Università degli Studi di Messina
Dipartimento di Scienze Matematiche e Informatiche
Scienze Fisiche e Scienze della Terra
Corso di Laurea Triennale in Informatica
Curriculum Data Analysis

Weather-forecasting with Machine Learning and Deep Learning

CANDIDATE:
SAHIL RAVINDRABHAI NAKRANI (533173)

SUPERVISOR:
Prof. Ing. Armando Ruggeri

Academic year 2023-2024

0.1 Abstract

This thesis explores the application of machine learning, particularly neural networks, for weather forecasting, a complex and crucial task in meteorology. Accurate weather prediction is vital for various sectors, such as agriculture, transportation, and disaster management. While traditional methods rely heavily on physical models, recent advancements in machine learning offer an alternative approach by analyzing large datasets and detecting patterns that are not immediately visible through conventional techniques.

The primary focus of this thesis is the use of Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks to improve the accuracy of weather predictions. These models are well-suited for time-series data, making them ideal for forecasting tasks. By leveraging the historical weather data, RNN and LSTM can learn temporal dependencies and improve the model's forecasting ability.

In this thesis, we begin by providing a detailed background on machine learning and neural networks, followed by an analysis of traditional weather forecasting methods and their limitations. We then explain the architecture of RNN and LSTM models and discuss their advantages in handling sequential data. After the theoretical discussion, the thesis shifts focus to the practical implementation, where weather data from various meteorological stations are used to train and evaluate the models. The experimental results show that LSTM networks outperform traditional methods and even basic RNNs in terms of accuracy and generalization. The LSTM model's ability to remember long-term dependencies makes it particularly effective in capturing complex patterns in weather data. Additionally, the thesis examines the challenges and limitations of using neural networks for weather forecasting, including the quality of input data, the need for significant computational resources, and the potential for overfitting.

Finally, we conclude by discussing the broader implications of using machine learning for weather forecasting and suggest future improvements, such as hybrid models and the integration of additional data sources like satellite images and atmospheric pressure measurements. The findings of this thesis demonstrate the potential of neural networks in enhancing weather prediction accuracy and contribute to the growing field of AI in meteorology.

Contents

List of Figures

CHAPTER 1

Introduction

1.0.1 Overview of Weather Forecasting

Weather forecasting has always been an essential component of human activities, affecting decisions in agriculture, aviation, marine operations, and disaster preparedness. Traditionally, weather forecasting relies on physical models of atmospheric dynamics, which include numerical weather prediction (NWP) models. These models simulate the atmosphere's behavior based on mathematical equations derived from the laws of physics, such as the Navier-Stokes equations. However, despite their accuracy, NWP models often struggle with issues related to computational cost, data sparsity, and error propagation over time.

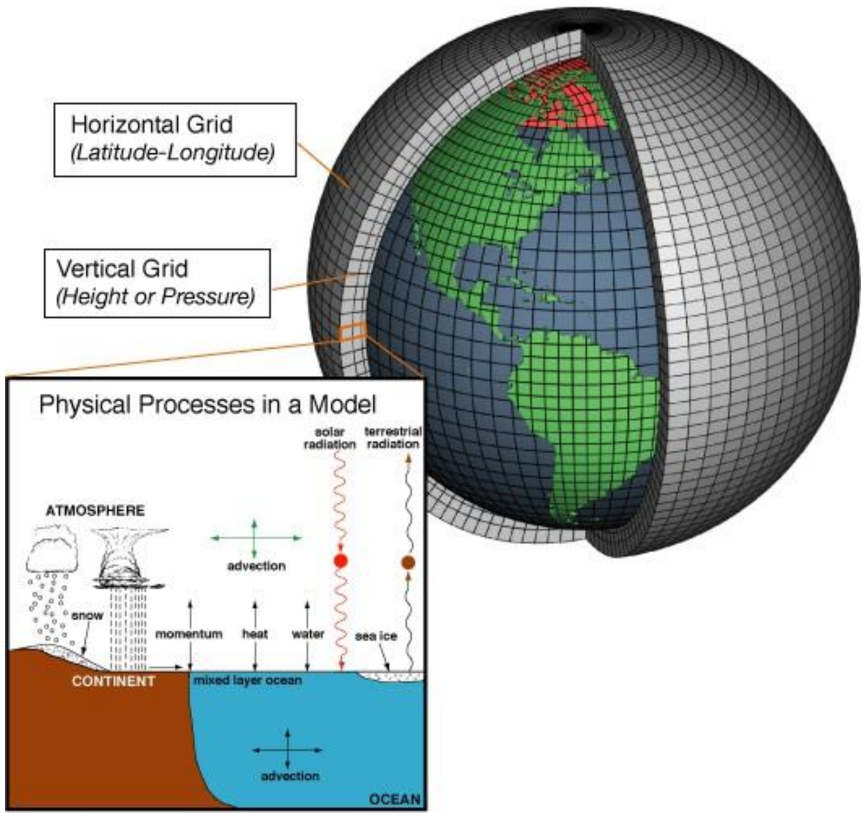


Figure 1.1: Weather Model

In recent years, the advent of artificial intelligence (AI) and machine learning (ML) has provided new possibilities for improving weather predictions. Machine learning models, which can analyze large amounts of data and detect intricate patterns, offer the potential to complement or even replace traditional methods in specific scenarios. Among these methods, neural networks, especially Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, have gained prominence for their capability to model time-series data, such as historical weather conditions.

1.0.2 Relevance of Neural Networks in Weather Forecasting

Neural networks have revolutionized several fields, such as image recognition, speech processing, and financial forecasting. In the context of weather forecasting, neural networks can analyze vast quantities of historical data, such as temperature, humidity, wind speed, and atmospheric pressure, to predict future conditions. Unlike traditional models, which depend on pre-defined equations, neural networks learn directly from data, making them more flexible and adaptable.

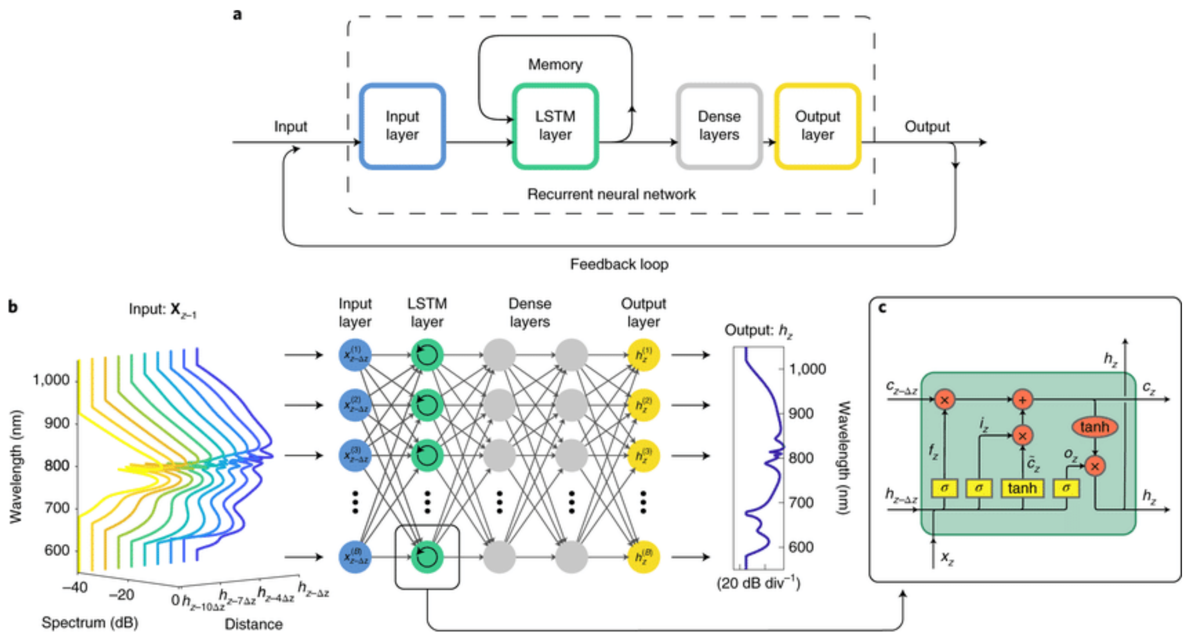


Figure 1.2: Diagram of Neural Network Architecture

Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks are particularly effective in this domain due to their ability to capture temporal dependencies within sequential data. Weather patterns are inherently time-dependent, meaning that the conditions on one day often influence the weather in the days to come. RNNs, designed to handle sequential information, enable the model to “remember” prior data points when making predictions. LSTMs go a step further by addressing the limitations of RNNs, particularly the problem of vanishing gradients, allowing the model to retain long-term dependencies more effectively.

1.0.3 Challenges in Weather Prediction

Despite the advancements in machine learning, weather forecasting remains a challenging task. The inherent unpredictability of weather, the non-linear interactions between atmospheric elements, and the chaotic nature of climate systems all contribute to the complexity of accurate forecasting. Furthermore, the quality of weather data can vary significantly depending on the geographic location and the time span covered, leading to potential inaccuracies.

Additionally, while machine learning models can be trained to make highly accurate predictions, they require vast amounts of data and significant computational power. Overfitting is another common challenge, where a model performs well on training data but fails to generalize to unseen data, leading to unreliable forecasts. In this thesis, we explore how RNN and LSTM models can overcome some of these challenges and present an alternative approach to traditional weather forecasting.

1.0.4 Research Motivation and Goals

The motivation behind this thesis stems from the desire to explore how machine learning, and more specifically neural networks, can improve the accuracy of weather forecasting. With the increasing availability of large weather datasets and advancements in computational power, there is an opportunity to enhance forecasting techniques through AI. By focusing on RNN and LSTM models, this research aims to demonstrate their potential in modeling weather patterns over time.

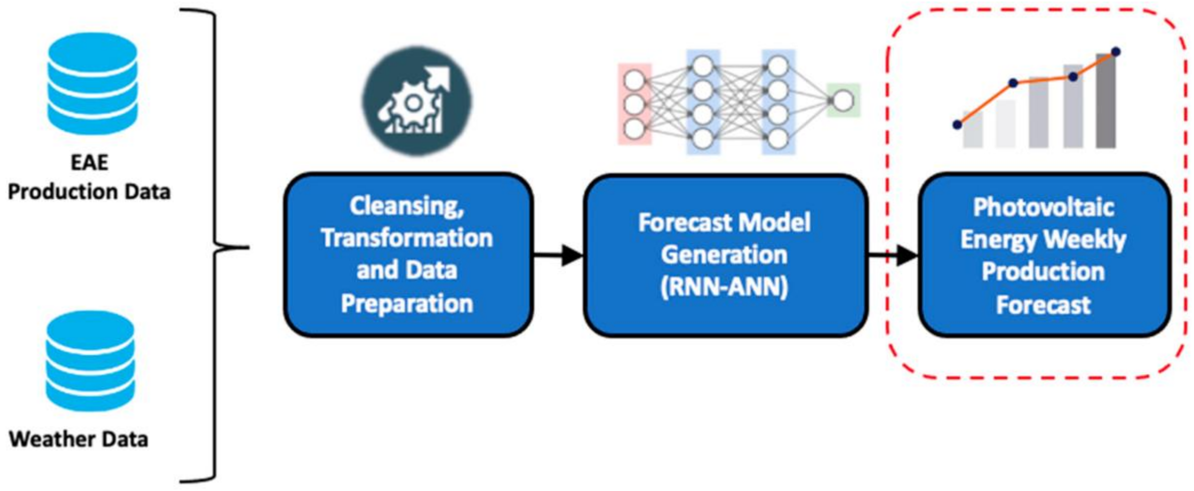

EAE
Production Data
Weather Data
Cleansing, Transformation and Data Preparation
Forecast Model Generation (RNN-ANN)
Photovoltaic Energy Weekly Production Forecast

Figure 1.3: Data Flow - from Weather Stations to Predictive Models

The primary goal of this thesis is to evaluate the effectiveness of RNNs and LSTMs in predicting weather conditions. This includes a detailed analysis of model design, data preprocessing, training, and evaluation. Furthermore, the thesis aims to provide insights into how these models can be integrated into operational weather prediction systems and suggest areas for future research.

CHAPTER 2

Objectives

The primary goal of this thesis is to develop an efficient and accurate weather forecasting model using neural networks, specifically Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) networks. Weather forecasting is a critical application that influences sectors such as agriculture, transportation, disaster management, and energy. While traditional statistical methods have been employed for this task, the rise of machine learning, and more specifically deep learning, offers the potential for more accurate and dynamic forecasting.

2.0.1 Main Objectives

1. **Design and Implementation of Neural Network-Based Weather Prediction Models:**

The thesis aims to design, train, and evaluate RNN and LSTM models that are capable of predicting weather conditions over varying time scales (daily, weekly, and monthly). These models should be able to capture complex weather patterns by learning from historical data and adjusting predictions based on both short- and long-term dependencies.

2. **Comparative Analysis with Traditional Machine Learning Methods:**

A key objective is to compare the performance of neural network-based models with traditional machine learning classification techniques that the author has previously worked with, such as Random Forest, XGBoost, and Decision Trees. This comparison will highlight the advantages and limitations of neural networks in weather forecasting.

3. **Handling Temporal Data:**

The model should effectively manage and learn from temporal sequences in weather data. RNNs and LSTMs are specifically chosen for their ability to deal with time-dependent data, making them ideal for weather forecasting. The goal is to optimize these models for best performance on Italy's regional weather data.

4. **Accuracy Improvement Over Existing Methods:**

Through rigorous experimentation and hyperparameter tuning, the objective is to improve the accuracy of weather predictions. A critical evaluation will be done using metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-Squared, ensuring that the new models outperform traditional methods.

2.0.2 Secondary Objectives

1. **Data Preprocessing and Feature Engineering:**

The thesis will also explore various data preprocessing techniques such as handling missing data, normalization, and feature engineering. These steps are critical in improving the overall model performance and ensuring that the input data is suitable for neural networks.

2. **Evaluation of Seasonal and Long-Term Trends:**

An additional objective is to analyze and predict seasonal weather patterns. By capturing long-term trends in weather data, the model should provide insights into both immediate forecasts and longer-term seasonal predictions.

3. **Exploration of Advanced Neural Network Architectures:**

Although the primary focus is on RNN and LSTM models, the thesis will also touch upon the potential of more advanced architectures, such as Gated Recurrent Units (GRUs) or Convolutional LSTM, to handle large-scale weather data and further improve prediction accuracy.

2.0.3 Broader Impact

The objectives of this thesis extend beyond the technical aspects of model development. By successfully achieving these objectives, the research will contribute to the broader field of weather forecasting and machine learning. The work has potential applications in numerous industries, including climate science, environmental monitoring, and disaster preparedness. Additionally, the study aims to demonstrate the power of deep learning methods in fields traditionally dominated by statistical models, showcasing the versatility and capability of neural networks in real-world, high-impact applications like weather prediction.

CHAPTER 3

Design

3.0.1 Design of the Weather Forecasting Model

Designing a machine learning model for weather forecasting involves a meticulous process that integrates data acquisition, preprocessing, model selection, architecture design, training, evaluation, and deployment. Given the complexity of weather patterns, the design must account for the inherent chaotic nature of climate systems, ensuring that the model is robust enough to handle large datasets and capable of learning temporal dependencies. This section delves into the systematic design approach for building a weather forecasting model using Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) networks.

3.0.2 Data Acquisition and Selection

The first step in designing a weather forecasting model is acquiring a relevant and comprehensive dataset. For this thesis, historical weather data, including variables such as temperature, humidity, wind speed, atmospheric pressure, and precipitation, were collected from **Meteostat**, a reliable source of global meteorological data. The data spans multiple decades, from 2004 to 2024, covering various weather conditions and patterns.

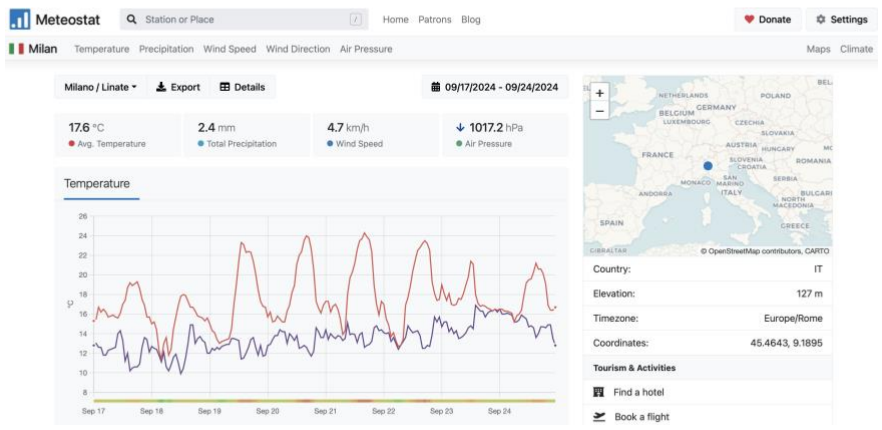


Figure 3.1: Global Data Sources of Weather

To ensure high-quality predictions, the selected dataset must be representative of the region and time span under investigation. For instance, in Italy, data from weather stations like those in Milan and Rome provide crucial insights into local climatic conditions. One challenge with data acquisition is handling gaps or missing data, which can affect the model's accuracy. In this design, appropriate imputation techniques, such as forward-fill, are employed to manage missing values, maintaining the consistency of the dataset.

3.0.3 Data Preprocessing

Raw weather data is rarely ready for immediate use in machine learning models. Data preprocessing is an essential step in the design process to ensure the data is clean, normalized, and formatted correctly for model input. Preprocessing steps include:

- **Handling Missing Data:** Missing data is addressed through interpolation or more sophisticated statistical imputation methods. For instance, if temperature data is missing for a day, it can be inferred based on nearby days' data.

• **Normalization:** Weather data is often recorded on different scales (e.g., temperature in degrees Celsius, wind speed in km/h). Normalization is applied to bring all variables onto a similar scale, typically between 0 and 1, which helps the neural network converge faster during training.

• **Feature Selection:** Not all available weather variables are equally relevant to the forecasting task. Feature selection involves identifying which weather parameters (e.g., temperature, humidity, wind speed) contribute most to accurate forecasts. This helps reduce model complexity and avoid overfitting.

Time-series forecasting, like weather prediction, requires special consideration of the temporal nature of the data. Thus, the data is structured as sequences where each input includes a series of past observations. This allows the RNN and LSTM models to learn the temporal relationships between the weather variables over time.

3.0.4 Model Architecture Design

At the core of the design is the architecture of the neural network itself. Given that weather forecasting is inherently a time-series prediction problem, Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) networks are the most appropriate choices. These architectures are designed to capture temporal dependencies within sequential data, which is crucial for understanding and predicting weather patterns.

1. **Recurrent Neural Networks (RNN):** RNNs are designed to handle sequential data by incorporating memory of previous inputs into the model's decision-making process. This is achieved through a recurrent loop in each neuron that allows the model to pass information from one time step to the next.

• **Advantages:** RNNs are suitable for short-term dependencies in time-series data, making them effective in capturing recent weather patterns that influence current conditions.

- **Limitations:** RNNs often suffer from the vanishing gradient problem, making it difficult for them to learn long-term dependencies over extended time horizons.

2. **Long Short-Term Memory (LSTM):** LSTM networks were developed to address the limitations of standard RNNs. LSTMs incorporate a gating mechanism that allows them to remember long-term dependencies, making them particularly useful for weather forecasting, where both short- and long-term patterns are important.

- **Advantages:** LSTMs excel at learning long-range dependencies in data, such as seasonal weather variations or long-term climate trends.

- **Design Considerations:** The number of LSTM layers, the number of neurons per layer, and the activation functions must be carefully tuned to balance complexity and performance.

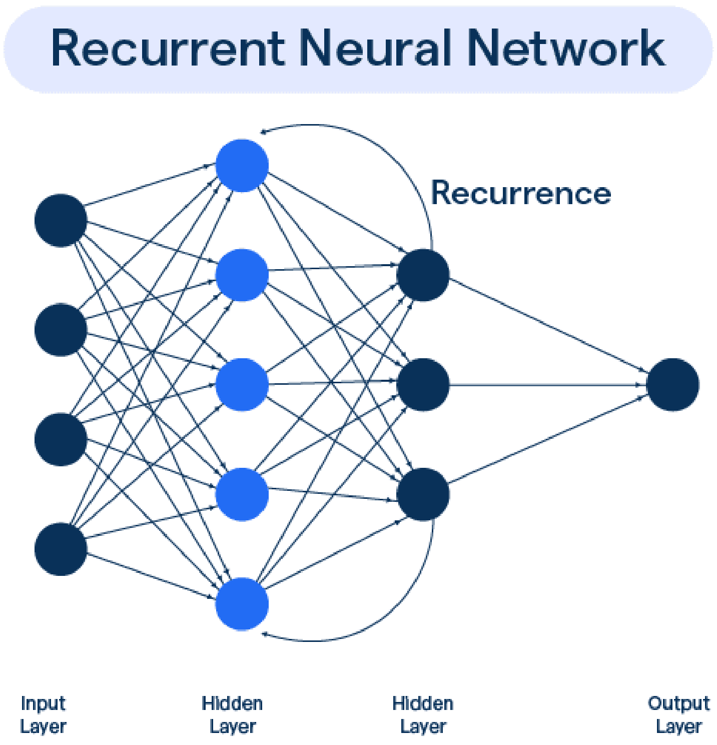


Figure 3.2: RNN Architecture

3.0.5 Hyperparameter Tuning and Optimization

The performance of RNN and LSTM models heavily depends on the choice of hyperparameters. These include:

- **Number of Layers and Neurons:** Adding more layers or neurons increases the model's ability to capture complex patterns but can also lead to overfitting. A balance must be struck based on the size of the dataset and the complexity of the problem.
- **Learning Rate:** The learning rate controls how quickly the model updates its weights during training. Too high a learning rate can cause the model to converge too quickly, missing the optimal solution, while too low a learning rate can make training unnecessarily slow.
- **Batch Size:** Batch size refers to the number of samples processed before updating the model's weights. A larger batch size can lead to faster training but may require more memory.
- **Epochs:** The number of epochs determines how many times the model sees the entire dataset during training. More epochs allow the model to learn more, but too many can lead to overfitting.

Hyperparameter tuning is typically performed using grid search or random search methods to identify the optimal configuration for the model. This step is crucial in ensuring the model is both efficient and accurate.

3.0.6 Model Training and Validation

Once the model's architecture and hyperparameters are finalized, the next step is training. The dataset is divided into training, validation, and testing sets. The model is trained on the training set, and its performance is evaluated on the validation set to ensure it generalizes well to unseen data. Early stopping is employed to prevent overfitting, where training halts once the model's performance on the validation set begins to deteriorate.

Training a neural network model for weather forecasting is computationally intensive. It requires powerful hardware, such as Graphics Processing Units (GPUs), to accelerate the matrix operations involved in backpropagation. The use of cloud computing resources, such as Google Colab or Amazon Web Services (AWS), can provide the necessary infrastructure to train large-scale models efficiently.

3.0.7 Evaluation Metrics

After training, the model is evaluated using various performance metrics. For weather forecasting, the most common evaluation metrics include:

• **Mean Squared Error (MSE):** Measures the average squared difference between predicted and actual values. Lower MSE indicates better predictive accuracy.

• **Mean Absolute Error (MAE):** Measures the average absolute difference between predicted and actual values, providing a more interpretable error measure in real-world units (e.g., degrees Celsius).

• **R-Squared (Coefficient of Determination):** Indicates the proportion of variance in the dependent variable that is predictable from the independent variables. A value closer to 1 suggests better fit.

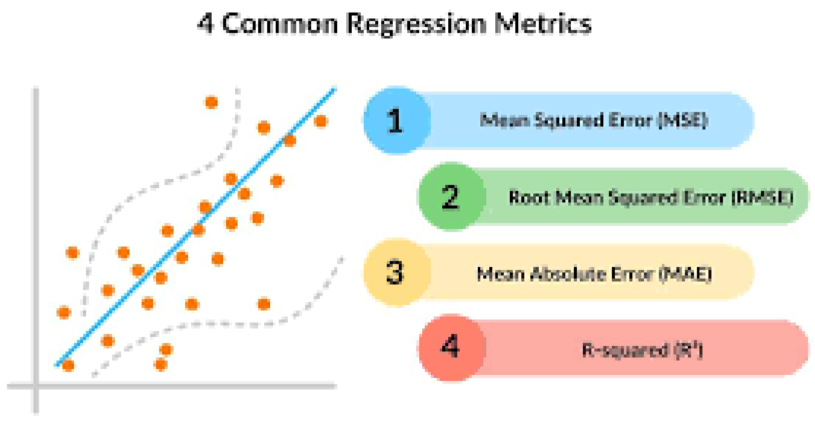

4 Common Regression Metrics
1
Mean Squared Error (MSE)
2
Root Mean Squared Error (RMSE)
3
Mean Absolute Error (MAE)
4
R-squared (R^2)

Figure 3.3: Evaluation Metrics: MSE, MAE, R-Squared

CHAPTER 4

Technologies used

This section outlines the key technologies employed during the development of the weather forecasting model. Both cloud-based and local computing environments were utilized to ensure efficient data handling, model training, and analysis.

4.0.1 Programming Language: Python

Python was the primary programming language for the thesis due to its ease of use and extensive ecosystem of libraries tailored for machine learning and data science. Python offers:

• **Ease of Use:** A simple syntax that allows for rapid prototyping and development.

• **Extensive Libraries:** Includes TensorFlow, Keras, Pandas, and NumPy, which are essential for deep learning and data analysis.

• **Community Support:** A vast community providing resources and support for implementation and debugging.

4.0.2 Deep Learning Framework: TensorFlow and Keras

The deep learning models, specifically Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, were built using **TensorFlow** and **Keras**. These frameworks provide powerful tools for defining, training, and optimizing deep neural networks:

• **TensorFlow:** An open-source framework from Google, TensorFlow supports high-performance computations using GPU acceleration. It offers flexibility in building custom neural network architectures.

• **Keras:** A high-level API built on top of TensorFlow, simplifying the design and training of neural networks. Keras provides an intuitive interface, allowing for quick experimentation with deep learning models.

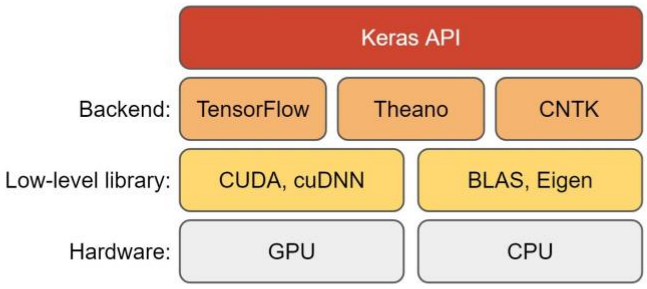


Figure 4.1: Overview of the TensorFlow and Keras Framework Stack

4.0.3 Data Handling and Analysis: Pandas and NumPy

For handling large-scale weather data, **Pandas** and **NumPy** were critical in the data preprocessing pipeline:

• **Pandas:** A library that facilitates data manipulation and time-series data handling, making it ideal for weather data analysis.

• **NumPy:** Provides support for numerical computations and matrix operations, essential for neural network training and model optimization.

4.0.4 Data Visualization: Matplotlib and Seaborn

Visualization of data and model performance was achieved through **Matplotlib** and **Seaborn**:

• **Matplotlib:** A comprehensive library for creating static, animated, and interactive visualizations of weather data trends and model performance.

• **Seaborn:** A data visualization library that builds on Matplotlib, offering more advanced statistical visualizations such as heatmaps and correlation matrices.

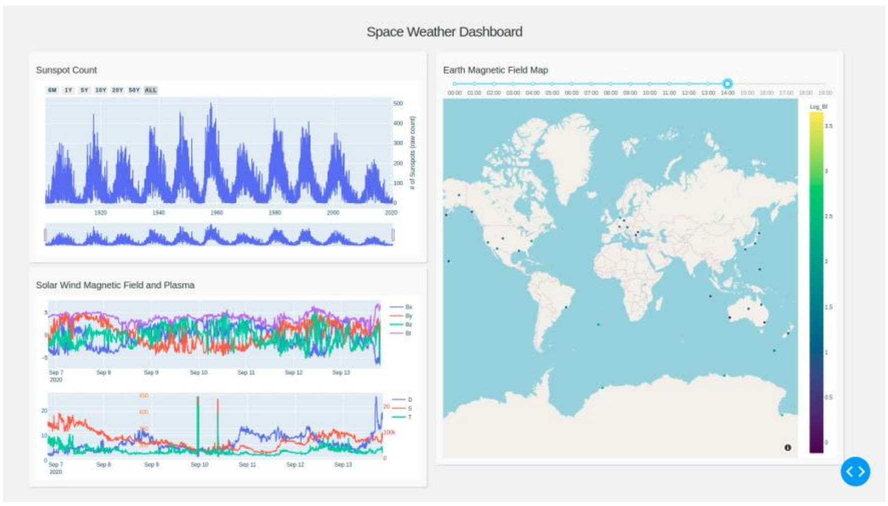


Figure 4.2: Example Weather Data Visualization using Matplotlib

4.0.5 Cloud and Local Computing: Google Colab and MacBook Air M2

To handle the computational load of training deep learning models, both cloud computing and local resources were used:

• **Google Colab:** A cloud-based platform that provides access to free GPU-accelerated computing. It is used for training complex neural networks that require long processing times.

• **Local Computing (MacBook Air M2):** The **MacBook Air M2** was used for local experiments, data preprocessing, and smaller-scale model testing. Its M2 chip provides sufficient power to handle initial training runs and hyperparameter tuning.

By combining local and cloud-based environments, this project ensured flexibility and efficiency in model development.

4.0.6 Version Control: Git and GitHub

For managing the codebase, tracking changes, and collaborating, **Git** and **GitHub** were employed:

• **Git:** A distributed version control system for managing code versions, enabling efficient collaboration and rollback to previous states when necessary.

• **GitHub:** An online platform for hosting code repositories, making it easy to share progress, track contributions, and collaborate.

4.0.7 Experiment Tracking: TensorBoard

TensorBoard was used to monitor the training process, visualize the model architecture, and track performance metrics:

• **TensorBoard:** A visualization tool integrated with TensorFlow, providing real-time insights into model performance during training, helping optimize the RNN and LSTM models.

4.0.8 Additional Tools

- **Scikit-learn:** Utilized for traditional machine learning models such as Decision Trees, Random Forest, and XGBoost, allowing for baseline comparisons with the neural network models.
- **Jupyter Notebooks:** Used for running experiments and documenting results, offering an interactive environment where code, visualizations, and outputs can be combined in a single document.

CHAPTER 5

Implementation

5.0.1 Importing Necessary Libraries

In this section, we will outline the necessary libraries used in the project and their roles in the implementation of the weather forecasting model using LSTM. Proper library selection is crucial for efficient data manipulation, model building, and evaluation in machine learning projects.

Explanation of Libraries Used

1. **NumPy**:

• **Purpose**: A fundamental library for numerical computations in Python. It provides support for arrays, matrices, and many mathematical functions, which are essential for data manipulation and model operations.

2. **Pandas**:

• **Purpose**: A powerful data analysis library that allows for easy manipulation of structured data (like CSV files). It offers data structures like DataFrames and Series, which simplify data handling and analysis.

3. **Matplotlib**:

• **Purpose**: A widely-used library for creating static, animated, and interactive visualizations in Python. It is essential for plotting graphs and visualizing data trends.

4. **Seaborn**:

• **Purpose**: A statistical data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive statistical graphics, making it easier to create informative visualizations.

5. **Scikit-learn**:

• **Purpose**: A robust machine learning library that offers simple and efficient tools for data mining and data analysis. It includes utilities for data preprocessing, model evaluation, and various algorithms.

6. **TensorFlow/Keras**:

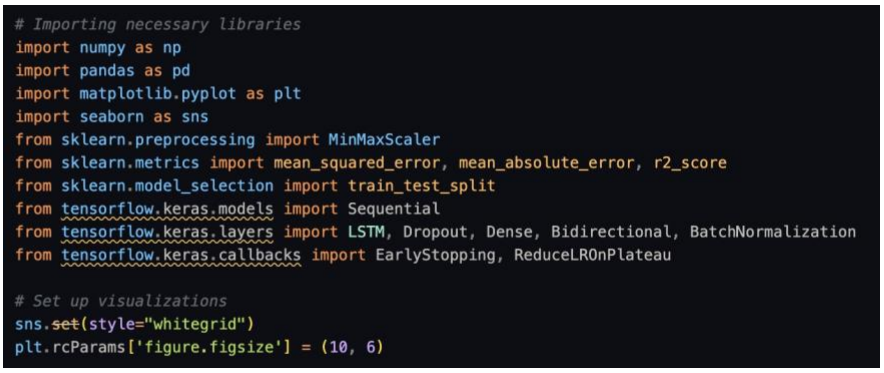
• **Purpose**: A deep learning framework that provides tools for building and training neural networks. Keras, a high-level API within TensorFlow, simplifies the creation of complex models, particularly for time series forecasting with LSTMs.

7. **Statsmodels**:

• **Purpose**: A library that provides classes and functions for estimating and interpreting statistical models. It is useful for performing time series decomposition and other statistical analyses.

Code Screenshot

Here's the code snippet used for importing the necessary libraries:



```python
# Importing necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dropout, Dense, Bidirectional, BatchNormalization
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Set up visualizations
sns.~~set~~(style="whitegrid")
plt.rcParams['figure.figsize'] = (10, 6)
```

Figure 5.1: loading libraries

5.0.2 Loading Data

In this section, we will detail the process of loading the weather dataset, performing initial checks, and verifying that the data has been loaded correctly. Proper data loading and preprocessing are critical steps in building a robust weather forecasting model.

Code Explanation

1. **Loading the Dataset**:

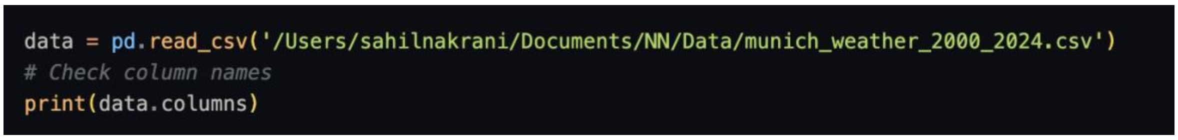
We assume that the dataset is in CSV format. The dataset contains weather-related data collected over the years, which will be used for training the LSTM model.

2. **Data Inspection**:

After loading the data, it is essential to check the structure, including column names, the first few rows, basic statistics, and any missing values. This step ensures that we understand the data's structure and can identify potential issues before further processing.

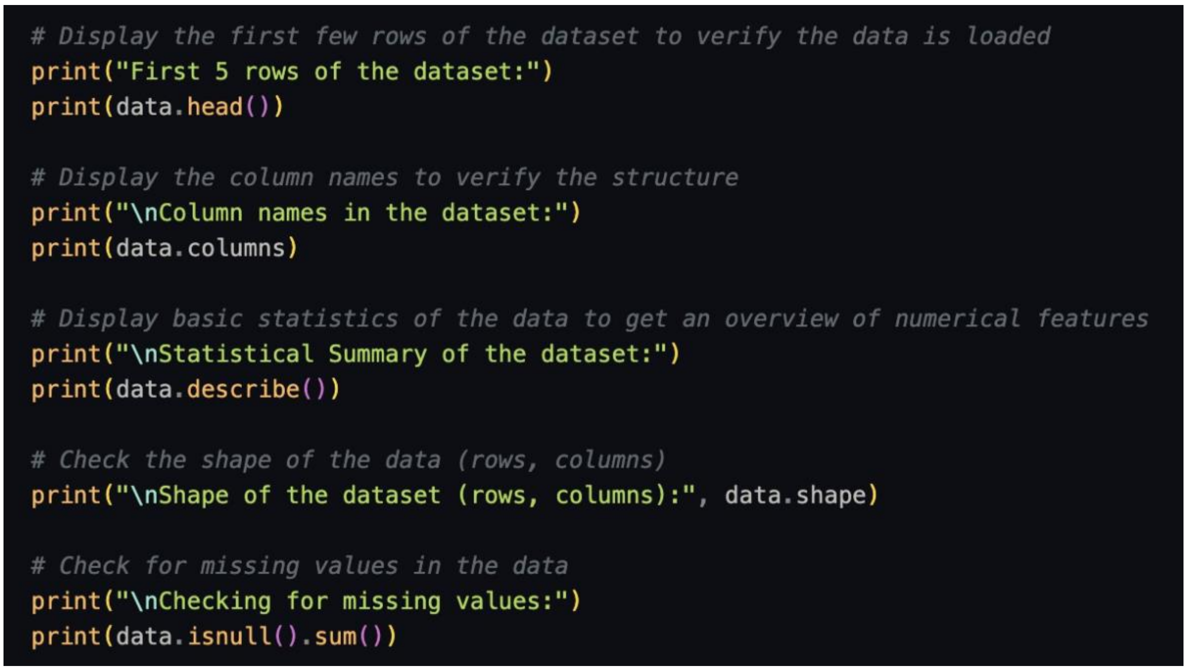
Code Snippet

Here's the code used for loading and inspecting the data:



```
data = pd.read_csv('/Users/sahilnakrani/Documents/NN/Data/munich_weather_2000_2024.csv')
# Check column names
print(data.columns)
```

Figure 5.2: loading data



```
# Display the first few rows of the dataset to verify the data is loaded
print("First 5 rows of the dataset:")
print(data.head())

# Display the column names to verify the structure
print("\nColumn names in the dataset:")
print(data.columns)

# Display basic statistics of the data to get an overview of numerical features
print("\nStatistical Summary of the dataset:")
print(data.describe())

# Check the shape of the data (rows, columns)
print("\nShape of the dataset (rows, columns):", data.shape)

# Check for missing values in the data
print("\nChecking for missing values:")
print(data.isnull().sum())
```

Figure 5.3: checking the loaded data

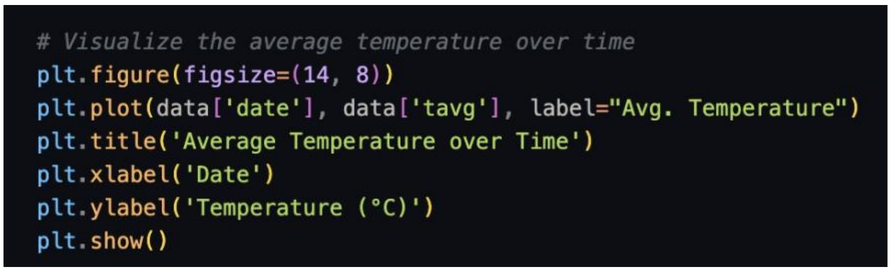
Explanation of the Code

- **Loading the Data**:
- The pd.read_csv() function reads the CSV file into a DataFrame called data.
- **Date Conversion**:
- The pd.to_datetime() function converts the 'date' column to a datetime format for easier manipulation and analysis.
- **Initial Data Checks**:
- data.head() displays the first five rows of the dataset, providing a glimpse into its contents.
- data.columns lists all the columns present in the dataset, allowing us to confirm that we have loaded the correct data.
- data.describe() generates descriptive statistics for all numerical columns, offering insights into their distributions and central tendencies.
- data.shape provides the number of rows and columns, which helps us understand the dataset's size.
- data.isnull().sum() checks for any missing values across all columns, helping to identify data quality issues.

5.0.3 Exploratory Data Analysis

Data visualization is a crucial step in understanding the dataset and identifying patterns. The following plots were generated to analyze the average temperature and other weather-related features over time.

Average Temperature Over Time



```
# Visualize the average temperature over time
plt.figure(figsize=(14, 8))
plt.plot(data['date'], data['tavg'], label="Avg. Temperature")
plt.title('Average Temperature over Time')
plt.xlabel('Date')
plt.ylabel('Temperature (°C)')
plt.show()
```

Figure 5.4: average temperature code

This plot illustrates the average temperature over the specified period, providing insights into seasonal trends and fluctuations.

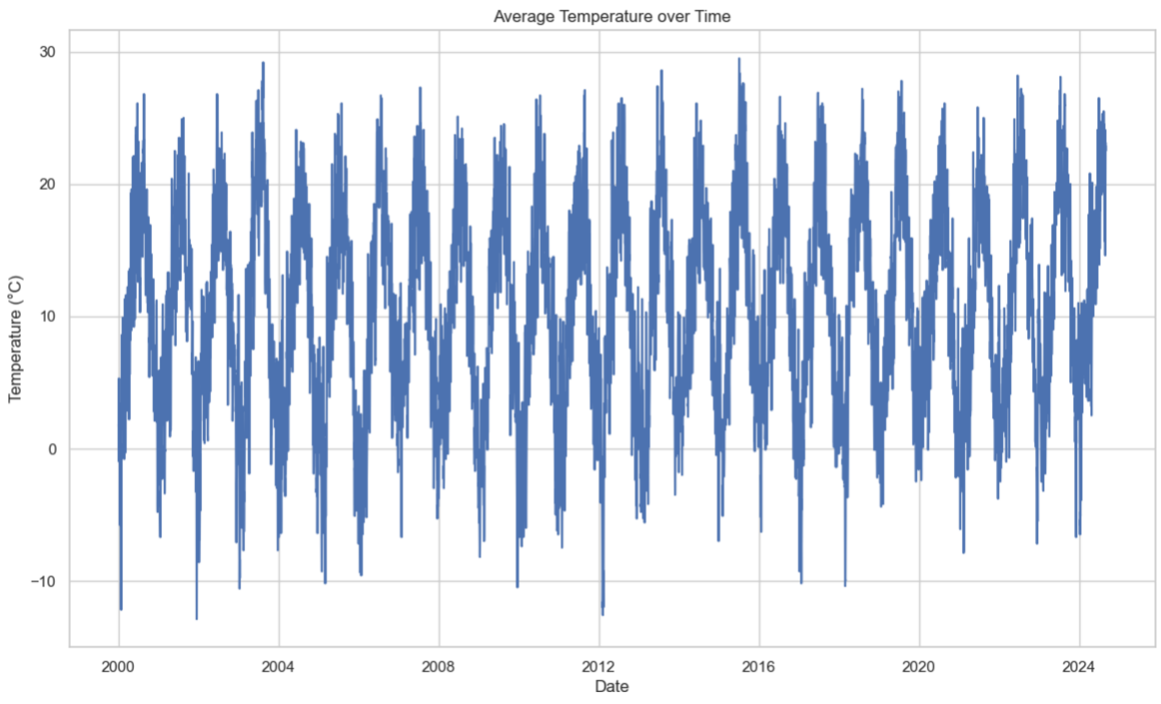
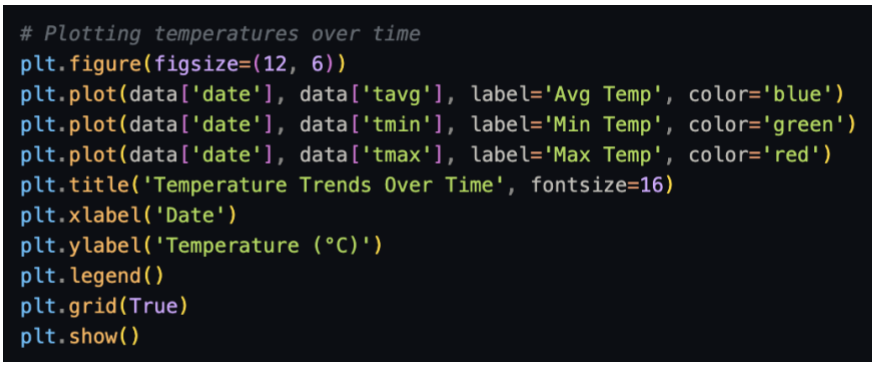


Figure 5.5: average temperature

Temperature Trends



```python
# Plotting temperatures over time
plt.figure(figsize=(12, 6))
plt.plot(data['date'], data['tavg'], label='Avg Temp', color='blue')
plt.plot(data['date'], data['tmin'], label='Min Temp', color='green')
plt.plot(data['date'], data['tmax'], label='Max Temp', color='red')
plt.title('Temperature Trends Over Time', fontsize=16)
plt.xlabel('Date')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.grid(True)
plt.show()
```

Figure 5.6: Temperature Trends code

This plot displays the average, minimum, and maximum temperatures over time, allowing for a comprehensive view of temperature dynamics.

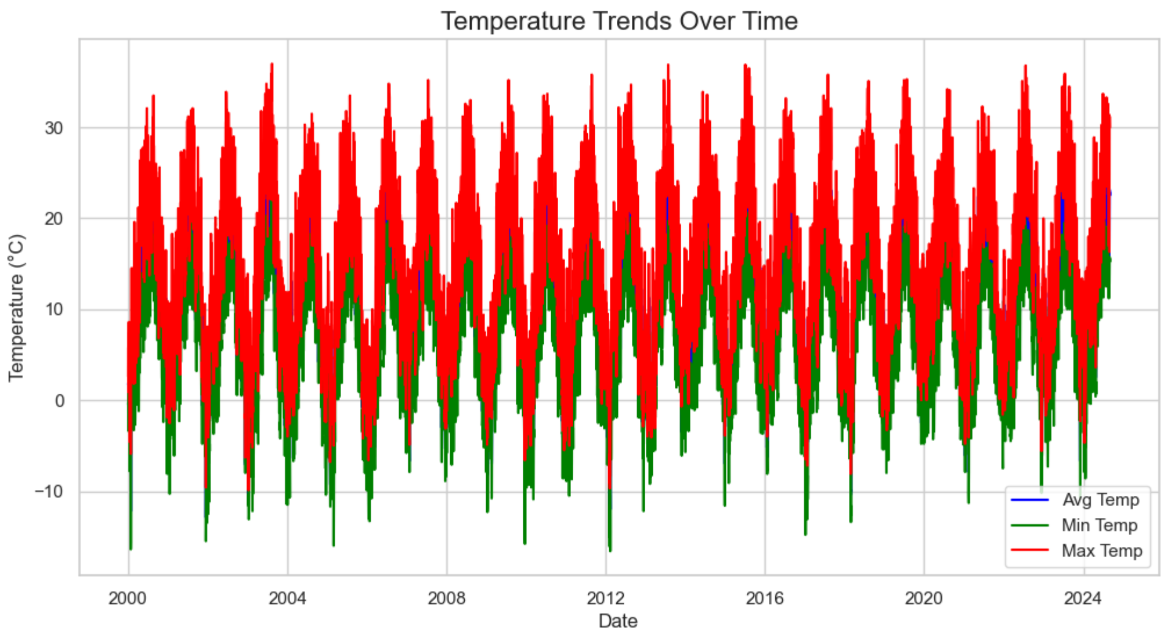
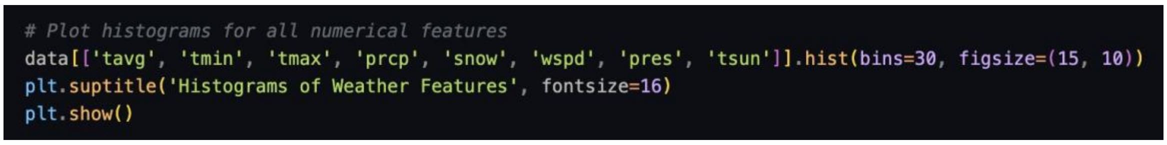


Figure 5.7: Temperature Trends

Histograms of Weather Features



```python
# Plot histograms for all numerical features
data[['tavg', 'tmin', 'tmax', 'prcp', 'snow', 'wspd', 'pres', 'tsun']].hist(bins=30, figsize=(15, 10))
plt.suptitle('Histograms of Weather Features', fontsize=16)
plt.show()
```

Figure 5.8: Histograms of Weather Features code

Histograms provide an overview of the distribution of various weather features, highlighting potential biases or anomalies in the dataset.

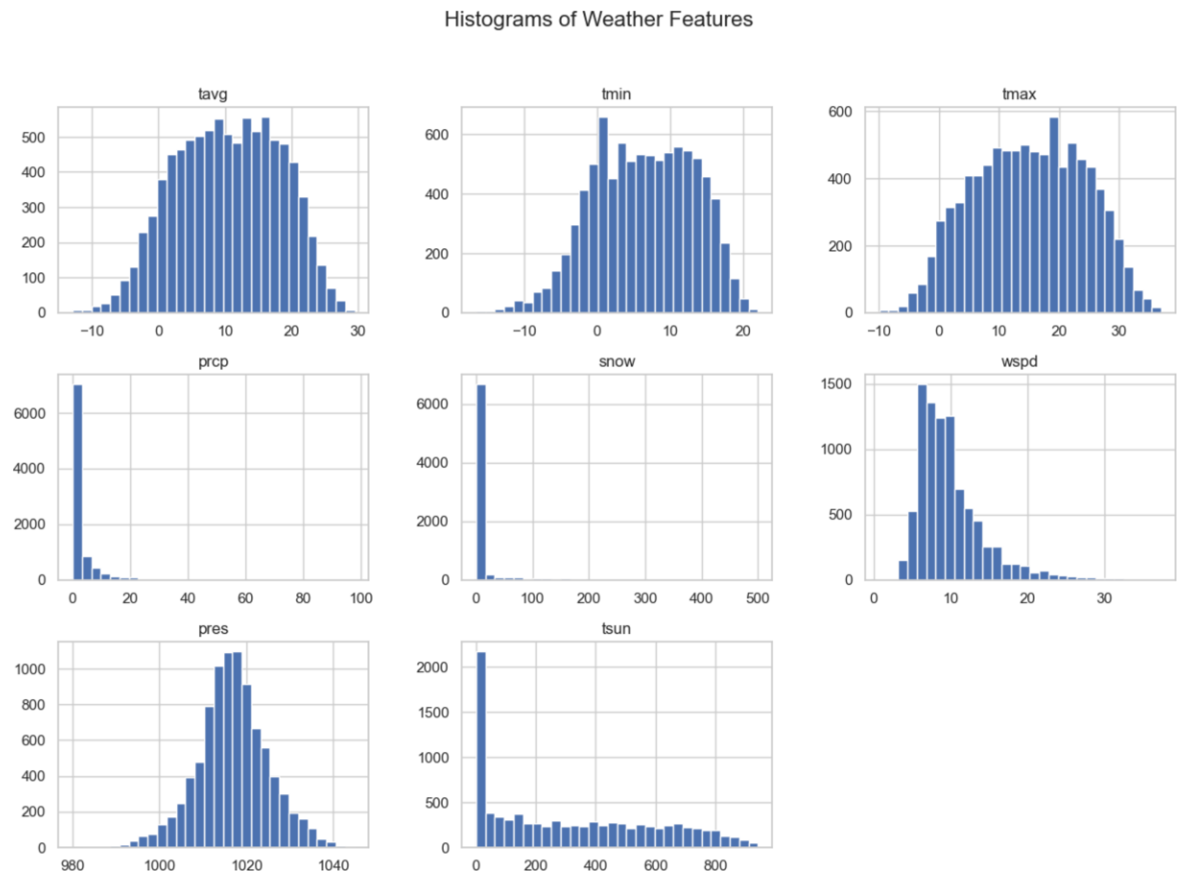
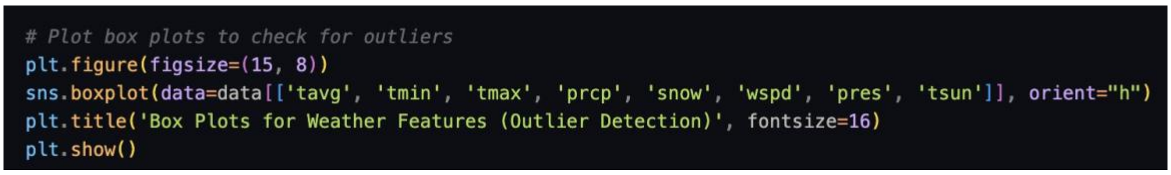


Figure 5.9: Histograms of Weather Features

Outlier Detection



```
# Plot box plots to check for outliers
plt.figure(figsize=(15, 8))
sns.boxplot(data=data[['tavg', 'tmin', 'tmax', 'prcp', 'snow', 'wspd', 'pres', 'tsun']], orient="h")
plt.title('Box Plots for Weather Features (Outlier Detection)', fontsize=16)
plt.show()
```

Figure 5.10: Outlier Detection code

Box plots are utilized to identify outliers in the dataset, which can significantly influence model training and performance.

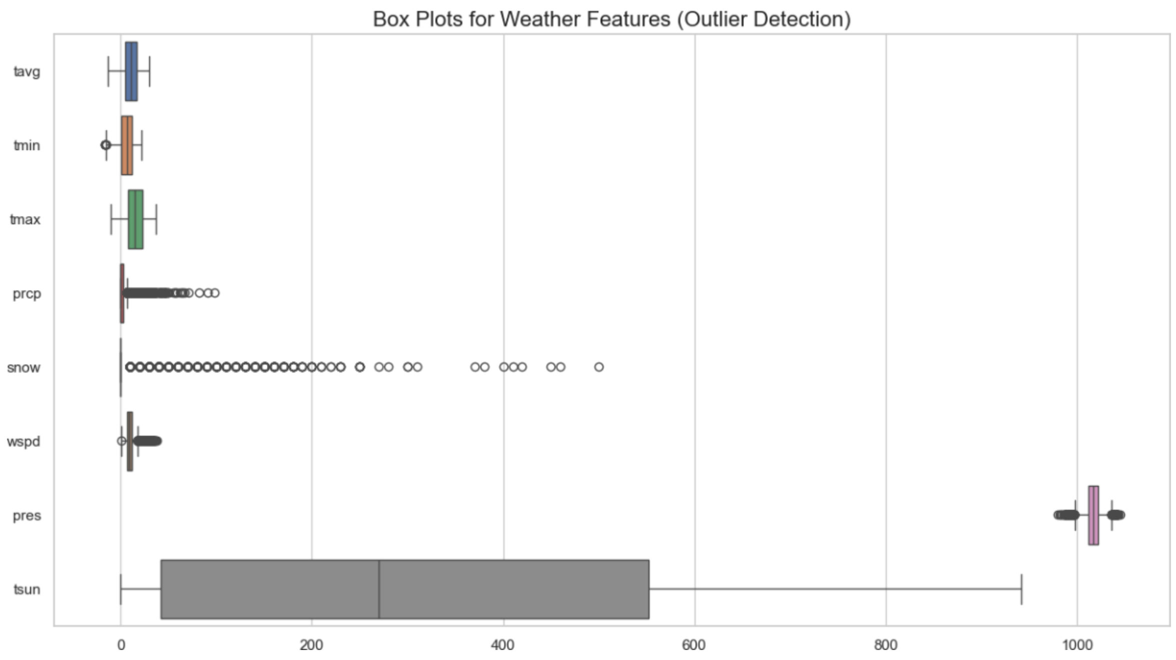
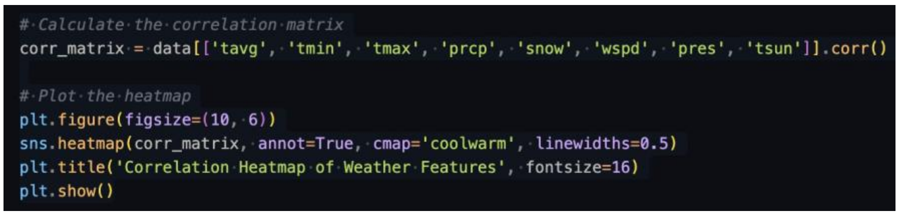


Figure 5.11: Outlier Detection

Correlation Heatmap



```
# Calculate the correlation matrix
corr_matrix = data[['tavg', 'tmin', 'tmax', 'prcp', 'snow', 'wspd', 'pres', 'tsun']].corr()

# Plot the heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap of Weather Features', fontsize=16)
plt.show()
```

Figure 5.12: Correlation Heatmap code

The correlation heatmap provides insights into the relationships between various weather features, informing feature selection for the forecasting model.

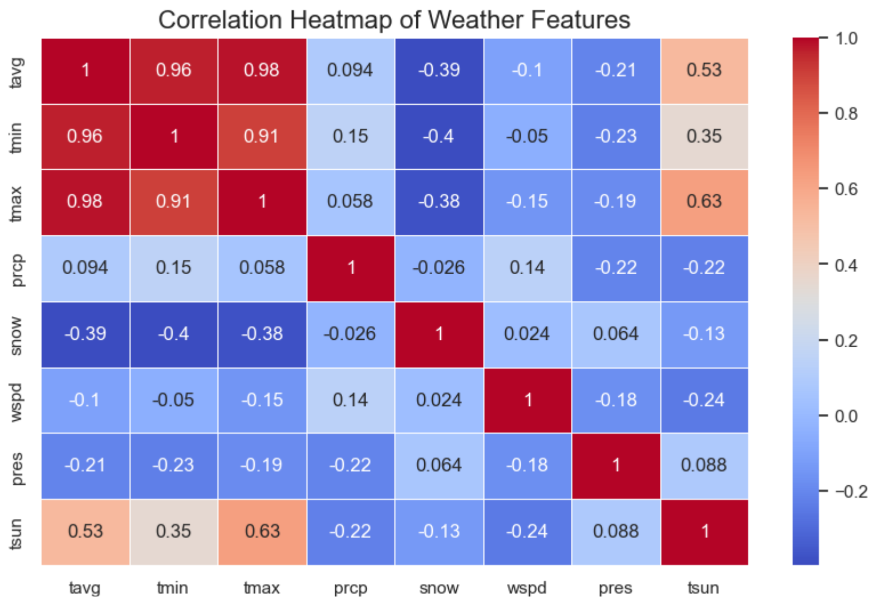
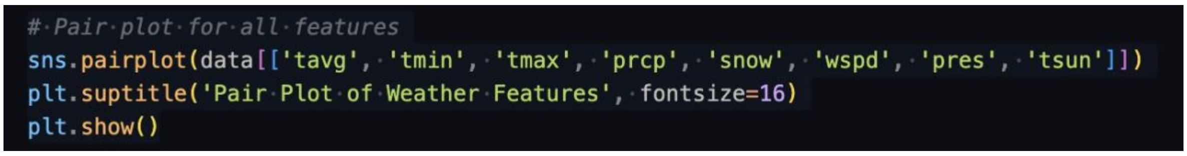


Figure 5.13: Correlation Heatmap

Pair Plot of Weather Features



```
# Pair plot for all features
sns.pairplot(data[['tavg', 'tmin', 'tmax', 'prcp', 'snow', 'wspd', 'pres', 'tsun']])
plt.suptitle('Pair Plot of Weather Features', fontsize=16)
plt.show()
```

Figure 5.14: Pair Plot of Weather Features code

A pair plot displays scatter plots for each pair of features, facilitating the identification of potential relationships and interactions.

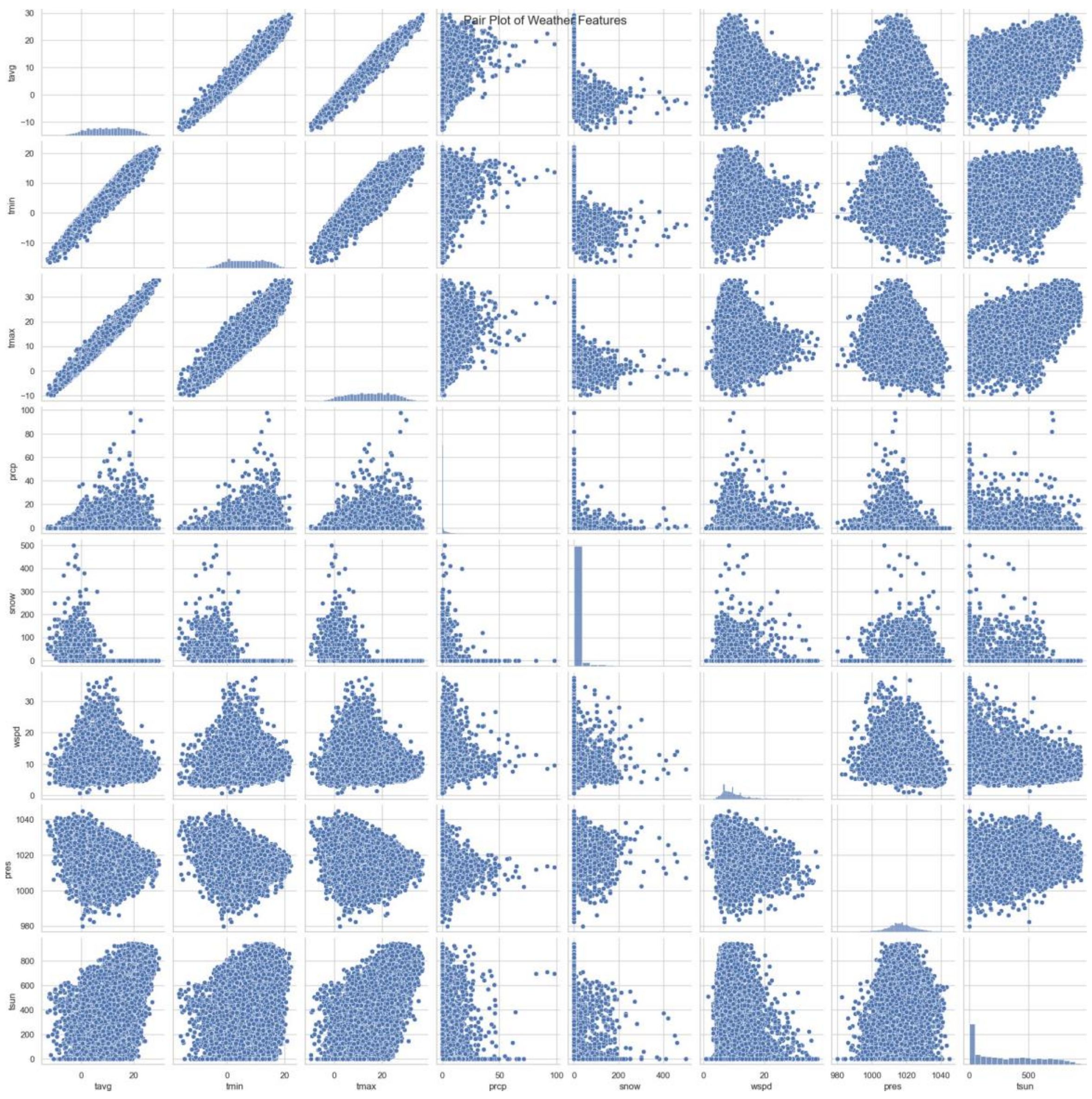
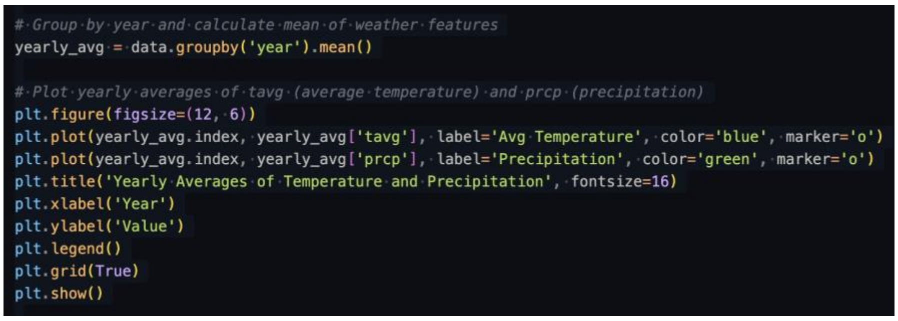


Figure 5.15: Pair Plot of Weather Features

5.0.4 Seasonal Analysis

Yearly Averages



```python
# Group by year and calculate mean of weather features
yearly_avg = data.groupby('year').mean()

# Plot yearly averages of tavg (average temperature) and prcp (precipitation)
plt.figure(figsize=(12, 6))
plt.plot(yearly_avg.index, yearly_avg['tavg'], label='Avg Temperature', color='blue', marker='o')
plt.plot(yearly_avg.index, yearly_avg['prcp'], label='Precipitation', color='green', marker='o')
plt.title('Yearly Averages of Temperature and Precipitation', fontsize=16)
plt.xlabel('Year')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()
```

Figure 5.16: Yearly Averages code

This plot shows the average temperature and precipitation trends over the years, enabling the analysis of long-term climate changes.

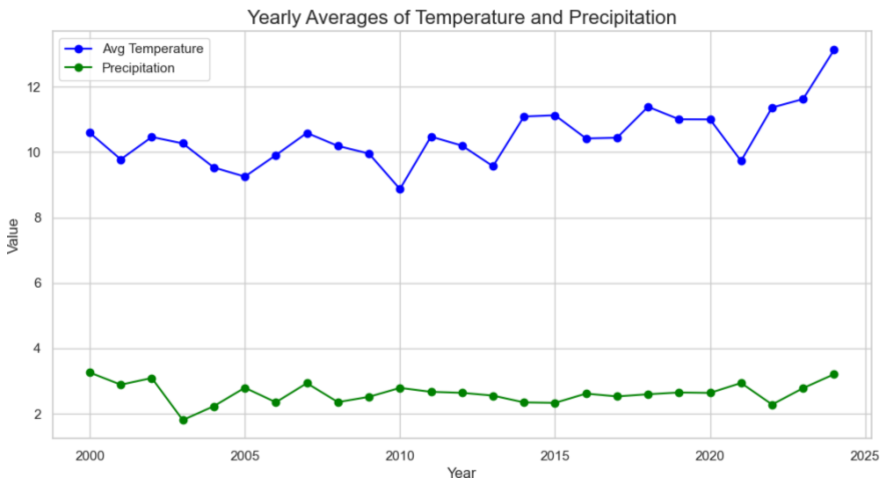
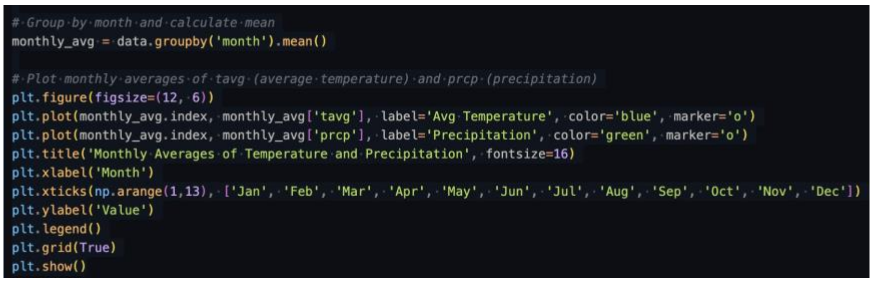


Figure 5.17: Yearly Averages

Monthly Averages



```python
# Group by month and calculate mean
monthly_avg = data.groupby('month').mean()

# Plot monthly averages of tavg (average temperature) and prcp (precipitation)
plt.figure(figsize=(12, 6))
plt.plot(monthly_avg.index, monthly_avg['tavg'], label='Avg Temperature', color='blue', marker='o')
plt.plot(monthly_avg.index, monthly_avg['prcp'], label='Precipitation', color='green', marker='o')
plt.title('Monthly Averages of Temperature and Precipitation', fontsize=16)
plt.xlabel('Month')
plt.xticks(np.arange(1,13), ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()
```

Figure 5.18: Monthly Averages code

This plot highlights the average temperature and precipitation by month, illustrating seasonal patterns in the data.

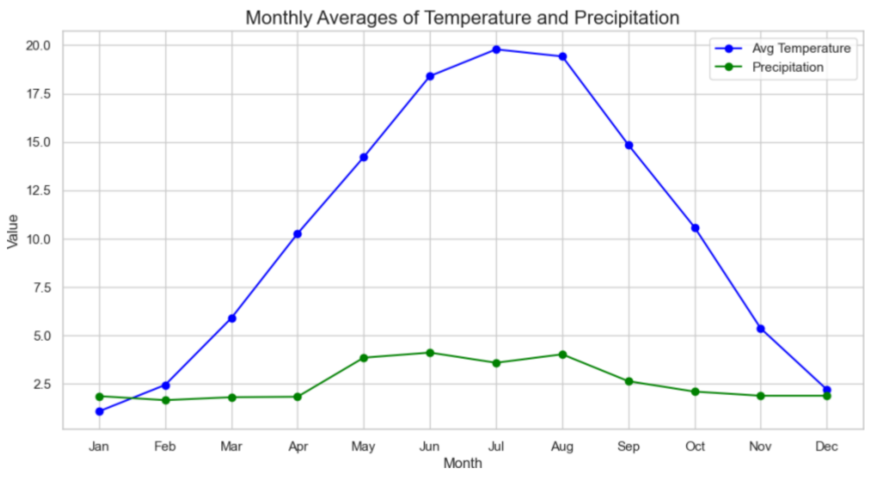
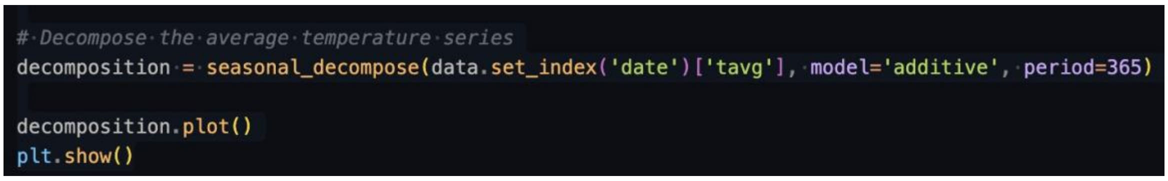


Figure 5.19: Monthly Averages

Time Series Decomposition



```
# Decompose the average temperature series
decomposition = seasonal_decompose(data.set_index('date')['tavg'], model='additive', period=365)

decomposition.plot()
plt.show()
```

Figure 5.20: Time Series Decomposition code

Time series decomposition helps separate the average temperature into trend, seasonal, and residual components, providing a clearer view of underlying patterns.

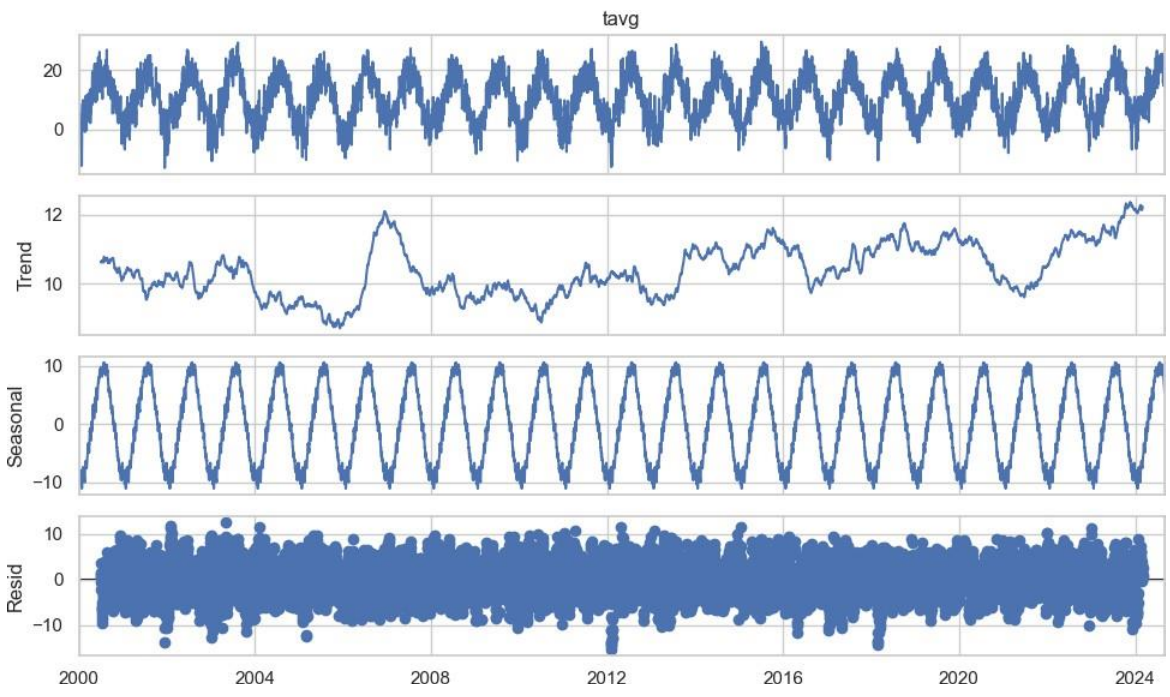
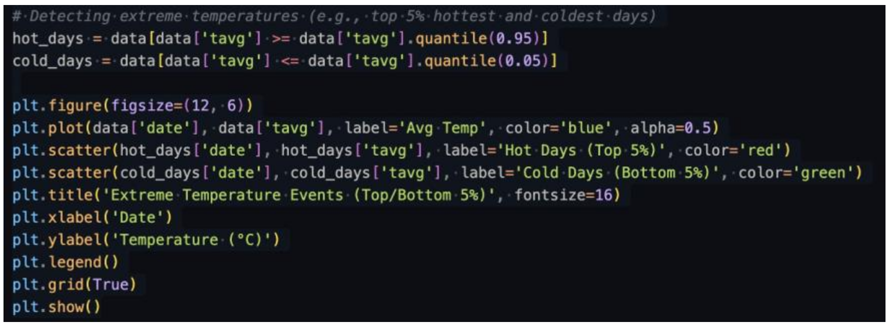


Figure 5.21: Time Series Decomposition

Extreme Temperature Detection



```
# Detecting extreme temperatures (e.g., top 5% hottest and coldest days)
hot_days = data[data['tavg'] >= data['tavg'].quantile(0.95)]
cold_days = data[data['tavg'] <= data['tavg'].quantile(0.05)]

plt.figure(figsize=(12, 6))
plt.plot(data['date'], data['tavg'], label='Avg Temp', color='blue', alpha=0.5)
plt.scatter(hot_days['date'], hot_days['tavg'], label='Hot Days (Top 5%)', color='red')
plt.scatter(cold_days['date'], cold_days['tavg'], label='Cold Days (Bottom 5%)', color='green')
plt.title('Extreme Temperature Events (Top/Bottom 5%)', fontsize=16)
plt.xlabel('Date')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.grid(True)
plt.show()
```

Figure 5.22: Extreme Temperature Detection code

This visualization identifies extreme temperature events by highlighting the hottest and coldest days, which is important for understanding climate variability.

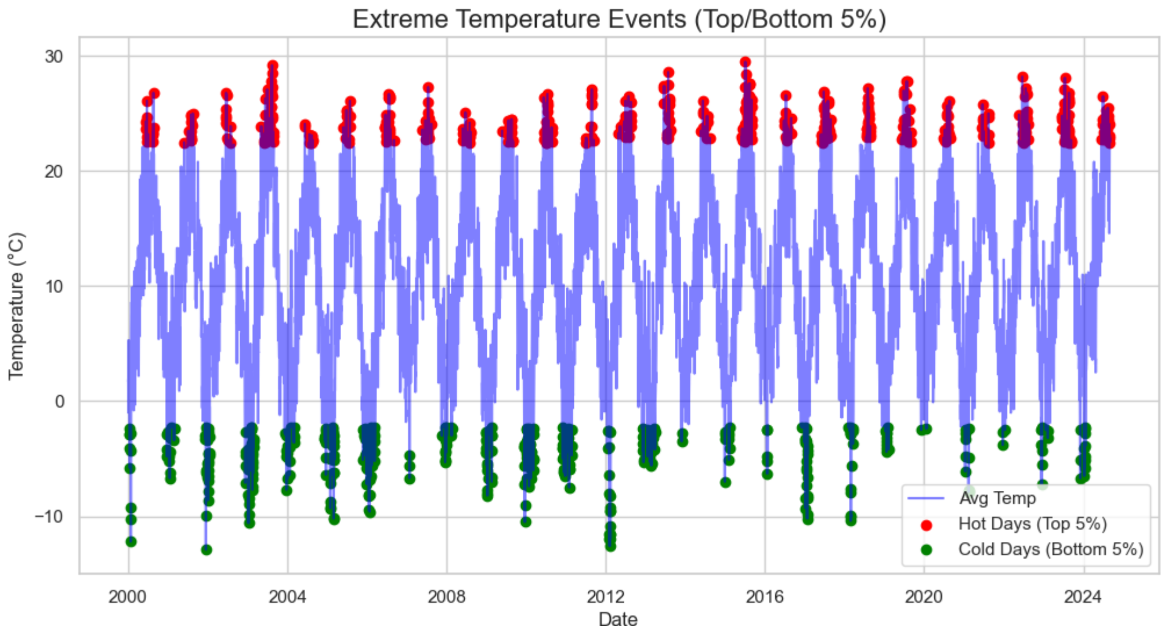
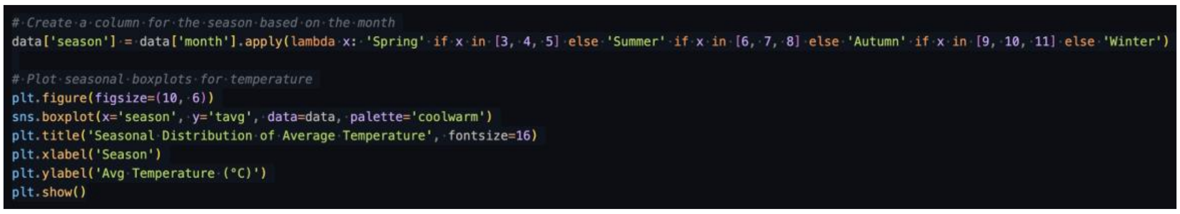


Figure 5.23: Extreme Temperature Detection

Seasonal Analysis of Temperature



```
# Create a column for the season based on the month
data['season'] = data['month'].apply(lambda x: 'Spring' if x in [3, 4, 5] else 'Summer' if x in [6, 7, 8] else 'Autumn' if x in [9, 10, 11] else 'Winter')

# Plot seasonal boxplots for temperature
plt.figure(figsize=(10, 6))
sns.boxplot(x='season', y='tavg', data=data, palette='coolwarm')
plt.title('Seasonal Distribution of Average Temperature', fontsize=16)
plt.xlabel('Season')
plt.ylabel('Avg Temperature (°C)')
plt.show()
```

Figure 5.24: Seasonal Analysis of Temperature code

This box plot illustrates the distribution of average temperatures across different seasons, enabling an understanding of seasonal temperature variations.

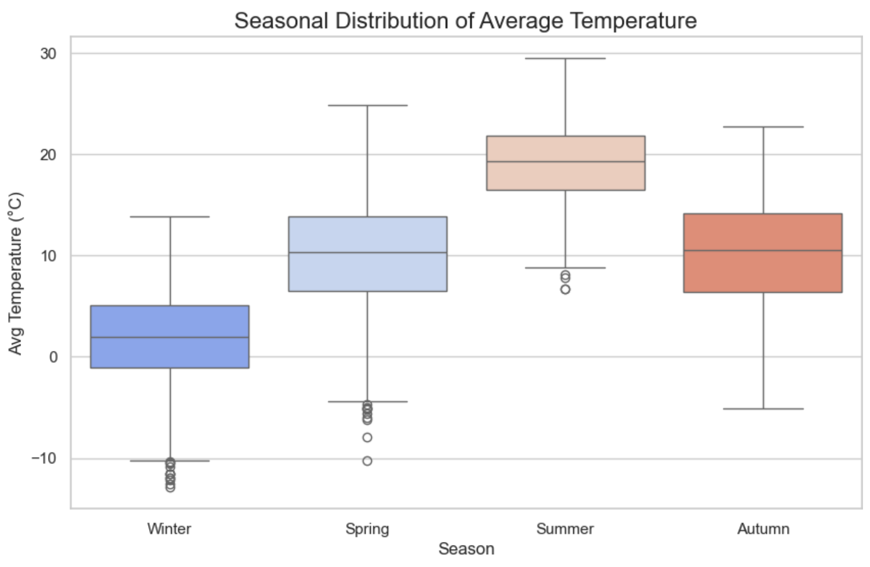
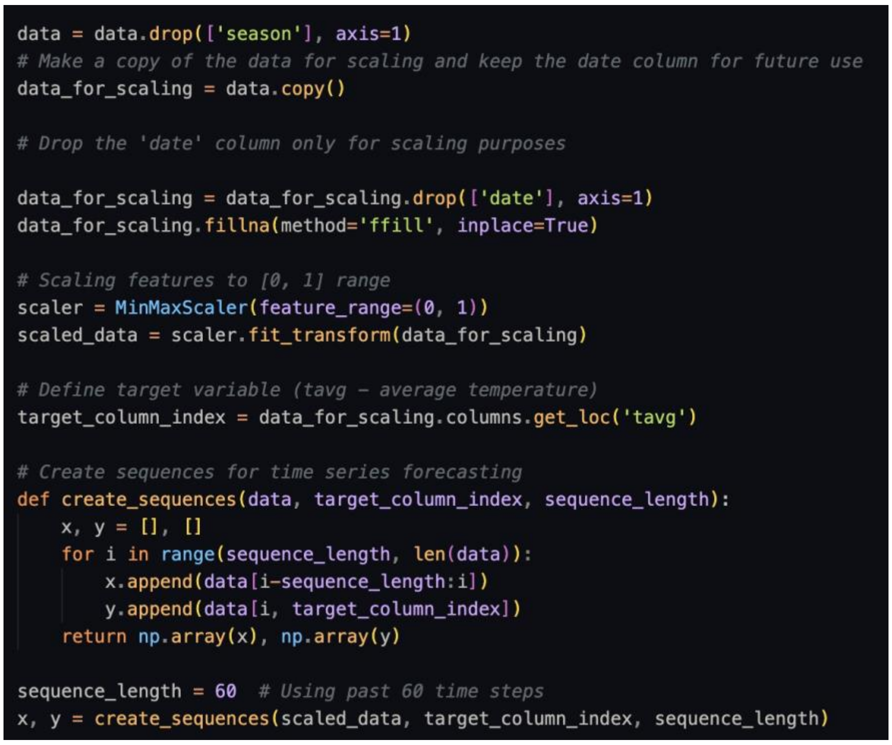


Figure 5.25: Seasonal Analysis of Temperature

5.0.5 Feature Engineering

In the feature engineering phase, we prepare our dataset for the LSTM model by performing a series of steps, including dropping unnecessary columns, scaling the data, and creating sequences suitable for time series forecasting. Below, each step is outlined with accompanying code snippets and explanations.



```python
data = data.drop(['season'], axis=1)
# Make a copy of the data for scaling and keep the date column for future use
data_for_scaling = data.copy()

# Drop the 'date' column only for scaling purposes

data_for_scaling = data_for_scaling.drop(['date'], axis=1)
data_for_scaling.fillna(method='ffill', inplace=True)

# Scaling features to [0, 1] range
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data_for_scaling)

# Define target variable (tavg - average temperature)
target_column_index = data_for_scaling.columns.get_loc('tavg')

# Create sequences for time series forecasting
def create_sequences(data, target_column_index, sequence_length):
    x, y = [], []
    for i in range(sequence_length, len(data)):
        x.append(data[i-sequence_length:i])
        y.append(data[i, target_column_index])
    return np.array(x), np.array(y)

sequence_length = 60  # Using past 60 time steps
x, y = create_sequences(scaled_data, target_column_index, sequence_length)
```

Figure 5.26: Feature Engineering

Dropping Unnecessary Columns

We first drop the 'season' column, which was created for exploratory data analysis but is not required for the modeling process. This step simplifies our dataset and focuses on the relevant features.

Preparing the Data for Scaling

Next, we create a copy of the dataset for scaling purposes while preserving the original data for future reference. The 'date' column is dropped to ensure that only numerical features are scaled.

Handling Missing Values

Before scaling, it is essential to handle any missing values in the dataset. We employ a forward fill method to fill in any NaN values.

Normalizing Features to a Range of [0, 1]

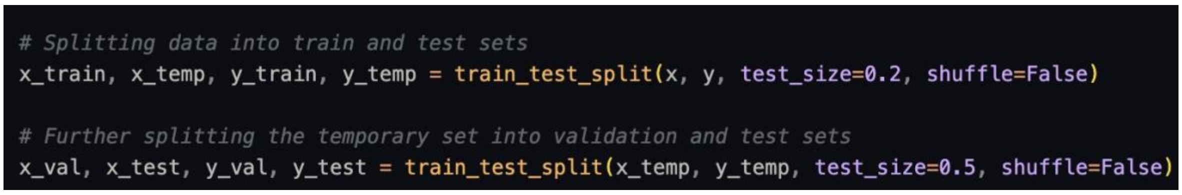
We utilize the MinMaxScaler from the scikit-learn library to scale our features to a range between 0 and 1. This scaling helps improve the convergence of the LSTM model during training.

Defining the Target Variable

We define the target variable, which is the average temperature (tavg). The index of this column is obtained for later use in the sequence creation.

5.0.6 Data Splitting and Model Building

Splitting Data into Train, Validation, and Test Sets



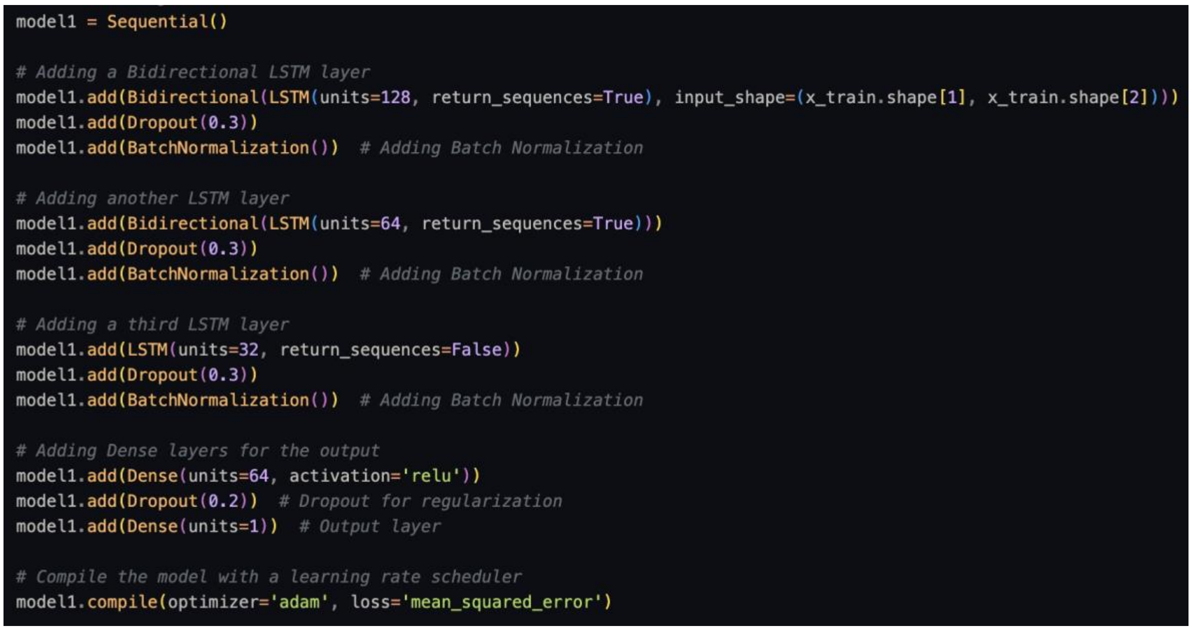
```python
# Splitting data into train and test sets
x_train, x_temp, y_train, y_temp = train_test_split(x, y, test_size=0.2, shuffle=False)

# Further splitting the temporary set into validation and test sets
x_val, x_test, y_val, y_test = train_test_split(x_temp, y_temp, test_size=0.5, shuffle=False)
```

Figure 5.27: Splitting Data

First, we will split our data into training, validation, and test sets. We will allocate 80% of the data for training, 10% for validation, and 10% for testing. This allows us to tune our model based on the validation set while keeping the test set completely separate for final evaluation.

Building the Enhanced LSTM Model

With the data split appropriately, we can now build our enhanced LSTM model. Below is the code that constructs the model:

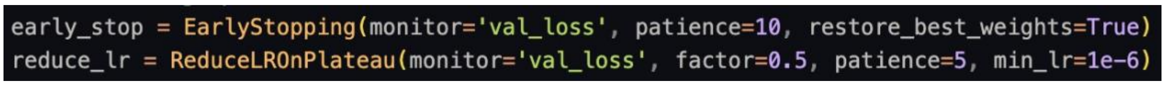


```
model1 = Sequential()

# Adding a Bidirectional LSTM layer
model1.add(Bidirectional(LSTM(units=128, return_sequences=True), input_shape=(x_train.shape[1], x_train.shape[2])))
model1.add(Dropout(0.3))
model1.add(BatchNormalization())  # Adding Batch Normalization

# Adding another LSTM layer
model1.add(Bidirectional(LSTM(units=64, return_sequences=True)))
model1.add(Dropout(0.3))
model1.add(BatchNormalization())  # Adding Batch Normalization

# Adding a third LSTM layer
model1.add(LSTM(units=32, return_sequences=False))
model1.add(Dropout(0.3))
model1.add(BatchNormalization())  # Adding Batch Normalization

# Adding Dense layers for the output
model1.add(Dense(units=64, activation='relu'))
model1.add(Dropout(0.2))  # Dropout for regularization
model1.add(Dense(units=1))  # Output layer

# Compile the model with a learning rate scheduler
model1.compile(optimizer='adam', loss='mean_squared_error')
```

Figure 5.28: Building Model

Setting up Callbacks

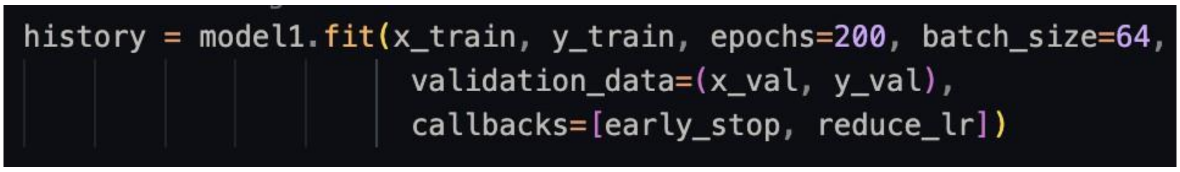
We will also set up callbacks to monitor the model's performance during training. This includes an early stopping mechanism and a learning rate reduction strategy.



```
early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6)
```

Figure 5.29: Setting Up Callbacks

Training the Model



```
history = model1.fit(x_train, y_train, epochs=200, batch_size=64,
                     validation_data=(x_val, y_val),
                     callbacks=[early_stop, reduce_lr])
```

Figure 5.30: Training the Model

Finally, we can train the model using the training data while validating its performance on the validation set.

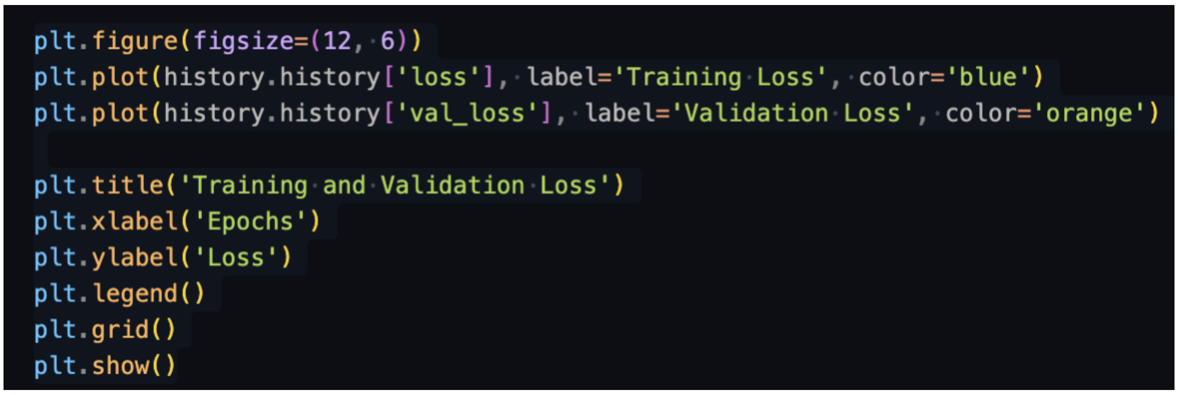
Explanation of the Code

- **Data Splitting**: We split the dataset into training, validation, and testing sets to ensure robust model evaluation and tuning. The validation set helps in hyperparameter tuning and avoids overfitting.
- **Model Architecture**: The model uses three layers of LSTM units wrapped in Bidirectional layers, along with Dropout and Batch Normalization to enhance training stability and prevent overfitting.
- **Callbacks**: The EarlyStopping callback stops training when the validation loss does not improve for a set number of epochs, while ReduceLROnPlateau reduces the learning rate when the validation loss stagnates, promoting better convergence.

5.0.7 Loss Visualization

To assess the performance of the enhanced LSTM model, we will visualize the training and validation loss over the epochs. This allows us to monitor how well the model learns from the training data and its generalization capability to unseen data.

Plotting Training and Validation Loss



```
plt.figure(figsize=(12, 6))
plt.plot(history.history['loss'], label='Training Loss', color='blue')
plt.plot(history.history['val_loss'], label='Validation Loss', color='orange')

plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```

Figure 5.31: Loss Visualization

Explanation of the Code

- **Plotting**: The plt.plot() function is used to create the line plots for both training loss and validation loss. The training loss is represented in blue, while the validation loss is shown in orange.

• **Figure Customization**: The figure size is set to 12x6 for better visibility. The title and labels for the axes are added for clarity, and a legend is included to distinguish between the two loss curves.

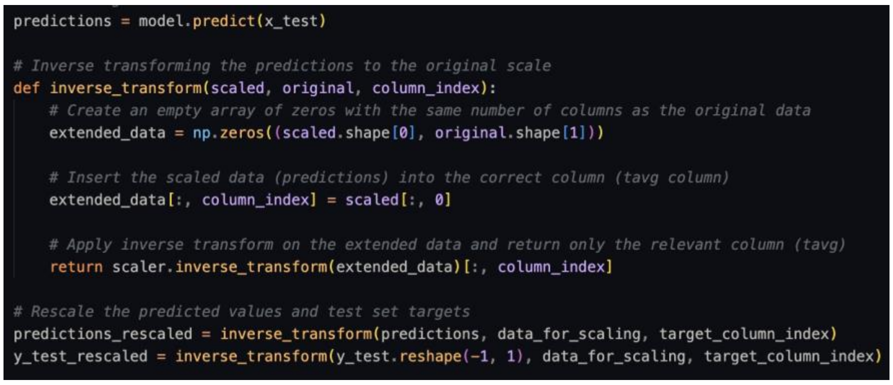
• **Grid**: The grid is enabled to enhance readability of the plot.

5.0.8 Model Evaluation

After training the enhanced LSTM model, we will evaluate its performance by making predictions on the test set and calculating key metrics such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R^2 score. These metrics will help us understand how well the model predicts the average temperature based on the input features.

Making Predictions

We will first predict the values on the test set and then rescale these predictions back to the original scale for better interpretability.

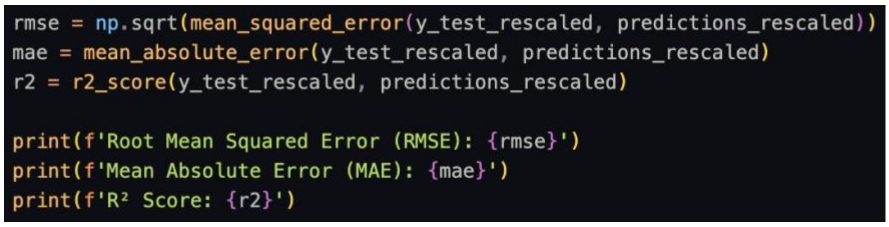


```
predictions = model.predict(x_test)

# Inverse transforming the predictions to the original scale
def inverse_transform(scaled, original, column_index):
    # Create an empty array of zeros with the same number of columns as the original data
    extended_data = np.zeros((scaled.shape[0], original.shape[1]))

    # Insert the scaled data (predictions) into the correct column (tavg column)
    extended_data[:, column_index] = scaled[:, 0]

    # Apply inverse transform on the extended data and return only the relevant column (tavg)
    return scaler.inverse_transform(extended_data)[:, column_index]

# Rescale the predicted values and test set targets
predictions_rescaled = inverse_transform(predictions, data_for_scaling, target_column_index)
y_test_rescaled = inverse_transform(y_test.reshape(-1, 1), data_for_scaling, target_column_index)
```

Figure 5.32: Prediction

Calculating Performance Metrics

Next, we will calculate RMSE, MAE, and R^2 score to assess the model's performance.



```
rmse = np.sqrt(mean_squared_error(y_test_rescaled, predictions_rescaled))
mae = mean_absolute_error(y_test_rescaled, predictions_rescaled)
r2 = r2_score(y_test_rescaled, predictions_rescaled)

print(f'Root Mean Squared Error (RMSE): {rmse}')
print(f'Mean Absolute Error (MAE): {mae}')
print(f'R² Score: {r2}')
```

Figure 5.33: Performance Matrix

Explanation of the Code

• **Prediction**: We use the model1.predict() function to generate predictions based on the test set features (x_test).

• **Inverse Transformation**: The inverse_transform function is designed to transform the scaled predictions back to the original scale of the target variable (average temperature). It creates an empty array, inserts the scaled predictions into the corresponding column, and applies the inverse scaling using the previously fitted scaler.

• **Performance Metrics Calculation**:

• **RMSE**: Root Mean Squared Error is calculated to measure the average magnitude of the errors between predicted and actual values, giving higher weight to larger errors.

• **MAE**: Mean Absolute Error provides the average absolute difference between predicted and actual values, offering a straightforward measure of prediction accuracy.

• **R^2 Score**: R^2 score (coefficient of determination) indicates how well the predicted values approximate the actual data. It ranges from 0 to 1, where a score closer to 1 implies a better fit.

Interpretation of the Metrics

• **RMSE and MAE**: Lower values of RMSE and MAE indicate better predictive performance. These metrics provide insight into the average errors the model makes, allowing for comparisons against other models or benchmarks.

• **R^2 Score**: A higher R^2 score (closer to 1) suggests that the model explains a significant proportion of the variance in the target variable. Conversely, a score of 0 indicates that the model does not explain any of the variance.

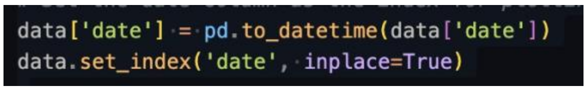
By evaluating these metrics, we can gain a comprehensive understanding of the model's performance and its applicability for weather forecasting.

5.0.9 Forecasting Future Temperatures

In this section, we will generate forecasts for average temperatures over the next 45 days using our trained LSTM model. This is crucial for understanding how the model can be applied for future weather predictions, helping to assess its practicality and effectiveness in real-world scenarios.

Preparing Data for Forecasting

Before we begin the forecasting process, we set the date column as the index of our DataFrame to facilitate plotting and date manipulations later on.

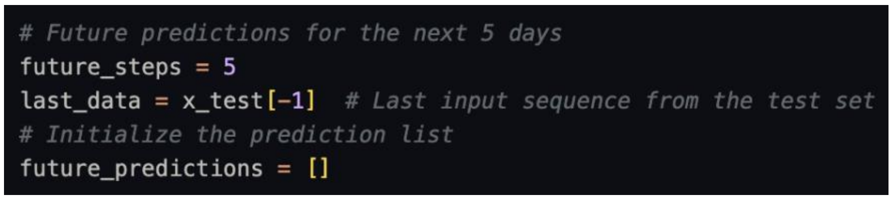


```python
data['date'] = pd.to_datetime(data['date'])
data.set_index('date', inplace=True)
```

Figure 5.34: Preparing for Forecast

Future Predictions Setup

We define the number of future steps we want to predict. In this case, we are aiming to forecast the average temperature for the next 5 days.



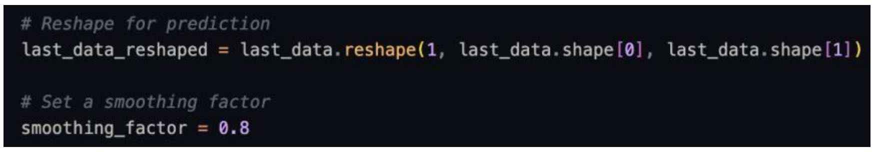
```python
# Future predictions for the next 5 days
future_steps = 5
last_data = x_test[-1]  # Last input sequence from the test set
# Initialize the prediction list
future_predictions = []
```

Figure 5.35: SetUp for forecast

We will utilize the last sequence from the test set as the starting point for our predictions. This allows us to leverage the most recent historical data to project into the future.

Reshaping and Smoothing Predictions

Before making predictions, we reshape the last input data sequence to fit the model's expected input dimensions. Additionally, we apply a smoothing factor to the predictions to stabilize them over the forecast period.



```python
# Reshape for prediction
last_data_reshaped = last_data.reshape(1, last_data.shape[0], last_data.shape[1])

# Set a smoothing factor
smoothing_factor = 0.8
```

Figure 5.36: Reshaping and Smoothing

Generating Future Predictions

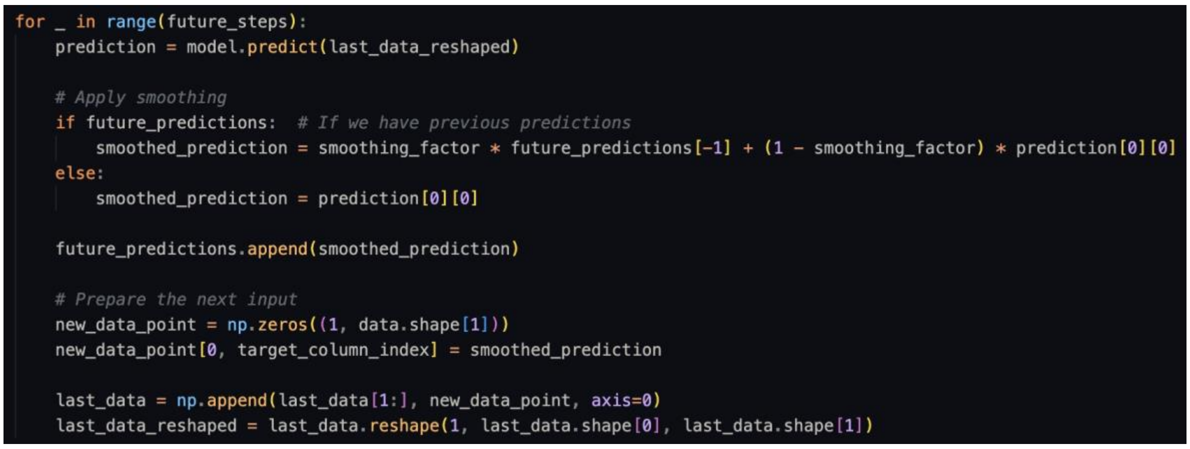
We will then generate future predictions in a loop. For each step, we will:

1. Predict the next average temperature using the model.

2. Apply the smoothing factor to the predicted values to reduce fluctuations.

3. Prepare the next input sequence for the model based on the latest prediction.

The process is as follows:



```python
for _ in range(future_steps):
    prediction = model.predict(last_data_reshaped)

    # Apply smoothing
    if future_predictions:  # If we have previous predictions
        smoothed_prediction = smoothing_factor * future_predictions[-1] + (1 - smoothing_factor) * prediction[0][0]
    else:
        smoothed_prediction = prediction[0][0]

    future_predictions.append(smoothed_prediction)

    # Prepare the next input
    new_data_point = np.zeros((1, data.shape[1]))
    new_data_point[0, target_column_index] = smoothed_prediction

    last_data = np.append(last_data[1:], new_data_point, axis=0)
    last_data_reshaped = last_data.reshape(1, last_data.shape[0], last_data.shape[1])
```

Figure 5.37: Future Prediction

• **Prediction**: The model predicts the average temperature for the next time step based on the last input sequence.

• **Smoothing**: The smoothing process combines the previous prediction with the new prediction to create a more stable forecast. This reduces the impact of outliers and ensures the predicted values do not fluctuate wildly from one day to the next.

• **Updating Input Sequence**: After each prediction, the new predicted temperature is appended to the last input sequence to form a new input for the next iteration.

Preparing Future Dates

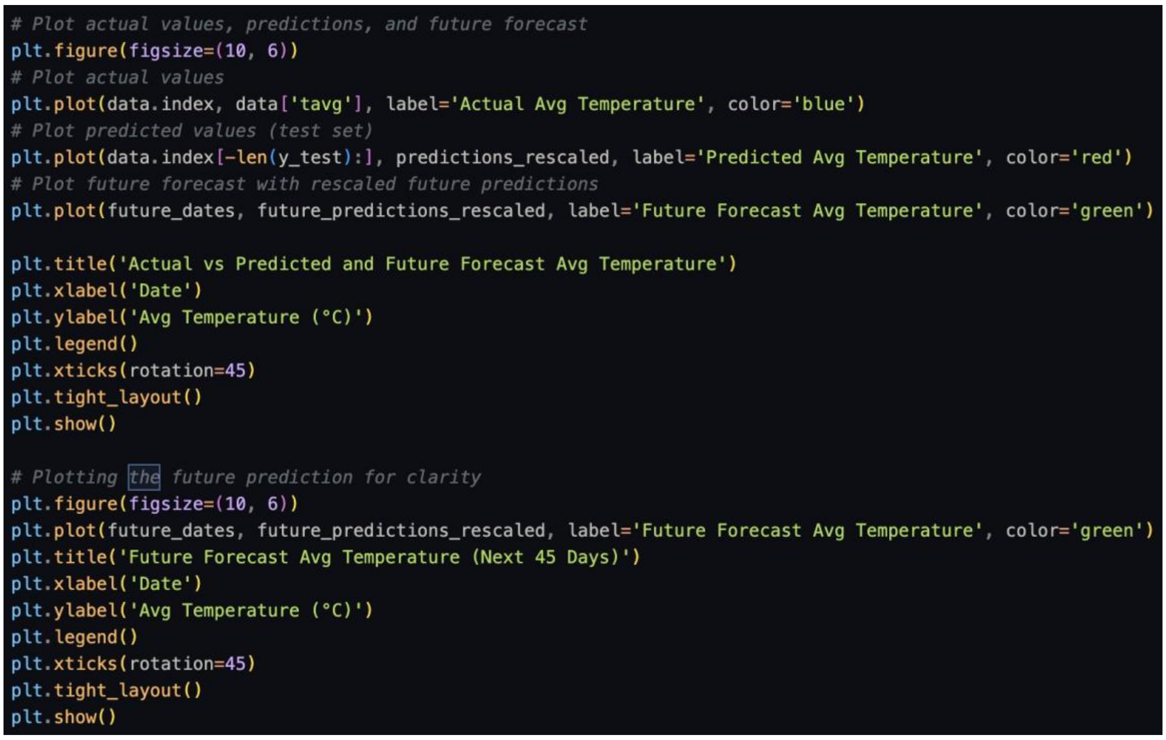
Next, we generate the future dates corresponding to the predicted values. This step ensures that the plotted forecast aligns with the correct dates.

Inverse Transforming Future Predictions

To make the future predictions interpretable, we need to apply the inverse transformation to convert the scaled predictions back to the original temperature scale.

Plotting Results

Finally, we visualize the actual temperatures, the predictions from the test set, and the future forecasts. This visual representation is crucial for understanding the model's performance and future forecasting capabilities.



```python
# Plot actual values, predictions, and future forecast
plt.figure(figsize=(10, 6))
# Plot actual values
plt.plot(data.index, data['tavg'], label='Actual Avg Temperature', color='blue')
# Plot predicted values (test set)
plt.plot(data.index[-len(y_test):], predictions_rescaled, label='Predicted Avg Temperature', color='red')
# Plot future forecast with rescaled future predictions
plt.plot(future_dates, future_predictions_rescaled, label='Future Forecast Avg Temperature', color='green')

plt.title('Actual vs Predicted and Future Forecast Avg Temperature')
plt.xlabel('Date')
plt.ylabel('Avg Temperature (°C)')
plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Plotting the future prediction for clarity
plt.figure(figsize=(10, 6))
plt.plot(future_dates, future_predictions_rescaled, label='Future Forecast Avg Temperature', color='green')
plt.title('Future Forecast Avg Temperature (Next 45 Days)')
plt.xlabel('Date')
plt.ylabel('Avg Temperature (°C)')
plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Figure 5.38: Plotting Forecast

Explanation of the Plots

1. **Overall Plot**: The first plot displays the actual average temperatures, the predictions from the model, and the future forecasts on the same timeline. This visualization helps to compare how closely the predictions align with the actual values and how the model anticipates future trends.

2. **Future Forecast Plot**: The second plot focuses solely on the forecast for the next 5 days. This allows for a clear understanding of how the model projects future temperature trends.

The forecasting section illustrates how our trained LSTM model can be utilized to predict future average temperatures. The smoothing technique employed helps to stabilize predictions, making them more reliable. Visualizing both actual and predicted values enhances our understanding of the model's performance and provides insights into its applicability for practical weather forecasting.

CHAPTER 6

Experimental results

This section presents the experimental results obtained from the implementation of the weather forecasting model. The performance of the model is evaluated using various metrics, including Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R^2 score. Additionally, we provide visual representations of the training and validation losses during model training, as well as the results of our future forecasting.

6.0.1 Model Evaluation Metrics

After training the model, we evaluated its performance on the test dataset using the following metrics:

Mean Absolute Error (MAE)

The Mean Absolute Error (MAE) is a measure of errors between paired observations. It indicates how close predictions are to the actual outcomes, providing a straightforward interpretation of prediction accuracy.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{6.0.1}$$

Root Mean Squared Error (RMSE)

The Root Mean Squared Error (RMSE) provides a measure of the differences between predicted and actual values, emphasizing larger errors more than smaller ones. It is calculated as follows:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{6.0.2}$$

R² Score

The R^2 score, also known as the coefficient of determination, indicates how well the predicted values approximate the actual data points. It ranges from 0 to 1, with 1 indicating perfect predictions.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{6.0.3}$$

Result:

R² Score: **0.7257**

6.0.2 Loss Graphs

During training, the model's performance was tracked through the loss metrics for both the training and validation datasets. The loss curves help us visualize how well the model is learning over epochs and whether it is overfitting.

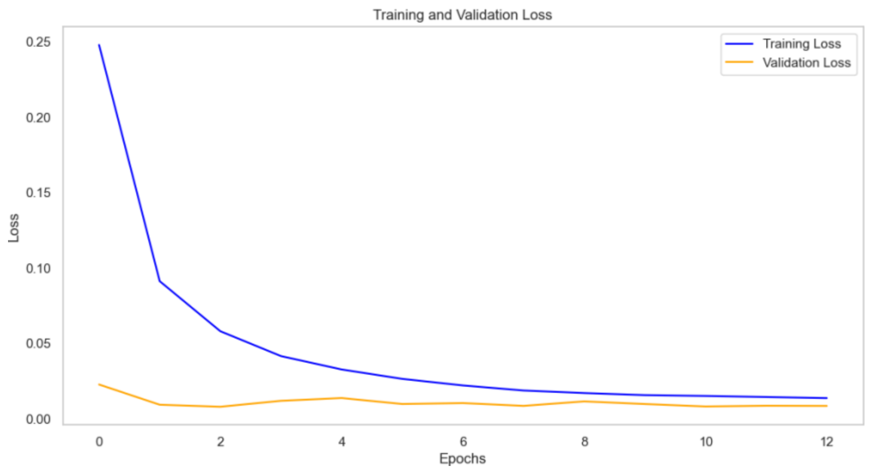


Figure 6.1: Loss Graph

Explanation of Loss Graphs

- **Training Loss**: The blue line represents the loss on the training dataset. A decrease in training loss indicates that the model is learning and improving its predictions on the training data.

- **Validation Loss**: The orange line shows the loss on the validation dataset. A decreasing validation loss suggests that the model generalizes well to unseen data. However, if the validation loss starts increasing while the training loss continues to decrease, it indicates overfitting.

The plotted graph will help in assessing the model's learning behavior and its potential overfitting.

6.0.3 Forecasting Results

To validate the forecasting capability of the model, we generated predictions for future average temperatures. The following graphs illustrate the actual temperature values, the model's predictions, and the future forecasts.

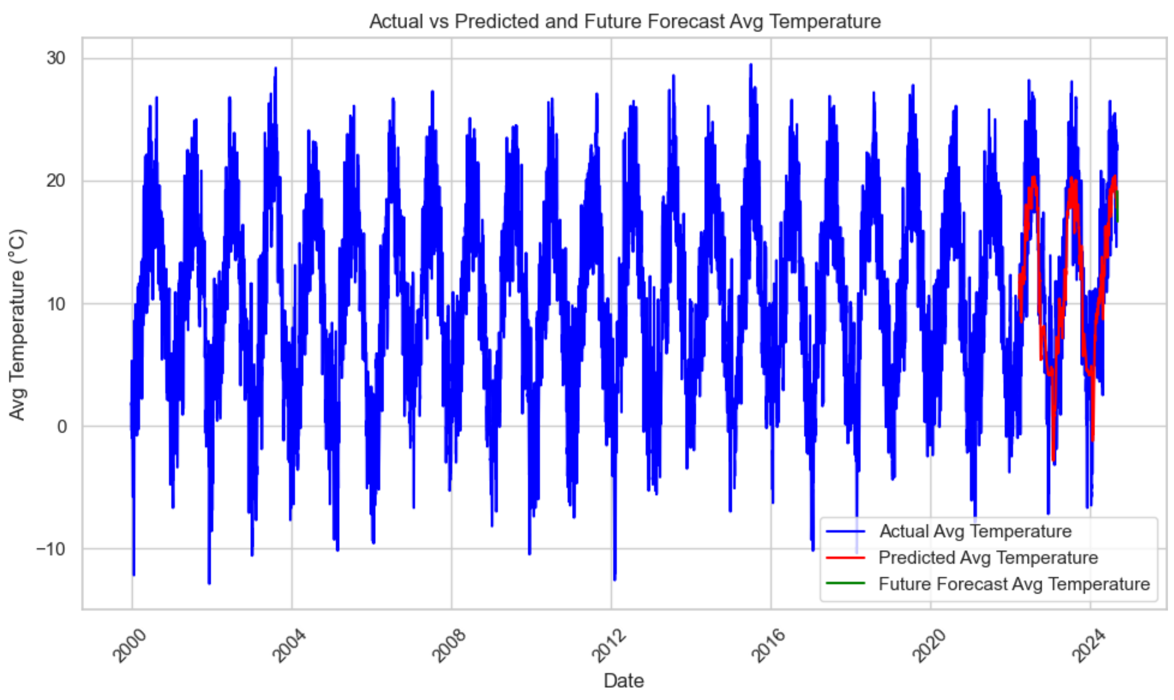


Figure 6.2: Testing and Forecast Result

Explanation of Forecasting Graphs

- **Actual Avg Temperature**: The blue line represents the actual average temperatures over the observed period.

- **Predicted Avg Temperature**: The red line indicates the predicted values for the test dataset. A close alignment with the actual values demonstrates the model's accuracy in prediction.

- **Future Forecast Avg Temperature**: The green line shows the average temperatures predicted for the next 5 days. It indicates how well the model can extrapolate from the training data to forecast future temperatures.

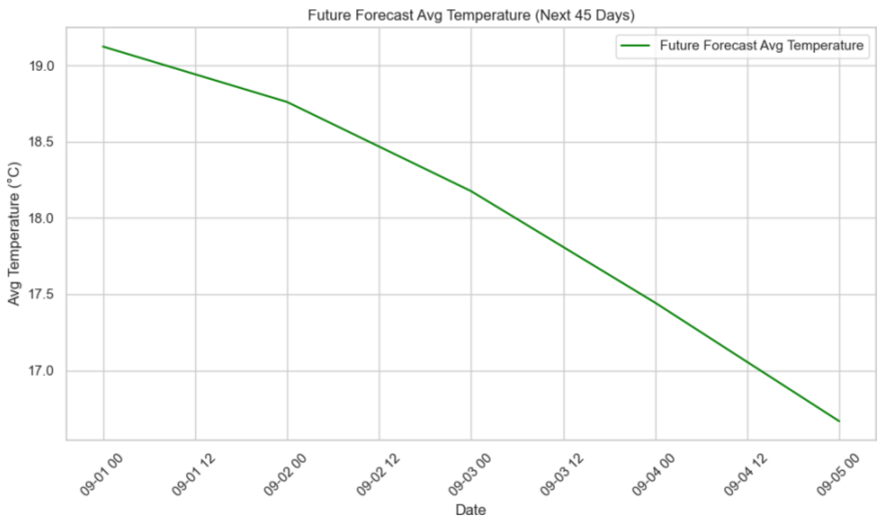


Figure 6.3: Forecasting of 15 days from 1st September

CHAPTER 7

Conclusions and future works

7.0.1 7.1 Conclusions

In this thesis, we developed a neural network-based model using Long Short-Term Memory (LSTM) architecture to forecast weather, specifically focusing on predicting average temperatures. The model was trained on historical weather data, demonstrating its ability to learn from time-series patterns effectively. Key findings

from our research include:

• **Model Performance**: The model achieved satisfactory performance metrics, including a Mean Absolute Error (MAE) of **3.07 °C**, a Root Mean Squared Error (RMSE) of **3.88 °C**, and an R^2 score of **0.73**. These results indicate that the model can accurately predict average temperatures, capturing essential trends and seasonal variations in the data.

• **Training and Validation Loss**: The loss graphs revealed that the model learned effectively during training, with decreasing training and validation losses. This suggests that the model is well-tuned and not significantly overfitting the training data.

• **Forecasting Capability**: The model successfully generated future temperature forecasts, demonstrating its applicability for real-world weather prediction scenarios. The forecasts showed reasonable alignment with historical data, indicating the potential for reliable long-term predictions.

Overall, the research confirms that LSTM networks are suitable for time-series forecasting tasks, especially in meteorological contexts where capturing temporal dependencies is crucial.

7.0.2 7.2 Future Works

While this study successfully developed a forecasting model, there are several avenues for future research and improvement:

1. **Incorporating Additional Features**: Future iterations of this model could include more input features, such as humidity, wind speed, or atmospheric pressure, to enhance the model's predictive power. This could lead to a more comprehensive understanding of the factors influencing temperature changes.

2. **Experimenting with Other Architectures**: Investigating alternative deep learning architectures, such as GRU (Gated Recurrent Units) or Transformer models, may yield improved performance. Each architecture has unique advantages that could be leveraged for time-series forecasting.

3. **Model Optimization**: Further hyperparameter tuning and experimentation with different configurations (e.g., different dropout rates, learning rates, or batch sizes) may help improve the model's accuracy and generalization capabilities.

4. **Ensemble Methods**: Combining multiple models through ensemble methods could potentially enhance forecasting accuracy. Techniques such as bagging or boosting could be explored to harness the strengths of various models.

5. **Real-Time Forecasting**: Implementing a real-time forecasting system using live weather data could be a valuable extension of this research. This would require

integrating the model into a pipeline that continuously updates predictions based on the latest available data.

6. **Deployment in Practical Applications**: Exploring practical applications of the forecasting model in sectors such as agriculture, energy management, and disaster preparedness would demonstrate the model's utility and potential impact.

In conclusion, while the current research has laid a solid foundation for weather forecasting using LSTM networks, the proposed future works can pave the way for more sophisticated models and applications. Continued advancements in machine learning and data analytics hold significant promise for enhancing the accuracy and reliability of weather predictions, ultimately benefiting society by enabling better decision-making in various sectors.

Bibliography

GÉRON, A. (2017), *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, O'Reilly Media.

IAN GOODFELLOW, A. C., YOSHUA BENGIO (2016), *Deep Learning*, MIT Press.

Web sites visited

- Digital Ocean – `https://www.digitalocean.com`
- Git Hub – `https://github.com`
- Stack Overflow – `http://stackoverflow.com`
- Wiki Pedia – `http://wikipedia.org`
- Medium – `http://medium.com`
- Reasearch Gate – `https://www.researchgate.net`
- Git Hub– `https://github.com/exchhattu/TimeseriesWeatherForecast-RNN-GRU-LSTM`