FINAL PROJECT REPORT

~ Unveiling Climate Change Dynamics through Earth Surface Temperature

1. INTRODUCTION

1.1. Project Overview

The project aims to comprehensively analyze climate change impacts using deep learning techniques like LSTM, empowering us to compose a future of resilience for our planet. By examining historical surface temperature data, the project seeks to uncover trends and anomalies that provide insights into the broader implications of climate change. The model addresses the complexities of spatial and temporal variability in temperature data, account for various influencing factors, and provide insights that can inform climate change mitigation and adaptation strategies.

1.2. Objectives

The main objectives include:

- Utilizing deep learning models (RNNs, GRUs, LSTMs) to predict future temperature trends.
- Providing actionable insights for policymakers, researchers, and the public regarding climate change dynamics.

2. PROJECT INITIALIZATION AND PLANNING PHASE

2.1. Define Problem Statement

Climate change profoundly affects ecosystems and societies worldwide.

This project focuses on leveraging surface temperature data to forecast future trends, crucial for developing effective mitigation and adaptation strategies, overcoming struggles of existing methods.

Problem Statement: Click Here

2.2. Project Proposal (Proposed Solution)

The project proposes using deep learning techniques to analyze surface temperature data. It involves data collection, preprocessing, model development (RNNs, GRUs, LSTMs), validation, and analysis to predict temperature trends accurately.

Project Proposal: Click Here

2.3. Initial Project Planning

Phases include data collection from reliable sources, rigorous preprocessing to ensure data quality, model development with emphasis on RNNs, GRUs, and LSTMs, and comprehensive validation and analysis of model outputs.

Project Planning: Click Here

3. DATA COLLECTION AND PREPROCESSING PHASE

3.1. Data Collection Plan and Sources

Data sourced from data.world, focusing on the "Global Climate Change Data from 1750-2015" dataset, covering land and ocean temperature anomalies. This dataset's CSV format allows for efficient data manipulation and analysis.

Data Collection Plan and Sources: Click Here

3.2. Data Quality Report

Steps included

- Addressing missing values through removal or mean imputation.
- Ensuring data integrity and consistency. Consistency checks were applied to maintain data quality standards.

Data Quality Report: Click Here

3.3. Data Preprocessing

Data normalization enhanced model convergence and performance. The dataset was split into training and testing sets, with additional feature engineering (of date to month and year) to capture temporal dependencies.

Data Preprocessing: Click Here

4. MODEL DEVELOPMENT PHASE

4.1. Model Selection Report

Focused on selecting and optimizing RNNs, GRUs, and LSTMs due to their suitability for sequential data analysis and forecasting temperature trends.

Model Selection Report: Click Here

4.2. Initial Model Training, Validation, and Evaluation Report

Models were trained using historical data, validated, and evaluated using metrics to ensure robustness and reliability in predicting temperature anomalies.

Initial Model Training, Validation, and Evaluation Report: Click Here

5. MODEL OPTIMIZATION AND TUNING PHASE

Detailed the process of hyperparameter tuning to optimize model performance, ensuring the models' accuracy in capturing climate change dynamics.

Model optimization and tuning phase: Click Here

6. RESULTS

The project demonstrated accurate predictions of temperature trends through interactive visualizations, facilitating a deeper understanding of climate change impacts.

7. ADVANTAGES & DISADVANTAGES

Highlighted advantages such as insightful predictions and scalable methodology, alongside challenges like data limitations and computational requirements.

8. CONCLUSION

In conclusion, the project "Unveiling Climate Change Dynamics through Earth Surface Temperature" has leveraged advanced data analysis and machine learning models to deepen our understanding of climate change dynamics. By analyzing historical surface temperature data, we've identified significant trends and anomalies, providing critical insights into the impacts of climate change. This approach underscores the importance of data-driven methodologies in shaping effective strategies for climate adaptation and mitigation.

9. FUTURE SCOPE

- Translate the model's findings into clear visualizations and reports for scientific community and policymakers to inform climate change mitigation strategies.
- Couple the deep learning model with Earth observation systems for real-time monitoring. This could trigger early warnings for extreme weather events like heatwaves or droughts based on predicted temperature deviations.
- Train the model to simulate the potential effects of theoretical large-scale climate engineering solutions (e.g., stratospheric aerosol injection) on future Earth surface temperatures. This would inform discussions on the feasibility and risks of such interventions.

10. APPENDIX

10.1. Source Code

MODELS/RNN

```
import numpy as np
```

```python
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN, Dropout, Input
import warnings

warnings.filterwarnings('ignore', category=pd.errors.SettingWithCopyWarning)
warnings.filterwarnings('ignore', category=FutureWarning)
```

```python
# Read data from CSV file
file_path = 'GlobalTemperatures.csv'
df = pd.read_csv(file_path)

# Impute LandAverageTemperature and LandAverageTemperatureUncertainty with mean
df['LandAverageTemperature'].fillna(df['LandAverageTemperature'].mean(), inplace=True)
df['LandAverageTemperatureUncertainty'].fillna(df['LandAverageTemperatureUncertainty'].
mean(), inplace=True)

# For columns with 1200 missing values, drop those rows
cols_to_dropna = ['LandMaxTemperature', 'LandMaxTemperatureUncertainty',
'LandMinTemperature', 'LandMinTemperatureUncertainty',
'LandAndOceanAverageTemperature', 'LandAndOceanAverageTemperatureUncertainty']
df.dropna(subset=cols_to_dropna, inplace=True)
```

```python
# Add Year and Month columns based on 'dt' column
df['Year'] = pd.to_datetime(df['dt']).dt.year
df['Month'] = pd.to_datetime(df['dt']).dt.month

# Prepare X (features) and y (target)
X = df.drop(['LandAverageTemperature', 'dt'], axis=1)
y = df['LandAverageTemperature']

# Split data 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)

# Scale X and y using MinMaxScaler
scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()

# Fit and transform the training data
X_train_scaled = scaler_x.fit_transform(X_train)
y_train_scaled = scaler_y.fit_transform(y_train.values.reshape(-1, 1))
```

```python
# Only transform the testing data
X_test_scaled = scaler_x.transform(X_test)
y_test_scaled = scaler_y.transform(y_test.values.reshape(-1, 1))

# Reshape X_train_scaled and X_test_scaled for RNN input
X_train_scaled = X_train_scaled.reshape((X_train_scaled.shape[0],
X_train_scaled.shape[1], 1))
X_test_scaled = X_test_scaled.reshape((X_test_scaled.shape[0], X_test_scaled.shape[1], 1))

# Print the shapes to verify
print("X_train_scaled shape:", X_train_scaled.shape)
print("X_test_scaled shape:", X_test_scaled.shape)
print("y_train_scaled shape:", y_train_scaled.shape)
print("y_test_scaled shape:", y_test_scaled.shape)
```

```python
# Define the RNN model
model = Sequential()
model.add(Input(shape=(X_train_scaled.shape[1], X_train_scaled.shape[2])))
model.add(SimpleRNN(100, activation='relu', return_sequences=True))
model.add(Dropout(0.2))
model.add(SimpleRNN(100, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

model.summary()
```

```python
# Train the model
history = model.fit(X_train_scaled, y_train_scaled, epochs=100, batch_size=32,
validation_split=0.2)
```

```python
# Make predictions
predictions = model.predict(X_test_scaled)
predictions = scaler_y.inverse_transform(predictions)
```

```python
# Compare predictions with actual values
actual = scaler_y.inverse_transform(y_test_scaled)
for i in range(len(predictions)):
    print(f"Actual: {actual[i][0]}, Predicted: {predictions[i][0]}")
```

```
from sklearn.metrics import mean_absolute_error, mean_squared_error
# Calculate metrics
mae = mean_absolute_error(actual, predictions)
mse = mean_squared_error(actual, predictions)
rmse = np.sqrt(mse)

print(f"Mean Absolute Error (MAE): {mae}")
print(f"Mean Squared Error (MSE): {mse}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
```

MODELS/GRU

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GRU, Dropout, Input
import warnings

warnings.filterwarnings('ignore', category=pd.errors.SettingWithCopyWarning)
warnings.filterwarnings('ignore', category=FutureWarning)
```

```
# Read data from CSV file
file_path = 'GlobalTemperatures.csv'
df = pd.read_csv(file_path)

# Impute LandAverageTemperature and LandAverageTemperatureUncertainty with mean
df['LandAverageTemperature'].fillna(df['LandAverageTemperature'].mean(), inplace=True)
df['LandAverageTemperatureUncertainty'].fillna(df['LandAverageTemperatureUncertainty']
.mean(), inplace=True)

# For columns with 1200 missing values, drop those rows
cols_to_dropna = ['LandMaxTemperature', 'LandMaxTemperatureUncertainty',
'LandMinTemperature', 'LandMinTemperatureUncertainty',
'LandAndOceanAverageTemperature', 'LandAndOceanAverageTemperatureUncertainty']
df.dropna(subset=cols_to_dropna, inplace=True)
```

```
# Add Year and Month columns based on 'dt' column
df['Year'] = pd.to_datetime(df['dt']).dt.year
df['Month'] = pd.to_datetime(df['dt']).dt.month
```

```python
# Prepare X (features) and y (target)
X = df.drop(['LandAverageTemperature', 'dt'], axis=1)
y = df['LandAverageTemperature']

# Split data 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)

# Scale X and y using MinMaxScaler
scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()

# Fit and transform the training data
X_train_scaled = scaler_x.fit_transform(X_train)
y_train_scaled = scaler_y.fit_transform(y_train.values.reshape(-1, 1))

# Only transform the testing data
X_test_scaled = scaler_x.transform(X_test)
y_test_scaled = scaler_y.transform(y_test.values.reshape(-1, 1))

# Reshape X_train_scaled and X_test_scaled for RNN input
X_train_scaled = X_train_scaled.reshape((X_train_scaled.shape[0], X_train_scaled.shape[1],
1))
X_test_scaled = X_test_scaled.reshape((X_test_scaled.shape[0], X_test_scaled.shape[1], 1))

# Print the shapes to verify
print("X_train_scaled shape:", X_train_scaled.shape)
print("X_test_scaled shape:", X_test_scaled.shape)
print("y_train_scaled shape:", y_train_scaled.shape)
print("y_test_scaled shape:", y_test_scaled.shape)
```

```python
# Define the GRU model
model = Sequential()
model.add(Input(shape=(X_train_scaled.shape[1], X_train_scaled.shape[2])))
model.add(GRU(100, activation='relu', return_sequences=True))
model.add(Dropout(0.2))
model.add(GRU(100, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

model.summary()
```

```
# Train the model
history = model.fit(X_train_scaled, y_train_scaled, epochs=100, batch_size=32,
validation_split=0.2)
```

```
# Make predictions
predictions = model.predict(X_test_scaled)
predictions = scaler_y.inverse_transform(predictions)
```

```
# Compare predictions with actual values
actual = scaler_y.inverse_transform(y_test_scaled)
for i in range(len(predictions)):
   print(f"Actual: {actual[i][0]}, Predicted: {predictions[i][0]}")
```

```
from sklearn.metrics import mean_absolute_error, mean_squared_error
# Calculate metrics
mae = mean_absolute_error(actual, predictions)
mse = mean_squared_error(actual, predictions)
rmse = np.sqrt(mse)

print(f"Mean Absolute Error (MAE): {mae}")
print(f"Mean Squared Error (MSE): {mse}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
```

MODELS/LSTM

```
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.offline as py
import plotly.graph_objs as go
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from math import sqrt
from scipy.stats import pearsonr
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, Bidirectional
```

```
from tensorflow.keras.callbacks import EarlyStopping

import warnings

warnings.filterwarnings('ignore', category=pd.errors.SettingWithCopyWarning)
warnings.filterwarnings('ignore', category=FutureWarning)
# Ignore warnings that match the pattern '.*SettingWithCopyWarning.*'.

py.init_notebook_mode(connected=True)  # Initialize Plotly's notebook mode to work in
Jupyter notebooks with interactive plots.
```

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

```
df.head()
```

```
df.tail()
```

```
df.shape
```

```
df.info()
```

```
df.isnull().sum()
```

```
import missingno as msno
msno.bar(df)
```

```
# Impute LandAverageTemperature and LandAverageTemperatureUncertainty with mean

# Fill missing values in the 'LandAverageTemperature' column with the mean of that
column
df['LandAverageTemperature'].fillna(df['LandAverageTemperature'].mean(), inplace=True)

# Fill missing values in the 'LandAverageTemperatureUncertainty' column with the mean of
that column
df['LandAverageTemperatureUncertainty'].fillna(df['LandAverageTemperatureUncertainty']
.mean(), inplace=True)
```

```
# For columns with 1200 missing values, drop those rows
```

```python
# List of columns to check for missing values
cols_to_dropna = ['LandMaxTemperature', 'LandMaxTemperatureUncertainty',
'LandMinTemperature', 'LandMinTemperatureUncertainty',
'LandAndOceanAverageTemperature', 'LandAndOceanAverageTemperatureUncertainty']

# Loop through each column in the list
for col in cols_to_dropna:
    # Drop rows where the specified column has missing values
    df.dropna(subset=[col], inplace=True)
```

```python
# Verify if there are any remaining missing values
print(df.isnull().sum())
```

```python
df.shape
```

```python
df.duplicated().sum()
```

```python
import matplotlib.pyplot as plt

# Create a 4x2 grid of subplots, with a figure size of 15x20 inches
fig, axs = plt.subplots(4, 2, figsize=(15, 20))

# List of columns to plot line plots for
columns = ['LandAverageTemperature', 'LandAverageTemperatureUncertainty',
'LandMaxTemperature', 'LandMaxTemperatureUncertainty', 'LandMinTemperature',
'LandMinTemperatureUncertainty', 'LandAndOceanAverageTemperature',
'LandAndOceanAverageTemperatureUncertainty']

# Loop through each subplot and the corresponding column
for i, ax in enumerate(axs.flatten()):
    # Plot line plot for the i-th column, excluding missing values
    ax.plot(df[columns[i]].dropna())
    # Set the title of the subplot to the column name
    ax.set_title(columns[i])

# Adjust the layout to prevent overlap
plt.tight_layout()

# Display the plot
plt.show()
```

```python
# Correlation Heatmap

# Calculate the correlation matrix of the DataFrame, considering only numeric columns
hm = df.corr(numeric_only=True)

# Create a heatmap of the correlation matrix with annotations
sns.heatmap(hm, annot=True)

# Display the plot
plt.show()
```

```python
# Select 'dt' and 'LandAverageTemperature' columns from the DataFrame
data = df[['dt', 'LandAverageTemperature']]

# Extract the year from the 'dt' column and create a new 'year' column
data['year'] = data['dt'].apply(lambda x: x[:4])

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

# Convert 'dt' column to datetime format
data['dt'] = pd.to_datetime(data['dt'])

# Set the 'dt' column as the index of the DataFrame
data.set_index('dt', inplace=True)
```

```python
pivot = data.pivot_table(values='LandAverageTemperature', index=data.index.year,
columns=data.index.month)

# Plot the monthly seasonality
monthly_seasonality = pivot.mean(axis=0)
monthly_seasonality.plot(figsize=(20, 6))
plt.title('Monthly Temperatures')
plt.xlabel('Months')
plt.ylabel('Temperature')
plt.xticks(range(1, 13))
plt.show()
```

```python
# Extract the month from the index and create a new 'month' column
data['month'] = data.index.month
```

```
# Extract the year from the index and create a new 'year' column
data['year'] = data.index.year

# Create a pivot table with 'LandAverageTemperature' as values,
# months as rows (index), and years as columns, aggregating by mean
pivot = pd.pivot_table(data, values='LandAverageTemperature', index='month',
columns='year', aggfunc='mean')

# Plot the pivot table
pivot.plot(figsize=(20, 6))

# Set the title of the plot
plt.title('Yearly Temperatures')

# Set the x-axis label
plt.xlabel('Months')

# Set the y-axis label
plt.ylabel('Temperatures')

# Set the x-axis ticks to represent the months (1 to 12)
plt.xticks(range(1, 13))

# Remove the legend
plt.legend().remove()

# Display the plot
plt.show()
```

```
# Create a new figure with a specified size of 22x6 inches
plt.figure(figsize=(22, 6))

# Plot a line graph using seaborn, with the x-axis as the index (datetime) and y-axis as
'LandAverageTemperature'
sns.lineplot(x=data.index, y=data['LandAverageTemperature'])

# Set the title of the plot
plt.title('Temperature Variation from 1760 until 2000')

# Display the plot
plt.show()
```

```
df.describe()
```

```
# Calculate the difference between consecutive values in the 'LandAverageTemperature' column
# and create a new column 'diff' to store these differences
data['diff'] = data['LandAverageTemperature'].diff().dropna()
```

```
# Create a copy of the DataFrame 'data' containing all rows except the last 60 (training set)
train = data[:-60].copy()

# Extract the 'diff' column from the training set as the target variable 'y'
y = train['diff'].dropna()

# Number of lags to plot in the autocorrelation and partial autocorrelation plots
lags_plots = 48

# Size of the figure for plotting
figsize = (22, 8)

# Create a figure with the specified figsize
fig = plt.figure(figsize=figsize)

# Define subplot positions within a grid of 3x3
ax1 = plt.subplot2grid((3, 3), (0, 0), colspan=2)  # Top-left subplot spanning 2 columns
ax2 = plt.subplot2grid((3, 3), (1, 0))             # Middle-left subplot
ax3 = plt.subplot2grid((3, 3), (1, 1))             # Middle-right subplot
ax4 = plt.subplot2grid((3, 3), (2, 0), colspan=2)  # Bottom subplot spanning 2 columns

# Plot the time series of 'y' on the top-left subplot
y.plot(ax=ax1)
ax1.set_title('Differenced Temperature Variation')

# Plot the autocorrelation function (ACF) of 'y' on the middle-left subplot
plot_acf(y, lags=lags_plots, zero=False, ax=ax2)

# Plot the partial autocorrelation function (PACF) of 'y' on the middle-right subplot
plot_pacf(y, lags=lags_plots, zero=False, ax=ax3)

# Plot the distribution (histogram with KDE) of 'y' on the bottom subplot
sns.histplot(y, bins=int(sqrt(len(y))), ax=ax4, kde=True)
ax4.set_title('Distribution Chart')

# Adjust layout to prevent overlapping of subplots
```

```
plt.tight_layout()

# Display the plot
plt.show()
```

```
# Print a header for the Dickey-Fuller test results
print('Results of Dickey-Fuller Test:')

# Perform the Augmented Dickey-Fuller test on the time series 'y' and store the results
adfinput = adfuller(y)

# Create a pandas Series to organize and round the test results
adftest = pd.Series(adfinput[0:4], index=['Test Statistic', 'p-value', 'Lags Used', 'Number of Observations Used'])
adftest = round(adftest, 4)

# Loop through the critical values and add them to the adftest Series
for key, value in adfinput[4].items():
    adftest[f"Critical Value ({key})"] = round(value, 4)

# Print the formatted Dickey-Fuller test results
print(adftest)

# Compare the Test Statistic with the Critical Value at 5% significance level
if adftest['Test Statistic'] < adftest['Critical Value (5%)']:
    print('\nThe Test Statistic is lower than the Critical Value of 5%. \nThe series seems to be stationary.')
else:
    print("\nThe Test Statistic is higher than the Critical Value of 5%. \nThe series isn't stationary.")
```

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Bidirectional, Dense, Dropout, Input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_absolute_error, mean_squared_error
import matplotlib.pyplot as plt
import numpy as np
```

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import joblib

# Add Year and Month columns based on 'dt' column
df['Year'] = pd.to_datetime(df['dt']).dt.year
df['Month'] = pd.to_datetime(df['dt']).dt.month

# Prepare X (features) and y (target)
X = df.drop(['LandAverageTemperature', 'dt'], axis=1)
y = df['LandAverageTemperature']

# Split data 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)

# Scale X and y using MinMaxScaler
scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()

# Fit and transform the training data
X_train_scaled = scaler_x.fit_transform(X_train)
y_train_scaled = scaler_y.fit_transform(y_train.values.reshape(-1, 1))  # Reshape y_train to a 2D array

# Only transform the testing data
X_test_scaled = scaler_x.transform(X_test)
y_test_scaled = scaler_y.transform(y_test.values.reshape(-1, 1))  # Reshape y_test to a 2D array

# Reshape X_train_scaled and X_test_scaled for LSTM input
X_train_scaled = X_train_scaled.reshape((X_train_scaled.shape[0], X_train_scaled.shape[1], 1))
X_test_scaled = X_test_scaled.reshape((X_test_scaled.shape[0], X_test_scaled.shape[1], 1))

# Print the shapes to verify
print("X_train_scaled shape:", X_train_scaled.shape)
print("X_test_scaled shape:", X_test_scaled.shape)
print("y_train_scaled shape:", y_train_scaled.shape)
print("y_test_scaled shape:", y_test_scaled.shape)

# Save the scaler
joblib.dump(scaler_x, 'scaler_x.pkl')
joblib.dump(scaler_y, 'scaler_y.pkl')
```

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, Bidirectional, Input
from tensorflow.keras.optimizers import Adam
from keras_tuner import HyperModel

class LSTMHyperModel(HyperModel):
    def build(self, hp):
        model = Sequential()
        model.add(Input(shape=(X_train_scaled.shape[1], 1)))

        # First Bidirectional LSTM Layer
        model.add(Bidirectional(LSTM(units=hp.Int('units_1', min_value=32, max_value=256,
step=32),
                                     return_sequences=True)))
        model.add(Dropout(rate=hp.Float('dropout_1', min_value=0.1, max_value=0.5,
step=0.1)))

        # Second Bidirectional LSTM Layer
        model.add(Bidirectional(LSTM(units=hp.Int('units_2', min_value=32, max_value=256,
step=32),
                                     return_sequences=True)))
        model.add(Dropout(rate=hp.Float('dropout_2', min_value=0.1, max_value=0.5,
step=0.1)))

        # Third Bidirectional LSTM Layer
        model.add(Bidirectional(LSTM(units=hp.Int('units_3', min_value=32, max_value=256,
step=32))))
        model.add(Dropout(rate=hp.Float('dropout_3', min_value=0.1, max_value=0.5,
step=0.1)))

        # Dense Layer
        model.add(Dense(units=hp.Int('dense_units', min_value=32, max_value=256, step=32),
activation='relu'))
        model.add(Dropout(rate=hp.Float('dropout_dense', min_value=0.1, max_value=0.5,
step=0.1)))

        # Output Layer
        model.add(Dense(1))

        # Compile the Model
        model.compile(optimizer=Adam(learning_rate=hp.Float('learning_rate', min_value=1e-
4, max_value=1e-2, sampling='LOG')),
                      loss='mean_squared_error',
                      metrics=['mae'])
```

```python
    return model
```

```python
from keras_tuner.tuners import RandomSearch

tuner = RandomSearch(
    LSTMHyperModel(),
    objective='val_loss',
    max_trials=10,
    executions_per_trial=2,
    directory='hyperparameter_tuning',
    project_name='lstm_temperature_prediction'
)

early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)

# Run the Hyperparameter Search
tuner.search(X_train_scaled, y_train_scaled, epochs=50, validation_data=(X_test_scaled,
y_test_scaled), callbacks=[early_stopping], verbose=2)
```

```python
# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the best model
best_model = tuner.hypermodel.build(best_hps)

# Summary of the best model
best_model.summary()
```

```python
# Train the model with the best hyperparameters
history = best_model.fit(
    X_train_scaled, y_train_scaled,
    batch_size=64, epochs=100,
    validation_data=(X_test_scaled, y_test_scaled),
    callbacks=[early_stopping],
    verbose=2
)

# Evaluate the best model
```

```
loss, mae = best_model.evaluate(X_test_scaled, y_test_scaled)
print(f"Validation Loss: {loss}, Validation MAE: {mae}")

# Save the best model
best_model.save('best_model.keras')
```

```
import matplotlib.pyplot as plt
from tensorflow.keras.models import load_model

# Generate predictions using the trained model on the test data

model = load_model('best_model.keras')
predictions = model.predict(X_test_scaled)

# Create a new figure with a size of 10x6 inches
plt.figure(figsize=(10, 6))

# Plot the actual values (ytest) as a line plot with label 'Actual'
plt.plot(y_test_scaled, label='Actual')

# Plot the predicted values (predictions) as a line plot with label 'Predicted'
plt.plot(predictions, label='Predicted')

# Set the title of the plot
plt.title('Actual vs Predicted')

# Set the x-axis label
plt.xlabel('Time')

# Set the y-axis label
plt.ylabel('Value')

# Add a legend to the plot
plt.legend()

# Display the plot
plt.show()
```

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

# Generate predictions using the trained model on the test data
```

```
predictions = model.predict(X_test_scaled)

# Calculate errors as the difference between ytest (actual) and predictions
errors = y_test_scaled - predictions

# Create a new figure with a size of 14x8 inches
plt.figure(figsize=(14, 8))

# Subplot 1: Actual values (ytest) plot
plt.subplot(3, 1, 1)
plt.plot(y_test_scaled, label='Actual', color='blue')
plt.title('Actual vs Predicted vs Errors')  # Set subplot title
plt.ylabel('Value')  # Set y-axis label
plt.legend()  # Display legend for this subplot

# Subplot 2: Predicted values (predictions) plot
plt.subplot(3, 1, 2)
plt.plot(predictions, label='Predicted', color='red')
plt.ylabel('Value')  # Set y-axis label
plt.legend()  # Display legend for this subplot

# Subplot 3: Errors plot
plt.subplot(3, 1, 3)
plt.plot(errors, label='Errors', color='green')
plt.ylabel('Error')  # Set y-axis label
plt.xlabel('Time')  # Set x-axis label
plt.legend()  # Display legend for this subplot

# Adjust layout to prevent overlapping of subplots
plt.tight_layout()

# Display the plot
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Generate predictions using the trained model on the test data
predictions = model.predict(X_test_scaled)

# Calculate errors as the difference between ytest (actual) and predictions
errors = y_test_scaled - predictions
```

```python
# Calculate evaluation metrics: Mean Absolute Error (MAE), Mean Squared Error (MSE), and
Root Mean Squared Error (RMSE)
mae = mean_absolute_error(y_test_scaled, predictions)
mse = mean_squared_error(y_test_scaled, predictions)
rmse = np.sqrt(mse)

# Create a new figure with a size of 14x8 inches
plt.figure(figsize=(14, 8))

# Plot actual values (ytest) as a line plot with label 'Actual' in blue
plt.plot(y_test_scaled, label='Actual', color='blue')

# Plot predicted values (predictions) as a line plot with label 'Predicted' in red
plt.plot(predictions, label='Predicted', color='red')

# Plot errors as a line plot with label 'Errors' in green
plt.plot(errors, label='Errors', color='green')

# Display MAE, MSE, and RMSE values as text annotations on the plot
plt.text(0, np.max(y_test_scaled), f"MAE: {mae:.2f}", fontsize=12, color='black')
plt.text(0, np.max(y_test_scaled)*0.9, f"MSE: {mse:.2f}", fontsize=12, color='black')
plt.text(0, np.max(y_test_scaled)*0.8, f"RMSE: {rmse:.2f}", fontsize=12, color='black')

# Set plot title, x-axis label, and y-axis label
plt.title('Actual vs Predicted vs Errors')
plt.ylabel('Value')
plt.xlabel('Time')

# Display legend
plt.legend()

# Enable grid on the plot
plt.grid(True)

# Adjust layout to prevent overlapping of elements
plt.tight_layout()

# Display the plot
plt.show()
```

10.2. GitHub & Project Demo Link

- GitHub Repo link : github
- Project Demo : Youtube