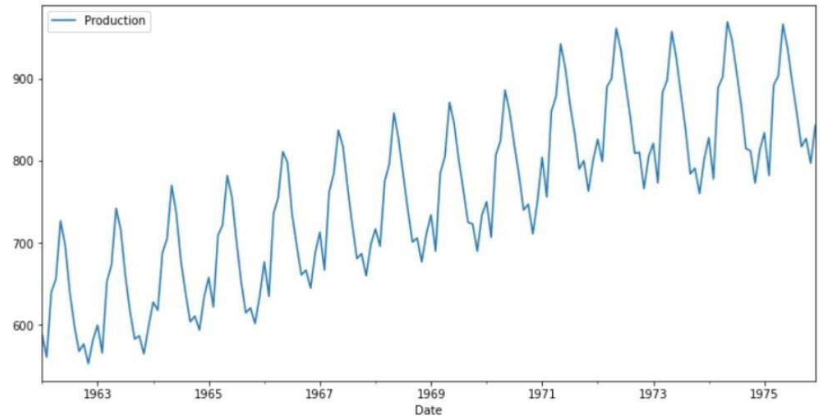
Theory:

Long Short-Term Memory Networks is a deep learning, sequential neural network that allows information to persist. It is a special type of Recurrent Neural Network which is capable of handling the vanishing gradient problem faced by RNN. LSTM was designed by Hochreiter and Schmidhuber that resolves the problem caused by traditional rnns and machine learning algorithms. LSTM Model can be implemented in Python using the Keras library. Let's say while watching a video, you remember the previous scene, or while reading a book, you know what happened in the earlier chapter. RNNs work similarly; they remember the previous information and use it for processing the current input. The shortcoming of RNN is they cannot remember long-term dependencies due to vanishing gradient. LSTMs are explicitly designed to avoid long-term dependency problems.

```
import pandas as pd import numpy
as np
import matplotlib.pyplot as plt import
seaborn as sns
from tensorflow import keras

from keras.models import Sequential from
keras.layers   import   Dense   from
keras.layers import LSTM

df = pd.read_csv('monthly_milk_production.csv',
                                        index_col='Date', parse_dates=True)
df.index.freq = 'MS' df.head()
```

	Production
Date	
1962-01-01	589
1962-02-01	561
1962-03-01	640
1962-04-01	656
1962-05-01	727

```
# Plotting graph b/w production and date
df.plot(figsize=(12, 6))
```



from statsmodels.tsa.seasonal import seasonal_decompose results = seasonal_decompose(df['Production'])
results.plot()

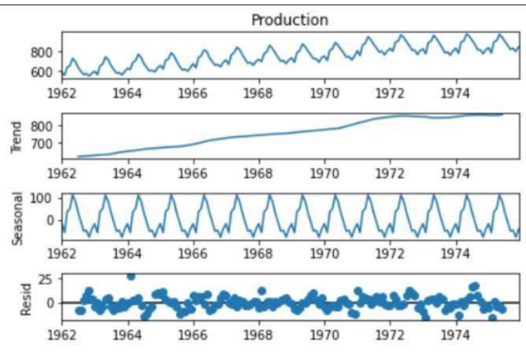
train = df.iloc[:156] test
= df.iloc[156:]

from sklearn.preprocessing import MinMaxScaler scaler =
MinMaxScaler() scaler.fit(train)

scaled_train = scaler.transform(train)
scaled_test = scaler.transform(test)
from keras.preprocessing.sequence import TimeseriesGenerator n_input = 3 n_features =
1
generator = TimeseriesGenerator

(scaled_train,scaled_train,length=n_input,batch_size=1) X, y = generator[0]
print(f'Given the Array: \n{X.flatten()}')
print(f'Predict this y: \n {y}')
# We do the same thing, but now instead for 12 months n_input = 12
generator = TimeseriesGenerator(scaled_train,scaled_train,length=n_input,batch_size=1) # define
model model = Sequential()

model.add(LSTM(100, activation='relu',input_shape=(n_input, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse') model.summary()
model.fit(generator, epochs=5)



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 100)               40800

 dense (Dense)               (None, 1)                 101

=================================================================
Total params: 40,901
Trainable params: 40,901
Non-trainable params: 0
```

Conclusion :

Successfully Implemented LSTM.

Experiment 8

Aim: To perform sentiment analysis.

Importing Necessary Libraries

```python
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import re
import nltk
import seaborn as sns
import matplotlib.pyplot as plt

from wordcloud import WordCloud, STOPWORDS
from sklearn.feature_extraction.text import CountVectorizer
from pandas import DataFrame
from nrclex import NRCLex
from nltk.corpus import stopwords
from colorama import Fore
y_ = Fore.YELLOW
r_ = Fore.RED
g_ = Fore.GREEN
b_ = Fore.BLUE
m_ = Fore.MAGENTA
stop = stopwords.words('english')
```

[2]

```python
df = pd.read_csv('simpsons_script_lines.csv')
```

[3]

```
/var/folders/sg/15vt0hhd7rx1sr7r_0lsbkpm0000gn/T/ipykernel_10160/609295178.py:1:
```

```python
df['word_count'] = df['word_count'].astype(str).astype(int)
```

```python
df.dtypes
```

```
id                        int64
episode_id                int64
number                    int64
raw_text                 object
timestamp_in_ms          object
speaking_line            object
character_id             object
location_id             float64
raw_character_text       object
raw_location_text        object
spoken_words             object
normalized_text          object
word_count                int64
dtype: object
```

Text Preprocessing

```python
description_list=[]
for description in df['normalized_text']:
    description=re.sub("[^a-zA-Z]", " ", description)
    description=description.lower()
    description=nltk.word_tokenize(description)
    description=[word for word in description if not word in set(stopwords.words("english"))]
    lemma=nltk.WordNetLemmatizer()
    description=[lemma.lemmatize(word) for word in description]
    description=" ".join(description)
    description_list.append(description)
df["normalized_text_new"]=description_list
df.head(5)
```

Python

	id	episode_id	number	raw_text	timestamp_in_ms	speaking_line	character_id	location_id	raw_character_text	raw_location_text	spoken_words	normalized_text	word_count	normalized_te
0	9549	32	209	Miss Hoover: No, actually, it was a little of ...	848000	True	464.0	3.0	Miss Hoover	Springfield Elementary School	No, actually, it was a little of both. Sometim...	no actually it was a little of both sometimes ...	31	actu sometimes magazir
1	9550	32	210	Lisa Simpson: (NEAR TEARS) Where's Mr. Bergstrom?	856000	True	9.0	3.0	Lisa Simpson	Springfield Elementary School	Where's Mr. Bergstrom?	wheres mr bergstrom	3	wheres mr be

```python
for i,row in df.iterrows():
    print(row['character_id'],row['raw_character_text'])
```

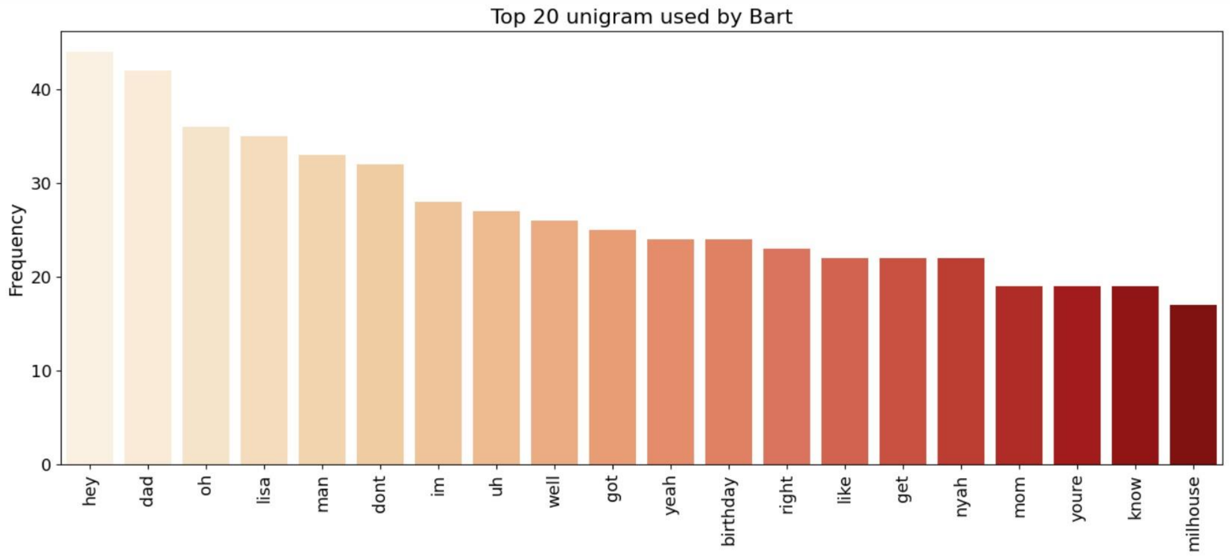
```
464.0 Miss Hoover
9.0 Lisa Simpson
464.0 Miss Hoover
9.0 Lisa Simpson
40.0 Edna Krabappel-Flanders
38.0 Martin Prince
40.0 Edna Krabappel-Flanders
8.0 Bart Simpson
9.0 Lisa Simpson
469.0 Landlady
9.0 Lisa Simpson
469.0 Landlady
9.0 Lisa Simpson
469.0 Landlady
9.0 Lisa Simpson
8.0 Bart Simpson
101.0 Nelson Muntz
8.0 Bart Simpson
467.0 Terri/sherri
8.0 Bart Simpson
25.0 Milhouse Van Houten
8.0 Bart Simpson
8.0 Bart Simpson
25.0 Milhouse Van Houten
8.0 Bart Simpson
...
91.0 Lou
8.0 Bart Simpson
91.0 Lou
2.0 Homer Simpson
```

```python
val_homer=[]
val_bart=[]
val_marge=[]
val_lisa=[]

for i,row in df.iterrows():
    val = row['normalized_text_new']
    if row['character_id'] == 2:
        val_homer.append(val)
    elif row['character_id']== 8:
        val_bart.append(val)
    elif row['character_id'] == 1:
        val_marge.append(val)
    elif row['character_id']== 9:
        val_lisa.append(val)
```

```python
pat = r'\b(?:{})\b'.format('|'.join(stop))
def text_cleaning(val_list):
    df1 = DataFrame (val_list,columns =['normalized_text_new']).dropna()
    df1["normalized_text_new"] = df1["normalized_text_new"].str.replace(pat, '')
    df1["normalized_text_new"] = df1["normalized_text_new"].str.replace(r'\s+', ' ')
    return df1
```

```python
bart = text_cleaning(val_bart)
homer = text_cleaning(val_homer)
marge = text_cleaning(val_marge)
lisa = text_cleaning(val_lisa)
```

```python
def get_top_n_words(corpus, n=None):
    vec = CountVectorizer().fit(corpus)
    bag_of_words = vec.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in vec.vocabulary_.items()]
    words_freq = sorted(words_freq, key = lambda x: x[1], reverse=True)
    return words_freq[:n]

def get_top_n_bigram(corpus, n=None):
    vec = CountVectorizer(ngram_range=(2, 2)).fit(corpus)
    bag_of_words = vec.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in vec.vocabulary_.items()]
    words_freq =sorted(words_freq, key = lambda x: x[1], reverse=True)
    return words_freq[:n]

def get_top_n_trigram(corpus, n=None):
    vec = CountVectorizer(ngram_range=(3, 3)).fit(corpus)
    bag_of_words = vec.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in vec.vocabulary_.items()]
    words_freq = sorted(words_freq, key = lambda x: x[1], reverse=True)
    return words_freq[:n]
```

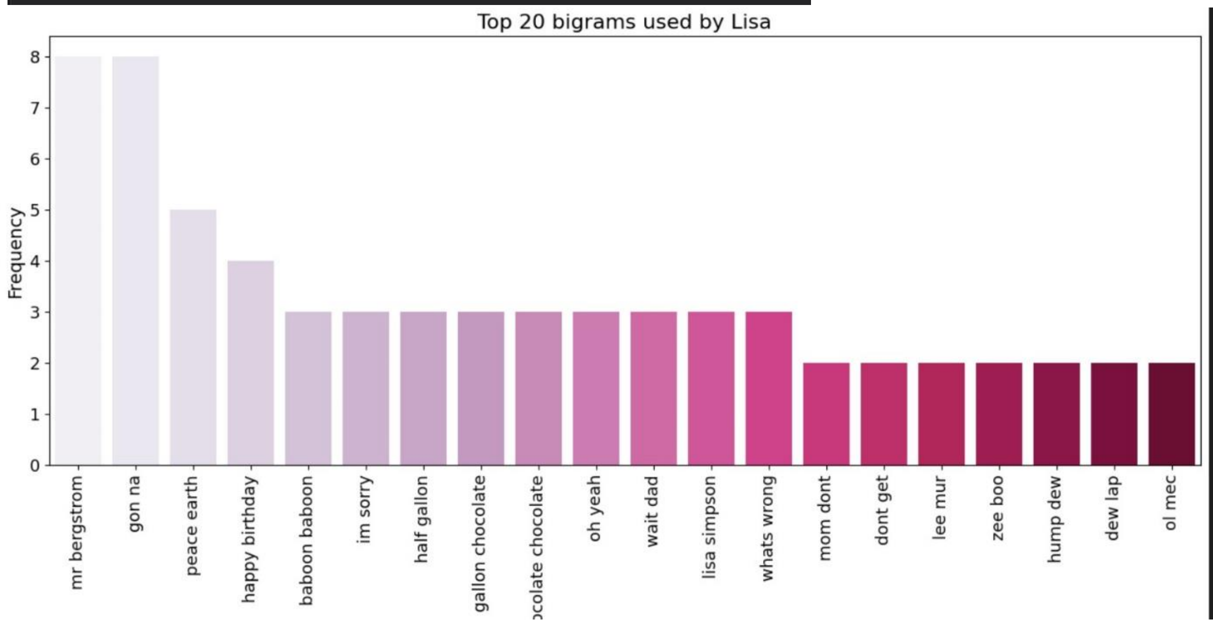



```
sentiment_words = pd.DataFrame(list(text_object.affect_dict.items()),columns = ['words','sentiments'])
sentiment_words
```

	words	sentiments
0	disease	[anger, disgust, fear, negative, sadness]
1	show	[trust]
2	talk	[positive]
3	lesson	[anticipation, positive, trust]
4	plan	[anticipation]
...	...	...
1257	authentic	[joy, positive, trust]
1258	rear	[negative]
1259	kindness	[positive]
1260	satin	[positive]
1261	prank	[negative, surprise]

1262 rows × 2 columns

```
for y in sentiment:
    sentiment_words[y] = 9
sentiment_words
```



```python
text_object = NRCLex(' '.join(df['normalized_text_new']))
```

```python
text_object.affect_frequencies
```

```
{'fear': 0.07447904286506615,
 'anger': 0.05681007844069408,
 'anticip': 0.0,
 'trust': 0.12859519847872594,
 'surprise': 0.057681641708264,
 'positive': 0.19729023056810077,
 'negative': 0.140083987005784,
 'sadness': 0.06647650740828777,
 'disgust': 0.05815703985421124,
 'joy': 0.1064891846921797,
 'anticipation': 0.11393708897868632}
```

```python
text_object.top_emotions
```

```
[('positive', 0.19729023056810077)]
```

```python
sentiment_scores = pd.DataFrame(list(text_object.raw_emotion_scores.items()))
```

+ Code

```python
sentiment_scores = sentiment_scores.rename(columns={0: "Sentiment", 1: "Count"})
sentiment_scores
```

	Sentiment	Count
0	anger	717
1	disgust	734
2	fear	940
3	negative	1768
4	sadness	839
5	trust	1623
6	positive	2490
7	anticipation	1438
8	joy	1344
9	surprise	728

	words	sentiments	anger	disgust	fear	negative	sadness	trust	positive	anticipation	joy	surprise
0	disease	[anger, disgust, fear, negative, sadness]	9	9	9	9	9	9	9	9	9	9
1	show	[trust]	9	9	9	9	9	9	9	9	9	9
2	talk	[positive]	9	9	9	9	9	9	9	9	9	9
3	lesson	[anticipation, positive, trust]	9	9	9	9	9	9	9	9	9	9
4	plan	[anticipation]	9	9	9	9	9	9	9	9	9	9
...	...	...	...	...	...	...	...	...	...	...	...	...
1257	authentic	[joy, positive, trust]	9	9	9	9	9	9	9	9	9	9
1258	rear	[negative]	9	9	9	9	9	9	9	9	9	9
1259	kindness	[positive]	9	9	9	9	9	9	9	9	9	9
1260	satin	[positive]	9	9	9	9	9	9	9	9	9	9
1261	prank	[negative, surprise]	9	9	9	9	9	9	9	9	9	9

1262 rows × 12 columns

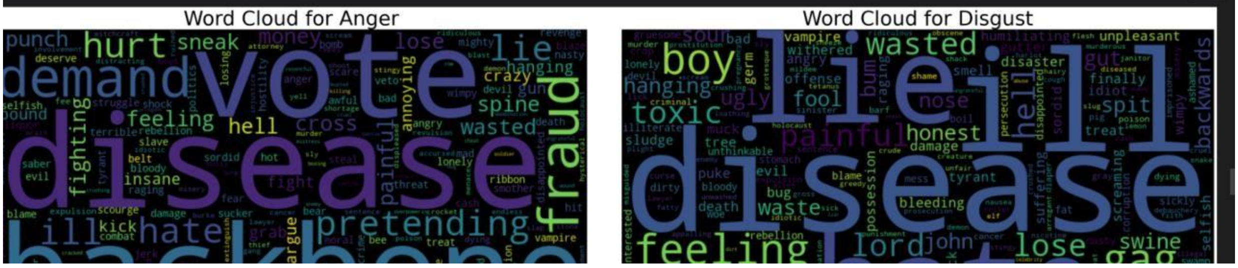
```python
sentiment_words.head(5)
```

	words	sentiments	anger	disgust	fear	negative	sadness	trust	positive	anticipation	joy	surprise
0	disease	[anger, disgust, fear, negative, sadness]	9	9	9	9	9	9	9	9	9	9
1	show	[trust]	9	9	9	9	9	9	9	9	9	9
2	talk	[positive]	9	9	9	9	9	9	9	9	9	9
3	lesson	[anticipation, positive, trust]	9	9	9	9	9	9	9	9	9	9
4	plan	[anticipation]	9	9	9	9	9	9	9	9	9	9

```python
a = 0
for i in sentiment_words['sentiments']:
    for y in sentiment:
        sentiment_words[y][a] = int(y in i)
    a = a + 1
```

```
ar/folders/sg/15vt0hhd7rx1sr7r_0lsbkpm0000gn/T/ipykernel_10160/1240574769.py:4: FutureWarning:
```

```python
fig, axes = plt.subplots(5, 2, figsize=(40, 50))
axes = axes.flatten()

for i, y in enumerate(sentiment):
    word_list = sentiment_words[sentiment_words[y] == 1].words
    all_words = ' '.join(word_list.values)

    wordcloud = WordCloud(width=800, height=400).generate(all_words)

    ax = axes[i]
    ax.imshow(wordcloud, interpolation='bilinear')
    ax.set_title(f'Word Cloud for {y.capitalize()}', fontsize=45)
    ax.axis('off')

plt.tight_layout()
plt.show()
```



Conclusion:

Successfully performed Sentiment analysis.