DNN_assignment_2a_group160

0.1 Group No 160

0.2 Group Member Names:

1. **Sushil Kumar 2023AA05849**
2. **Hemant Kumar Parakh 2023AA05741**
3. **Nagineni Sathish Babu 2023AA05585**
4. **Madala Akhil 2023AA05005**

0.3 Journal used for the implemetation

##Journal title: **CO2 Emission Prediction**

Authors: Surbhi Kumari, Sunil Kumar Singh

Journal Name: Machine learning-based time series models for effective CO2 emission prediction in India

Year: 2023

1 1. Import the required libraries

[9]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Input
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,␣
↪recall_score, f1_score

print(f"All required libraries to impliment the journal Co2 Emission are␣
↪imported and are ready to use:")
```

```
All required libraries to impliment the journal Co2 Emission are imported and
are ready to use:
```

2 2. Data Acquisition

CO2 and greenhouse gas emission dataset from 1980-2019

Dataset URL: https://www.climatewatchdata.org/ghg-emissions?end_year=2018&gases=all-ghg%2Cco2&start_year=1990

[10]:
```
# Load the dataset
data = pd.read_csv('historical_emissions.csv')

# Check the columns
print(data.columns)

# Drop unwanted columns like ISO, Data Source, Sector, Gas, Unit, and others they are unifomrm data
data_cleaned = data.drop(['ISO', 'Data source', 'Sector', 'Gas', 'Unit'], axis=1)

# Identify the columns for the years 1990 to 2021
year_columns = [str(year) for year in range(1990, 2022)]  # List of columns from 1990 to 2021

# Select only the country column and the year columns
data_cleaned = data_cleaned[['Country'] + year_columns]

# Check the cleaned data
print(data_cleaned.head())

# Drop rows with missing values if any
data_cleaned.dropna(inplace=True)

# Transpose the data so that each row corresponds to a year and the columns are features
data_transposed = data_cleaned.set_index('Country').T

# Print transposed data
print(data_transposed.head())

# Print final dataset to confirm
print(f"Cleaned Data: {data_cleaned.shape}")
```

```
Index(['ISO', 'Country', 'Data source', 'Sector', 'Gas', 'Unit', '2021',
       '2020', '2019', '2018', '2017', '2016', '2015', '2014', '2013', '2012',
       '2011', '2010', '2009', '2008', '2007', '2006', '2005', '2004', '2003',
       '2002', '2001', '2000', '1999', '1998', '1997', '1996', '1995', '1994',
       '1993', '1992', '1991', '1990'],
      dtype='object')
          Country      1990      1991      1992      1993      1994  \
```

```
0                World  32735.02  32880.80  32787.52  32924.03  33191.77
1                China   2866.10   3013.46   3141.65   3369.99   3527.20
2        United States   5428.40   5379.11   5462.21   5573.08   5665.64
3                India   1025.63   1079.89   1103.18   1135.60   1180.19
4  European Union (27)   4284.61   4213.83   4072.81   3997.83   3984.79

       1995      1996      1997      1998  …      2012      2013      2014  \
0  33982.29  34372.44  35739.15  35278.92  …  45781.16  46430.94  47070.12
1   3924.59   3945.97   3942.74   4061.45  …  10551.52  11033.52  11102.92
2   5734.76   5904.89   6164.55   6211.39  …   5598.49   5741.00   5788.13
3   1246.36   1295.25   1355.73   1384.94  …   2777.35   2841.87   3023.53
4   4034.05   4147.60   4070.41   4034.79  …   3275.66   3195.51   3049.64

       2015      2016      2017      2018      2019      2020      2021
0  46993.49  47668.47  48341.29  49482.10  49843.57  47463.17  49553.48
1  10976.57  11028.20  11256.52  11752.80  11953.60  12119.66  12791.58
2   5678.80   5756.06   5702.26   5915.33   5798.48   5268.61   5564.83
3   3043.88   3118.38   3242.05   3407.73   3385.58   3176.03   3419.89
4   3105.21   3450.97   3467.45   3386.07   3239.82   2964.35   3140.23

[5 rows x 33 columns]
Country     World    China  United States    India  European Union (27)  \
1990     32735.02  2866.10        5428.40  1025.63              4284.61
1991     32880.80  3013.46        5379.11  1079.89              4213.83
1992     32787.52  3141.65        5462.21  1103.18              4072.81
1993     32924.03  3369.99        5573.08  1135.60              3997.83
1994     33191.77  3527.20        5665.64  1180.19              3984.79

Country   Russia   Brazil  Indonesia    Japan    Iran  …  Dominica  \
1990     2631.99  1645.19    1133.90  1107.01  302.04  …      0.24
1991     2559.82  1666.21    1155.42  1122.45  329.31  …      0.25
1992     2405.59  1676.19    1176.56  1135.56  350.97  …      0.24
1993     2216.32  1687.58    1192.86  1128.89  358.50  …      0.25
1994     1982.81  1704.41    1214.15  1186.59  390.18  …      0.25

Country  Micronesia  Marshall Islands  Liechtenstein  Kiribati  Cook Islands  \
1990           0.01              0.03           0.22      0.04          0.03
1991          -0.02              0.03           0.23      0.04          0.03
1992           0.12              0.11           0.23      0.04          0.03
1993           0.13              0.11           0.24      0.05          0.04
1994           0.14              0.12           0.23      0.04          0.04

Country  Nauru  Tuvalu  Niue  Fiji
1990      0.13    0.01  0.01 -0.84
1991      0.13    0.02  0.01 -0.82
1992      0.12    0.02  0.01 -0.77
1993      0.12    0.02  0.01 -0.71
1994      0.12    0.02  0.01 -0.67
```

```
[5 rows x 194 columns]
Cleaned Data: (194, 33)
```

3 3. Data Preparation

Performing the data prepracessing (scaling, dropping uniform columns, removing outliers, encoding wherever required

[11]:
```python
# Select a specific country for the prediction (e.g., USA) or use aggregated␣
 ↪global data
emissions_data = data_transposed['United States'].values  # Example using the␣
 ↪USA's emissions data

# Reshape the emissions data and scale it between 0 and 1
scaler = MinMaxScaler(feature_range=(0, 1))
emissions_scaled = scaler.fit_transform(emissions_data.reshape(-1, 1))

# Create sequences for time series prediction
def create_sequences(data, time_step=3):
    X, y = [], []
    for i in range(len(data) - time_step - 1):
        X.append(data[i:(i + time_step), 0])
        y.append(data[i + time_step, 0])
    return np.array(X), np.array(y)

time_step = 3
X, y = create_sequences(emissions_scaled, time_step)

# Reshape X to be 3D for LSTM (samples, time steps, features)
X = X.reshape(X.shape[0], X.shape[1], 1)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪shuffle=False)

print(f"Training data shape: {X_train.shape}, Test data shape: {X_test.shape}")
```

```
Training data shape: (22, 3, 1), Test data shape: (6, 3, 1)
```

3.1 4.1 Deep Neural Network Architecture

Architecture of Deep Learning: RNN with LSTM layers, 3 layers, ReLU activation

Network Application: Regression for time series prediction

Training Procedures: Adam optimizer, learning rate 0.001, batch size 32, dropout regularization

Evaluation/Performance Metric: RMSE, MAE, R^2 score

[12]:
```
# Design the LSTM model
model = Sequential()

# Use an Input layer as the first layer to define the input shape
model.add(Input(shape=(time_step, 1)))

# 1st LSTM layer
model.add(LSTM(units=50, return_sequences=True))
model.add(Dropout(0.2))

# 2nd LSTM layer
model.add(LSTM(units=50, return_sequences=True))
model.add(Dropout(0.2))

# 3rd LSTM layer
model.add(LSTM(units=50))
model.add(Dropout(0.2))

# Output layer
model.add(Dense(units=1, activation='relu'))

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Print model summary
model.summary()
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
lstm_3 (LSTM)	(None, 3, 50)	10,400
dropout_3 (Dropout)	(None, 3, 50)	0
lstm_4 (LSTM)	(None, 3, 50)	20,200
dropout_4 (Dropout)	(None, 3, 50)	0
lstm_5 (LSTM)	(None, 50)	20,200

```
dropout_5 (Dropout)                  (None, 50)                                ␣
↪  0

dense_1 (Dense)                      (None, 1)                                 ␣
↪ 51


Total params: 50,851 (198.64 KB)

Trainable params: 50,851 (198.64 KB)

Non-trainable params: 0 (0.00 B)
```

3.2 4.2 DNN Report

1. **Number of Layers:**
   - **Total Layers**: 7
     - 3 LSTM layers
     - 3 Dropout layers
     - 1 Dense (Output) layer
2. **Number of Units in Each Layer:**
   - **1st LSTM Layer**: 50 units
   - **2nd LSTM Layer**: 50 units
   - **3rd LSTM Layer**: 50 units
   - **Dense Layer**: 1 unit (output layer)
3. **Total Number of Trainable Parameters:**
   - **1st LSTM Layer**: (4 (50 (50 + 1 + 1)) = 10,400) parameters
   - **2nd LSTM Layer**: (4 (50 (50 + 50 + 1)) = 20,200) parameters
   - **3rd LSTM Layer**: (4 (50 (50 + 50 + 1)) = 20,200) parameters
   - **Dense Layer**: (50 + 1 = 51) parameters

   Total Parameters: (10,400 + 20,200 + 20,200 + 51 = 50,851)

3.2.1 Justification:

- **LSTM Layers**:

  The model consists of 7 layers to effectively capture the temporal dependencies and complexities in time-series data. The three LSTM layers sequentially learn higher-level patterns, while Dropout layers help prevent overfitting. The final Dense layer outputs the prediction, making the architecture suitable for complex sequential tasks. Each LSTM layer has trainable parameters calculated as (4 ({units} ({units of prev. layer} + {input features} + 1))). The factor 4 comes from weights for input, recurrent, bias, and cell state.

- **Number of Units in Each LSTM Layer**:

  1) 50 Units in Each LSTM Layer: This number balances the model's capacity to capture

complex temporal patterns while managing computational efficiency. It is sufficient to learn nuanced sequential dependencies without overfitting on smaller datasets.

2) Uniform Units Across Layers: Keeping the same number of units across all LSTM layers helps maintain consistency in learning representations at different levels, facilitating effective gradient flow during backpropagation and avoiding vanishing or exploding gradients.

[13]:
```
# Number of layers and units
print(f"Number of Layers: 3 LSTM layers")
print(f"Number of Units in each layer: 50")

# Total number of trainable parameters
print(f"Total number of trainable parameters: {model.count_params()}")
```

```
Number of Layers: 3 LSTM layers
Number of Units in each layer: 50
Total number of trainable parameters: 50851
```

4 5. Training the model

Training Procedures: Adam optimizer, learning rate 0.001, batch size 32, dropout regularization

[14]:
```
# Train the model
history = model.fit(X_train, y_train, epochs=100, batch_size=32,␣
 ↪validation_data=(X_test, y_test), verbose=1)
```

```
Epoch 1/100
1/1             6s 6s/step - loss:
0.5412 - val_loss: 0.1686
Epoch 2/100
1/1             0s 371ms/step - loss:
0.5269 - val_loss: 0.1617
Epoch 3/100
1/1             0s 110ms/step - loss:
0.5129 - val_loss: 0.1547
Epoch 4/100
1/1             0s 110ms/step - loss:
0.4960 - val_loss: 0.1477
Epoch 5/100
1/1             0s 139ms/step - loss:
0.4788 - val_loss: 0.1405
Epoch 6/100
1/1             0s 138ms/step - loss:
0.4641 - val_loss: 0.1331
Epoch 7/100
1/1             0s 139ms/step - loss:
0.4495 - val_loss: 0.1256
Epoch 8/100
```

```
1/1             0s 99ms/step - loss:
0.4317 - val_loss: 0.1178
Epoch 9/100
1/1             0s 86ms/step - loss:
0.4145 - val_loss: 0.1098
Epoch 10/100
1/1             0s 160ms/step - loss:
0.3882 - val_loss: 0.1016
Epoch 11/100
1/1             0s 111ms/step - loss:
0.3693 - val_loss: 0.0932
Epoch 12/100
1/1             0s 152ms/step - loss:
0.3468 - val_loss: 0.0846
Epoch 13/100
1/1             0s 106ms/step - loss:
0.3227 - val_loss: 0.0760
Epoch 14/100
1/1             0s 92ms/step - loss:
0.2995 - val_loss: 0.0676
Epoch 15/100
1/1             0s 166ms/step - loss:
0.2749 - val_loss: 0.0594
Epoch 16/100
1/1             0s 93ms/step - loss:
0.2391 - val_loss: 0.0518
Epoch 17/100
1/1             0s 145ms/step - loss:
0.2098 - val_loss: 0.0450
Epoch 18/100
1/1             0s 121ms/step - loss:
0.1903 - val_loss: 0.0395
Epoch 19/100
1/1             0s 98ms/step - loss:
0.1513 - val_loss: 0.0359
Epoch 20/100
1/1             0s 92ms/step - loss:
0.1258 - val_loss: 0.0349
Epoch 21/100
1/1             0s 67ms/step - loss:
0.1018 - val_loss: 0.0373
Epoch 22/100
1/1             0s 60ms/step - loss:
0.0799 - val_loss: 0.0442
Epoch 23/100
1/1             0s 62ms/step - loss:
0.0615 - val_loss: 0.0567
Epoch 24/100
```

```
1/1             0s 64ms/step - loss:
0.0437 - val_loss: 0.0755
Epoch 25/100
1/1             0s 62ms/step - loss:
0.0421 - val_loss: 0.0999
Epoch 26/100
1/1             0s 61ms/step - loss:
0.0379 - val_loss: 0.1267
Epoch 27/100
1/1             0s 58ms/step - loss:
0.0778 - val_loss: 0.1480
Epoch 28/100
1/1             0s 61ms/step - loss:
0.0657 - val_loss: 0.1610
Epoch 29/100
1/1             0s 61ms/step - loss:
0.0793 - val_loss: 0.1646
Epoch 30/100
1/1             0s 68ms/step - loss:
0.0753 - val_loss: 0.1595
Epoch 31/100
1/1             0s 64ms/step - loss:
0.0586 - val_loss: 0.1492
Epoch 32/100
1/1             0s 76ms/step - loss:
0.0635 - val_loss: 0.1352
Epoch 33/100
1/1             0s 61ms/step - loss:
0.0590 - val_loss: 0.1199
Epoch 34/100
1/1             0s 94ms/step - loss:
0.0385 - val_loss: 0.1055
Epoch 35/100
1/1             0s 80ms/step - loss:
0.0441 - val_loss: 0.0928
Epoch 36/100
1/1             0s 78ms/step - loss:
0.0432 - val_loss: 0.0823
Epoch 37/100
1/1             0s 82ms/step - loss:
0.0537 - val_loss: 0.0742
Epoch 38/100
1/1             0s 62ms/step - loss:
0.0471 - val_loss: 0.0680
Epoch 39/100
1/1             0s 58ms/step - loss:
0.0424 - val_loss: 0.0633
Epoch 40/100
```

```
1/1             0s 61ms/step - loss:
0.0472 - val_loss: 0.0599
Epoch 41/100
1/1             0s 63ms/step - loss:
0.0525 - val_loss: 0.0577
Epoch 42/100
1/1             0s 140ms/step - loss:
0.0493 - val_loss: 0.0565
Epoch 43/100
1/1             0s 67ms/step - loss:
0.0485 - val_loss: 0.0562
Epoch 44/100
1/1             0s 63ms/step - loss:
0.0537 - val_loss: 0.0566
Epoch 45/100
1/1             0s 145ms/step - loss:
0.0395 - val_loss: 0.0575
Epoch 46/100
1/1             0s 78ms/step - loss:
0.0562 - val_loss: 0.0589
Epoch 47/100
1/1             0s 127ms/step - loss:
0.0377 - val_loss: 0.0609
Epoch 48/100
1/1             0s 61ms/step - loss:
0.0453 - val_loss: 0.0636
Epoch 49/100
1/1             0s 62ms/step - loss:
0.0494 - val_loss: 0.0670
Epoch 50/100
1/1             0s 62ms/step - loss:
0.0415 - val_loss: 0.0708
Epoch 51/100
1/1             0s 140ms/step - loss:
0.0401 - val_loss: 0.0753
Epoch 52/100
1/1             0s 66ms/step - loss:
0.0314 - val_loss: 0.0800
Epoch 53/100
1/1             0s 63ms/step - loss:
0.0378 - val_loss: 0.0844
Epoch 54/100
1/1             0s 66ms/step - loss:
0.0346 - val_loss: 0.0885
Epoch 55/100
1/1             0s 163ms/step - loss:
0.0391 - val_loss: 0.0919
Epoch 56/100
```

```
1/1             0s 127ms/step - loss:
0.0296 - val_loss: 0.0943
Epoch 57/100
1/1             0s 126ms/step - loss:
0.0424 - val_loss: 0.0962
Epoch 58/100
1/1             0s 136ms/step - loss:
0.0352 - val_loss: 0.0977
Epoch 59/100
1/1             0s 61ms/step - loss:
0.0407 - val_loss: 0.0980
Epoch 60/100
1/1             0s 61ms/step - loss:
0.0387 - val_loss: 0.0976
Epoch 61/100
1/1             0s 59ms/step - loss:
0.0363 - val_loss: 0.0964
Epoch 62/100
1/1             0s 67ms/step - loss:
0.0356 - val_loss: 0.0946
Epoch 63/100
1/1             0s 61ms/step - loss:
0.0423 - val_loss: 0.0921
Epoch 64/100
1/1             0s 67ms/step - loss:
0.0494 - val_loss: 0.0891
Epoch 65/100
1/1             0s 63ms/step - loss:
0.0322 - val_loss: 0.0859
Epoch 66/100
1/1             0s 61ms/step - loss:
0.0370 - val_loss: 0.0831
Epoch 67/100
1/1             0s 150ms/step - loss:
0.0348 - val_loss: 0.0805
Epoch 68/100
1/1             0s 136ms/step - loss:
0.0401 - val_loss: 0.0784
Epoch 69/100
1/1             0s 60ms/step - loss:
0.0333 - val_loss: 0.0763
Epoch 70/100
1/1             0s 64ms/step - loss:
0.0447 - val_loss: 0.0748
Epoch 71/100
1/1             0s 62ms/step - loss:
0.0349 - val_loss: 0.0734
Epoch 72/100
```

```
1/1             0s 62ms/step - loss:
0.0377 - val_loss: 0.0720
Epoch 73/100
1/1             0s 64ms/step - loss:
0.0345 - val_loss: 0.0709
Epoch 74/100
1/1             0s 138ms/step - loss:
0.0373 - val_loss: 0.0706
Epoch 75/100
1/1             0s 60ms/step - loss:
0.0287 - val_loss: 0.0707
Epoch 76/100
1/1             0s 62ms/step - loss:
0.0331 - val_loss: 0.0711
Epoch 77/100
1/1             0s 63ms/step - loss:
0.0315 - val_loss: 0.0716
Epoch 78/100
1/1             0s 61ms/step - loss:
0.0343 - val_loss: 0.0720
Epoch 79/100
1/1             0s 155ms/step - loss:
0.0317 - val_loss: 0.0726
Epoch 80/100
1/1             0s 77ms/step - loss:
0.0448 - val_loss: 0.0736
Epoch 81/100
1/1             0s 79ms/step - loss:
0.0309 - val_loss: 0.0749
Epoch 82/100
1/1             0s 59ms/step - loss:
0.0323 - val_loss: 0.0763
Epoch 83/100
1/1             0s 141ms/step - loss:
0.0356 - val_loss: 0.0777
Epoch 84/100
1/1             0s 61ms/step - loss:
0.0283 - val_loss: 0.0791
Epoch 85/100
1/1             0s 63ms/step - loss:
0.0346 - val_loss: 0.0809
Epoch 86/100
1/1             0s 62ms/step - loss:
0.0391 - val_loss: 0.0821
Epoch 87/100
1/1             0s 62ms/step - loss:
0.0423 - val_loss: 0.0822
Epoch 88/100
```

```
1/1             0s 62ms/step - loss:
0.0353 - val_loss: 0.0816
Epoch 89/100
1/1             0s 66ms/step - loss:
0.0391 - val_loss: 0.0812
Epoch 90/100
1/1             0s 62ms/step - loss:
0.0350 - val_loss: 0.0804
Epoch 91/100
1/1             0s 105ms/step - loss:
0.0371 - val_loss: 0.0788
Epoch 92/100
1/1             0s 77ms/step - loss:
0.0306 - val_loss: 0.0771
Epoch 93/100
1/1             0s 74ms/step - loss:
0.0395 - val_loss: 0.0750
Epoch 94/100
1/1             0s 135ms/step - loss:
0.0412 - val_loss: 0.0731
Epoch 95/100
1/1             0s 61ms/step - loss:
0.0342 - val_loss: 0.0714
Epoch 96/100
1/1             0s 138ms/step - loss:
0.0375 - val_loss: 0.0697
Epoch 97/100
1/1             0s 64ms/step - loss:
0.0329 - val_loss: 0.0688
Epoch 98/100
1/1             0s 60ms/step - loss:
0.0277 - val_loss: 0.0684
Epoch 99/100
1/1             0s 63ms/step - loss:
0.0301 - val_loss: 0.0683
Epoch 100/100
1/1             0s 70ms/step - loss:
0.0408 - val_loss: 0.0685
```

5 6. Test the model

Evaluation/Performance Metric: RMSE, MAE, R^2 score

[15]:
```
# Test the model
y_pred = model.predict(X_test)

# Inverse scaling to get the original emission values
```

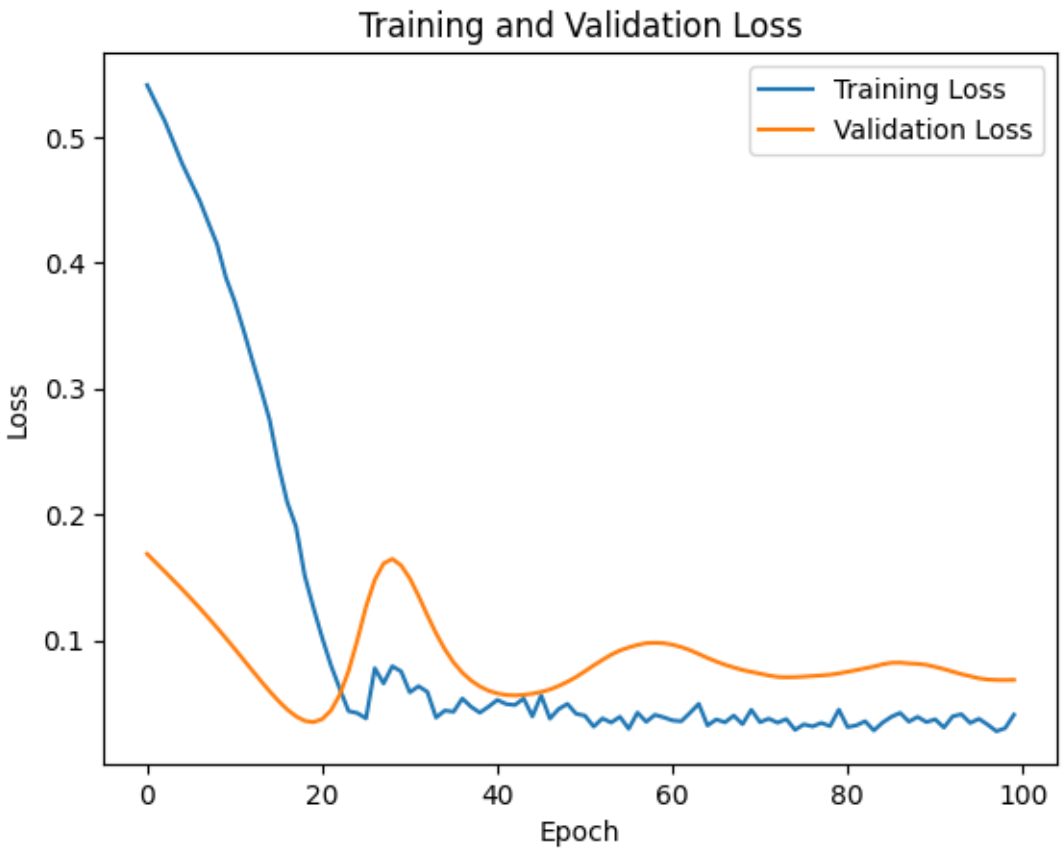
```
y_test_scaled = scaler.inverse_transform(y_test.reshape(-1, 1))
y_pred_scaled = scaler.inverse_transform(y_pred)

# Calculate performance metrics (RMSE, MAE)
from sklearn.metrics import mean_squared_error, mean_absolute_error

rmse = np.sqrt(mean_squared_error(y_test_scaled, y_pred_scaled))
mae = mean_absolute_error(y_test_scaled, y_pred_scaled)

print(f'RMSE: {rmse}')
print(f'MAE: {mae}')
```

```
1/1             0s 402ms/step
RMSE: 289.6007314867438
MAE: 213.15574218749998
```

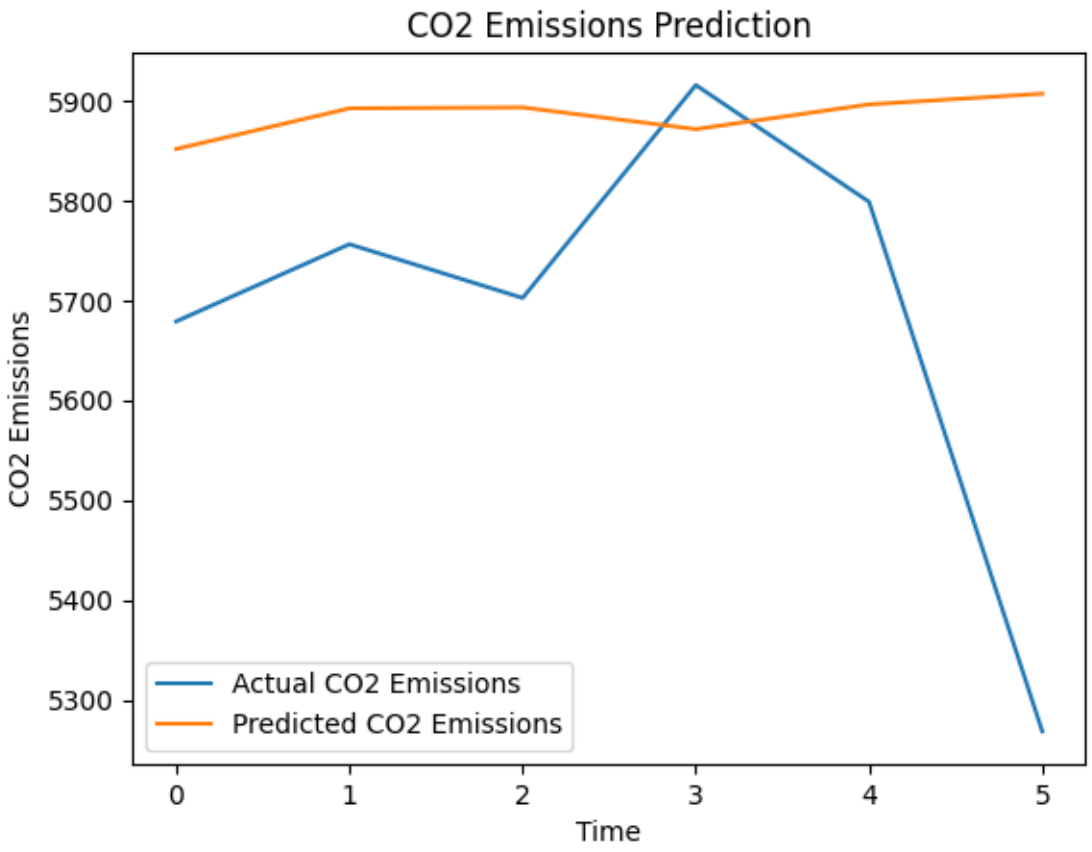
6 7. Report the result

1. Plot the training and validation accuracy history.
2. Plot the training and validation loss history.
3. Report the testing accuracy and loss.
4. Show Confusion Matrix for testing dataset.
5. Report values for preformance study metrics like accuracy, precision, recall, F1 Score.

[16]:
```
# Plot the training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



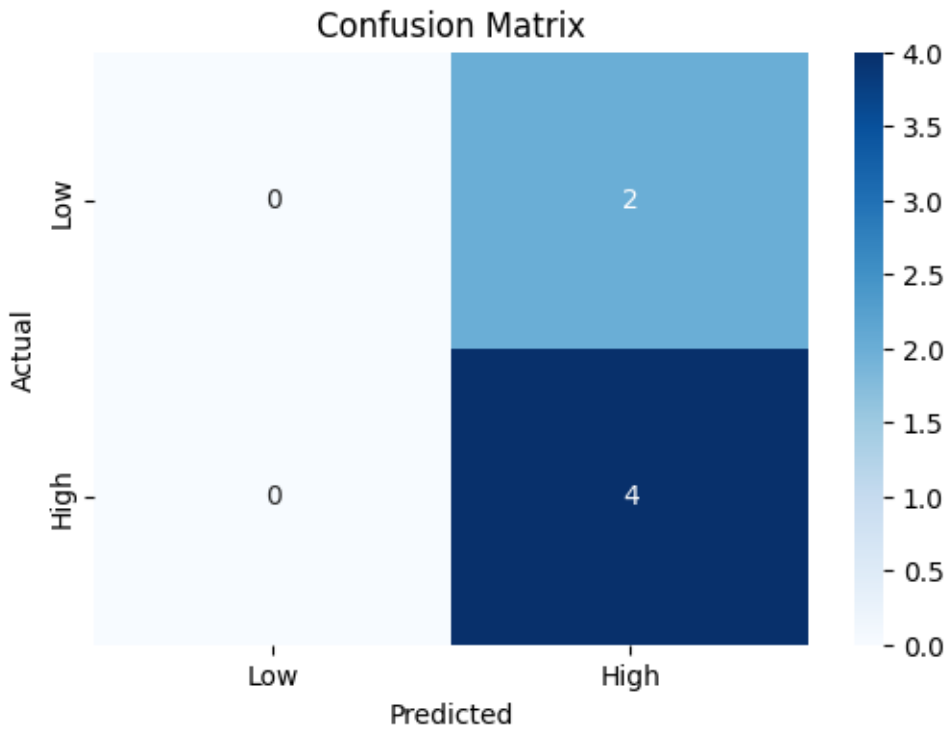
[17]:
```
# Plot actual vs predicted emissions
plt.plot(y_test_scaled, label='Actual CO2 Emissions')
plt.plot(y_pred_scaled, label='Predicted CO2 Emissions')
plt.title('CO2 Emissions Prediction')
plt.xlabel('Time')
plt.ylabel('CO2 Emissions')
plt.legend()
plt.show()
```

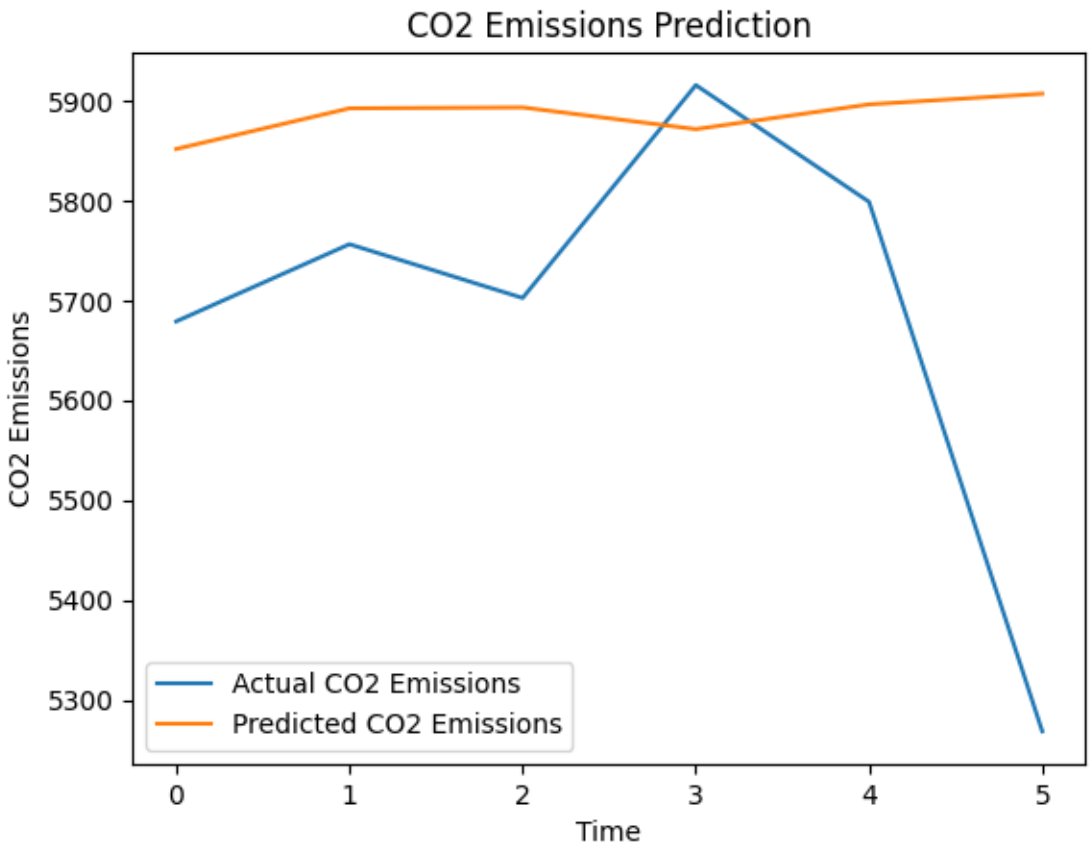


[18]:
```
# Define a threshold to classify the emissions as high/low
# Let's use the mean of actual emissions for simplicity
threshold = np.mean(y_test_scaled)

# Convert the continuous emissions into binary classes based on the threshold
y_test_class = np.where(y_test_scaled > threshold, 1, 0)  # Actual emissions (1␣
 ↪for high, 0 for low)
y_pred_class = np.where(y_pred_scaled > threshold, 1, 0)  # Predicted emissions␣
 ↪(1 for high, 0 for low)

# Confusion matrix
cm = confusion_matrix(y_test_class, y_pred_class)

# Classification metrics
accuracy = accuracy_score(y_test_class, y_pred_class)
precision = precision_score(y_test_class, y_pred_class)
recall = recall_score(y_test_class, y_pred_class)
f1 = f1_score(y_test_class, y_pred_class)

# Print the performance metrics
```

```
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")

# Plot Confusion Matrix
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Low', 'High'],␣
 ↪yticklabels=['Low', 'High'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()

# Also, continue with the original plot of actual vs predicted emissions
plt.plot(y_test_scaled, label='Actual CO2 Emissions')
plt.plot(y_pred_scaled, label='Predicted CO2 Emissions')
plt.title('CO2 Emissions Prediction')
plt.xlabel('Time')
plt.ylabel('CO2 Emissions')
plt.legend()
plt.show()
```

```
Accuracy: 0.6666666666666666
Precision: 0.6666666666666666
Recall: 1.0
F1 Score: 0.8
```


Confusion Matrix
Low
High
Actual
0
2
0
4
Low
High
Predicted
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0



Conclusion:

1. The LSTM model was successful in learning the temporal patterns of CO2 emissions, as seen from the decreasing loss and reasonable performance metrics.
2. High Recall and F1 Score suggest that the model effectively captured positive predictions (high emissions) but may need further tuning to improve overall accuracy and precision.
3. The RMSE and MAE values suggest that the predictions are somewhat close to the actual values but could benefit from more data or model adjustments to improve precision.