Phase 4: Development Part 2 - Model Training and Evaluation

Objective:

In this phase, we will continue building the "Measure Energy Consumption" project by performing Feature Engineering , Model Training and Evaluation on the dataset obtained by Kaggle.The primary goal is to have a model with high accuracy and integration with other functionalities.

Team Details:

1. College Name: Madras Institute of Technology, Anna University

2. Team Members:

Name	Email	NM ID
Fowzaan Abdur Razzaq Rasheed	fowzaan.rasheed@gmail.com	8E4AF1FB4D2CAD089814D6BED938AC27
Mohit S	smohit28.04@gmail.com	B80CBC310CADE36AB9A4F5A439515636
Pronoy Kundu	pronoykundu513@gmail.com	11306F001F2B3639BBE4CB15C475F9EC

Importing Libraries and Setting Up the Environment

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
import sklearn.preprocessing
from sklearn.metrics import r2_score
from keras.layers import Dense, Dropout, SimpleRNN, LSTM, Bidirectional, Input , Re
from keras.models import Sequential, Model
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, ReduceLROnPlateau, EarlyStopping
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import math
import os
import seaborn as sns
from google.colab import drive
drive.mount('/content/drive')
print("Libraries imported successfully")
```

```
Mounted at /content/drive
Libraries imported successfully
```

.....

.....

Loading The Dataset

The dataset AEP_hourly.csv is loaded using the pandas library's read_csv function. The parse_dates parameter is set to ['Datetime'] to ensure that the 'Datetime' column is recognized as a datetime object. The index_col parameter is set to 'Datetime' to set the 'Datetime' column as the index of the DataFrame.

In [ ]:
```
# Load AEP_hourly.csv
aep_data = pd.read_csv('/content/drive/MyDrive/NM Proj/AEP_hourly.csv', parse_dates
print("Data loaded successfully.")
print("AEP Data Shape:", aep_data.shape)
```

```
Data loaded successfully.
AEP Data Shape: (121273, 1)
```

Data Preprocessing: Handling Missing Values and Outliers in AEP Time Series Data

Checking for missing values

- We check for missing values in the DataFrame using the isnull().sum() function. If any missing values are found, they are removed using the dropna() function. If no missing values are found, the original DataFrame is used.

In [ ]:
```
# Count missing values
missing_values = aep_data.isnull().sum()

if missing_values.any():
    # Remove rows with missing values
    aep_data_cleaned = aep_data.dropna()

    print("Number of Missing Values:")
    print(missing_values)

    print("Data cleaned successfully.")
    print("AEP Data Shape (Cleaned):", aep_data_cleaned.shape)
else:
    print("No missing values found. Data is already clean.")
    aep_data_cleaned = aep_data
```

```
No missing values found. Data is already clean.
```

Checking for Outliers

- We check for outliers in the 'AEP_MW' column of the DataFrame. Outliers are identified using the Z-score method, where any data point with a Z-score greater than a threshold (set to 3 in this case) is considered an outlier. If any outliers are found, they are removed from the DataFrame

In [ ]:
```
# Count outliers
outlier_threshold = 3  # You can adjust this threshold as needed
```

```
z_scores = (aep_data_cleaned['AEP_MW'] - aep_data_cleaned['AEP_MW'].mean()) / aep_c
outliers = (z_scores.abs() > outlier_threshold)
num_outliers = outliers.sum()

if num_outliers > 0:
    # Remove rows with outliers
    aep_data_cleaned = aep_data_cleaned[~outliers]

    print("Number of Outliers:", num_outliers)
    print("Data cleaned successfully.")
    print("AEP Data Shape (Cleaned):", aep_data_cleaned.shape)
else:
    print("No outliers found. Data is already clean.")
```

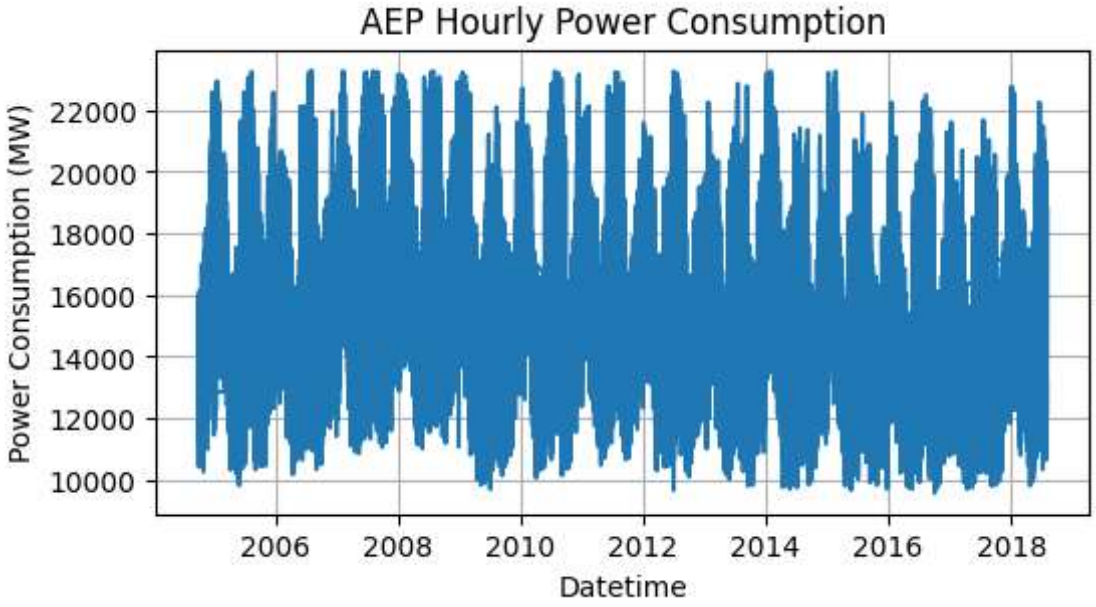
```
Number of Outliers: 39
Data cleaned successfully.
AEP Data Shape (Cleaned): (120975, 10)
```

Preliminary Data Analysis

Visualization are created to better understand the trends in the dataset.

Time Series Plot

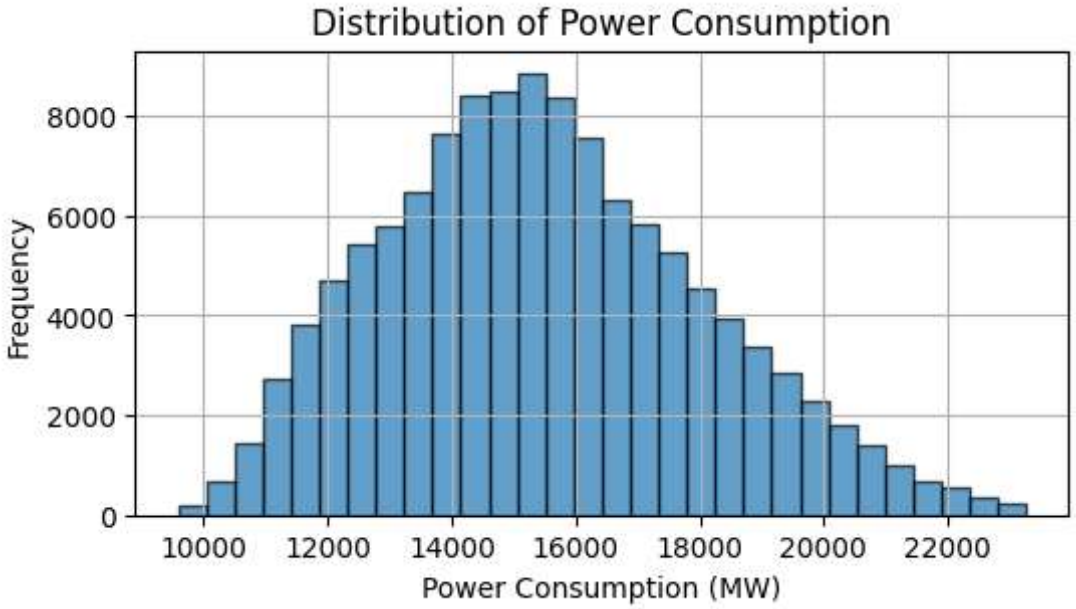
In [ ]:
```
plt.figure(figsize=(6, 3))
plt.plot(aep_data_cleaned.index, aep_data_cleaned['AEP_MW'], label='AEP_MW')
plt.title('AEP Hourly Power Consumption')
plt.xlabel('Datetime')
plt.ylabel('Power Consumption (MW)')
#plt.legend()
plt.grid(True)
plt.show()
```



Histogram Plot

In [ ]:
```
plt.figure(figsize=(6, 3))
plt.hist(aep_data_cleaned['AEP_MW'], bins=30, edgecolor='k', alpha=0.7)
plt.title('Distribution of Power Consumption')
plt.xlabel('Power Consumption (MW)')
plt.ylabel('Frequency')
```

```
plt.grid(True)
plt.show()
```



Feature Extraction

- In this section , we extract various features from the 'Datetime' index, such as the hour of the day, the day of the week, the quarter of the year, the month, the year, the day of the year, and the week of the year. These features are added as new columns in the DataFrame. Additionally, a 'season' feature is created based on the 'month' feature.

In [ ]:
```
aep_data_cleaned = aep_data_cleaned.copy()
# Feature Engineering
aep_data_cleaned['hour'] = aep_data_cleaned.index.hour
aep_data_cleaned['dayofweek'] = aep_data_cleaned.index.dayofweek
aep_data_cleaned['quarter'] = aep_data_cleaned.index.quarter
aep_data_cleaned['month'] = aep_data_cleaned.index.month
aep_data_cleaned['year'] = aep_data_cleaned.index.year
aep_data_cleaned['dayofyear'] = aep_data_cleaned.index.dayofyear
aep_data_cleaned['dayofmonth'] = aep_data_cleaned.index.day
aep_data_cleaned['weekofyear'] = aep_data_cleaned.index.isocalendar().week
aep_data_cleaned['season'] = aep_data_cleaned['month'] % 12 // 3 + 1

# Print the updated DataFrame to CSV
print(aep_data_cleaned.head())
aep_data_cleaned.to_csv('AEP_hourly_cleaned.csv')
print("Feature Extraction Completed Successfully")
```

```
                     AEP_MW  hour  dayofweek  quarter  month  year  \
Datetime
2004-12-31 01:00:00  13478.0     1          4        4     12  2004
2004-12-31 02:00:00  12865.0     2          4        4     12  2004
2004-12-31 03:00:00  12577.0     3          4        4     12  2004
2004-12-31 04:00:00  12517.0     4          4        4     12  2004
2004-12-31 05:00:00  12670.0     5          4        4     12  2004

                     dayofyear  dayofmonth  weekofyear  season
Datetime
2004-12-31 01:00:00        366          31          53       1
2004-12-31 02:00:00        366          31          53       1
2004-12-31 03:00:00        366          31          53       1
2004-12-31 04:00:00        366          31          53       1
2004-12-31 05:00:00        366          31          53       1
Feature Extraction Completed Successfully
```

Exploratory Data Analysis

- The code performs exploratory data analysis on the cleaned and preprocessed DataFrame. It prints the first few rows of the DataFrame, the basic statistics of the numerical columns, the shape of the DataFrame, and information about the DataFrame. Additionally, it visualizes the data using box plots and heatmaps to understand the distribution and correlation of different features

Note : The seasons follow this numerical representation

1. Spring
2. Summer
3. Autumn
4. Winter

In [ ]:
```python
print("First few rows of the dataset:")
print(aep_data_cleaned.head())

print("\n\nBasic statistics of numerical columns:")
print(aep_data_cleaned.describe())

print("\n\nDataset shape (rows, columns):\n", aep_data_cleaned.shape)

print("\n\nInformation about the dataset:")
print(aep_data_cleaned.info())
```

First few rows of the dataset:

Datetime	AEP_MW	hour	dayofweek	quarter	month	year
2004-12-31 01:00:00	13478.0	1	4	4	12	2004
2004-12-31 02:00:00	12865.0	2	4	4	12	2004
2004-12-31 03:00:00	12577.0	3	4	4	12	2004
2004-12-31 04:00:00	12517.0	4	4	4	12	2004
2004-12-31 05:00:00	12670.0	5	4	4	12	2004

Datetime	dayofyear	dayofmonth	weekofyear	season
2004-12-31 01:00:00	366	31	53	1
2004-12-31 02:00:00	366	31	53	1
2004-12-31 03:00:00	366	31	53	1
2004-12-31 04:00:00	366	31	53	1
2004-12-31 05:00:00	366	31	53	1

Basic statistics of numerical columns:

	AEP_MW	hour	dayofweek	quarter
count	121014.000000	121014.000000	121014.000000	121014.000000
mean	15481.650991	11.493612	3.001380	2.503297
std	2565.116348	6.924680	2.000497	1.122607
min	9581.000000	0.000000	0.000000	1.000000
25%	13626.000000	5.000000	1.000000	2.000000
50%	15303.000000	11.000000	3.000000	2.000000
75%	17186.000000	17.000000	5.000000	4.000000
max	23271.000000	23.000000	6.000000	4.000000

	month	year	dayofyear	dayofmonth	weekofyear
count	121014.000000	121014.000000	121014.000000	121014.000000	121014.0
mean	6.501785	2011.176186	182.466591	15.725784	26.504256
std	3.462516	4.011232	105.876069	8.800901	15.135374
min	1.000000	2004.000000	1.000000	1.000000	1.0
25%	4.000000	2008.000000	91.000000	8.000000	13.0
50%	6.000000	2011.000000	181.000000	16.000000	26.0
75%	10.000000	2015.000000	276.000000	23.000000	40.0
max	12.000000	2018.000000	366.000000	31.000000	53.0

	season
count	121014.000000
mean	2.491596
std	1.114854
min	1.000000
25%	2.000000
50%	2.000000
75%	3.000000
max	4.000000

Dataset shape (rows, columns):
(121014, 10)

Information about the dataset:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 121014 entries, 2004-12-31 01:00:00 to 2018-01-02 00:00:00
Data columns (total 10 columns):

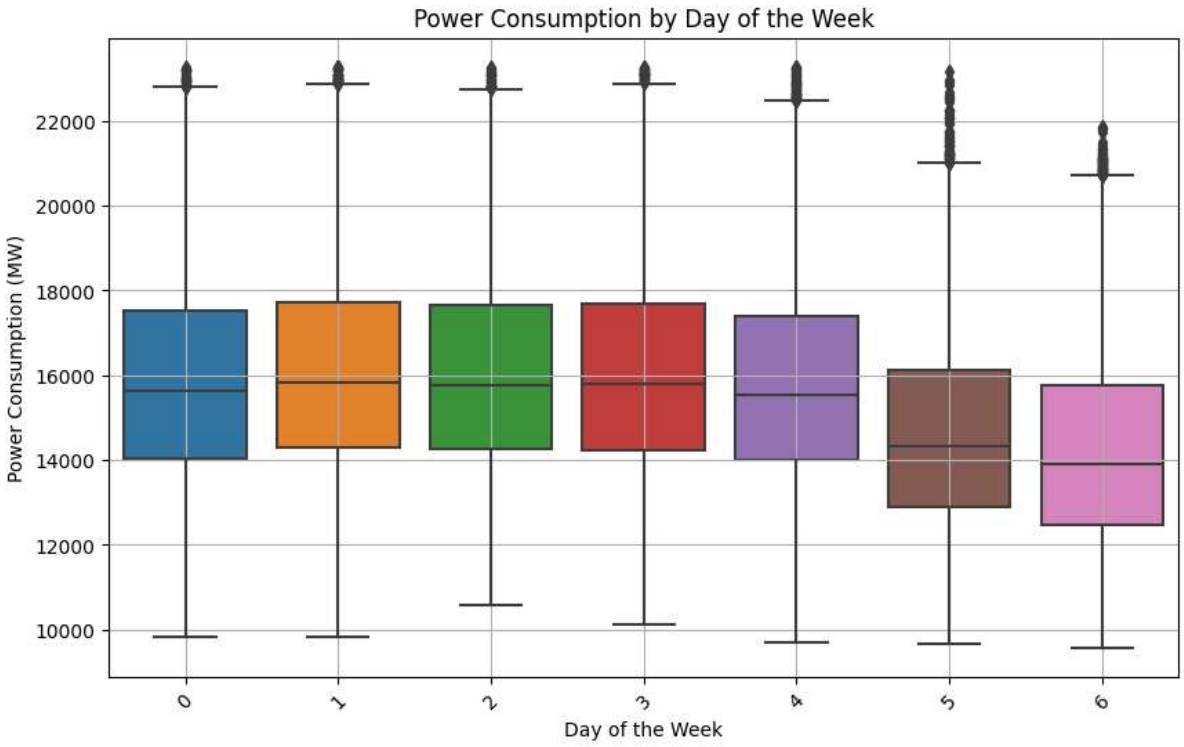
#	Column	Non-Null Count	Dtype
0	AEP_MW	121014 non-null	float64
1	hour	121014 non-null	int64
2	dayofweek	121014 non-null	int64
3	quarter	121014 non-null	int64

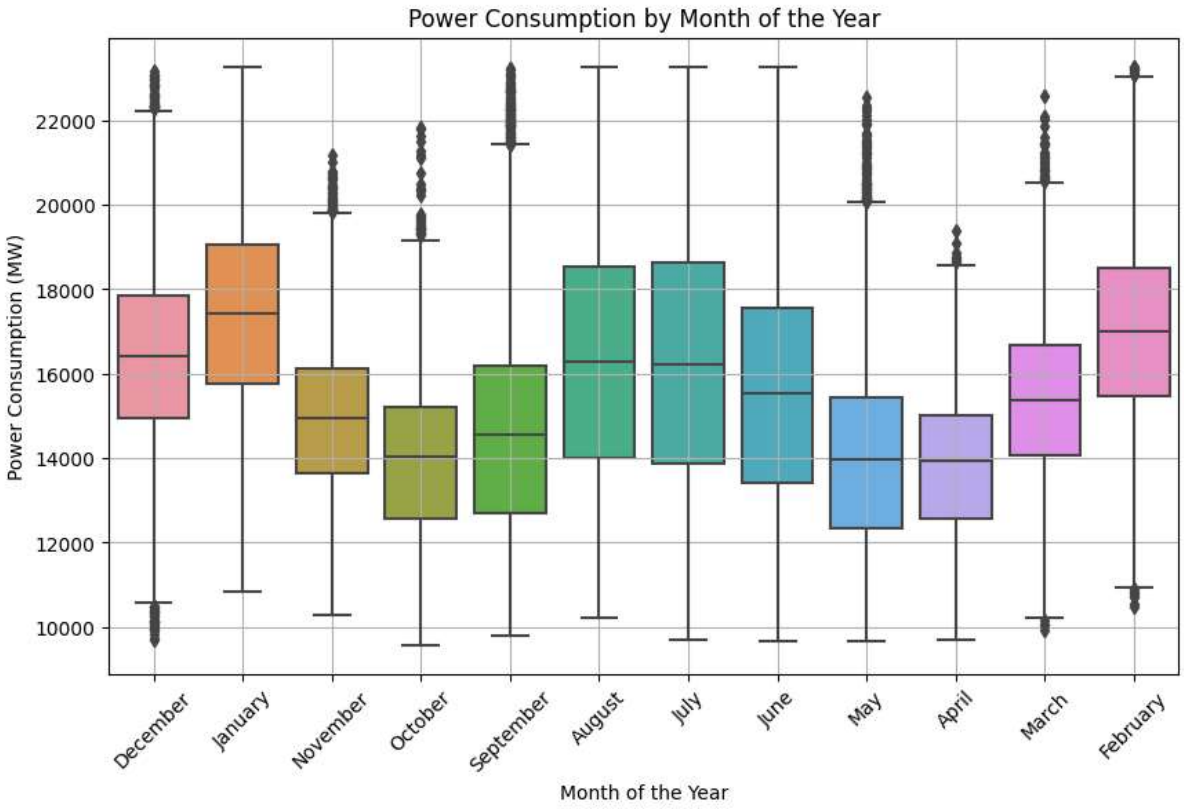
```
 4   month       121014 non-null  int64
 5   year        121014 non-null  int64
 6   dayofyear   121014 non-null  int64
 7   dayofmonth  121014 non-null  int64
 8   weekofyear  121014 non-null  UInt32
 9   season      121014 non-null  int64
dtypes: UInt32(1), float64(1), int64(8)
memory usage: 9.8 MB
None
```

Box Plot

In [ ]:

```python
# Create a copy of the DataFrame
aep_data_copy = aep_data_cleaned.copy()

# Plot without changing the original DataFrame
plt.figure(figsize=(10, 6))
sns.boxplot(x='dayofweek', y='AEP_MW', data=aep_data_copy)
plt.title('Power Consumption by Day of the Week')
plt.xlabel('Day of the Week')
plt.ylabel('Power Consumption (MW)')
plt.xticks(rotation=45)
plt.grid(True)
plt.show()


# Create a copy of the DataFrame
aep_data_copy = aep_data_cleaned.copy()

# Add the 'MonthOfYear' column to the copy
aep_data_copy['monthofyear'] = aep_data_cleaned.index.month_name()

# Plot without changing the original DataFrame
plt.figure(figsize=(10, 6))
sns.boxplot(x='monthofyear', y='AEP_MW', data=aep_data_copy)
plt.title('Power Consumption by Month of the Year')
plt.xlabel('Month of the Year')
plt.ylabel('Power Consumption (MW)')
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```

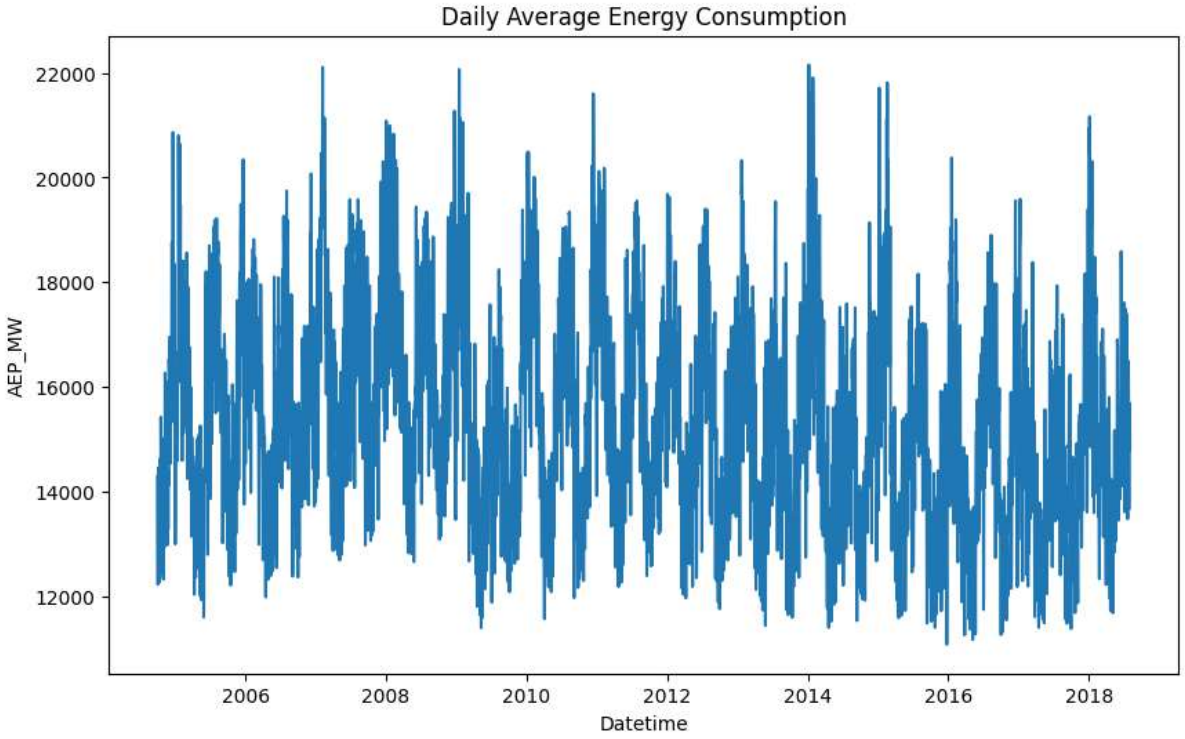


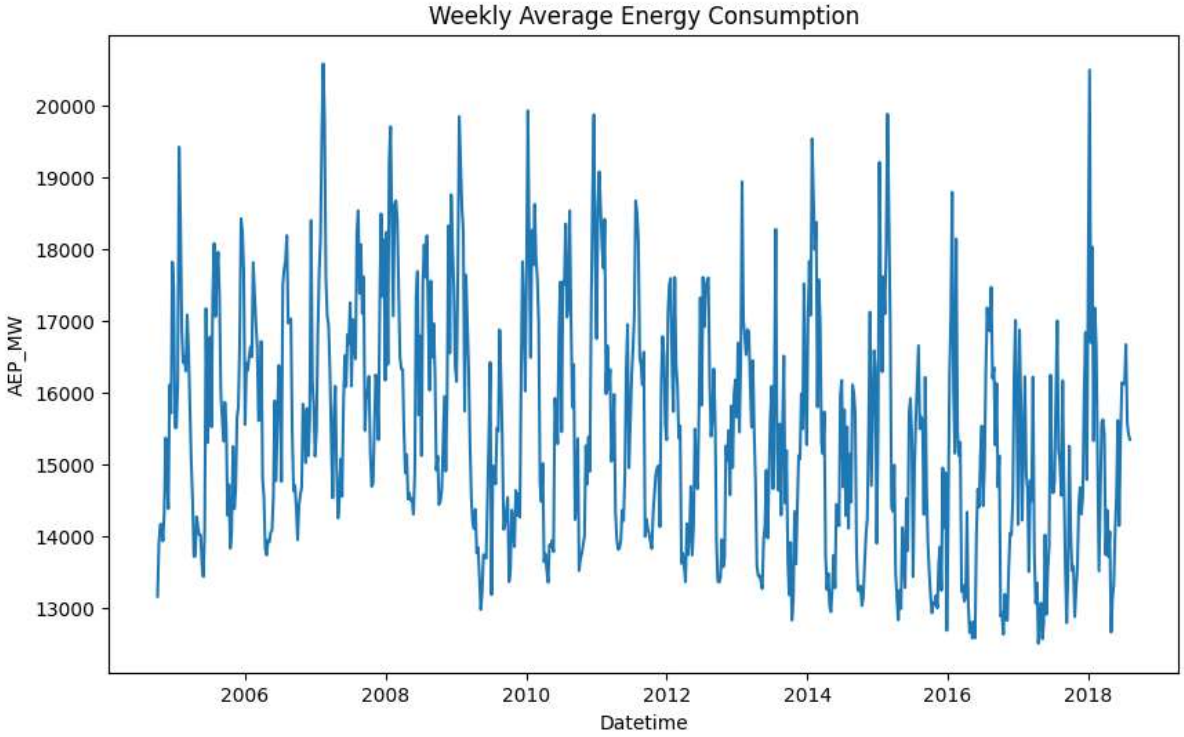


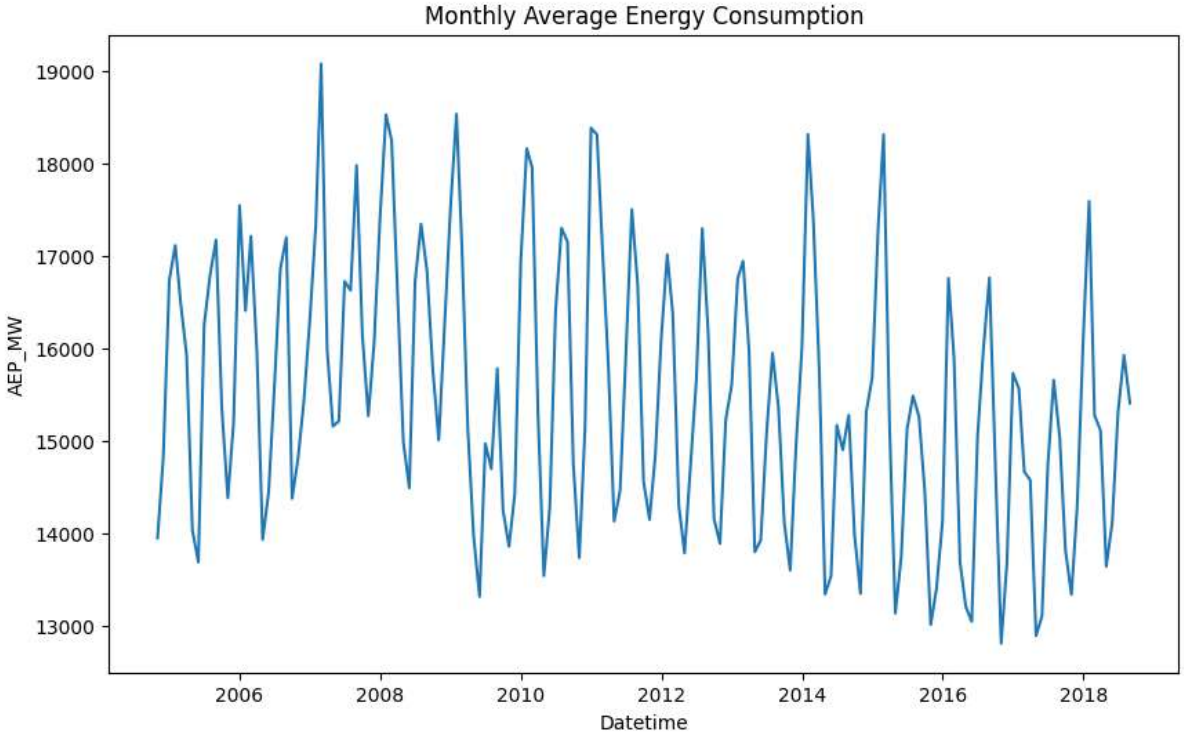
Plotting Seasonality in the Data

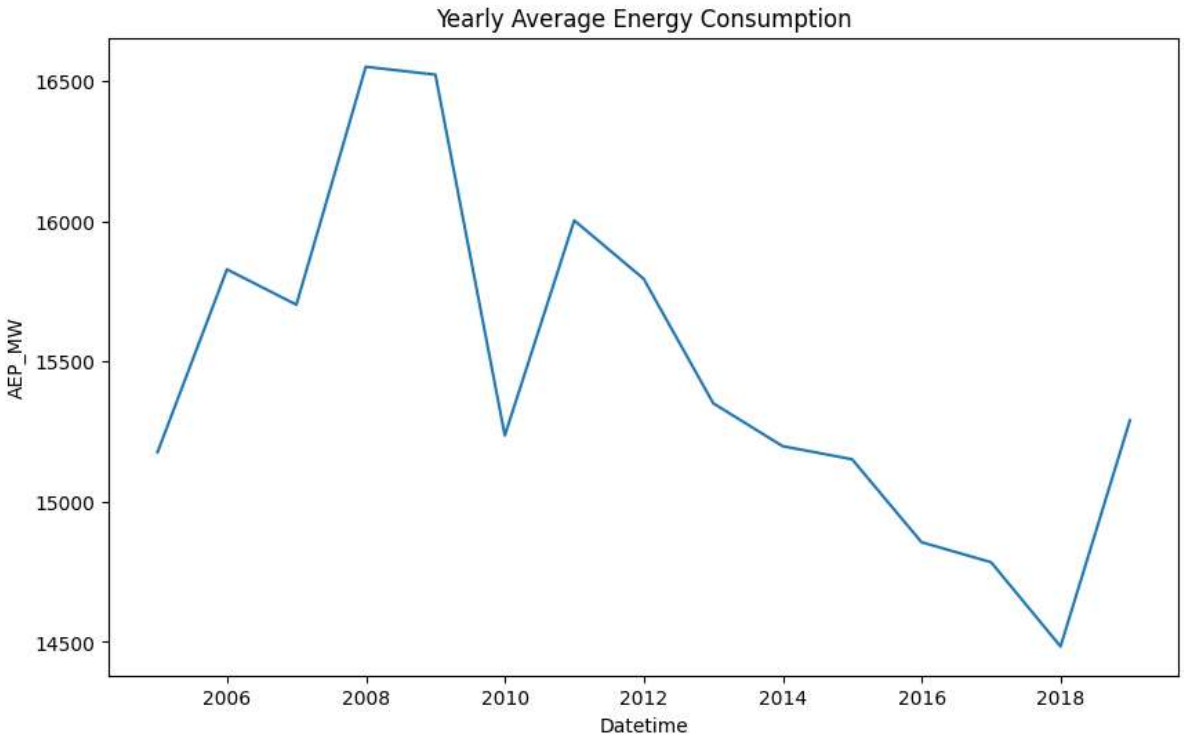
In [ ]:
```
aep_data_cleaned = pd.read_csv('AEP_hourly_cleaned.csv', parse_dates=True, index_co
plt.figure(figsize=(10, 6))
plt.plot(aep_data_cleaned.resample('D')['AEP_MW'].mean())
plt.title('Daily Average Energy Consumption')
plt.xlabel('Datetime')
plt.ylabel('AEP_MW')
plt.show()

# Weekly
plt.figure(figsize=(10, 6))
plt.plot(aep_data_cleaned.resample('W')['AEP_MW'].mean())
plt.title('Weekly Average Energy Consumption')
```

```
plt.xlabel('Datetime')
plt.ylabel('AEP_MW')
plt.show()

# Monthly
plt.figure(figsize=(10, 6))
plt.plot(aep_data_cleaned.resample('M')['AEP_MW'].mean())
plt.title('Monthly Average Energy Consumption')
plt.xlabel('Datetime')
plt.ylabel('AEP_MW')
plt.show()

# Yearly
plt.figure(figsize=(10, 6))
plt.plot(aep_data_cleaned.resample('Y')['AEP_MW'].mean())
plt.title('Yearly Average Energy Consumption')
plt.xlabel('Datetime')
plt.ylabel('AEP_MW')
plt.show()
```




Weekly Average Energy Consumption
AEP_MW
20000
19000
18000
17000
16000
15000
14000
13000
2006
2008
2010
2012
2014
2016
2018
Datetime

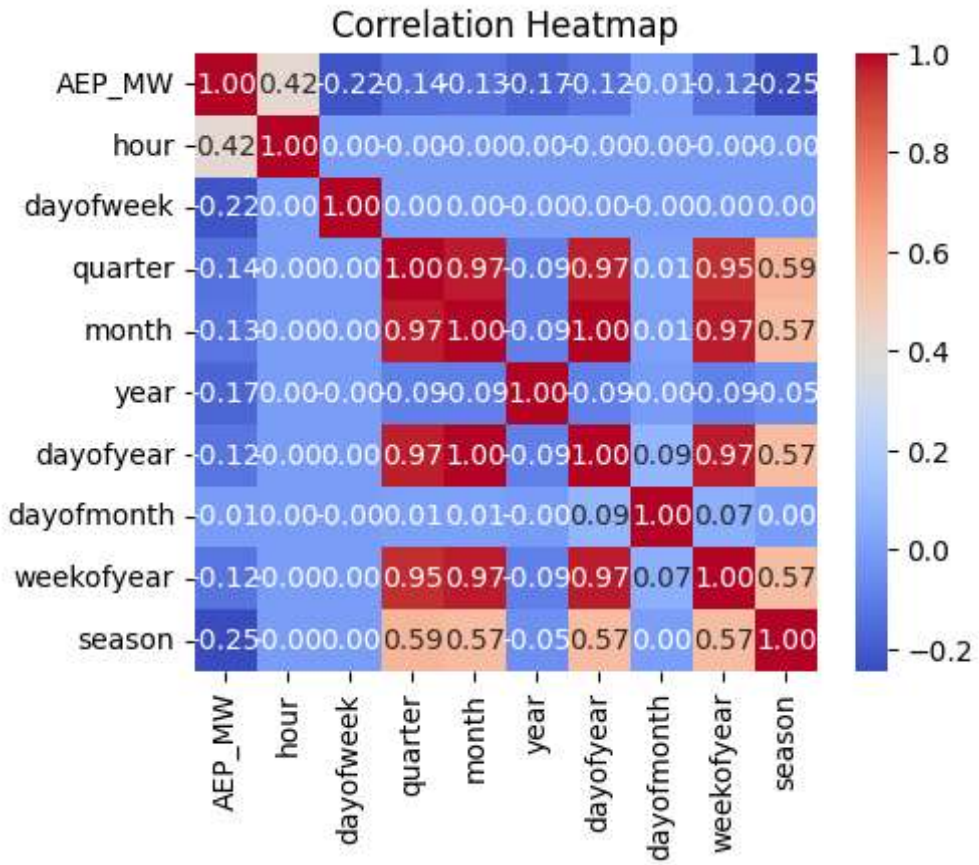

Monthly Average Energy Consumption
AEP_MW
19000
18000
17000
16000
15000
14000
13000
2006
2008
2010
2012
2014
2016
2018
Datetime

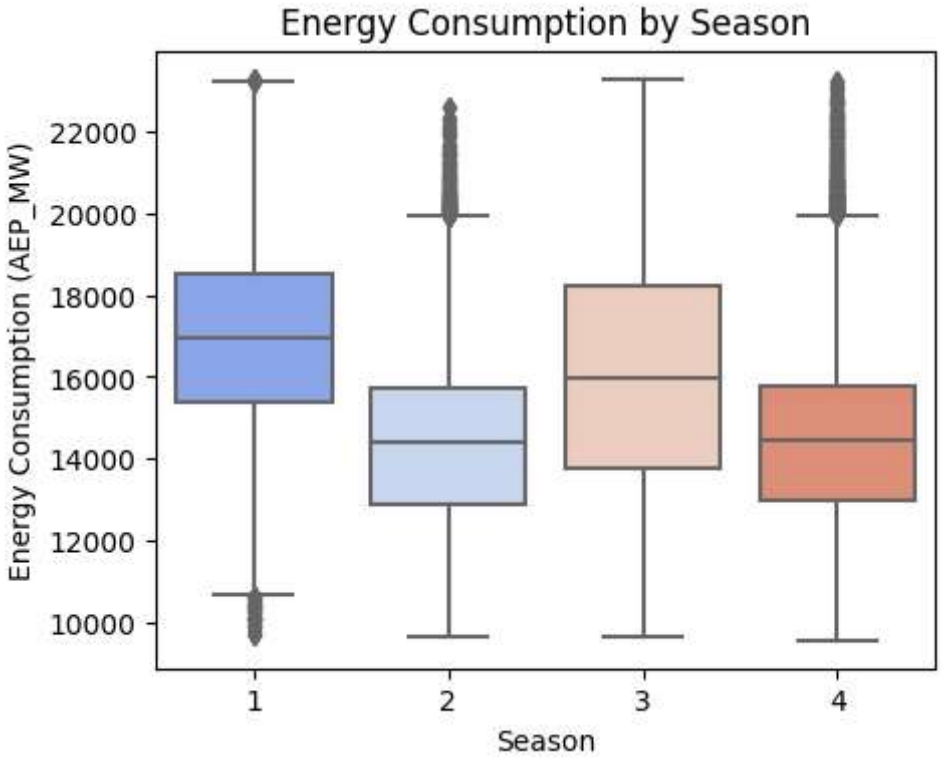


Visualizing the Features

In [ ]:

```
plt.figure(figsize=(5, 4))
corr_matrix = aep_data_cleaned.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap')
plt.show()

print("\n\n")

plt.figure(figsize=(5, 4))
sns.boxplot(x='season', y='AEP_MW', data=aep_data_cleaned, palette='coolwarm')
plt.title('Energy Consumption by Season')
plt.xlabel('Season')
plt.ylabel('Energy Consumption (AEP_MW)')
plt.show()
```





Note : The seasons follow this numerical representation

1. Spring
2. Summer
3. Autumn
4. Winter

Data Preprocessing for Time Series Analysis: Normalization and Time Series Splitting

Data Loading and Preprocessing for Time Series Analysis

- Here we load the cleaned and preprocessed DataFrame from a CSV file. The 'Datetime' column is converted to datetime format using the pd.to_datetime() function. The 'Datetime' column is set as the index of the DataFrame using the set_index() function. The DataFrame is sorted by 'Datetime' using the sort_index() function

In [ ]:
```
 # Load cleaned CSV file
aep_data_cleaned = pd.read_csv('AEP_hourly_cleaned.csv')

# Convert 'Datetime' to datetime format if necessary
aep_data_cleaned['Datetime'] = pd.to_datetime(aep_data_cleaned['Datetime'])

# Set 'Datetime' as the index
aep_data_cleaned.set_index('Datetime', inplace=True)

# Sort the DataFrame by 'Datetime'
aep_data_cleaned.sort_index(inplace=True)

print("Data Processed Successfully")
```

```
Data Processed Successfully
```

Data Normalization

The data is then normalized using the MinMaxScaler from sklearn.preprocessing. This scales the data to a range between 0 and 1. This is important for many machine learning algorithms that perform time series forecasting.

In [ ]:
```
# Initialize a scaler
scaler = MinMaxScaler()

# Fit and transform the data
aep_data_cleaned = pd.DataFrame(scaler.fit_transform(aep_data_cleaned), columns=aep
print("The data has been Normalized successfully")
print(aep_data_cleaned.head())
```

```
The data has been Normalized successfully
     AEP_MW      hour  dayofweek  quarter     month  year  dayofyear  \
0  0.204383  0.043478   0.666667      1.0  0.818182   0.0   0.750685
1  0.171950  0.086957   0.666667      1.0  0.818182   0.0   0.750685
2  0.154200  0.130435   0.666667      1.0  0.818182   0.0   0.750685
3  0.147261  0.173913   0.666667      1.0  0.818182   0.0   0.750685
4  0.153397  0.217391   0.666667      1.0  0.818182   0.0   0.750685

   dayofmonth  weekofyear  season
0         0.0        0.75     1.0
1         0.0        0.75     1.0
2         0.0        0.75     1.0
3         0.0        0.75     1.0
4         0.0        0.75     1.0
```

Time Series Split and Sequencing

- In this section, we set up a time series split for our data. We define a sequence length of 24, which means that our model will consider 24 time steps in the past to predict the next time step. We also initialize a TimeSeriesSplit object with 5 splits.
- We then separate our target variable 'AEP_MW' from the rest of the dataset. The target variable is what we aim to predict, and the rest of the dataset is our features or predictors.
- Next, we iterate through the splits, separating our data into training and testing sets. We then reshape our data according to the sequence length we defined earlier. This is done by sliding a window of length sequence_length over our data and appending the result to a list. We then convert these lists to numpy arrays for easier manipulation later on.

In [ ]:
```
# Define the sequence length
sequence_length = 24  # You can adjust this based on your needs (e.g., 24 for daily
n_splits = 5
# Initialize the TimeSeriesSplit
tscv = TimeSeriesSplit(n_splits=n_splits)

y = aep_data_cleaned['AEP_MW'].values
X = aep_data_cleaned.drop('AEP_MW', axis=1).values

# Initialize lists to store training and testing data
X_train_list = []
X_test_list = []
y_train_list = []
y_test_list = []

# Iterate through the splits
for train_index, test_index in tscv.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Reshape the data with the defined sequence length
    for i in range(0, len(X_train) - sequence_length + 1):
        X_train_list.append(X_train[i:i+sequence_length])
        y_train_list.append(y_train[i+sequence_length-1])
    for i in range(0, len(X_test) - sequence_length + 1):
        X_test_list.append(X_test[i:i+sequence_length])
        y_test_list.append(y_test[i+sequence_length-1])

# Convert lists to numpy arrays
X_train = np.array(X_train_list)
X_test = np.array(X_test_list)
y_train = np.array(y_train_list)
y_test = np.array(y_test_list)
print("Data Has been Succesfully Split and Timesteps Have been Added")
```

```
Data Has been Succesfully Split and Timesteps Have been Added
```

Shape checking

In [ ]:
```
print("Shape of X_train:", X_train.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_test:", y_test.shape)
```

```
Shape of X_train: (302420, 24, 9)
Shape of y_train: (302420,)
Shape of X_test: (100730, 24, 9)
Shape of y_test: (100730,)
```

LSTM Model Architecture and Training for Time Series Forecasting

- In this section, we define and train our LSTM model. The model is a sequential model, meaning that the layers are stacked on top of each other. The model consists of several bidirectional LSTM layers, each followed by a dropout layer for regularization. The final layer is a dense layer with a single unit, which will output our predicted value.
- We compile our model with the Adam optimizer and a learning rate of 0.001. We use mean squared error as our loss function since this is a regression problem.
- We then fit our model to our training data. We use a batch size of 64 and run for 25 epochs. We also specify a validation split of 0.2, meaning that 20% of our training data will be used for validation. We also use several callbacks, including ModelCheckpoint to save the best model, ReduceLROnPlateau to reduce the learning rate when the validation loss stops improving, and EarlyStopping to stop training early if the validation loss stops improving for a certain number of epochs.

In [ ]:
```
 # Initialize the model
model = Sequential()

# Add the first Bidirectional LSTM layer
model.add(Bidirectional(LSTM(units=64, return_sequences=True, input_shape=(X_train.
model.add(Dropout(0.2))  # Dropout for regularization

# Add 4 more Bidirectional LSTM layers using a for loop
for i in range(4):
    model.add(Bidirectional(LSTM(units=64, return_sequences=True)))
    model.add(Dropout(0.2))  # Dropout for regularization

# Add the output layer
model.add(Bidirectional(LSTM(units=64, return_sequences=False)))
model.add(Dropout(0.2))
model.add(Dense(units=1))

# Compile the model
optimizer = Adam(learning_rate=0.001)
model.compile(optimizer=optimizer, loss='mean_squared_error')
early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=Tr

callbacks = [
     ModelCheckpoint("/content/drive/MyDrive/NM Proj/save.h5", verbose=1, save_best
     ReduceLROnPlateau(monitor="val_loss", patience=3, factor=0.1, verbose=1, min_l
     EarlyStopping(monitor="val_loss", patience=10, verbose=1)
]
# Train the model
history = model.fit(
    X_train,
    y_train,
    epochs=25,
    batch_size=64,
```

```
    validation_split=0.2,
    callbacks=callbacks

)
model.summary()
print("Training has been Completed")
```

```
Epoch 1/25
3779/3781 [============================>.] - ETA: 0s - loss: 0.0115
Epoch 1: val_loss improved from inf to 0.01101, saving model to /content/drive/MyD
rive/NM Proj/save.h5
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3079: UserWar
ning: You are saving your model as an HDF5 file via `model.save()`. This file form
at is considered legacy. We recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')`.
  saving_api.save_model(
```

```
3781/3781 [==============================] - 141s 29ms/step - loss: 0.0115 - val_l
oss: 0.0110 - lr: 0.0010
Epoch 2/25
3781/3781 [==============================] - ETA: 0s - loss: 0.0071
Epoch 2: val_loss improved from 0.01101 to 0.00933, saving model to /content/driv
e/MyDrive/NM Proj/save.h5
3781/3781 [==============================] - 105s 28ms/step - loss: 0.0071 - val_l
oss: 0.0093 - lr: 0.0010
Epoch 3/25
3780/3781 [============================>.] - ETA: 0s - loss: 0.0046
Epoch 3: val_loss improved from 0.00933 to 0.00837, saving model to /content/driv
e/MyDrive/NM Proj/save.h5
3781/3781 [==============================] - 106s 28ms/step - loss: 0.0046 - val_l
oss: 0.0084 - lr: 0.0010
Epoch 4/25
3781/3781 [==============================] - ETA: 0s - loss: 0.0032
Epoch 4: val_loss did not improve from 0.00837
3781/3781 [==============================] - 105s 28ms/step - loss: 0.0032 - val_l
oss: 0.0096 - lr: 0.0010
Epoch 5/25
3779/3781 [============================>.] - ETA: 0s - loss: 0.0023
Epoch 5: val_loss improved from 0.00837 to 0.00754, saving model to /content/driv
e/MyDrive/NM Proj/save.h5
3781/3781 [==============================] - 105s 28ms/step - loss: 0.0023 - val_l
oss: 0.0075 - lr: 0.0010
Epoch 6/25
3779/3781 [============================>.] - ETA: 0s - loss: 0.0018
Epoch 6: val_loss did not improve from 0.00754
3781/3781 [==============================] - 104s 28ms/step - loss: 0.0018 - val_l
oss: 0.0081 - lr: 0.0010
Epoch 7/25
3780/3781 [============================>.] - ETA: 0s - loss: 0.0015
Epoch 7: val_loss did not improve from 0.00754
3781/3781 [==============================] - 105s 28ms/step - loss: 0.0015 - val_l
oss: 0.0080 - lr: 0.0010
Epoch 8/25
3780/3781 [============================>.] - ETA: 0s - loss: 0.0012
Epoch 8: val_loss improved from 0.00754 to 0.00727, saving model to /content/driv
e/MyDrive/NM Proj/save.h5
3781/3781 [==============================] - 104s 28ms/step - loss: 0.0012 - val_l
oss: 0.0073 - lr: 0.0010
Epoch 9/25
3779/3781 [============================>.] - ETA: 0s - loss: 0.0010
Epoch 9: val_loss improved from 0.00727 to 0.00717, saving model to /content/driv
e/MyDrive/NM Proj/save.h5
3781/3781 [==============================] - 105s 28ms/step - loss: 0.0010 - val_l
oss: 0.0072 - lr: 0.0010
Epoch 10/25
3781/3781 [==============================] - ETA: 0s - loss: 9.5618e-04
Epoch 10: val_loss improved from 0.00717 to 0.00693, saving model to /content/driv
e/MyDrive/NM Proj/save.h5
3781/3781 [==============================] - 104s 28ms/step - loss: 9.5618e-04 - v
al_loss: 0.0069 - lr: 0.0010
Epoch 11/25
3780/3781 [============================>.] - ETA: 0s - loss: 0.0010
Epoch 11: val_loss improved from 0.00693 to 0.00692, saving model to /content/driv
e/MyDrive/NM Proj/save.h5
3781/3781 [==============================] - 105s 28ms/step - loss: 0.0010 - val_l
oss: 0.0069 - lr: 0.0010
Epoch 12/25
3780/3781 [============================>.] - ETA: 0s - loss: 8.1375e-04
Epoch 12: val_loss did not improve from 0.00692
```

```
3781/3781 [==============================] - 105s 28ms/step - loss: 8.1379e-04 - v
al_loss: 0.0070 - lr: 0.0010
Epoch 13/25
3781/3781 [==============================] - ETA: 0s - loss: 7.0266e-04
Epoch 13: val_loss did not improve from 0.00692

Epoch 13: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
3781/3781 [==============================] - 104s 28ms/step - loss: 7.0266e-04 - v
al_loss: 0.0073 - lr: 0.0010
Epoch 14/25
3780/3781 [============================>.] - ETA: 0s - loss: 4.0498e-04
Epoch 14: val_loss improved from 0.00692 to 0.00684, saving model to /content/driv
e/MyDrive/NM Proj/save.h5
3781/3781 [==============================] - 104s 28ms/step - loss: 4.0498e-04 - v
al_loss: 0.0068 - lr: 1.0000e-04
Epoch 15/25
3781/3781 [==============================] - ETA: 0s - loss: 3.6871e-04
Epoch 15: val_loss improved from 0.00684 to 0.00672, saving model to /content/driv
e/MyDrive/NM Proj/save.h5
3781/3781 [==============================] - 106s 28ms/step - loss: 3.6871e-04 - v
al_loss: 0.0067 - lr: 1.0000e-04
Epoch 16/25
3779/3781 [============================>.] - ETA: 0s - loss: 3.5199e-04
Epoch 16: val_loss did not improve from 0.00672
3781/3781 [==============================] - 106s 28ms/step - loss: 3.5198e-04 - v
al_loss: 0.0069 - lr: 1.0000e-04
Epoch 17/25
3779/3781 [============================>.] - ETA: 0s - loss: 3.4300e-04
Epoch 17: val_loss improved from 0.00672 to 0.00669, saving model to /content/driv
e/MyDrive/NM Proj/save.h5
3781/3781 [==============================] - 106s 28ms/step - loss: 3.4299e-04 - v
al_loss: 0.0067 - lr: 1.0000e-04
Epoch 18/25
3780/3781 [============================>.] - ETA: 0s - loss: 3.3422e-04
Epoch 18: val_loss did not improve from 0.00669

Epoch 18: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
3781/3781 [==============================] - 106s 28ms/step - loss: 3.3423e-04 - v
al_loss: 0.0068 - lr: 1.0000e-04
Epoch 19/25
3780/3781 [============================>.] - ETA: 0s - loss: 3.1875e-04
Epoch 19: val_loss did not improve from 0.00669
3781/3781 [==============================] - 105s 28ms/step - loss: 3.1874e-04 - v
al_loss: 0.0067 - lr: 1.0000e-05
Epoch 20/25
3781/3781 [==============================] - ETA: 0s - loss: 3.1503e-04
Epoch 20: val_loss did not improve from 0.00669
3781/3781 [==============================] - 104s 27ms/step - loss: 3.1503e-04 - v
al_loss: 0.0067 - lr: 1.0000e-05
Epoch 21/25
3780/3781 [============================>.] - ETA: 0s - loss: 3.1094e-04
Epoch 21: val_loss did not improve from 0.00669

Epoch 21: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
3781/3781 [==============================] - 116s 31ms/step - loss: 3.1095e-04 - v
al_loss: 0.0067 - lr: 1.0000e-05
Epoch 22/25
3781/3781 [==============================] - ETA: 0s - loss: 3.0984e-04
Epoch 22: val_loss did not improve from 0.00669
3781/3781 [==============================] - 123s 33ms/step - loss: 3.0984e-04 - v
al_loss: 0.0067 - lr: 1.0000e-06
Epoch 23/25
3780/3781 [============================>.] - ETA: 0s - loss: 3.1006e-04
Epoch 23: val_loss did not improve from 0.00669
```

```
3781/3781 [==============================] - 135s 36ms/step - loss: 3.1005e-04 - v
al_loss: 0.0067 - lr: 1.0000e-06
Epoch 24/25
3780/3781 [============================>.] - ETA: 0s - loss: 3.0828e-04
Epoch 24: val_loss did not improve from 0.00669

Epoch 24: ReduceLROnPlateau reducing learning rate to 1.0000001111620805e-07.
3781/3781 [==============================] - 127s 33ms/step - loss: 3.0828e-04 - v
al_loss: 0.0067 - lr: 1.0000e-06
Epoch 25/25
3780/3781 [============================>.] - ETA: 0s - loss: 3.0958e-04
Epoch 25: val_loss did not improve from 0.00669
3781/3781 [==============================] - 130s 34ms/step - loss: 3.0958e-04 - v
al_loss: 0.0067 - lr: 1.0000e-07
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 bidirectional (Bidirection  (None, 24, 128)           37888
 al)

 dropout (Dropout)           (None, 24, 128)           0

 bidirectional_1 (Bidirecti  (None, 24, 128)           98816
 onal)

 dropout_1 (Dropout)         (None, 24, 128)           0

 bidirectional_2 (Bidirecti  (None, 24, 128)           98816
 onal)

 dropout_2 (Dropout)         (None, 24, 128)           0

 bidirectional_3 (Bidirecti  (None, 24, 128)           98816
 onal)

 dropout_3 (Dropout)         (None, 24, 128)           0

 bidirectional_4 (Bidirecti  (None, 24, 128)           98816
 onal)

 dropout_4 (Dropout)         (None, 24, 128)           0

 bidirectional_5 (Bidirecti  (None, 128)               98816
 onal)

 dropout_5 (Dropout)         (None, 128)               0

 dense (Dense)               (None, 1)                 129

=================================================================
Total params: 532097 (2.03 MB)
Trainable params: 532097 (2.03 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Training has been Completed
```

LSTM Autoencoder for Anomaly Detection

- In this section, we create an LSTM autoencoder for anomaly detection. An autoencoder is a type of neural network that is trained to copy its input to its output. It can be used to learn a compressed representation of the input data, and is often used for anomaly detection.

- We define our autoencoder with an encoder part, consisting of two LSTM layers, and a decoder part, which mirrors the encoder. We compile the model with the Adam optimizer and mean squared error loss, and fit it to our training data.

In [ ]:
```
# Creating copy of X_train for the autoencoder model
X_train_copy = X_train.copy()

# LSTM Autoencoder
inputs = Input(shape=(X_train_copy.shape[1], X_train_copy.shape[2]))
encoded = LSTM(64, return_sequences=True)(inputs)
encoded = LSTM(32, return_sequences=False)(encoded)

decoded = RepeatVector(X_train_copy.shape[1])(encoded)
decoded = LSTM(32, return_sequences=True)(decoded)
decoded = LSTM(X_train_copy.shape[2], return_sequences=True)(decoded)

# Defining the autoencoder model
autoencoder = Model(inputs, decoded)

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

# Fit the model
autoencoder.fit(X_train_copy, X_train_copy, epochs=20, batch_size=64, validation_sp
```

```
Epoch 1/20
4253/4253 [==============================] - 69s 14ms/step - loss: 0.0161 - val_lo
ss: 0.0087
Epoch 2/20
4253/4253 [==============================] - 50s 12ms/step - loss: 0.0033 - val_lo
ss: 0.0023
Epoch 3/20
4253/4253 [==============================] - 54s 13ms/step - loss: 0.0014 - val_lo
ss: 0.0013
Epoch 4/20
4253/4253 [==============================] - 55s 13ms/step - loss: 0.0010 - val_lo
ss: 9.6572e-04
Epoch 5/20
4253/4253 [==============================] - 57s 13ms/step - loss: 7.2170e-04 - va
l_loss: 0.0012
Epoch 6/20
4253/4253 [==============================] - 49s 12ms/step - loss: 6.3570e-04 - va
l_loss: 7.2479e-04
Epoch 7/20
4253/4253 [==============================] - 53s 12ms/step - loss: 5.9795e-04 - va
l_loss: 7.9809e-04
Epoch 8/20
4253/4253 [==============================] - 47s 11ms/step - loss: 5.4643e-04 - va
l_loss: 5.7236e-04
Epoch 9/20
4253/4253 [==============================] - 47s 11ms/step - loss: 5.2450e-04 - va
l_loss: 6.7706e-04
Epoch 10/20
4253/4253 [==============================] - 46s 11ms/step - loss: 5.0631e-04 - va
l_loss: 5.5986e-04
Epoch 11/20
4253/4253 [==============================] - 52s 12ms/step - loss: 4.8061e-04 - va
l_loss: 5.5278e-04
Epoch 12/20
4253/4253 [==============================] - 50s 12ms/step - loss: 4.6974e-04 - va
l_loss: 5.0566e-04
Epoch 13/20
4253/4253 [==============================] - 51s 12ms/step - loss: 4.5210e-04 - va
l_loss: 4.7835e-04
Epoch 14/20
4253/4253 [==============================] - 52s 12ms/step - loss: 4.4419e-04 - va
l_loss: 4.7456e-04
Epoch 15/20
4253/4253 [==============================] - 45s 11ms/step - loss: 4.4969e-04 - va
l_loss: 6.4047e-04
Epoch 16/20
4253/4253 [==============================] - 52s 12ms/step - loss: 4.3703e-04 - va
l_loss: 4.6720e-04
Epoch 17/20
4253/4253 [==============================] - 47s 11ms/step - loss: 4.4270e-04 - va
l_loss: 0.0011
Epoch 18/20
4253/4253 [==============================] - 49s 12ms/step - loss: 4.2205e-04 - va
l_loss: 4.5380e-04
Epoch 19/20
4253/4253 [==============================] - 57s 13ms/step - loss: 4.2564e-04 - va
l_loss: 4.5538e-04
Epoch 20/20
4253/4253 [==============================] - 52s 12ms/step - loss: 4.1432e-04 - va
l_loss: 4.6015e-04
```

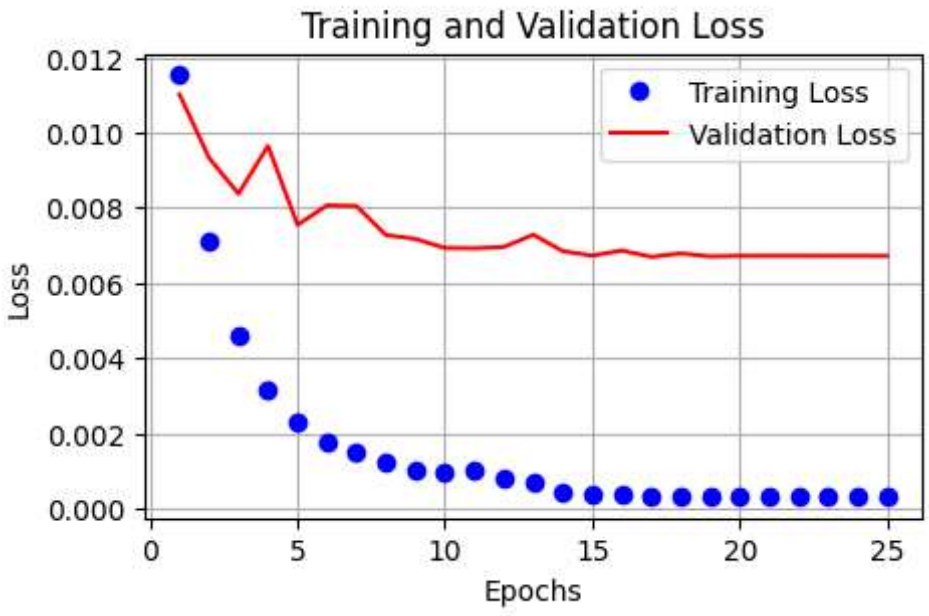
Out[ ]: <keras.src.callbacks.History at 0x7a8a27e1b220>

Model Evaluation and Performance Visualization for Time Series Forecasting

In this section, we evaluate the performance of our model. We plot the training and validation loss over each epoch to visualize how our model learned over time. We then make predictions on our testing data and calculate several metrics, including the mean absolute error, mean squared error, root mean squared error, and R-squared score. We plot the actual vs. predicted values to visually assess the performance of our model.

Training History

In [ ]:
```
# Get the training history
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)

# Plot the training and validation loss
plt.figure(figsize=(5, 3))
plt.plot(epochs, loss, 'bo', label='Training Loss')
plt.plot(epochs, val_loss, 'r', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

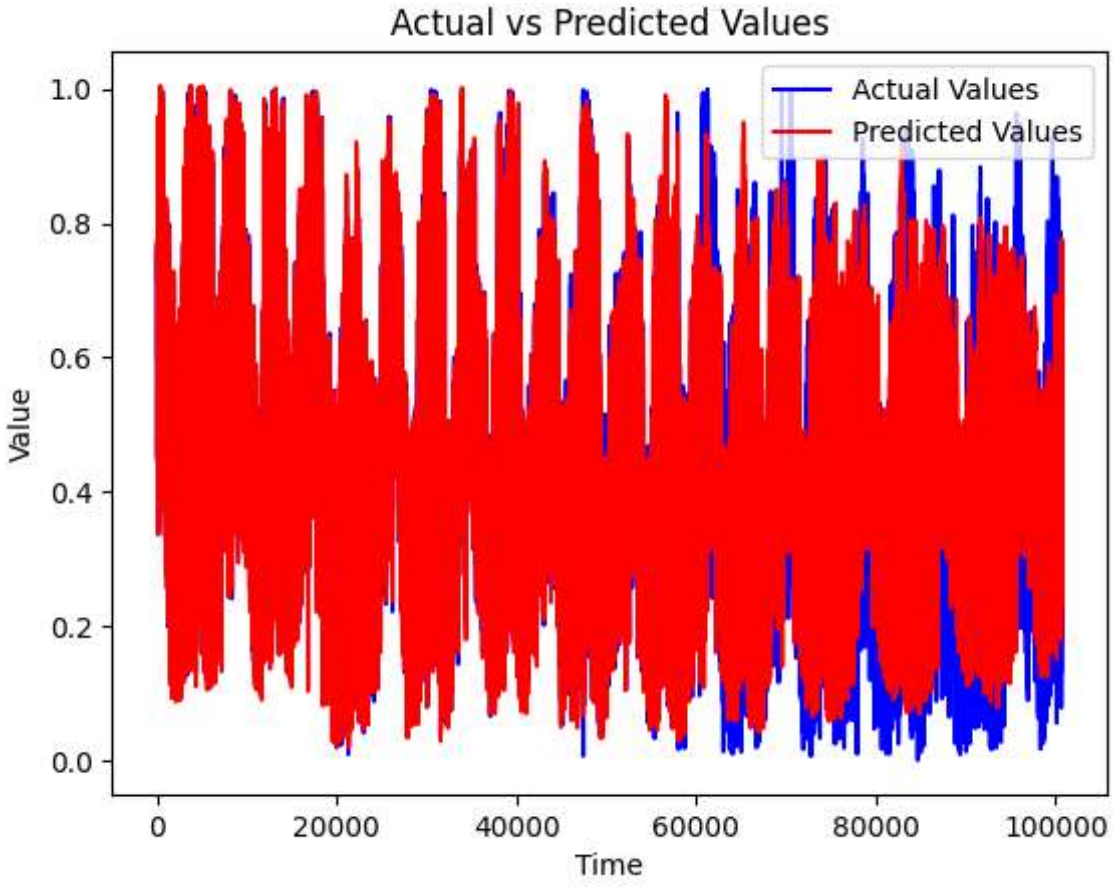


Evaluating Performance on the Testing Set

In [ ]:
```
# Prediction
y_pred = model.predict(X_test)

# Calculate the mean absolute error
mae = mean_absolute_error(y_test, y_pred)
print('Mean Absolute Error:', mae)

# Calculate the Mean Squared Error
```

```
mse = mean_squared_error(y_test, y_pred)
rmse = math.sqrt(mse)
print("Mean Squared Error (MSE):", mse)
print("Root Mean Squared Error (RMSE):", rmse)

# Calculate the R-squared score
r2 = r2_score(y_test, y_pred)
print('R-squared (R2) Score:', r2)

# Plotting Actual vs. Predicted Values
plt.plot(y_test, label='Actual Values', color='blue')
plt.plot(y_pred, label='Predicted Values', color='red')
plt.xlabel('Time')
plt.ylabel('Value')
plt.legend()
plt.title('Actual vs Predicted Values')
plt.show()
```

```
3148/3148 [==============================] - 36s 10ms/step
Mean Absolute Error: 0.051179707494769786
Mean Squared Error (MSE): 0.008290061756812605
Root Mean Squared Error (RMSE): 0.09104977625899256
R-squared (R2) Score: 0.7689952638701818
```



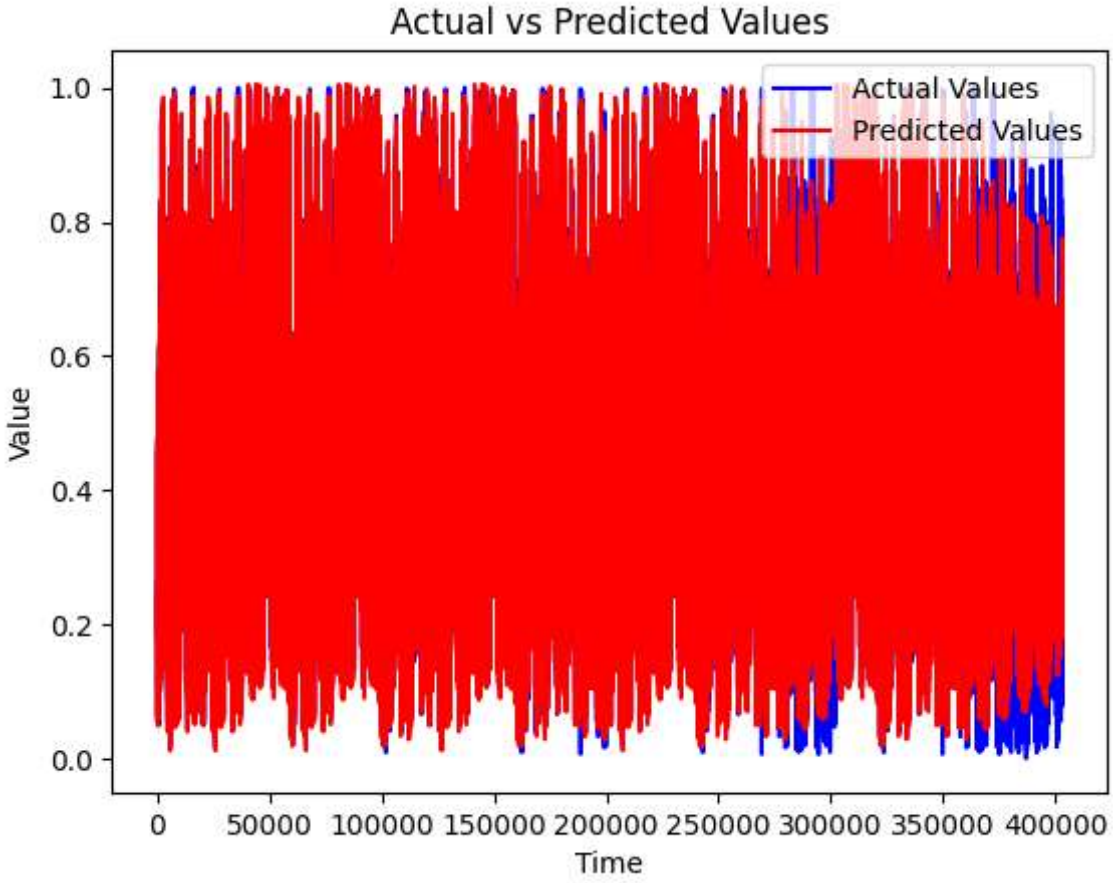
Evaluating Performance Over the Entire Dataset

In [ ]:

```
X = np.concatenate((X_train, X_test), axis=0)
y = np.concatenate((y_train, y_test), axis=0)

# Prediction
y_pred = model.predict(X)

# Calculate the mean absolute error
mae = mean_absolute_error(y, y_pred)
print('Mean Absolute Error:', mae)
```

```python
# Calculate the Mean Squared Error
mse = mean_squared_error(y, y_pred)
rmse = math.sqrt(mse)
print("Mean Squared Error (MSE):", mse)
print("Root Mean Squared Error (RMSE):", rmse)

# Calculate the R-squared score
r2 = r2_score(y, y_pred)
print('R-squared (R2) Score:', r2)

# Plotting Actual vs. Predicted Values
plt.plot(y, label='Actual Values', color='blue')
plt.plot(y_pred, label='Predicted Values', color='red')
plt.xlabel('Time')
plt.ylabel('Value')
plt.legend()
plt.title('Actual vs Predicted Values')
plt.show()
```

```
12599/12599 [==============================] - 138s 11ms/step
Mean Absolute Error: 0.02405409769231152
Mean Squared Error (MSE): 0.003144156021178198
Root Mean Squared Error (RMSE): 0.05607277433102627
R-squared (R2) Score: 0.9100920272976961
```



Anomaly Detection

- Finally, we used our trained autoencoder for anomaly detection. We made predictions on our testing data and calculated the mean squared error between the actual and predicted values. We then defined a threshold for anomalies as the 99.9th percentile of the mean squared error. Any data point with a mean squared error greater than this

threshold was considered an anomaly. We printed the indices of these anomalous data points.

In [ ]:
```
# Using the autoencoder to make predictions on the test data
X_test_copy = X_test.copy()
X_test_pred = autoencoder.predict(X_test_copy)

# Calculating the mean squared error (MSE) between the actual and predicted values
mse = np.mean(np.power(X_test - X_test_pred, 2), axis=(1, 2))

# Defining a threshold for anomaly detection
threshold = np.quantile(mse, 0.999)

# Detecting anomalies in the test data
anomalies = mse > threshold

# Printing the indices of anomalous data points
anomalous_indices = np.where(anomalies)[0]
print("Anomalous Data Points (Indices):", anomalous_indices)
```

```
3148/3148 [==============================] - 19s 5ms/step
Anomalous Data Points (Indices): [   356    357    358    359   3512   4008   4009
4010   4555   4556
   4573   4574   4575   4593   4660   4661   4705   4706   4707   4708
   4709   4726   4866   5025   5026   5027   5042   5043   5044   5045
   5154   5155   5156   8100   8719   8720  11977  11978  11979  16818
  17264  17265  17266  17267  17268  25679  30499  30500  34406  39178
  39179  39195  39196  39197  39198  39212  39213  39214  39353  43095
  51826  60537  61151  61152  61267  61268  69254  70538  70539  70540
  78145  86858  95594  95762  95930  96098  96266  96434  96602  96770
  96938  97106  97273  97441  97609  97777  97945  98113  98281  98449
  98617  98785  98953  99121  99289  99457  99625  99793  99961 100129
 100633]
```