ZANSKAR

COMPREHENSIVE TIME SERIES EXPLORATION AND FUTURE FORECASTS

Prepared By :
Lakshya Singh

+91-8433005189

singhi7.lakshya@gmail.com

lakshyasingh07

122, Rama Enclave, Bulandshahr-203001, Uttar Pradesh

PREFACE

This report encapsulates a comprehensive analysis of the work undertaken to forecast future values of two critical parameters, `t1` and `t2`, using time series data. The objective of this project was to design, implement, and evaluate predictive models that are capable of forecasting these values over different time windows, with a focus on understanding underlying patterns and trends within the dataset. The report is organized into several key sections, each addressing a critical component of the analysis and modeling pipeline.

1. **Preprocessing Steps**: A detailed description of the preprocessing pipeline is provided, including key steps such as data scaling using `MinMaxScaler`, applying Exponential Moving Average (EMA) for smoothing, and the generation of lagged features to capture temporal dependencies. These steps were essential to prepare the data for machine learning models, ensuring the input features were suitable for effective model training. Additionally, techniques like time series normalization were implemented to facilitate better convergence of the predictive models.
2. **Exploratory Data Analysis (EDA)**: The report documents the extensive exploratory data analysis (EDA) performed on the dataset. Through visualizations and statistical analysis, we identified key trends, seasonalities, and anomalies within the time series. This analysis provided critical insights into the underlying structure of the data, such as periodic fluctuations and potential external factors influencing the observed variables. By understanding these data patterns, we were able to inform the subsequent modeling decisions and ensure the models could generalize well to unseen data.
3. **Modeling Techniques and Results**: The predictive models developed in this report employ advanced deep learning techniques, primarily using Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures. These models were trained to forecast future values of `t1` and `t2` over various time horizons, including short-term (N=1), medium-term (N=10), and long-term forecasts (N=60, N=300). The models were evaluated on multiple metrics, including Mean Squared Error (MSE) and Mean Absolute Error (MAE), to assess their performance and robustness. The results reveal the effectiveness of the models in capturing temporal dependencies and generating accurate forecasts over different time windows.
4. **Streamlit Application for Visualization**: A key component of this work is the development of an interactive Streamlit application designed to visualize the forecasting results in real-time. This application allows users to input the most recent observed data and view the corresponding forecasts generated by the model over various time windows. The visualizations, built using Plotly, offer dynamic, interactive plots that allow users to explore the forecasting behavior and assess the model's performance interactively.
5. **Insights into Data Patterns**: Throughout the analysis, we uncovered several important insights regarding the time series data. Patterns such as trends, seasonality, and irregular fluctuations were identified and analyzed. These insights not only helped in fine-tuning the predictive models but also provided valuable information on how `t1` and `t2` interact and how they are influenced by different factors over time. The identification of these patterns is crucial for enhancing our understanding of the system and for refining future forecasting strategies.

Contents

1 Exploratory Data Analysis Findings

Exploratory Data Analysis (EDA) serves as the cornerstone of any data investigation. For time series, EDA not only includes traditional descriptive statistics but also emphasizes temporal patterns that can reveal significant insights. This section will provide a detailed analysis of the time series data for t1 and t2, aiming to uncover any trends, seasonal patterns, correlations, and anomalies.

1.1 Preprocessing steps

STEP 1: Folder Iteration and File Reading:

```
for file_name in os.listdir(folder_path):
    if file_name.endswith('.csv'):
        file_path = os.path.join(folder_path, file_name)
        df = pd.read_csv(file_path)
```

The function iterates through all CSV files within a specified folder path and reads each file into a Pandas DataFrame. This step ensures that only files with a .csv extension are processed.

STEP 2: Column Check for Required Data:

```
required_columns = ['Time', 't1', 't2']
if not all(col in df.columns for col in required_columns):
    raise ValueError(f"File {file_name} is missing one or more required columns:
    ↪ {required_columns}")
```

Each CSV file is checked to confirm it contains the necessary columns: Time, t1, and t2. This step is crucial to ensure the data has the required structure for consistent processing. If any required column is missing, an error is raised, prompting a review of the dataset.

STEP 3: Index Setting and Sorting by Time

```
df.set_index('Time', inplace=True)
df.sort_index(inplace=True)
```

The Time column is set as the DataFrame index to facilitate time series analysis. Sorting the index ensures the time data is in chronological order, which is essential for sequential analysis and prevents any time-based misalignment.

STEP 4: Appending to the List and Data Concatenation

```
all_data.append(df)
consolidated_df = pd.concat(all_data)
```

Each processed DataFrame is appended to a list. After iterating through all files, the individual DataFrames are concatenated into a single DataFrame, consolidated_df. This results in one unified DataFrame containing all time series data across the files.

STEP 5: Loading Data with Run ID for Source Tracking

```
for i, file in enumerate(glob(file_pattern)):
df['Run_ID'] = i  # Assign a unique ID for each file
dataframes.append(df)
```

Each DataFrame is assigned a unique Run_ID to keep track of the file source. This is helpful for identifying the origin of data, especially in scenarios where data characteristics need to be traced back to specific files.

STEP 6: Concatenating with ignore_index

```
consolidated_df = pd.concat(dataframes, axis=0, ignore_index=True)
df = consolidated_df.copy(deep = True)
```

The ignore_index = True parameter resets the index after concatenation. This is useful when merging data from multiple sources to create a seamless DataFrame without duplicating index labels.
This preprocessing strategy results in a consolidated DataFrame that is ready for exploratory data analysis, enabling consistent time-based analysis across all loaded files.

1.2 Time Series Plotting and Descriptive Statistics

1.2.1 Summary Statistics for t1 and t2

```
summary_stats = df[['t1', 't2']].describe(percentiles=[0.25, 0.5, 0.75])
print("Summary Statistics for t1 and t2:")
print(summary_stats)
```

The describe() function provides key summary statistics for t1 and t2, including mean, standard deviation, minimum, and maximum. This gives a snapshot of the distribution and variability of each series. These statistics help assess the central tendency, spread, and general behavior of the data, allowing you to spot potential issues such as high variability or skewness in the data.

```
Summary Statistics for t1 and t2:
                t1             t2
count  981414.000000  981414.000000
mean       -0.000125       0.002857
std         0.576996       0.301435
min       -19.600000      -1.000094
max         9.861404       1.000003
```

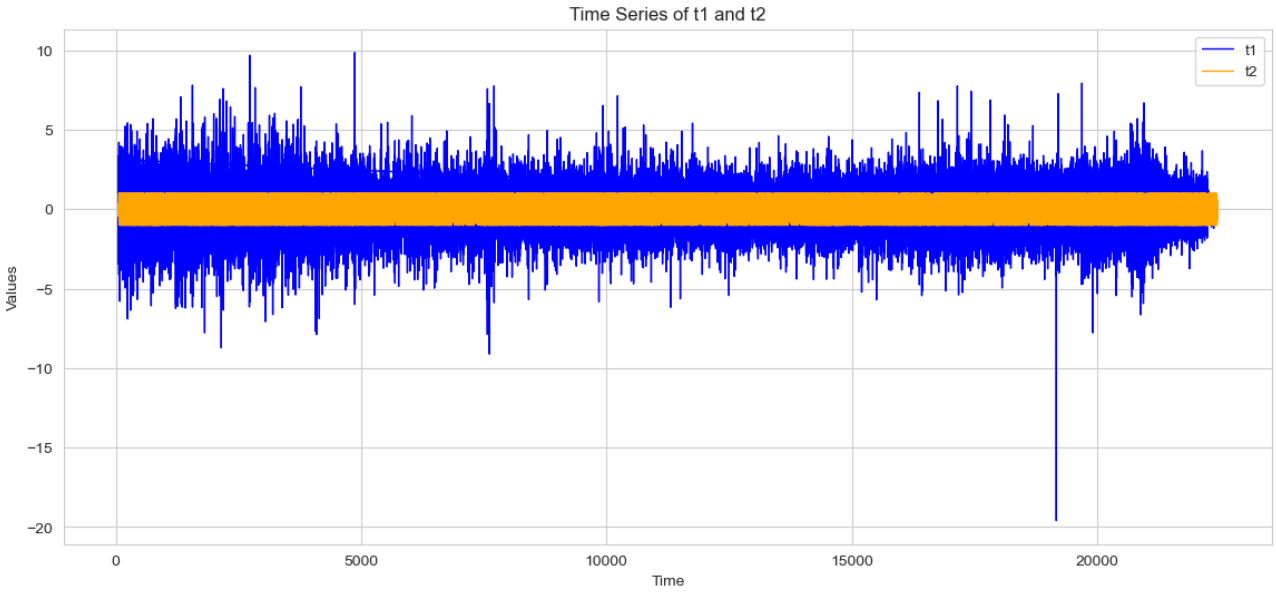

Figure 1.1: Time Series Plot of t1 and t2

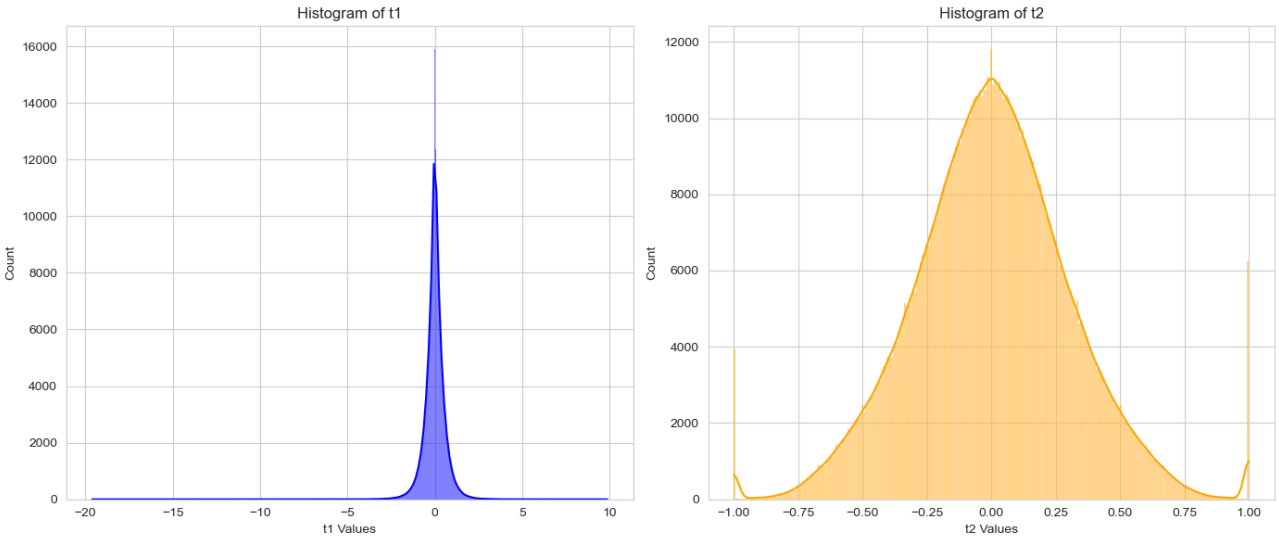

Figure 1.2: Histograms of t1 and t2

1.2.2 Time Series Plot of t1 and t2

```
plt.figure(figsize=(14, 6))
plt.plot(df['Time'], df['t1'], label='t1', color='blue', linewidth=1)
plt.plot(df['Time'], df['t2'], label='t2', color='orange', linewidth=1)
plt.title('Time Series of t1 and t2')
plt.xlabel('Time')
plt.ylabel('Values')
plt.legend()
plt.show()
```

This plot (Figure 1.1) visualizes how t1 and t2 change over time, allowing for direct comparison of both series on the same timeline.

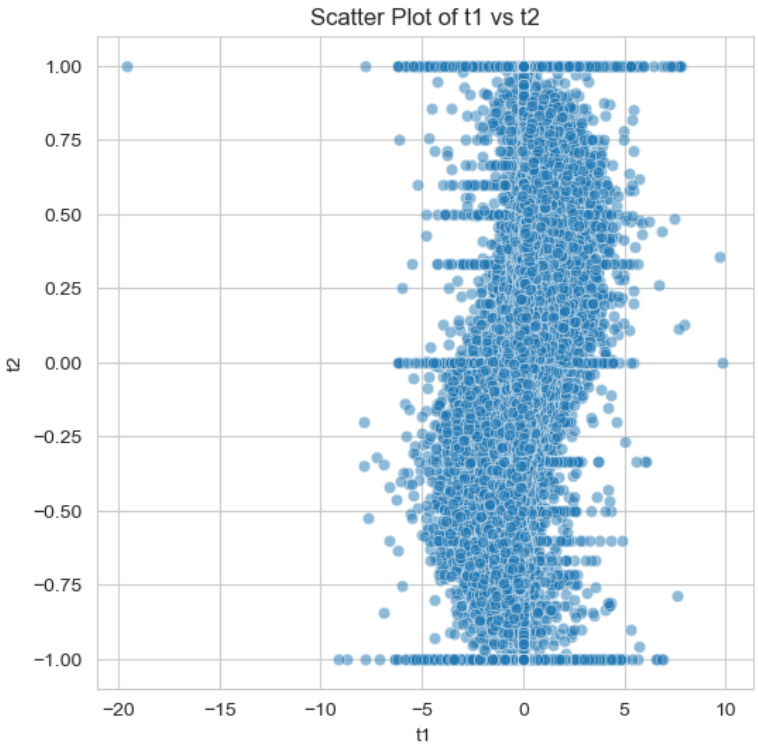

Figure 1.3: Scatter Plot of t1 vs t2

1.2.3 Histograms of t1 and t2

Histograms display the frequency distribution of t1 and t2, with a Kernel Density Estimate (KDE) line to show the probability density.
Clearly these plots (Figure 1.2) show that these two time series approximately follow a normal distribution.

1.2.4 Scatter Plot of t1 vs t2

The scatter plot (Figure 1.3) suggests a non-linear relationship.

1.3 Rolling Mean and Standard Deviation Plots

- **Rolling Mean:** Helps identify the underlying trend in the time series by smoothing out short-term fluctuations. It shows how the average value of the series changes over time, making it easier to see trends or long-term movements.
- **Rolling Standard Deviation:** Measures the variability of the data within a specified window, showing how much the data deviates from the mean over time. High standard deviation indicates more variability, while low standard deviation suggests more consistency.

The choice of the window size (window_size = 600, representing a 10-minute interval in this case) affects the level of smoothing. Larger windows will provide smoother trends but may obscure short-term fluctuations, while smaller windows will show more detail but can be noisy.

```
# Setting a window size (e.g., 3600 for 1-hour moving average, 600 for 10 min)
window_size = 600
```

```
# Rolling mean and standard deviation for t1 and t2
df['t1_rolling_mean'] = df['t1'].rolling(window=window_size).mean()
df['t1_rolling_std'] = df['t1'].rolling(window=window_size).std()

df['t2_rolling_mean'] = df['t2'].rolling(window=window_size).mean()
df['t2_rolling_std'] = df['t2'].rolling(window=window_size).std()

plt.figure(figsize=(8, 8))

# Plot for t1
plt.subplot(2, 1, 1)
plt.plot(df['Time'], df['t1'], label='t1', color='blue', alpha=0.5)
plt.plot(df['Time'], df['t1_rolling_mean'], label='t1 Rolling Mean', color='orange')
plt.plot(df['Time'], df['t1_rolling_std'], label='t1 Rolling Std Dev', color='green',
↪  linestyle='--')
plt.title('t1: Raw Data and Rolling Statistics')
plt.legend()

# Plot for t2
plt.subplot(2, 1, 2)
plt.plot(df['Time'], df['t2'], label='t2', color='red', alpha=0.5)
plt.plot(df['Time'], df['t2_rolling_mean'], label='t2 Rolling Mean', color='orange')
plt.plot(df['Time'], df['t2_rolling_std'], label='t2 Rolling Std Dev', color='green',
↪  linestyle='--')
plt.title('t2: Raw Data and Rolling Statistics')
plt.legend()

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

Looking at the plots (Figure 1.4), A rolling mean that remains flat suggests a stationary series without significant trends.

High variability (spikes in rolling standard deviation) might indicate periods of volatility or significant changes in the time series.

Steady periods with low variability might indicate stability or less fluctuation in the series.

There are no synchronized spikes or dips, which could suggest that both series are influenced by similar factors at specific times.

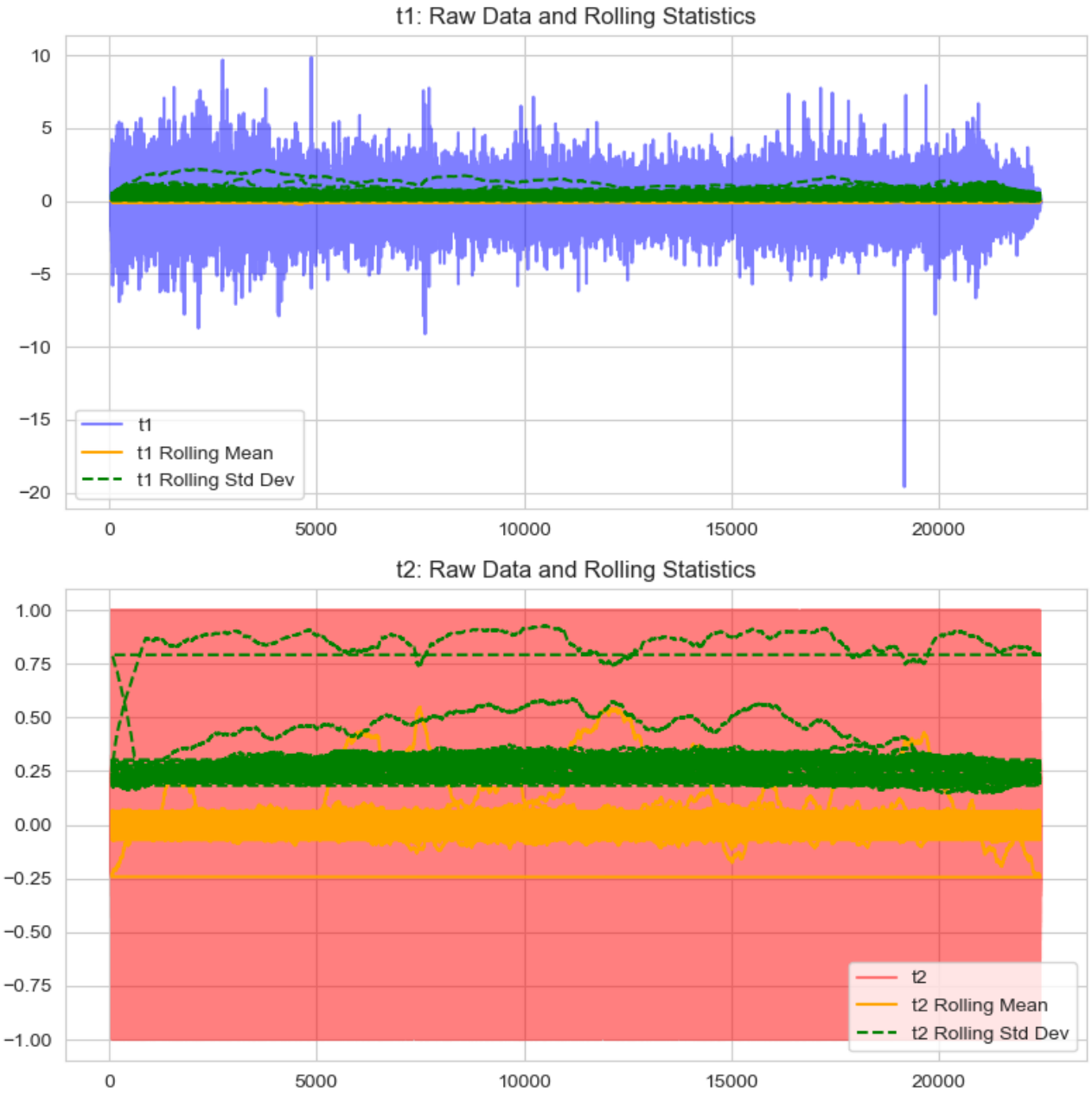

Figure 1.4: Rolling Mean and Standard Deviation Plots

2 Predictive models

This section delves into predictive modeling for multivariate time series, providing code explanations and discussing the significance, advantages, and disadvantages of the models, as well as the relevance of evaluation metrics used.

2.1 Simple RNNs

Simple RNNs are a baseline approach for multivariate time series modeling. They work well for shorter sequences but are limited when handling long-term dependencies. They struggle with learning long-term dependencies due to the vanishing gradient problem, making them less effective for data with long sequences compared to more advanced RNN architectures like LSTM and GRU.

```
import pandas as pd
import os
from google.colab import drive, files
import numpy as np
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential, load_model
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from tqdm.keras import TqdmCallback
from tqdm import tqdm
from keras.models import Sequential
from keras.layers import SimpleRNN, Dropout, Dense

# Checking if a GPU is available and set the device accordingly
device = tf.device('/device:GPU:0' if tf.test.is_gpu_available() else '/device:CPU:0')
device
```

1. **Preprocessing Function (`preprocess_data`)**:
    - Reads a CSV file into a DataFrame.
    - Scales the data using `MinMaxScaler` to keep the feature values between 0 and 1.

- Applies an Exponential Moving Average (EMA) to smooth the data.
- Creates lagged features to capture historical dependencies up to a specified number of lags (5 in this case).
- Reshapes the input `X` to be compatible with the RNN structure (`[samples, timesteps, features]`).
- Splits the data into training and testing sets without shuffling (to maintain the time sequence).

2. **Exponential Moving Average (`apply_ema`)**:
   - Computes an EMA for specified columns (`t1` and `t2`) to highlight trends over a specified window (span).
3. **Lagged Feature Creation (`create_lagged_features`)**:
   - Generates lagged versions of columns `t1` and `t2` up to 5 lags, aligning them for model input to help capture sequential information.
4. **Model Architecture (`build_model`)**:
   - Constructs a neural network with:
     - **SimpleRNN layers** to process the sequential data.
     - **Dropout layers** for regularization to reduce the risk of overfitting.
     - **Dense layer** with 2 units for predicting two targets (`t1` and `t2`).
   - Uses `Adam` as the optimizer for efficient gradient descent, and `mean_squared_error` as the loss function, with `mean absolute error (mae)` as a performance metric.

```
# Function to load and preprocess the data
def preprocess_data(file_path):
    df = pd.read_csv(file_path)
    # Scaling and EMA
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(df.values)
    df_ema = apply_ema(pd.DataFrame(scaled_data, columns=df.columns))

    # Create lagged features
    lagged_df = create_lagged_features(df_ema, n_lags=5)

    X = lagged_df.dropna().drop(columns=['t1', 't2'])
    y = lagged_df[['t1', 't2']]

    # Reshaping input for RNN
    X = np.array(X)
    y = np.array(y)
    X = X.reshape(X.shape[0], 1, X.shape[1])

    # Splitting the data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    ↪  shuffle=False)
```

```
    return X_train, X_test, y_train, y_test

# Function to create EMA
def apply_ema(data, span=10):
    data['t1_ema'] = data['t1'].ewm(span=span, adjust=False).mean()
    data['t2_ema'] = data['t2'].ewm(span=span, adjust=False).mean()
    return data

# Function to create lagged features
def create_lagged_features(data, n_lags=5):
    lagged_data = data.copy()
    for lag in range(1, n_lags + 1):
        lagged_data = pd.concat([lagged_data, data[['t1',
        ↪  't2']].shift(lag).add_suffix(f'_lag{lag}')], axis=1)
    lagged_data = lagged_data.dropna()
    return lagged_data

# Initializing model building with Simple RNN
def build_model(X_train):
    model = Sequential([
        SimpleRNN(128, input_shape=(X_train.shape[1], X_train.shape[2]),
        ↪  return_sequences=True),
        Dropout(0.5),
        SimpleRNN(64, return_sequences=False),
        Dropout(0.2),
        Dense(2)
    ])
    model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mae'])
    return model
```

2.1.1 Initial Training

1. **File Directory and File List Creation**:
   - The `file_directory` variable points to the directory containing the CSV files used for training.
   - The `files` list is generated using `os.listdir()`, filtering only CSV files in the specified directory.
2. **Training on the Initial Data File**:
   - The first CSV file from the `files` list is chosen for the initial training.
   - The `preprocess_data()` function processes this file to prepare `X_train`, `X_test`, `y_train`, and `y_test` sets for training and validation.
3. **Building and Training the Initial Model**:
   - The `build_model()` function constructs the initial Simple RNN model with layers described earlier.
   - The model is trained using the `fit()` method with the following parameters:
     - `epochs=50`: The number of complete passes through the training data.

- `batch_size=32`: The number of samples per gradient update.
- `validation_data=(X_test, y_test)`: A portion of data set aside for validating the model during training.
- `verbose=0`: Suppresses the default progress output.
- `callbacks=[TqdmCallback(verbose=1)]`: Adds a progress bar with `tqdm` to show the training progress interactively.

4. **Saving the Model**:
   - The trained model is saved to a file (`pretrained_model.h5`) for future use. This allows loading the model later for prediction or further training.

Significance of Initial Training: This step prepares a baseline model trained on the initial data. The model can then be fine-tuned or used for further analysis with other datasets.

```
file_directory = '/content/drive/My Drive/Colab Notebooks/intern/'
files = [f for f in os.listdir(file_directory) if f.endswith('.csv')]

# Initializing lists to store predictions and results
predictions = []
true_values = []

# Step 1: Training initial model on the first data file
initial_file_path = os.path.join(file_directory, files[0])
X_train, X_test, y_train, y_test = preprocess_data(initial_file_path)

initial_model = build_model(X_train)
initial_model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_test,
↪  y_test), verbose=0,  callbacks=[TqdmCallback(verbose=1)] ) # Adding tqdm callback)

# Saving the trained model
initial_model.save('pretrained_model.h5')
```

2.1.2 Fine-Tuning Model

1. **Loading the Pre-trained Model**

```
 pretrained_model_path = 'pretrained_model.h5'
fine_tuned_model = load_model(pretrained_model_path)
fine_tuned_model.compile(optimizer='adam', loss='mean_squared_error',
↪  metrics=['mae'])
```

- This code loads a previously trained model saved at `pretrained_model.h5`.
- After loading, the model is compiled with the Adam optimizer, `mean_squared_error` loss function, and `mean_absolute_error` (MAE) as an evaluation metric. This setup is common for regression tasks like time series forecasting, where both MSE and MAE provide insight into the prediction error.

2. **Fine-Tuning the Model in Batches**

```
batch_size = 20
predictions = []
true_values = []

for i in range(0, len(files), batch_size):
    batch_files = files[i:i + batch_size]
    print(f"Fine-tuning on batch {i // batch_size + 1}")
```

- The model is fine-tuned in batches of 20 files. Batch processing helps avoid memory overload and ensures the model is exposed to various data patterns.
- `predictions` and `true_values` lists are initialized to store results for later evaluation.
- For each batch of files, `batch_files` extracts the current set of files to be processed, and `i // batch_size + 1` displays the batch number in the console.

3. **Processing Each File in a Batch**

```
for file in tqdm(batch_files, desc=f"Processing batch {i // batch_size + 1}"):
    file_path = os.path.join(file_directory, file)
    X_train, X_test, y_train, y_test = preprocess_data(file_path)
```

- Each file in the batch undergoes preprocessing (e.g., feature scaling, train-test splitting) to ensure the data is in the right format for the model.

4. **Fine-Tuning the Model on Each File**

```
fine_tuned_model.fit(X_train, y_train, epochs=10, batch_size=32,
↪  validation_data=(X_test, y_test), verbose=0)
```

- The model is trained on each file's training set (`X_train`, `y_train`) for 10 epochs with a batch size of 32, while validation data (`X_test`, `y_test`) monitors performance.
- Setting `verbose=0` suppresses output, making the process cleaner.

5. **Making Predictions and Collecting Results**

```
y_pred = fine_tuned_model.predict(X_test)
predictions.append(y_pred)
true_values.append(y_test)
```

- After training on each file, predictions are made on the test set (`X_test`), and results are stored for later evaluation.
- Collecting predictions and true values across batches provides data for calculating error metrics or generating visual comparisons.
- Optional Plotting of Training History can be done.

Fine-tuning allows the model to adapt to new patterns by leveraging knowledge from previous training (the pre-trained model). Batch-wise fine-tuning ensures gradual and memory-efficient adaptation to potentially diverse datasets.

2.1.3 Model Evaluation

In this code, we are valuating the fine-tuned model using two main metrics: **Mean Absolute Error (MAE)** and **R-squared (R^2)**.

1. **Mean Absolute Error (MAE):**

```
test_loss, test_mae = fine_tuned_model.evaluate(X_test, y_test, verbose=2)
print(f"Test Loss: {test_loss}, Test MAE: {test_mae}")
```

- **Definition**: MAE calculates the average absolute difference between predicted and actual values. It is expressed as:

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

  where y_i is the actual value and $\hat{y}_i$ is the predicted value.

- **Interpretation**: A lower MAE indicates that the model's predictions are close to the true values, which is desirable in time series forecasting where accurate predictions are critical. In this example, the `test_mae` of 0.0381 suggests that, on average, the model's predictions deviate by around 0.0381 from the true values in the test set.

2. **R-squared (R^2):**

```
from sklearn.metrics import r2_score

# Calculating R-squared
y_pred = fine_tuned_model.predict(X_test)
r2_t1 = r2_score(y_test[:, 0], y_pred[:, 0])  # R² for t1
r2_t2 = r2_score(y_test[:, 1], y_pred[:, 1])  # R² for t2

print(f"R-squared for t1: {r2_t1}")
print(f"R-squared for t2: {r2_t2}")
```

- **Definition**: R-squared, or the coefficient of determination, measures the proportion of variance in the dependent variable that is predictable from the independent variable(s). It's calculated as:

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

  where y_i is the true value, $\hat{y}_i$ is the predicted value, and $\bar{y}$ is the mean of true values.

- **Interpretation**: R-squared values range from 0 to 1, where values closer to 1 indicate that the model explains a high proportion of the variance in the data. Here:
  - `r2_t1 = 0.7576` implies that 75.76% of the variance for target variable `t1` is captured by the model.
  - `r2_t2 = 0.7524` shows that 75.24% of the variance for `t2` is explained by the model.

2.1.4 Plotting the Results

This section visualizes the model's predictions against the true values, which is helpful for understanding how well the model captures the trends and patterns in the time series data.

1. **Concatenating Predictions and True Values**

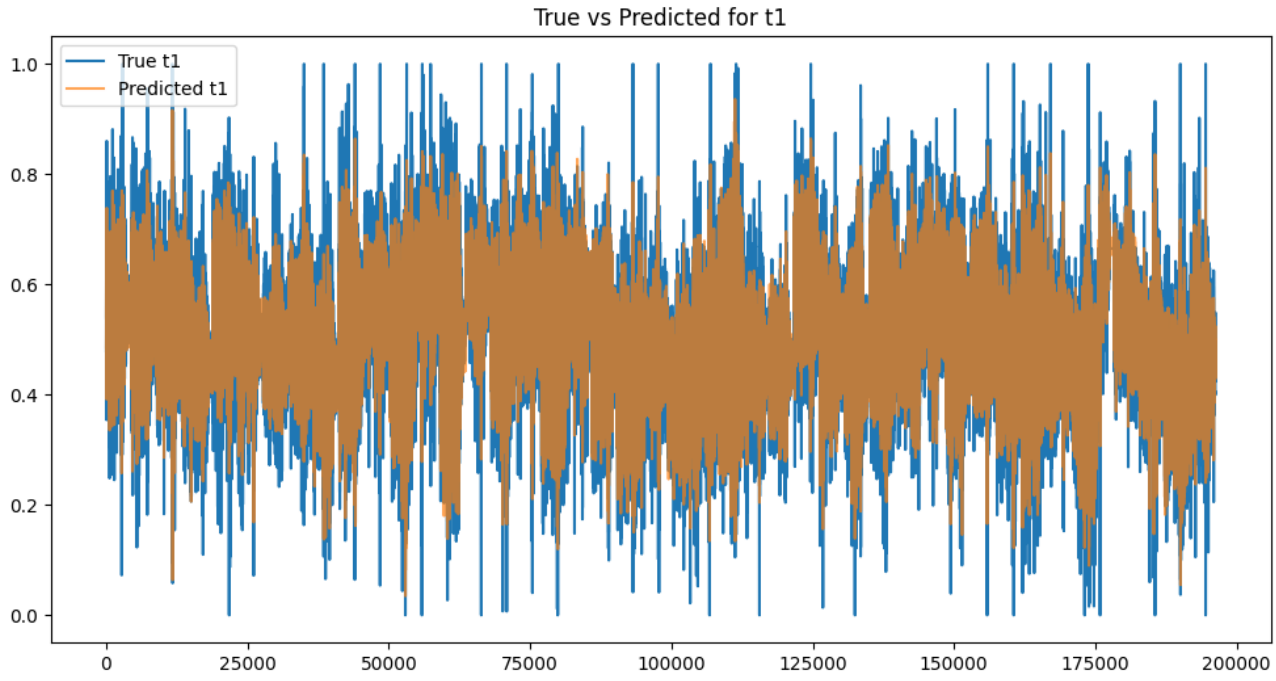

Figure 2.1: Plotting for t1

```
predictions = np.concatenate(predictions, axis=0)
true_values = np.concatenate(true_values, axis=0)
```

- The lists `predictions` and `true_values`, which were populated in batches during fine-tuning, are concatenated along the 0-axis. This step aggregates the results from all batches, creating a complete set of predictions and corresponding true values for further analysis and plotting.

2. **Plotting for t1 (First Target Variable)**

```
plt.figure(figsize=(12, 6))
plt.plot(true_values[:, 0], label='True t1')
plt.plot(predictions[:, 0], label='Predicted t1', alpha=0.7)
plt.title('True vs Predicted for t1')
plt.legend()
plt.show()
```

Patterns, peaks, and troughs in the time series can indicate if the model captures seasonality, trends, or sudden changes.

3. **Plotting for t2 (Second Target Variable)**

```
plt.figure(figsize=(12, 6))
plt.plot(true_values[:, 1], label='True t2')
plt.plot(predictions[:, 1], label='Predicted t2', alpha=0.7)
plt.title('True vs Predicted for t2')
plt.legend()
plt.show()
```

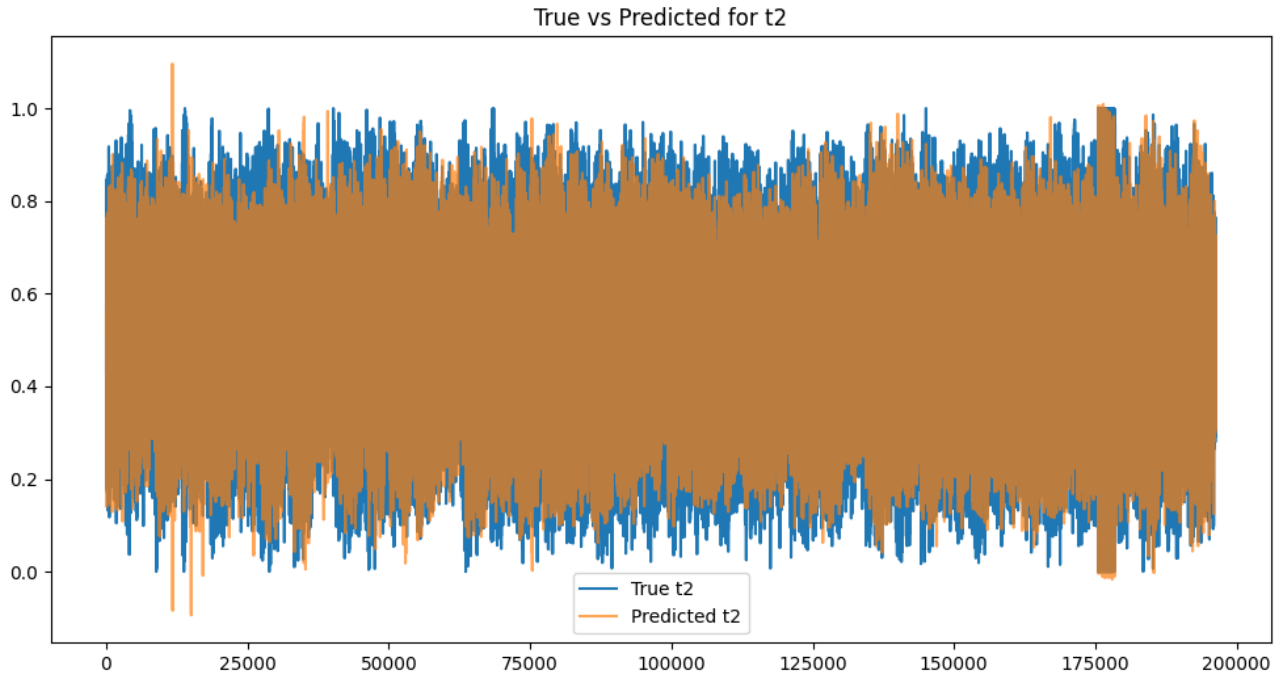

Figure 2.2: Plotting for t2

- This plot follows the same structure as for `t1`, but here we are visualizing `t2` (second target variable).

These plots are a visual diagnostic tool, offering a quick and intuitive sense of model accuracy and reliability in capturing the dynamics of `t1` and `t2`. Together with numerical metrics, they complete the picture of model performance.

2.1.5 Forecasting

To forecast the results for a specific window size after choosing a particular data file we following the same pre-processing steps:

```
# Function to load and preprocess the data
def preprocess_data(file_path):
    df = pd.read_csv(file_path)

    df_ema = apply_ema(df)

    # Create lagged features
    lagged_df = create_lagged_features(df_ema, n_lags=5)

    X = lagged_df.dropna().drop(columns=['t1', 't2'])  # Dropping target columns
    y = lagged_df[['t1', 't2']]  # Target columns are the original columns

    X = np.array(X)
    y = np.array(y)
    X = X.reshape(X.shape[0], 1, X.shape[1])
```

```
    return X, y



# Function to create EMA
def apply_ema(data, span=10):
    data['t1_ema'] = data['t1'].ewm(span=span, adjust=False).mean()
    data['t2_ema'] = data['t2'].ewm(span=span, adjust=False).mean()
    return data

# Function to create lagged features
def create_lagged_features(data, n_lags=5):
    lagged_data = data.copy()
    for lag in range(1, n_lags + 1):
        lagged_data = pd.concat([lagged_data, data[['t1',
        ↪  't2']].shift(lag).add_suffix(f'_lag{lag}')], axis=1)
    lagged_data = lagged_data.dropna()  # Drop rows with missing values after lagging
    return lagged_data



# Load the data and preprocess
file_path = '/content/intern_v1_0703.csv'  # Replace with your actual file path
X, y = preprocess_data(file_path)
```

Here, the forecast function is used to generate forecasts for time series data over different window sizes, allowing flexibility in prediction lengths:

1. **Function Parameters**

```
def forecast(model, X, scaler, windows=[60], batch_size=32):
```

   - `model`: The fine-tuned model used for making predictions.
   - `X`: The input data for forecasting, which has already been preprocessed into sequences with lagged features.
   - `windows`: A list of different window sizes, where each value represents the number of time steps (e.g., days, hours) to predict ahead.
   - `batch_size`: Defines the number of samples processed simultaneously for memory efficiency.

2. **Initializing Forecast Storage**

```
forecasts = {}
```

   - An empty dictionary `forecasts` is created to store forecasted values for each window size. This enables predictions over multiple horizons in a single run.

3. **Looping Through Each Window Size**

```
for N in windows:
    forecasted_values = []
```

- The loop iterates over each window size in `windows` (e.g., `60` steps ahead).
- `forecasted_values` is initialized as an empty list to store predictions for each batch in the current window.

4. **Batch-wise Forecasting**

```
for i in tqdm(range(0, len(X), batch_size), desc=f"Predicting for window size {N}"):
    batch_input = X[i:i+batch_size, :, :]
    batch_predictions = model.predict(batch_input)
    forecasted_values.append(batch_predictions)
```

- To manage large datasets, forecasting is done in batches. `batch_input` selects a batch of data for predictions, and `model.predict(batch_input)` generates the forecast.
- `tqdm` provides a progress bar, which is especially useful for monitoring long-running forecasts.

5. **Returning the Forecasts**

```
return forecasts
```

- The function returns the `forecasts` dictionary, where each key is a window size (e.g., `60`), and each value is an array of forecasted values.

```
forecasted_values = forecast(fine_tuned_model, X, scaler)
```

The output, `forecasted_values`, contains the predicted values over the desired time horizon in the original data scale, which is essential for practical application and interpretation.

2.1.6 Interactive Plot for Forecasts

```
# Plotting with Plotly (Interactive Plot)
def plot_forecasts(forecasts, true_values):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x=np.arange(len(true_values)), y=true_values[:, 0],
    ↪  mode='lines', name='Actual t1'))
    fig.add_trace(go.Scatter(x=np.arange(len(true_values)), y=true_values[:, 1],
    ↪  mode='lines', name='Actual t2'))

    for window, forecasted in forecasts.items():
        forecasted_x = np.arange(len(true_values), len(true_values) + len(forecasted))

        fig.add_trace(go.Scatter(x=forecasted_x, y=forecasted[:, 0], mode='lines',
        ↪  name=f'Forecasted t1 (N={window})'))
```

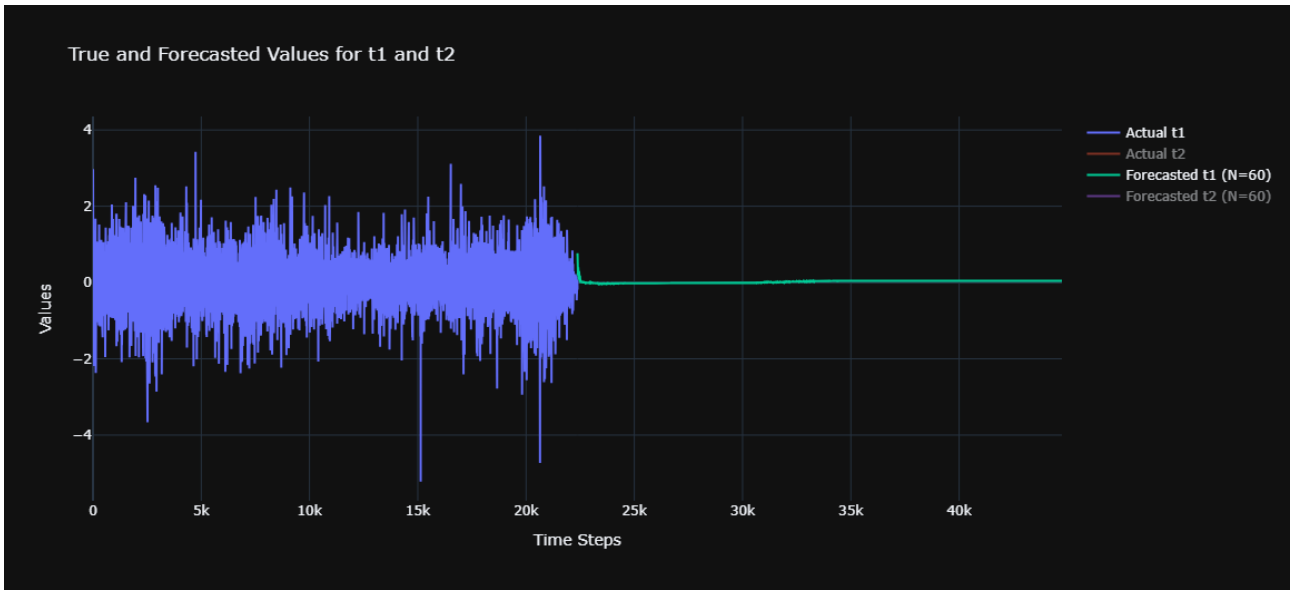

Figure 2.3: Interactive Plot for Forecasts using Simple RNNs

```
        fig.add_trace(go.Scatter(x=forecasted_x, y=forecasted[:, 1], mode='lines',
        ↪  name=f'Forecasted t2 (N={window})'))

    fig.update_layout(title='True and Forecasted Values for t1 and t2',
                      xaxis_title='Time Steps',
                      yaxis_title='Values',
                      template='plotly_dark')
    fig.show()

plot_forecasts(forecasted_values, y)
```

2.2 Gated Recurrent Unit (GRU)

For the implementation of GRU, we will follow the same pre-processing steps and training approach but with the following change:

```
# Initializing model building with GRU
def build_model(X_train):
    model = Sequential([
        GRU(128, input_shape=(X_train.shape[1], X_train.shape[2]),
        ↪  return_sequences=True),
        Dropout(0.5),
        GRU(64, return_sequences=True),
        Dropout(0.3),
        GRU(32, return_sequences=False),
        Dropout(0.2),
        Dense(2)
```

```
    ])
    model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mae'])
    return model
```

The **Gated Recurrent Unit (GRU)** model enhances performance over a traditional RNN by addressing key limitations of standard RNNs, particularly with handling long-term dependencies and vanishing gradients. Here's how GRUs improve on RNNs:

1. **Gates for Controlling Information Flow**
   - GRUs use **update** and **reset gates** to control what information passes through the network and what is forgotten, which is crucial for capturing long-term dependencies in time series data.
   - The update gate helps GRUs retain information over longer sequences, mitigating the **vanishing gradient problem** that often affects standard RNNs. This makes GRUs better suited for capturing patterns over extended time steps.
2. **Simplified Structure Compared to LSTMs**
   - While GRUs are similar to Long Short-Term Memory (LSTM) units, they have a simpler architecture with fewer parameters, making them faster to train than LSTMs while still effectively handling long sequences.
   - The simpler gating mechanism in GRUs allows for a balance between **computational efficiency** and **memory capability**.
3. **Performance on Sequential Data**
   - The gated structure enables GRUs to perform better than RNNs on complex sequential data, as they learn to retain essential information across longer sequences without the risk of "forgetting" earlier information as easily as standard RNNs.

In the code provided, replacing the standard RNN layers with stacked GRU layers enables better learning of temporal dependencies in the data, particularly for sequential forecasting tasks. The dropout layers added between GRU layers help prevent overfitting, and the final dense layer outputs predictions for `t1` and `t2`.

2.2.1 Model Evaluation

The evaluation results of the GRU model indicate an improvement over the RNN, reflected by lower error metrics and higher R-squared values, which show better model accuracy and fit. Let's break down these metrics:

1. **Test Loss (MSE) and MAE**
   - **Loss (Mean Squared Error)**: `0.0027`
   - **Mean Absolute Error (MAE)**: `0.0381`
     Both the loss and MAE values are slightly lower than those from the RNN, suggesting that the GRU model better minimizes errors during predictions.
2. **R-squared for `t1` and `t2`**
   - **R-squared for `t1`**: `0.7587` and for `t2`: `0.7592` - These values indicate that the GRU model explains approximately 75.8% of the variance in `t1` and 75.9% in `t2`. Higher R-squared values show a stronger fit, with the GRU capturing more variability in the data than the RNN.

Overall, the GRU's improved results highlight its ability to capture long-term dependencies more effectively than RNNs, reducing errors and fitting the data with greater accuracy.

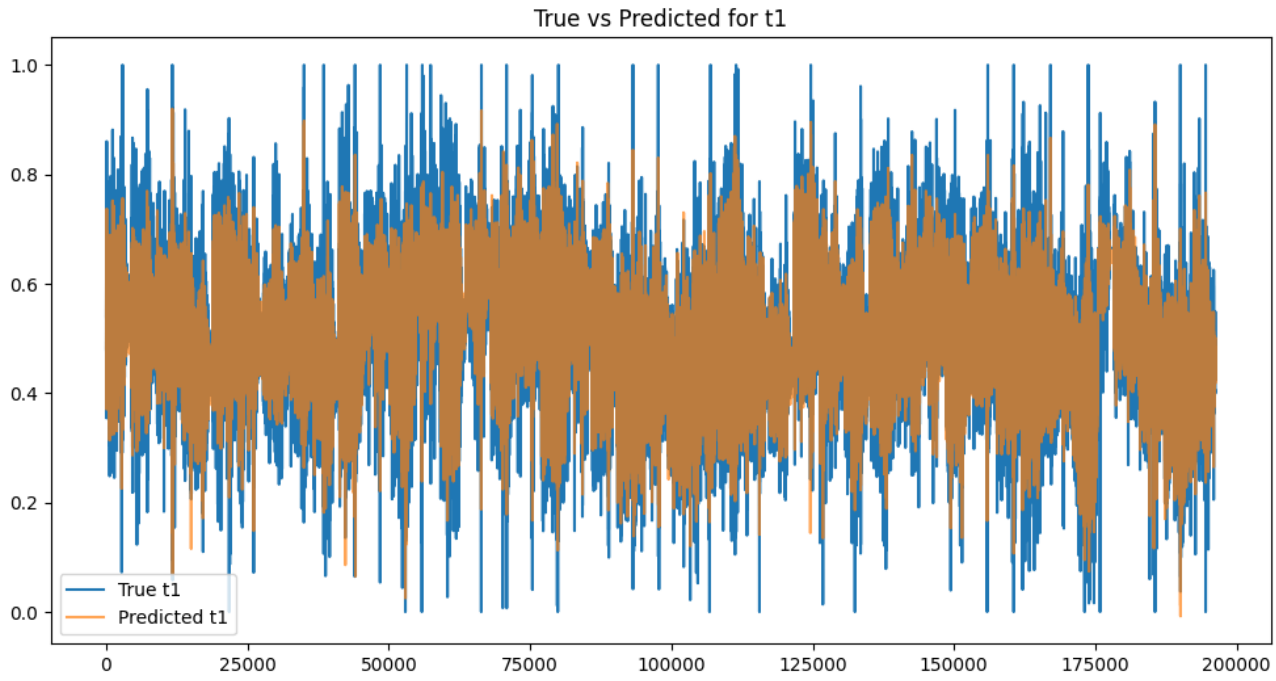

Figure 2.4: Plot for t1

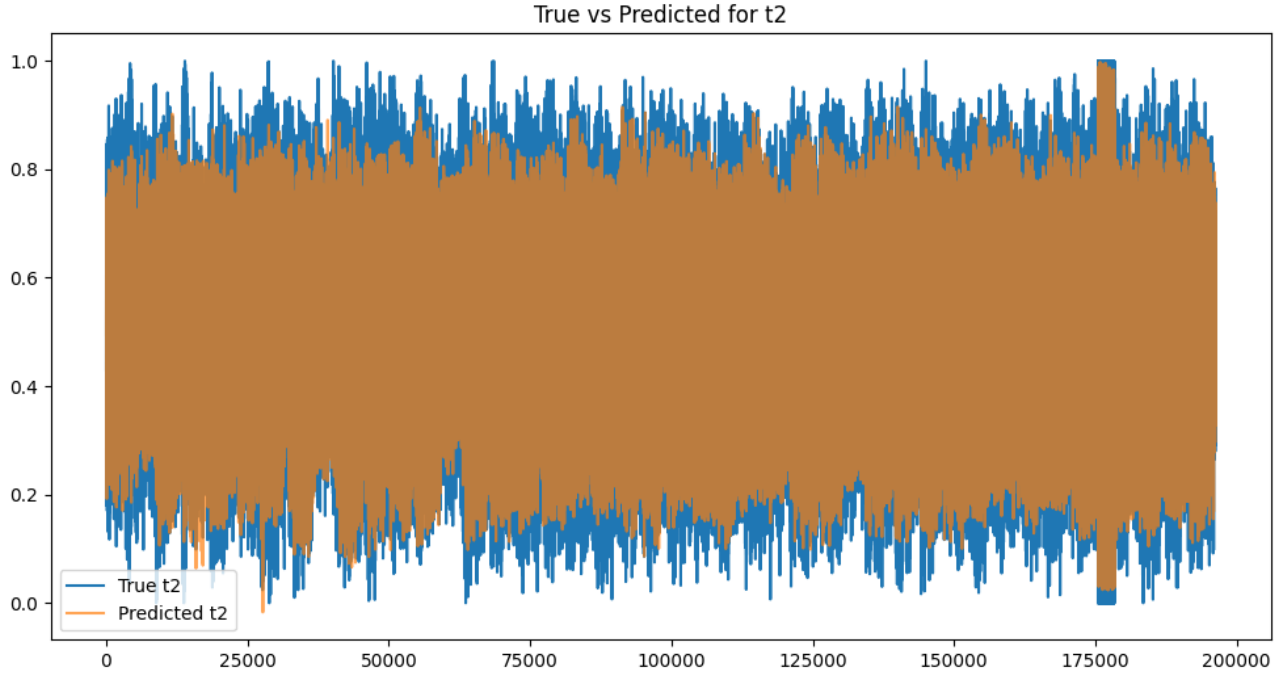

Figure 2.5: Plot for t2

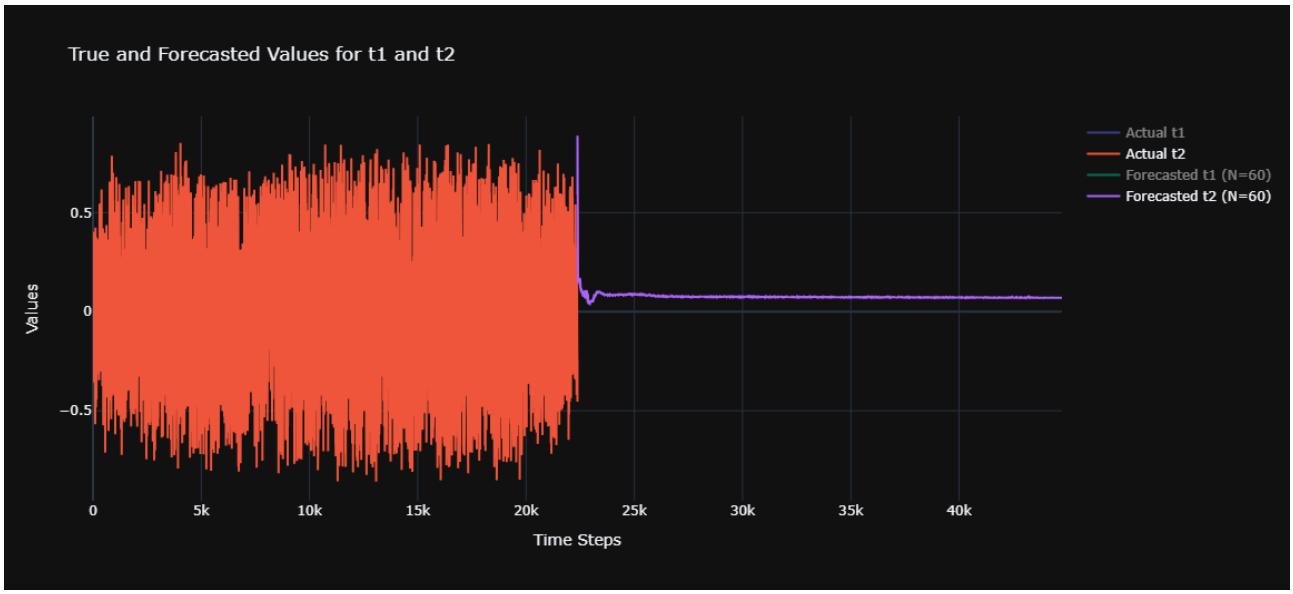

Figure 2.6: Interactive Plot for Forecasts using GRU

2.2.2 Forecasting

(see Figure 2.6)

2.3 Bidirectional Long Short Term Memory (Bidirectional LSTM)

Following the same procedure we now

```
# Initializing model building
def build_model(X_train):
    model = Sequential([
        Bidirectional(LSTM(128, input_shape=(X_train.shape[1], X_train.shape[2]),
        ↪  return_sequences=True)),
        Dropout(0.5),
        Bidirectional(LSTM(128, return_sequences=True)),
        Dropout(0.4),
        LSTM(64, return_sequences=True),
        Dropout(0.3),
        LSTM(32, return_sequences=False),
        Dropout(0.2),
        Dense(2)
    ])
    model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mae'])
    return model
```

While **GRUs (Gated Recurrent Units)** are similar to LSTMs, they are a simplified version, and each has its strengths. However, LSTM has certain advantages over GRUs, particularly in complex tasks like time series forecasting:

- **Complexity vs. Efficiency**:

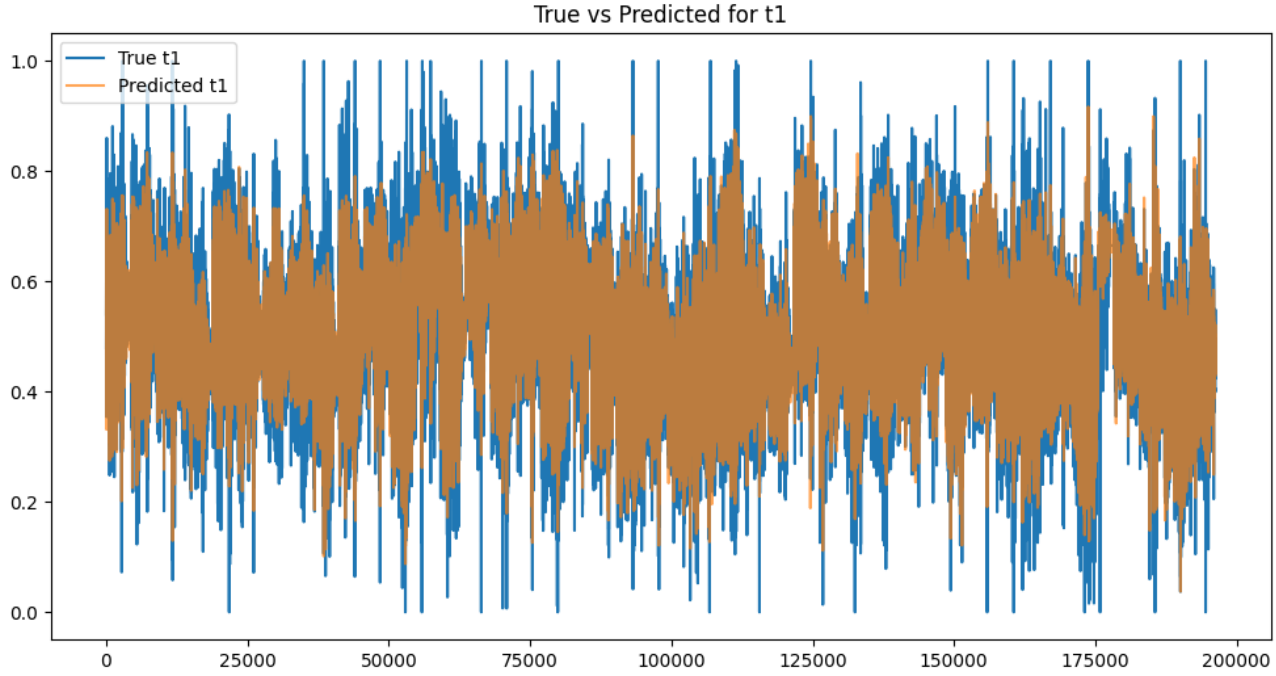

Figure 2.7: Plot for t1

  - **GRU**: Has fewer gates (only **reset** and **update** gates) compared to LSTM. This simplicity can be an advantage in terms of computational efficiency and speed, making GRUs often faster to train. However, this simplicity can also be a limitation when dealing with more complex sequential dependencies.
  - **LSTM**: With its three gates (forget, input, and output), LSTM can capture more detailed dependencies in the data, making it more powerful in certain contexts, especially when the data has very long-term dependencies.

- **Bidirectional Layers:** The model uses **Bidirectional LSTMs**, which means that the model processes the input sequence both forwards and backwards. This gives it access to both past and future context when making predictions, which can be particularly useful in time series tasks where both past and future values are important for predicting the current state.
  - **Forward and Backward Context**: By processing sequences in both directions, the LSTM can leverage future context as well as past context, which is critical for tasks where both past and future information can help make better predictions.
- **Regularization (Dropout)** The model includes several **Dropout layers**, which help to prevent overfitting by randomly deactivating a fraction of neurons during training. This is a standard technique to improve generalization and prevent the model from memorizing the training data (overfitting).
  - **Dropout at Different Layers**: In this model, dropout rates are applied at various stages (0.5, 0.4, 0.3, and 0.2), with higher rates in the early layers and lower rates in the deeper layers, which is a common approach to balance between preventing overfitting and allowing the model to learn complex features in deeper layers.

This LSTM-based model, with its complex architecture and regularization, is therefore well-suited for forecasting tasks involving time series data with long-term dependencies and intricate patterns.

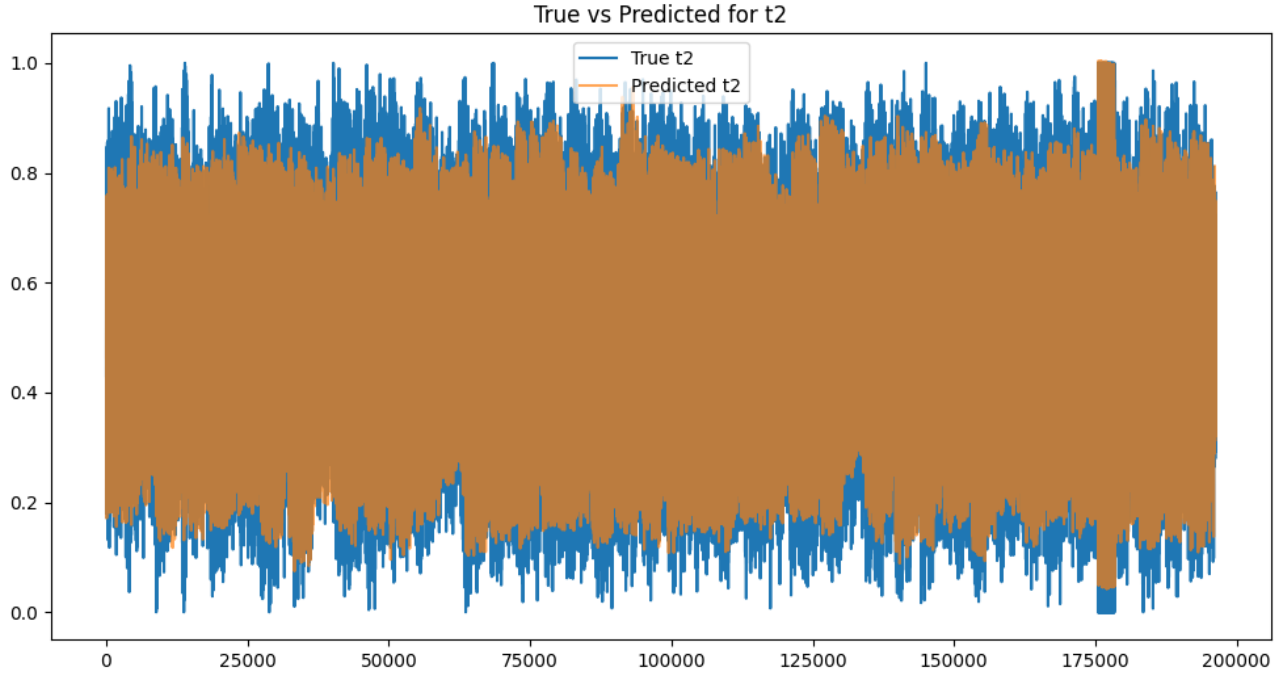

Figure 2.8: Plot for t2

2.3.1 Model Evaluation

1. **Test Loss**

```
Test Loss: 0.002703288337215781
```

   - The **Test Loss** is `0.0029`, which represents the mean squared error (MSE) between the predicted values and the actual values on the test data. A low value indicates that the model's predictions are close to the actual values, meaning the model is performing well in terms of minimizing errors.

2. **Test MAE (Mean Absolute Error)**

```
, Test MAE: 0.03661471977829933
```

   - The **Test MAE** is `0.0380`, which represents the average absolute difference between the predicted and actual values. This is another measure of prediction accuracy, where lower values indicate better performance. An MAE of `0.038` suggests that, on average, the model's predictions are off by about `0.038` units, which is relatively small.

3. **R-squared (R^2) for `t1` and `t2`**

```
R-squared for t1: 0.7950593269736068
R-squared for t2: 0.7615569217187673
```

   - **R-squared** is a statistical measure that indicates how well the model explains the variance in the data. It ranges from `0` to `1`, where values closer to `1` indicate a better fit:
     - **R-squared for `t1`** is `0.7950`, meaning the model explains about 77% of the variance in the target variable `t1`. This indicates a good fit.
     - **R-squared for `t2`** is `0.7615`, meaning the model explains about 74% of the variance in the

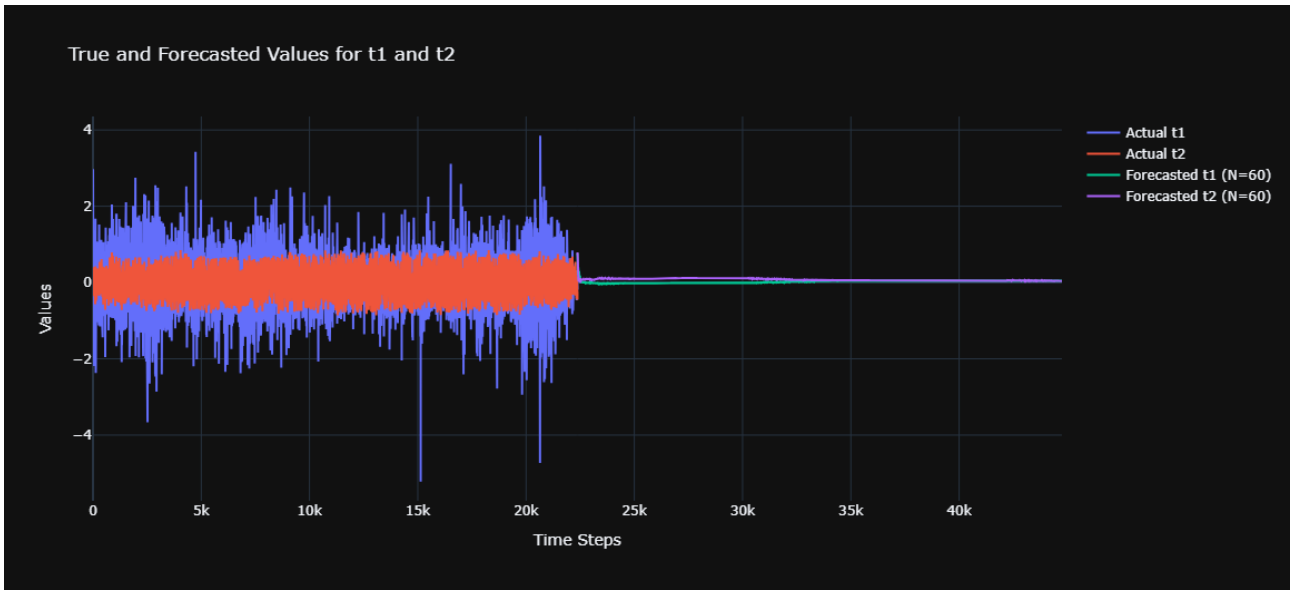

Figure 2.9: Interactive Plot for Forecasts using Bidirectional LSTM

target variable `t2`, which is also a good result but slightly lower than for `t1`.

Overall Interpretation

- The **Test Loss** and **Test MAE** indicate that the model is making predictions with relatively small errors.
- The **R-squared values** show that the model is capturing a significant portion of the variance in both target variables (`t1` and `t2`), indicating strong predictive power.

In summary, the LSTM model demonstrates good performance on the test data with low errors and strong explanatory power for both target variables.

2.3.2 Forecasts

(see Figure 2.9)

2.4 Combined Evaluation Results

Model	Test Loss	Test MAE	R-squared (t1)	R-squared (t2)
RNN	0.0028	0.0381	0.7576	0.7524
GRU	0.0027	0.0381	0.7587	0.7592
LSTM	0.0027	0.0366	0.7950	0.7615

Table 2.1: Evaluation Results of RNN, GRU, and LSTM Models

2.5 Introduction to the Time Series Forecasting App

This Streamlit application provides an interactive platform for time series forecasting using deep learning models. Users can upload a time series data file, select a model (Simple RNN or GRU), and specify a forecasting window size. The app preprocesses the data by applying an Exponential Moving Average (EMA) and generating lagged

features, which improve forecasting accuracy. Using the selected model and window size, the app generates forecasts that are visualized alongside actual data to show the model's performance.

2.5.1 Model and Background Image Paths

- **Model Files**: The pre-trained models are stored as `.h5` files and loaded based on user selection. The paths to these models are as follows (vary from system to sytem):
  - Simple RNN: `C:\Users\Lakshya Singh\Downloads\RNN_model.h5`
  - GRU: `C:\Users\Lakshya Singh\Downloads\GRU_model.h5`
- **Background Image**: The app uses a custom background image located at `C:\Users\Lakshya Singh\Downloads\bg_tsa` (vary from system to sytem), enhancing the visual appeal and improving readability with white text overlay.

2.5.2 How to Access and Use the App

1. **Open the Streamlit App**: In the project directory, run the app using:

```
streamlit run streamlit_tsa.py
```

2. **Upload a Data File**: Use the "Upload your CSV file" button to upload a CSV file containing columns `t1` and `t2`.
3. **Select Model and Window Size**:
   - Choose between Simple RNN and GRU (Figure 2.10).
   - Adjust the window size slider to set the forecasting window.
4. **View Forecast**: The app will display an interactive plot with true values and forecasted values for both `t1` and `t2`, allowing easy evaluation of the model's predictions (Figure 2.11).

This application offers a straightforward interface, enabling users to explore deep learning forecasting models and adjust parameters interactively for time series analysis.

Figure 2.10: Screenshot of Streamlit APP (I)

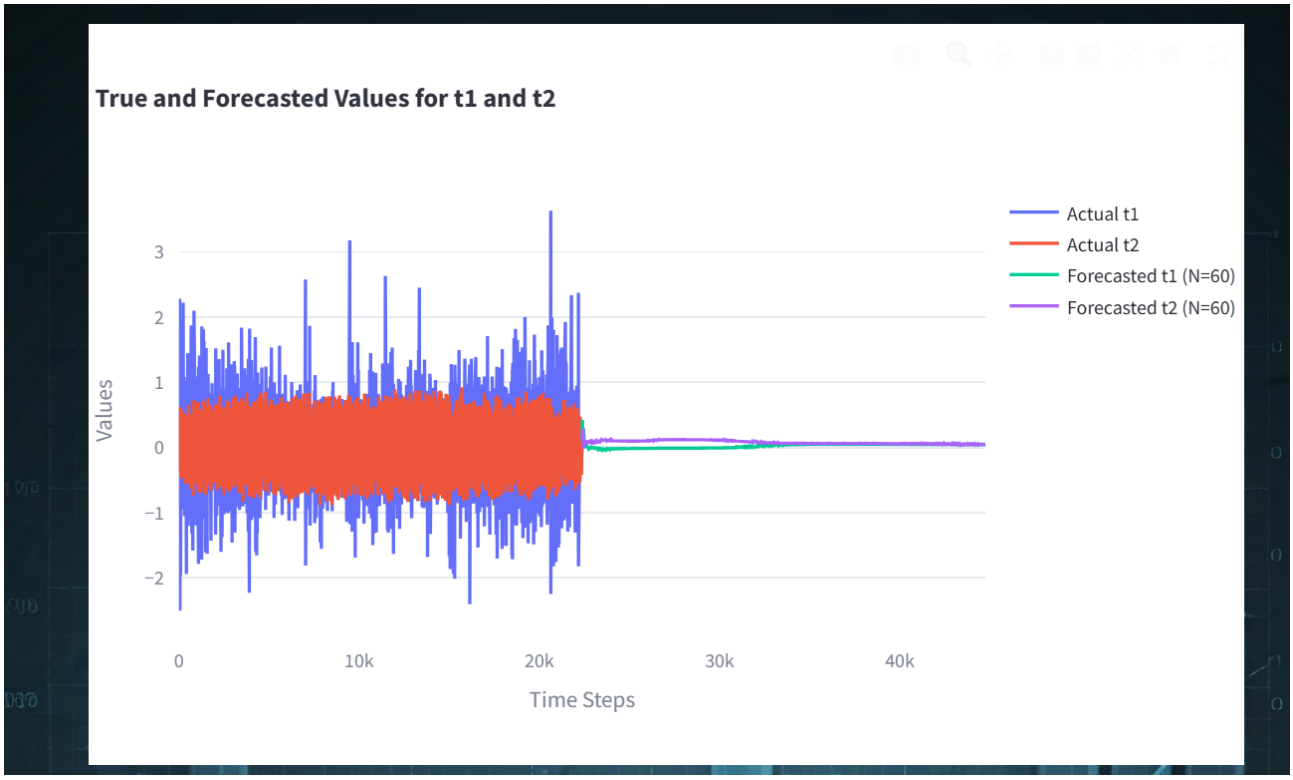

Figure 2.11: Screenshot of Streamlit APP (II)

3

Insights into Various Data Patterns of Time Series

For a time series analysis, uncovering data patterns is essential to understanding the underlying trends, seasonality, cycles, and irregular fluctuations within the data. These insights reveal the behaviors and structures that shape the time series over time, offering context that enhances forecasting accuracy and guides decision-making. By examining these patterns, we can identify regular trends, seasonal shifts, unexpected anomalies, and other features that affect predictions. This section will delve into the key patterns observed, interpreting their implications for the data and how they inform both the analytical approach and model selection.

3.1 Working with Consolidated Data File

```
import pandas as pd
import os
from glob import glob
import numpy as np
from google.colab import drive, files
import matplotlib.pyplot as plt
import seaborn as sns


# Function to load and preprocess data
def load_and_preprocess_data(folder_path):
    all_data = []

    for file_name in os.listdir(folder_path):
        if file_name.endswith('.csv'):
            file_path = os.path.join(folder_path, file_name)

            df = pd.read_csv(file_path)

            # Checking for essential columns
            required_columns = ['Time', 't1', 't2']
            if not all(col in df.columns for col in required_columns):
```

```
                raise ValueError(f"File {file_name} is missing one or more required
                ↪  columns: {required_columns}")

            # Set 'Time' as the index
            df.set_index('Time', inplace=True)

            df.sort_index(inplace=True)
            all_data.append(df)

    consolidated_df = pd.concat(all_data)

    return consolidated_df

data = load_and_preprocess_data(folder_path)

print("Data successfully loaded and preprocessed!")
```

This pre-processing function, `load_and_preprocess_data`, is designed to efficiently load, validate, and consolidate multiple CSV files containing time series data. Here's an overview of each step:

1. **Folder Scanning**:
   - The function scans the specified `folder_path` to retrieve all CSV files, allowing for easy loading of large, segmented time series data from multiple files.
2. **File Loading and Column Validation**:
   - For each CSV file, it loads the data using `pandas`. It then checks for the essential columns (`Time`, `t1`, and `t2`), which are necessary for consistent analysis. If any file is missing these columns, the function raises an error to alert the user.
3. **Indexing and Sorting by Time**:
   - The `Time` column is set as the index to structure the data in a time-ordered format, essential for chronological data analysis. Sorting the index ensures that the data remains consistent across the combined dataset.
4. **Concatenating Data**:
   - After processing each file individually, all the data frames are combined into a single consolidated data frame. This unified dataset allows for more efficient analysis of the entire time series.

This pre-processing step enables seamless handling of time series data across multiple files, ensuring the data is structured, validated, and ready for deeper analysis.

3.1.1 Decomposition of Time Series

```
from statsmodels.tsa.seasonal import seasonal_decompose

# Decompose the time series for t1 and t2
```

```
decomposition_t1 = seasonal_decompose(df['t1'], model='additive', period=86400) #
↪  Assuming 1 day period (24*60*60 seconds)
decomposition_t2 = seasonal_decompose(df['t2'], model='additive', period=86400)

# Plot decomposition for t1
plt.figure(figsize=(12, 8))
plt.subplot(4, 1, 1)
plt.plot(df['Time'], df['t1'], label='Original (t1)')
plt.title('Original Time Series (t1)')
plt.subplot(4, 1, 2)
plt.plot(df['Time'], decomposition_t1.trend, label='Trend', color='orange')
plt.title('Trend Component (t1)')
plt.subplot(4, 1, 3)
plt.plot(df['Time'], decomposition_t1.seasonal, label='Seasonality', color='green')
plt.title('Seasonal Component (t1)')
plt.subplot(4, 1, 4)
plt.plot(df['Time'], decomposition_t1.resid, label='Residuals', color='red')
plt.title('Residuals (t1)')
plt.tight_layout()
plt.show()
```

1. **Library Import and Decomposition Setup**:
   - The function `seasonal_decompose` from `statsmodels` is used to decompose the time series into three main components. We specify an `additive` model, meaning that each component (trend, seasonal, residual) is added together to form the original series.
   - The `period` parameter is set to `86400` (24 hours in seconds) to account for daily cycles, assuming the data has a daily seasonal pattern.
2. **Decomposing Components**:
   - For each target variable (`t1` and `t2`), the code decomposes the time series into:
     - **Trend**:No overall direction or movement in the data over a longer period.
     - **Seasonality**: No Repeated patterns or cycles occurring at a consistent frequency.
     - **Residuals**: Many irregular fluctuations or noise after removing the trend and seasonality.

Significance of Seasonal Decomposition

- This decomposition allows us to analyze each pattern individually, identifying the effects of trends, seasonal fluctuations, and irregular noise. By isolating each component, we gain insights into how seasonality or long-term trends impact the data, which can help inform forecasting and data-driven decisions.

ARIMA (AutoRegressive Integrated Moving Average) and SARIMA (Seasonal ARIMA) models are most effective when the time series data has clear, stable patterns in trend and seasonality. When a time series lacks such stable trend and seasonal components and instead exhibits high irregularity, ARIMA and SARIMA are often insufficient for accurate modeling and forecasting. Hence we will avoid these two models for t1 and t2.

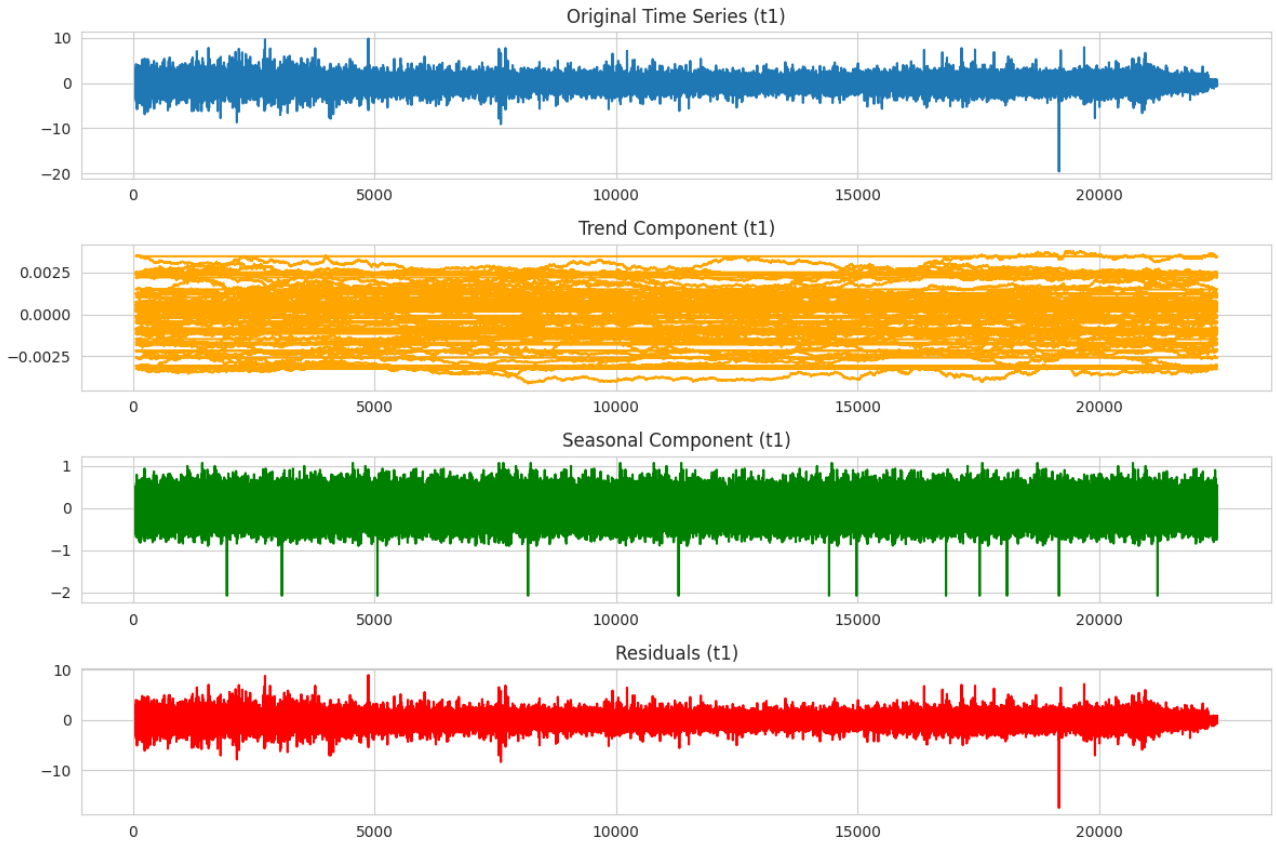

Figure 3.1: Seasonal Decomposition of t1

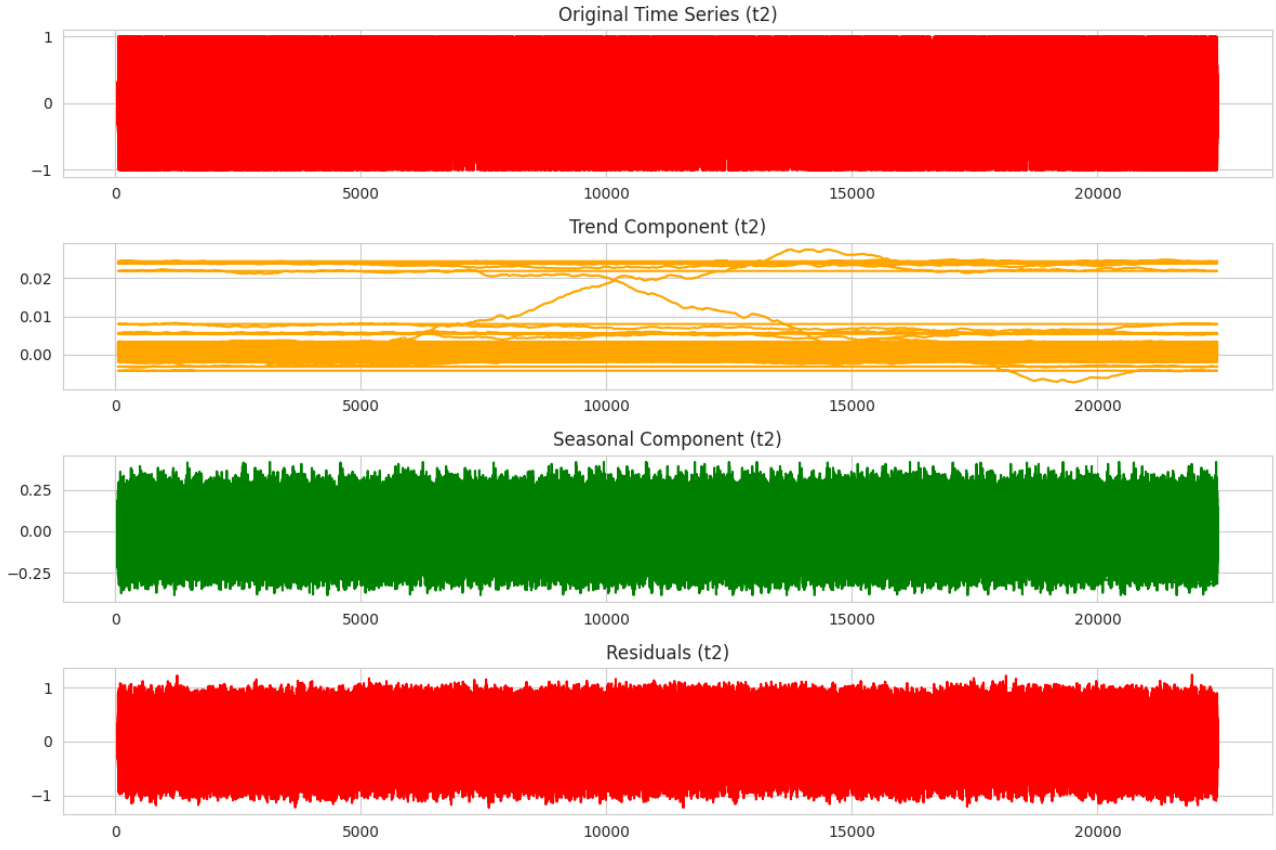

Figure 3.2: Seasonal Decomposition of t2

3.1.2 ACF & PACF Plots (using R)

```
library(readr)
df <- read_csv("C:/Users/Lakshya Singh/Downloads/consolidated_df.csv")

# Convert t1 and t2 to time series objects
ts_t1 <- ts(df$t1, frequency=600)
ts_t2 <- ts(df$t2, frequency=600)

# ACF and PACF for t1
acf(ts_t1, main="ACF for t1")
pacf(ts_t1, main="PACF for t1")

# ACF and PACF for t2
acf(ts_t2, main="ACF for t2")
pacf(ts_t2, main="PACF for t2")
```

(a) **Autocorrelation Function (ACF)**:

- The **ACF** measures the correlation between a time series and its lagged versions across multiple time lags.
- **Insights from ACF of t1**:
  - **Seasonality**: Peaks at regular intervals in the ACF plot suggest slight seasonal patterns in the data.
  - **Trend**: A fast decay in autocorrelation values over lags indicates no trend in the data.
  - **Moving Average (MA) Component**: No Significant spikes at specific lags followed by a sharp drop.
- **Insights from ACF of t2**:
  - **Seasonality**: No Peaks at regular intervals in the ACF plot.
  - **Trend**: A slow decay in autocorrelation values over lags indicates slighttrend in the data.
  - **Moving Average (MA) Component**: No Significant spikes at specific lags followed by a sharp drop.

(b) **Partial Autocorrelation Function (PACF)**:

- The **PACF** measures the correlation between a time series and its lagged values, after removing the effect of intermediate lags. It shows the unique contribution of each lag to the series, isolating direct correlations without influence from other lags.
- **Insights from PACF of t1**:
  - **AR Component**: A significant spike at lag 1 (or another specific lag) followed by a rapid decline suggests the presence of an autoregressive (AR) component. Clearly no AR component is there.
- **Insights from PACF of t2**:
  - **AR Component**: A significant spike at lag 1 (or another specific lag) followed by a rapid decline suggests the presence of an autoregressive (AR) component.

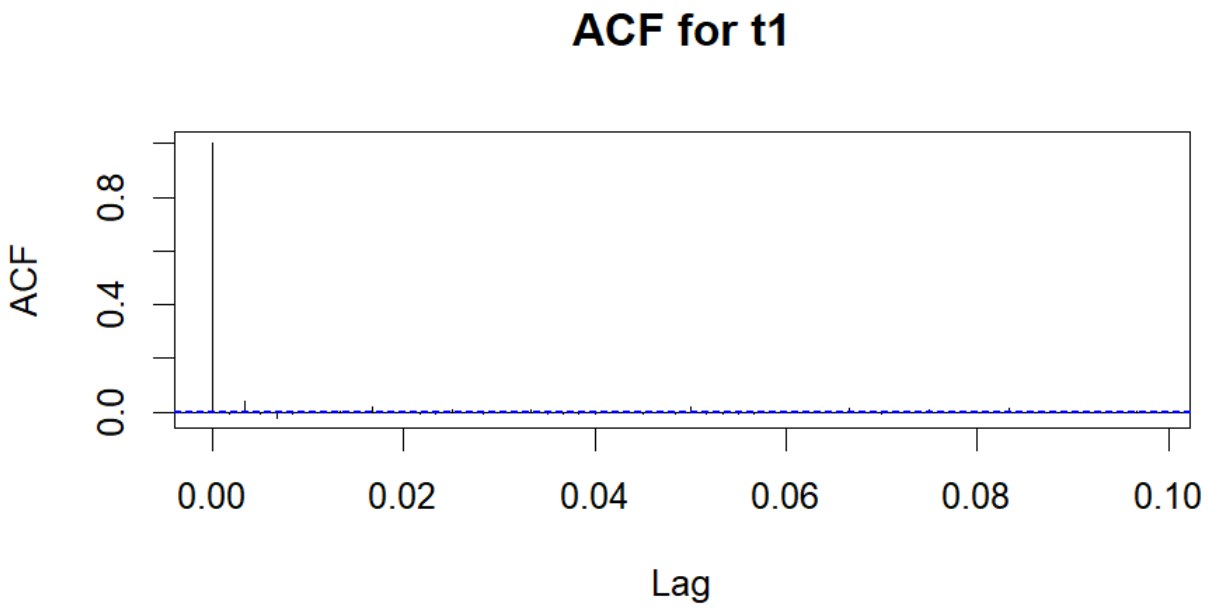

Figure 3.3: ACF of t1

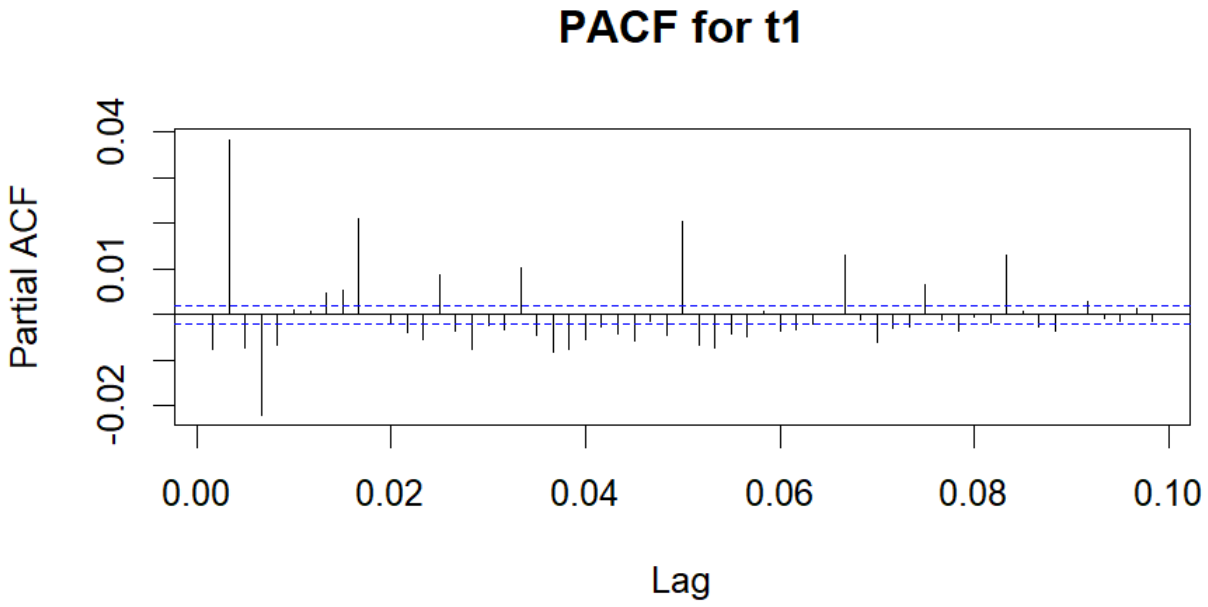

Figure 3.4: PACF of t1

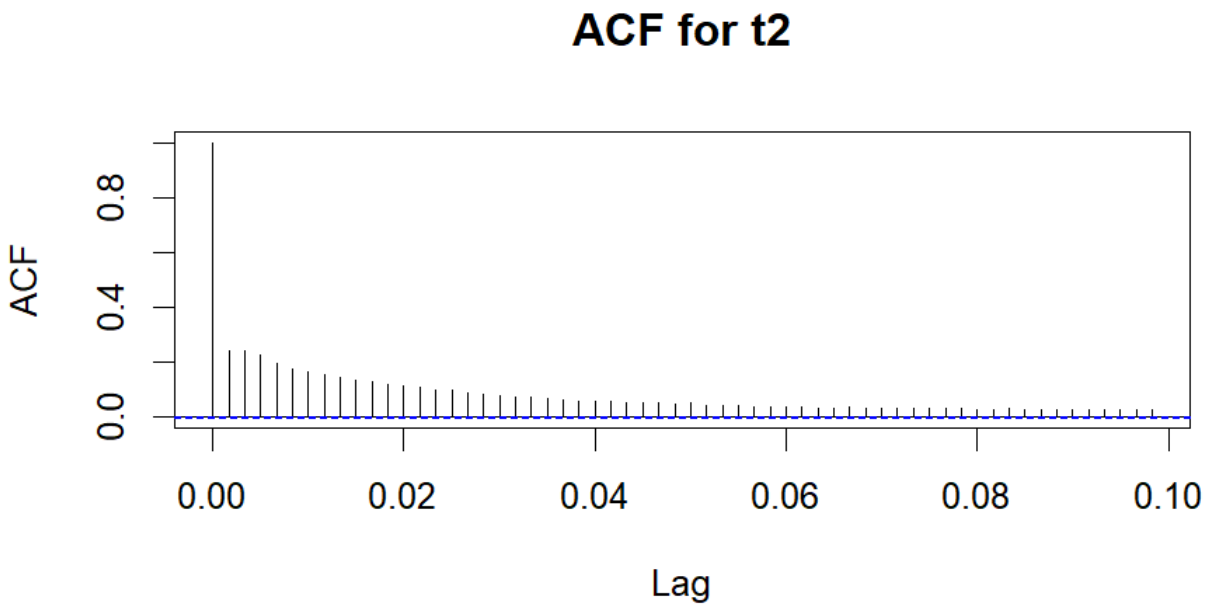

Figure 3.5: ACF of t2

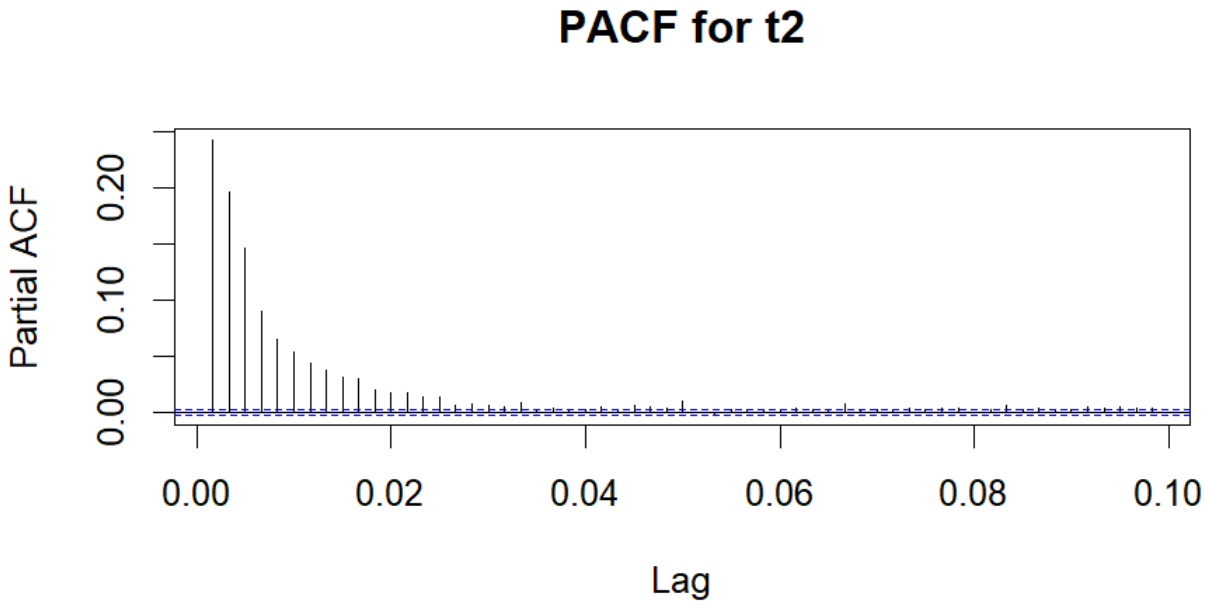

Figure 3.6: PACF of t2

The analysis done so far can also be performed on a single data file. This allows for a more focused examination of the individual time series patterns within that file, which might reveal unique dependencies, trends, or seasonal patterns specific to that dataset. In this project (`'TSA_EDA2.ipynb'`), similar code could be applied to any single dataset to assess these patterns before consolidating insights across multiple files. This file-specific analysis could be beneficial for detecting particular lags or auto-correlations that vary between different time series files.

CONCLUSION

In conclusion, this project successfully developed and evaluated predictive models capable of forecasting future values for `t1` and `t2` over varying time windows. Through comprehensive pre-processing, including scaling, smoothing, and feature engineering, we ensured the data was well-prepared for modeling. The deep learning models, particularly LSTM and GRU architectures, demonstrated strong performance in capturing temporal dependencies and producing accurate forecasts. Additionally, the creation of an interactive Streamlit application allowed for real-time visualization of the forecasted results, providing a user-friendly interface for exploring the predictions. The insights gained from exploratory data analysis further enhanced our understanding of the data's underlying patterns, which informed model tuning and improved prediction accuracy. Overall, the work presented in this report offers a robust framework for time series forecasting and serves as a foundation for further refinement and application in real-world scenarios.