Long Short-Term Memory Networks for Weather Prediction: Case Study on Delhi's Temperature

1. *Introduction*

Recurrent Neural Networks and LSTMs (a type of RNN) are now commonly used to observe temporal data. They are helpful for data analysis applications where one of the main objectives is to analyze patterns over time, for instance stock analysis or observing weather patterns. Using these methods, "historical" observations are stored and included in analyses of later events. Language modeling, machine translation, handwriting recognition, and question-answering are a few of the machine learning applications of LSTM that are most frequently used. In this paper, we will be describing the most basic cell types and structures for RNN and LSTM. Then, the described methods will be used to analyze real data, with the primary purpose being the of prediction of weather data such as mean temperature in Delhi, India. There are many different types of LSTM that are also being applied these days and we will describe a couple of those as well.

One advantage of LSTM over a simple RNN is its capacity to learn longer-term dependencies. In simple RNNs, as more layers with specific activation functions are added to neural networks, a common issue arises where the gradients of the loss function tend to approach zero. This phenomenon can make it challenging to train the network effectively. In contrast, LSTM solves this vanishing gradient problem, which can cause the network weights to either become very small or very large in RNNs.

LSTMs have 2 important components in them. One is the element of the network and the other is the structure of the network. The elements of an LSTM network include the cell state, which stores long-term memory, and the hidden state, which serves as the network's short-term or working memory and enables it to store and make use of knowledge from earlier processing steps while handling sequential data. An LSTM network consists of LSTM units arranged in a recurrent manner, allowing them to handle input sequences and track relationships over time. For efficient data processing, these units have input, forget, and output gates that can help them manage the information flow and allow them to update their internal states. These gates enable the network to effectively capture and use information from previous steps when processing sequential data. Although LSTM networks have advantages such as enhanced long-term dependency learning and robustness to time gaps, they also require additional training data and can encounter challenges with data that is not linear or that is very noisy. Nonetheless, ongoing research continues to advance the architectures of LSTM, which is being done to expand its applications in other fields such as finance and traffic prediction.

In this project, we will be using LSTM to try and predict weather conditions as accurately as possible. Specifically, we will be trying to predict the average temperature of a certain day in a certain city, Delhi, India, given the average temperature from a certain number of days before as well as the atmospheric conditions of those days. We will try to make a model that is as efficient and as accurate as possible. The ARIMA model, which is a popular choice for time series analysis in statistics, will also be compared to the predictive outcomes of LSTM.

The LSTM prediction technique used in this weather study can be extended to other regions. This increases the model's applicability to a wider range of geographical areas, enabling the comparison of temperature trends and an in-depth comprehension of climatic patterns. Therefore, these insights can be used by researchers and policymakers to create climate adaptation plans.

1.1. **Background.**

Within the field of artificial neural networks, recurrent neural networks (RNNs) are widely utilized for analyzing time series data. One crucial advantage of RNNs is their ability to store and utilize information from previous time steps within the hidden layer's memory. This memory serves as additional inputs for subsequent moments in the sequence. In practice, the memory is initialized with initial values and updated at each time step with the output of the hidden layer, as seen in the Elman Network. Similarly, the output layer's output can also be stored in memory, as demonstrated by the Jordan Network. By leveraging this memory mechanism, RNNs effectively capture and exploit the temporal dependencies inherent in time series data, making them a valuable tool in time series analysis.

Despite the potential of generic RNNs, they encounter two significant challenges. Firstly, these networks have limited memory capacity and struggle to retain information from earlier steps in the sequence, rendering them inadequate for handling longer sequences of data (Siami-Namini and Namin, 2018). Secondly, the issue of vanishing gradients consistently affects the training process, impacting the final outcome. Therefore, long short-term memory (LSTM) as a kind of Recurrent Neural Network with the capability of memorizing longer steps in the sequence is used in order to predict time series and reduce update times.

The LSTM model consists of specialized neurons, each comprising a memory cell with four inputs and one output. In addition to receiving input from other parts of the network and transmitting output to other parts of the network, these neurons incorporate three distinct gates. The input gate, characterized by an activation function f, mimics the behavior of an opening and closing gate to regulate the input flow. Which is usually a sigmoid function. The output gate contains an activation function f to mimic the opening and closing gate to control the output. It is usually a sigmoid function. Forget gate possesses an activation function f to mimic the open and close gate to decide whether to remember. It is usually a sigmoid function as well (Hochreiter and Schmidhuber, 1997).

1.2. **Data Description.**

The data that we will be analyzing contains information about the weather conditions in the city of Delhi, India from January 2013 to April 2017. There are 1537 total observations and 5 variables, including the date of the observation, the mean temperature (in degrees Celsius), the humidity, the wind speed, and the mean atmospheric pressure. The data was

downloaded from Kaggle, a website that contains numerous open-source data sets from a variety of sources. There was no missing data that needed to be dealt with.

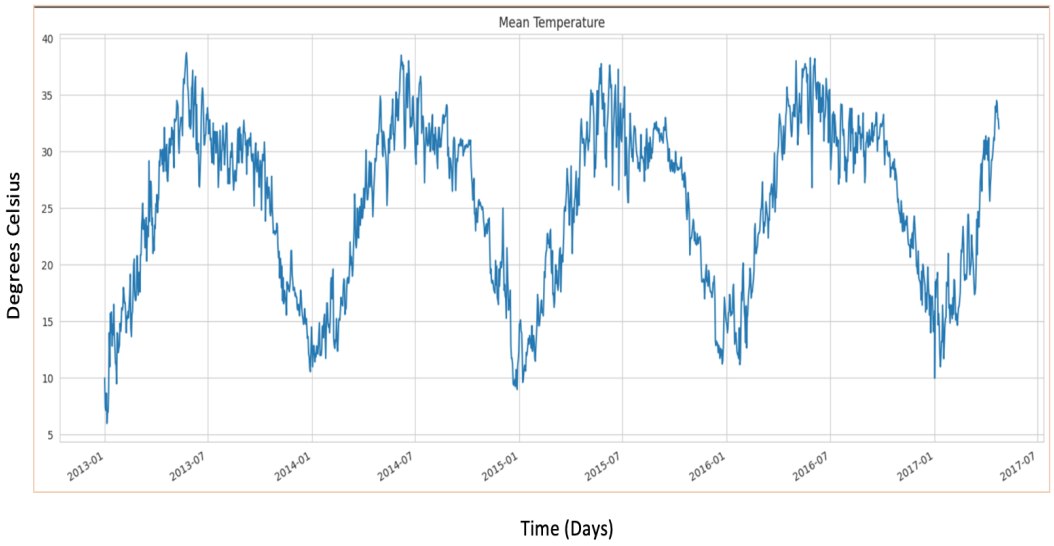


FIGURE 1. Distribution of Mean Temperature

Figure 1 shows the distribution of the variable of interest: Mean Temperature. A few initial observations that can be made are that, while there appears to be a clear overall trend, the curve itself is not completely smooth.

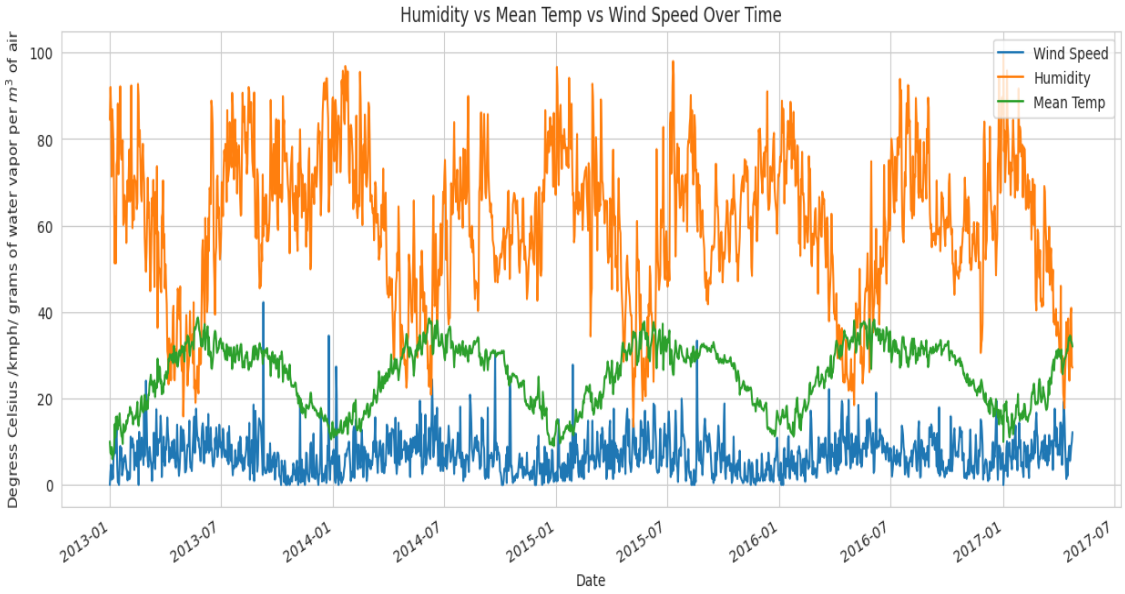


FIGURE 2. Distribution of Mean Temperature compared to Wind Speed and Humidity

Figure 2 shows the distribution of the variable of interest: Mean Temperature compared to Humidity and Wind Speed. It is clear that all 3 variables follow a cyclical pattern.

The curves are not completely smooth but do show an obvious trend. For instance, when the mean temperature drops, humidity increases, and wind speed tends to decrease. The humidity, wind speed, and mean temperature will be used as variables in a multivariate LSTM model and compared to the univariate LSTM model to determine which one results in better predictions.

2. *Methodology*

2.1. **Algorithm and Model Used.**

In this project, we used a stacked LSTM with 3 layers. Using multiple layers in an LSTM network allows for a hierarchical representation of the input sequence, where lower layers capture short-term dependencies and higher layers capture longer-term dependencies. This helps the network learn more complex patterns and capture dependencies at different scales, leading to improved performance in modeling sequential data. In the LSTM model, the activation function used was a tanh activation function and the recurrent activation function was the sigmoid function. Using the tanh activation function in the LSTM's hidden state allows for better gradient flow during training, while the sigmoid activation function as the recurrent activation helps control the flow of information through the cell state. Additionally, the optimizer used was Adam. It is an expanded form of Stochastic Gradient Descent which allows weights to be assigned to each level and be optimized as the model is trained. Finally, mini-batch processing was used since it increases training speed due to multiple batches being trained at once. Also, it adds regularization and stability by adding noise into the model due to the randomized effect of the batch processing and balancing that noise as well. As described earlier, LSTM models can take on a variety of structures and components. A few of the structures that we will be used in this analysis are described below.

2.2. **Standard Recurrent Cell.**

Recurrent Neural Networks (RNNs) are a type of neural network that are designed to process sequential data by utilizing a series of interconnected cells. These cells are specifically designed to handle information in a way that enables the detection of temporal relationships and sequential patterns. The core components of recurrent neural networks are cells that incorporate activation functions, commonly the sigmoid or tanh function. They consist of interconnected cells arranged in a sequential manner, allowing them to iterate over the data at each "time" point within the sequence. They perform the designated task consistently, ensuring continuity throughout the network. By incorporating backpropagation, they leverage previous information to influence the current iteration's inputs and outputs. This enables them to retain a form of "memory" and effectively capture temporal dependencies within the data. These enhancements to their basic structure allow for improved information processing and modeling of sequential patterns.

The mathematical expressions of the standard recurrent sigma cell are written as follows: $h_t = \sigma(W_h h_{t-1} + W_x x_t + b)$, $y_t = h_t$;

x_t ht and y_t are the input, the recurrent information, and the output of the cell at time t. W_h and W_x are the weights and b is the bias.

2.3. **LSTM with a Forget Gate.**
In the LSTM cell with a forget gate, an additional gate is introduced to control the removal of information from the cell state. The mathematical equations for this model are:

(1) Forget gate: $f_t = \sigma(W_{fh}h_{t-1} + W_{fx}x_t + b_f)$

(2) Input gate: $i_t = \sigma(W_{ih}h_{t-1} + W_{ix}x_t + b_i)$

(3) Candidate cell state: $c_t = tanh(W_{ch}h_{t-1} + W_{cx}x_t + b_c)$

(4) Updated cell state:$c_t = f_t \cdot c_{t-1} + i_t \cdot c_t$

(5) Output gate: $o_t = \sigma(W_{oh}h_{t-1} + W_{ox}x_t + b_o)$

(6) Hidden state: $h_t = o_t \cdot tanh(c_t)$

The forget gate in the LSTM cell plays an important role in determining which information is discarded from the cell state. When the value of the forget gate, ft, is equal to 1, it means that all the information is retained. However, when the value is 0, it indicates that all the information is discarded. Research by Jozefowicz, Zaremba, and Sutskever in 2015 revealed that increasing the bias of the forget gate, bf, often leads to improved performance of the LSTM network. The "dropout" of the information is used to prevent overfitting of the model on the data, which greatly improves the model's overall flexibility. In addition, Schmidhuber, Wierstra, Gagliolo, and Gomez proposed in 2007 that LSTM networks can benefit from being trained using evolutionary algorithms in combination with other techniques, rather than relying only on traditional gradient descent methods. This highlights the potential of alternative training approaches to further enhance the performance of LSTM networks.

2.4. **Stacked LSTM.**
The technique of employing a stacked LSTM network is widely used to enhance the capacity and depth of LSTM networks. It involves stacking multiple LSTM layers on top of each other, making it a basic and straightforward structure for LSTM networks. The network can also be seen as a multilayered fully-connected structure.

N, the depth dimension of the stacked LSTM, the output of the LSTM layer that is the $(L-1)^{th}$ layer at time h_t $L-1$. The output of that layer becomes input X_{tL} in the next layer, i.e the L^{th} layer. The output-input connections are the primary relationships between two adjacent layers and allow information to flow between the layers. However, when it comes to the time dimension, the recurrent connections within one layer are constrained. The L^{th} LSTM layer could be expressed as follows:

In each cell, we are performing the same basic steps. First, the input data (x_t) is inputted and concatenated with the hidden state from the previous iteration. The inputs provided to the function associated with the "forget gate" determine the weights assigned to each observation, ranging from 1 to 0. Following that, the input gate comes into play, utilizing the same inputs but employing a distinct set of functions. In these steps, the "input gate" and "candidate cell state" functions are applied. The product of these steps is computed by adding the product of the output from the forget gate and the previous cell state to the product of the input and candidate cell state. This calculation generates the new cell state. In the final step, the new cell state undergoes a tanh transformation, and the resulting value

is multiplied by the output of the "output" gate function. This element-wise multiplication produces the new hidden state. Both the new hidden state and the new cell state are then utilized as inputs for the subsequent cell in the sequence.

3. ***Results***

The metric we will be using to compare the prediction accuracy of the models is the root mean squared error (RMSE). The RMSE is commonly used to compare the error rates of different predictive models. First, three models were fitted, all with varying numbers of time steps. A few of the parameters used in the Keras package, the package used to implement neural networks in Python, are time steps and epochs. Epochs are simply the number of times the algorithm will go through the data in an attempt to optimize the weights that are being fitted. Given an appropriate optimizer, the model should, theoretically, converge to an optimal result. Here, the optimal result is one that minimizes the loss. Each time the models are fitted, plots of the loss over each iteration will be made in order to confirm that this assumption is indeed true and that we are indeed reaching an optimal model at some point over the iterations. If not, then the number of epochs may have to be increased or an alternative optimizer would have to be used. For this analysis, each model was fitted using 50 epochs. Time steps are the number of "historical" observations that are used to predict the desired outcome of the next day. So, if we set time steps to 7, in our case, this would correspond to using the data from seven days before to predict the average temperature of the next day. The results of fitting the models with different numbers of time steps can be found below.

Number of Time Steps	RMSE
60	.42
14	.50
7	.15
3	.31

TABLE 1. RMSE values at different numbers of time steps

As shown in Table 1, the best prediction results were found when only 7 days worth of data was used. Intuitively, this result makes sense as we expect more recent observations to be more informative for predicting the next day's data. In less formal terms, in general, if has been extremely hot for the past week, you might also expect the next day's average temperature to be relatively high. However, as Table 1 also shows, using too few data points to make predictions makes the predictive power of the model worse. When using too few data points, there isn't enough data to be able to accurately observe any trends present. Especially in the context of having a single observation per day, time_step = 3 is much too short.

Below, Figure 3 shows how the loss changes over the iterations. As expected, the loss decreases and then converges, indicating that the algorithm is reaching an optimal model. In the model with the best predictive power (timestep = 7), the lines converge within about 10 iterations. The same pattern is observed in the model using tiemstep = 3. The models using larger timesteps take a bit longer to converge. Figure 4 shows the distributions of the

actual and predicted data for the test set. The predictions are obtained from the model using timestep = 7 as that was the model with the best predictive power. As indicated by the low RMSE, the blue line (the distribution for the predictions) follows the orange line (the true distribution) very closely.

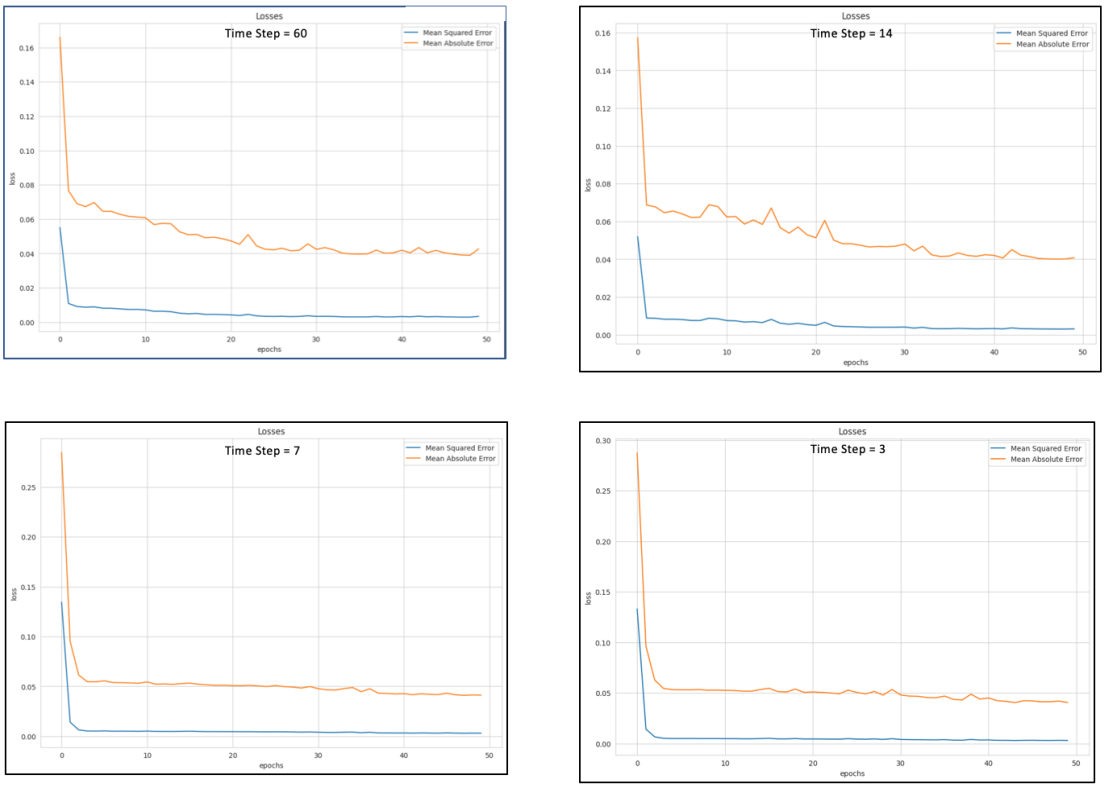


FIGURE 3. Loss over Epochs Uni-variate Example

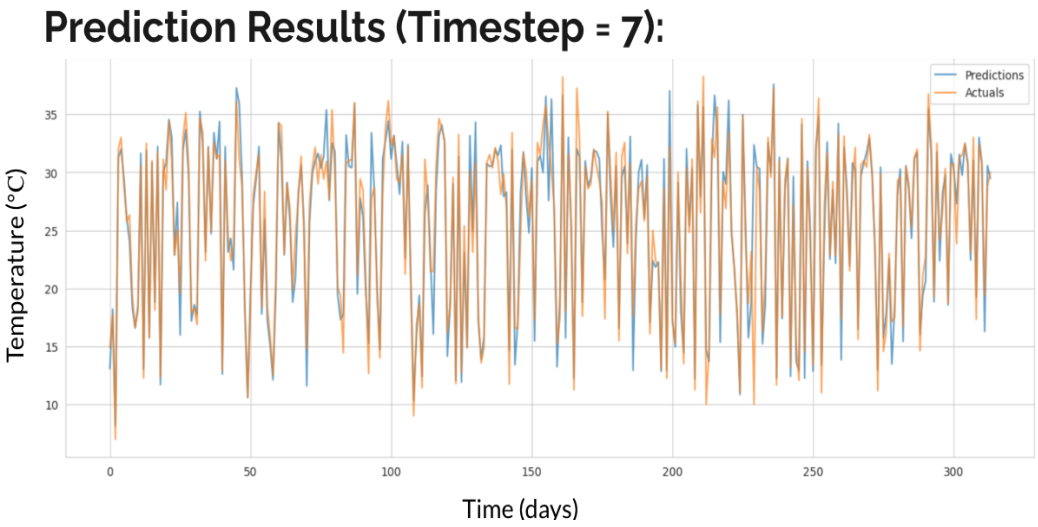


FIGURE 4. Distribution of Predicted vs Actual Values for the Test Set

3.1. **Model Comparisons.**

3.1.1. *Multivariate LSTM.*

Figures 4 and 5 show that the univariate and multivariate models have very similar predictive powers. The model incorporating wind speed, humidity, and mean temperature to predict mean temperature yielded slightly improved results compared to using only the patterns of mean temperature. This improvement was particularly evident when the optimal time step of 7 days was selected, as indicated by the smaller RMSE. The outcome is logical given that wind speed and humidity are important variables that can affect temperature variations.

However, when using a very small time step (3 days), the univariate prediction model demonstrated a much smaller RMSE compared to the multivariate prediction. This implies that when data availability is limited and only a few days' worths of data is accessible, relying on a univariate model produces more accurate predictions. The mean pressure was not included in this particular model, as it did not exhibit the same cyclic patterns observed in the other variables.

Additionally, Figure 6 demonstrates the loss's change through each iteration. The loss decreases and then converges as anticipated, showing that the algorithm has found an ideal model. The lines in the model with the highest predictive ability (timestep = 7) converge after roughly 10 iterations. The model with timestep = 3 exhibits the same pattern. Larger timestep models converge more slowly than smaller timestep models. This is similar to how the univariate model that had only mean temperature behaved.

Number of Time Steps	RMSE
60	.44
14	.21
7	.12
3	.79

TABLE 2. RMSE values at different numbers of time steps (MultiVariate)

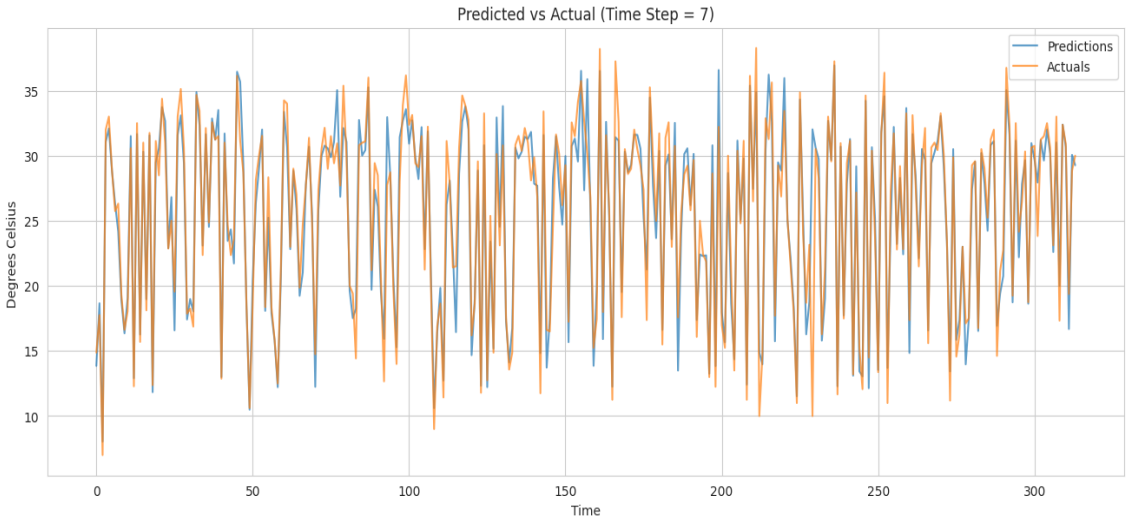


FIGURE 5. Accurate Vs Predicted - Multivariate Case

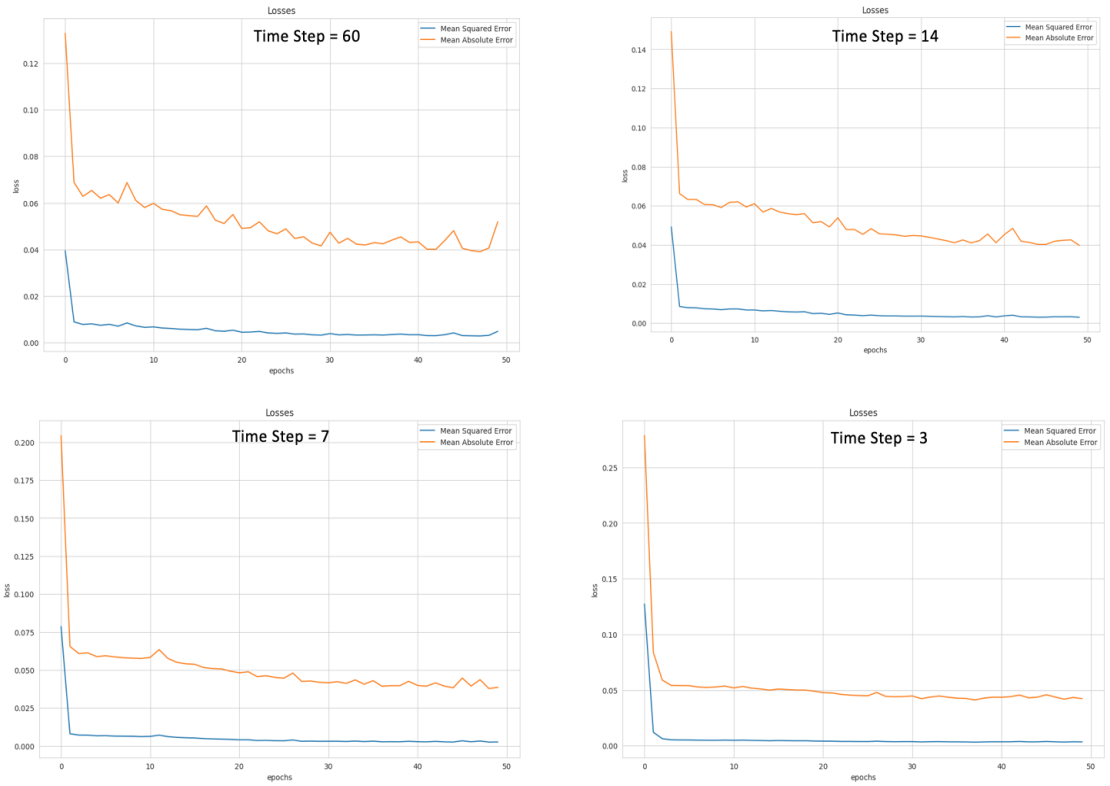


FIGURE 6. Loss over Epochs Multivariate Example

3.1.2. *Auto-regressive Integrated Moving Average (ARIMA).*

It is evident, as seen in Figure 7, that the forecasting ability of ARIMA is only accurate for a short period. However, after a trend change point, the ARIMA model fails to capture this change. This is because one of the disadvantages of ARIMA is its inability to effectively capture sudden shifts or structural changes in the underlying data. ARIMA models rely on the assumption of stationarity, which means that the statistical properties of the data remain constant over time. As we know, our original time series is not stationary. The presence of non-stationarity in the data makes it challenging for ARIMA to effectively capture and forecast the underlying trend. To overcome this limitation, alternative models such as machine learning algorithms like neural networks, specifically Long Short-Term Memory (LSTM), can be employed. These models are capable of capturing non-linear relationships and adapting to changing patterns in the data, making them suitable for forecasting tasks that involve abrupt changes or long-term trends.

While ARIMA models can be effective for short-term forecasting in relatively stable data environments, they are less suitable for capturing abrupt changes or long-term trends. This can especially be seen in something like predicting weather patterns. As seen in the figure below, the ARIMA model tended to underestimate the mean temperature. Employing LSTM or considering alternative models is recommended when dealing with data that exhibits significant shifts or structural changes.

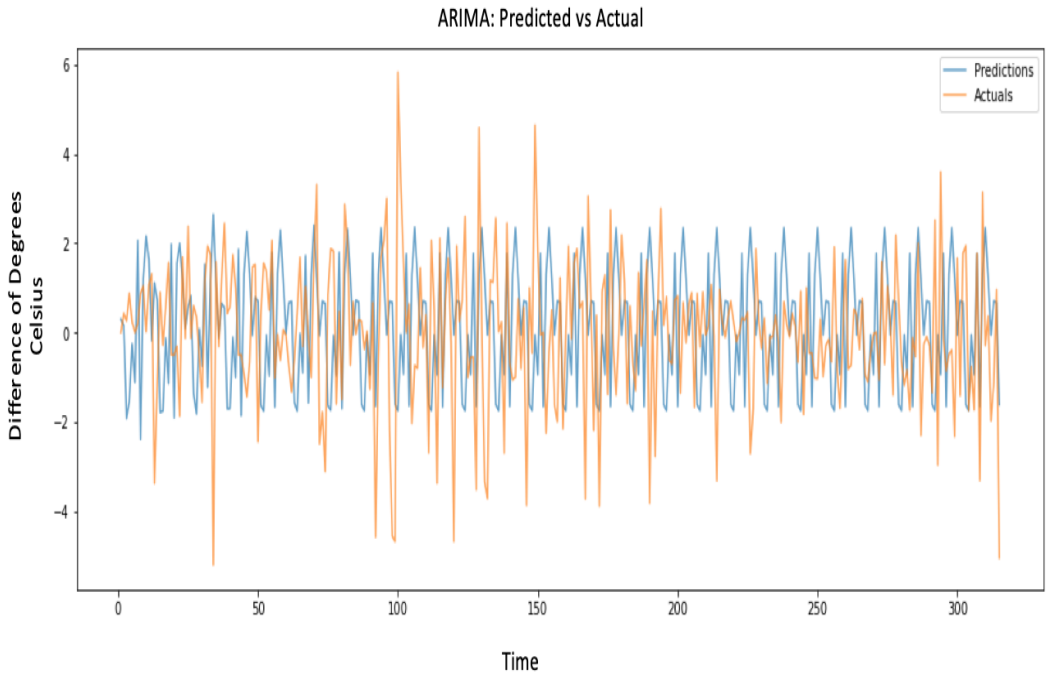


FIGURE 7. Accurate Vs Predicted - ARIMA Case

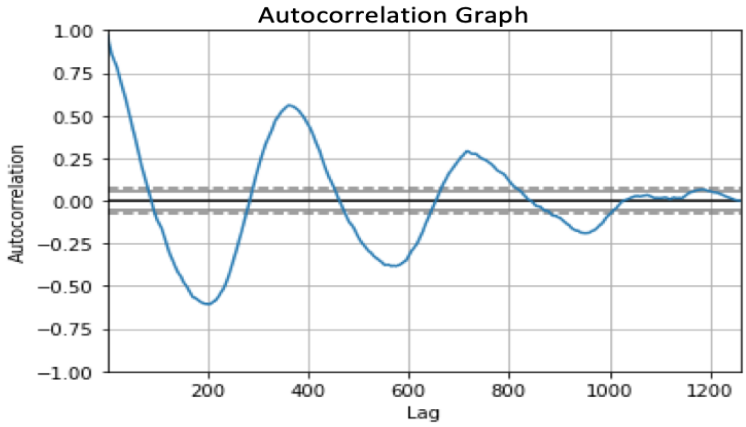


FIGURE 8. Auto-Correlation Graph

The damped sinusoidal pattern in the ACF suggests that an ARIMA model is suitable for predicting mean temperature. By determining the AR and MA orders, fitting the model, and evaluating its fit, predictions can be made.

4. *Conclusion&Discussion*

Predicting the mean temperature in Delhi has provided us with several key insights. Firstly, smaller time windows but not too small, particularly 7 days, have been shown to yield better prediction results. This is because recent observations hold more relevance when forecasting the temperature for the next day. Striking a balance is crucial, as using too few data points can diminish the predictive power of the model, hindering its ability to capture underlying patterns and trends accurately.

In the context of temperature prediction, ARIMA models have their limitations. While they are effective for short-term forecasting in stable data environments, they may struggle to capture abrupt changes or long-term trends. As a result, alternative models such as LSTM networks are recommended, especially when dealing with data that exhibits significant shifts or structural changes. This is especially clear when dealing with something like weather data such as mean temperature.

Nevertheless, one limitation of LSTM networks, in comparison to other methods, is their higher demand for training data to achieve effective learning. The presence of additional gates in LSTM introduces a greater number of parameters that need to be learned, thereby increasing the model's complexity. Consequently, training an LSTM on a substantial amount of data requires a significant investment of time and computational resources to accurately learn the parameters of the LSTM cells. Furthermore, not all types of data may be suitable for LSTMs. For example, they might have trouble with data that is highly nonlinear or has or is noisy. In such scenarios, alternative models or preprocessing techniques may be more suitable for extracting meaningful patterns from the data. It is important to note that LSTM is not well-suited for tasks such as prediction or classification when the input data does not follow a sequential format.

Another downfall of the LSTM model is that it is difficult to interpret the parameters of the model directly because there are so many. As such, we cannot be certain of which kinds of data the LSTM model thinks is important or not. However, based on the results of fitting the model and how the predicted data fits the true data, we can say that there is a sort of pattern to the daily temperature present that the algorithm picked up and used to fit the model. However, the average temperature per day is not so homogeneously cyclical that the data can be predicted using the ARIMA model.

The inclusion of additional features, which are humidity and wind speed, has proven beneficial for improving the performance of the LSTM model in certain step sizes. This is seen when forecasting mean temperature with step sizes of 14 and 7, these features provide valuable information that improves the predictive power of the model. However, their impact may be limited for shorter step sizes (e.g., 3), as such a short time step may not be able to accurately capture any patterns visible in the data. Therefore, using a time step of 7 days and a multivariate model with wind speed, humidity, and mean temperature as predictors results in the smallest RMSE for the best prediction accuracy for future temperature.

Adding more layers by incorporating spatial data analysis, such as including the latitude and altitude of the particular area, are potential ways to further improve the performance of the LSTM model.

In summary, predicting the mean temperature in Delhi has highlighted the importance of selecting an appropriate time window, considering additional features, and employing suitable models. By leveraging these insights and further refining the modeling approach, more accurate and effective temperature predictions can be achieved for Delhi's climate.

Further applications of these kinds of analyses could be used to observe trends in weather data all over the world. With sufficient data, it could be possible to predict data from many more days in the future. In terms of research into climate change, it could be possible to use such predictive models to determine the trajectory of the Earth's climate and be prepared for more extreme situations. Perhaps, in the short term, LSTM could be used in daily weather predictions.

5. ***References***

[1] Hochreiter, S. and Schmidhuber, J. (1997): "Long Short-Term Memory," Neural Computation, 9(8), pp.1735-1780. DOI: 10.1162/neco.1997.9.8.1735.

[2] Siami-Namini, S. and Namin, A. S. (2018): "Forecasting economics and financial time series: ARIMA vs. LSTM", arXiv preprint arXiv:1803.06386.

[3] Daily Climate time series data. Available at: https://www.kaggle.com/datasets/sumanthvrao/daily-climate-time-series-data

[4] Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever. "An empirical exploration of recurrent network architectures." In International conference on machine learning, pp. 2342-2350. PMLR, 2015.

[5] Saxena. "Learn about Long Short-Term Memory (LSTM) Algorithms." Analytics Vidhya, March 16, 2021. https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/.

[6] Schmidhuber, Jürgen, Daan Wierstra, Matteo Gagliolo, and Faustino Gomez. "Training recurrent networks by evolino." Neural computation 19, no. 3 (2007): 757-779.

Sugandhi. "A Guide to Long Short Term Memory (LSTM) Networks." A Guide to Long Short Term Memory (LSTM) Networks, 2023. https://www.knowledgehut.com/blog/web-development/long-short-term-memory.

6. *Appendix*

6.1. LSTM analysis.

[ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_style('whitegrid')
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.callbacks import EarlyStopping
from keras.layers import Dense, LSTM, Dropout

from sklearn.preprocessing import MinMaxScaler
```

[ ]:
```
# multi-step data preparation
from google.colab import files
uploaded = files.upload()
```

[ ]:
```
### load data
my_data_1 = pd.read_csv('DailyDelhiClimateTest.csv', header = 0)
my_data_2 = pd.read_csv('DailyDelhiClimateTrain.csv', header = 0)



# Concatenate the dataframes vertically
combined_data = pd.concat([my_data_1, my_data_2], axis=0)

# Reset the index of the combined dataframe
combined_data.reset_index(drop=True, inplace=True)
data= combined_data
```

6.2. Data Exploration.

6.2.1. *Set Date column as an index.*

[ ]:
```
df= data

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

[ ]:
```
plt.figure(figsize=(15, 6))
df['meantemp'].plot()
plt.ylabel(None)
plt.xlabel(None)
```

```
plt.title("Mean Temperature")
plt.tight_layout()
plt.show()
```

[ ]:
```
plt.figure(figsize=(15, 6))
df['humidity'].plot()
plt.ylabel(None)
plt.xlabel(None)
plt.title("Humidity")
plt.tight_layout()
plt.show()
```

[ ]:
```
plt.figure(figsize=(15, 6))
df['wind_speed'].plot()
plt.ylabel(None)
plt.xlabel(None)
plt.title("Wind Spead")
plt.tight_layout()
plt.show()
```

[ ]:
```
plt.figure(figsize = (15,6))
df['wind_speed'].plot(label = "Wind Speed")
df['humidity'].plot(label = "Humidity")
df['meantemp'].plot(label = "Mean Temp")
plt.legend(["Wind Speed","Humidity", "Mean Temp"])
plt.xlabel('Date')
```

[ ]:
```
plt.figure(figsize=(15, 6))
df['meanpressure'].plot()
plt.ylabel(None)
plt.xlabel(None)
plt.title("Mean Pressure")
plt.tight_layout()
plt.show()
```

6.3. **Data Preprocessing.**

6.3.1. *Choosing Prediction Column.*

[ ]:
```
#n_cols = 1
#dataset = df["meantemp"]
#dataset = pd.DataFrame(dataset)
#data = dataset.values

#data.shape
```

```
df
data=(df.iloc[:, 0:1] )
print(data.shape)
```

[ ]:
```
#print(data)
```

6.3.2. *Normalizing Data.*

[ ]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range= (0, 1))
scaled_data = scaler.fit_transform(np.array(data))
```

6.3.3. *Splitting Data.*

[ ]:
```
train_size = int(len(data) * 0.80)
test_size = len(data) - train_size
print("Train Size :",train_size,"Test Size :",test_size)
```

[ ]:
```
train_data = scaled_data[0:train_size, :]
train_data.shape
```

[ ]:
```
train_data
```

[ ]:
```

```

[ ]:
```
from google.colab import files
df.to_csv('test_data.csv', encoding = 'utf-8-sig')
files.download('test_data.csv')
```

[ ]:
```
train_data.shape
```

6.3.4. *Creating training set.*

[ ]:
```
# Creating a Training set with 7 time-steps
x_data = []
y_data = []
time_steps = 3
n_cols = 1

for i in range(time_steps, len(scaled_data)):
    x_data.append(scaled_data[i-time_steps:i, :n_cols])
    y_data.append(scaled_data[i, :n_cols])
    if i<=time_steps:
        print('x_data_temp: ', x_data)
        print('y_data_temp:' , y_data)
```

[ ]:
```
# Convert to numpy array
x_data, y_data = np.array(x_data), np.array(y_data)
```

[ ]:
```
# Reshaping the input to (n_samples, time_steps, n_feature)
x_data = np.reshape(x_data, (x_data.shape[0], x_data.shape[1], n_cols))
```

[ ]:
```
x_data.shape , y_data.shape
```

[ ]:
```
y_data
```

7. LSTM Model

7.0.1. *Model Structure.*

[ ]:
```
from re import VERBOSE
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Bidirectional, LSTM, Dense


n_steps = time_steps
n_features = 1
n_steps_out = 1

# Define the model architecture
model = Sequential()

# Add the first layer (stacked LSTM)
# Default Activation function = "tanh"
# Default recurrent activation function = "sigmoid", therefore don't␣
 ↪specify new ones
model.add(LSTM(units=64, return_sequences=True, input_shape=(n_steps,␣
 ↪n_features)))

# Add the second layer (stacked LSTM)
model.add(LSTM(units=64, return_sequences=True))

# Add the third layer (stacked LSTM)
model.add(LSTM(units=64))

# Add the output layer
model.add(Dense(units=n_steps_out))

# Compile the model
model.compile(optimizer='adam', loss='mse',␣
 ↪metrics=['mean_absolute_error'])
```

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

[ ]:
```
# LSTM

# Split the data into training and validation sets
x_train_temp, x_test_temp, y_train_temp, y_test_temp =␣
↪train_test_split(x_data, y_data, test_size=0.2, random_state=3)
```

LSTM TRAINING

[ ]:
```
# Fitting the LSTM to the Training set
# Define the early stopping criteria
early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.00001,␣
↪patience=5, mode='min', verbose=1)
history = model.fit(x_train_temp, y_train_temp, epochs=50, batch_size=32,␣
↪verbose=1, validation_data=(x_test_temp, y_test_temp))
```

7.0.2. *Model Evaluation.*

[ ]:
```
plt.figure(figsize=(12, 8))
plt.plot(history.history["loss"])
plt.plot(history.history["mean_absolute_error"])
plt.legend(['Mean Squared Error','Mean Absolute Error'])
plt.title("Losses")
plt.xlabel("epochs")
plt.ylabel("loss")
plt.show()
```

7.1. Prediction.

7.1.1. *Creating testing set.*

[ ]:
```
# Get Prediction
predictions_temp = model.predict(x_test_temp)
```

[ ]:
```
predictions_temp.shape
```

[ ]:
```
#inverse predictions scaling


# Inverse transform the predictions
predictions_temp = scaler.inverse_transform(predictions_temp)

# The predictions will now have the shape (n_samples, n_features)
```

```
# Print the predictions
#print(predictions_temp)
```

7.1.2. *Root mean square error.*

[ ]:
```
#inverse y_test_temp scaling
y_test_temp = scaler.inverse_transform(y_test_temp)
#print(y_test_temp)
```

[ ]:
```
RMSE = np.sqrt(np.mean( y_test_temp - predictions_temp )**2).round(2)
print("The RMSE is",RMSE)
```

[ ]:
```
preds_acts = pd.DataFrame(data={'Predictions':predictions_temp.flatten(),␣
↪'Actuals':y_test_temp.flatten()})
preds_acts
```

[ ]:
```
plt.figure(figsize = (16, 6))
plt.plot(preds_acts['Predictions'], alpha = 0.7)
plt.plot(preds_acts['Actuals'], alpha = 0.7)
plt.legend(['Predictions', 'Actuals'])
plt.show()
```

Multivariate LSTM - Predict Temp with other variables

[ ]:
```
data_mult=combined_data
```

[ ]:
```
n_col = 60
#␣
↪date        meantemp        humidity        wind_speed        meanpressure
cols = list(data_mult.loc[:, ['meantemp', 'humidity', 'wind_speed',␣
↪'meanpressure']])
data_mult = data_mult[cols]
data_mult = pd.DataFrame(data_mult)
data_val_mult = data_mult.values
```

[ ]:
```
data_mult

# Select the input features and the target (temperature) column
input_features = ['meantemp', 'humidity', 'wind_speed']
target_column = 'meantemp'
input_data = data_mult[input_features].values
target_data = data_mult[target_column].values
target_data = target_data.reshape(-1, 1)
```

[ ]:
```
#noramlize
```

```
scaler = MinMaxScaler(feature_range= (0, 1))
scaled_data_mult_input = scaler.fit_transform(input_data)
scaled_data_mult_output = scaler.fit_transform(target_data)
```

[ ]:
```
scaled_data_mult_output
```

[ ]:
```
# Creating a Training set with 7 time-steps
x_data_mult = []
y_data_mult = []
time_steps = 14
n_cols = 3

input_data = scaled_data_mult_input
target_data = scaled_data_mult_output

for i in range(time_steps, len(input_data)):
    x_data_mult.append(input_data[i-time_steps:i])
    y_data_mult.append(target_data[i])
    if i<=time_steps:
        print('x_data_mult: ', x_data_mult)
        print('y_data_mult:' , y_data_mult)
```

[ ]:
```
# Convert to numpy array
x_data_mult, y_data_mult = np.array(x_data_mult), np.array(y_data_mult)
```

[ ]:
```
x_train_mult, x_test_mult, y_train_mult, y_test_mult =␣
↪train_test_split(x_data_mult, y_data_mult, test_size=0.2,␣
↪random_state=3)
```

[ ]:
```
x_train_mult.shape
```

[ ]:
```
from re import VERBOSE
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Bidirectional, LSTM, Dense


n_steps = time_steps
n_features = 3
n_steps_out = 1

# Define the model architecture
model_mult = Sequential()

# Add the first layer (stacked LSTM)
```

```
# Default Activation function = "tanh"
# Default recurrent activation function = "sigmoid", therefore don't␣
 ↪specify new ones
model_mult.add(LSTM(units=64, return_sequences=True,␣
 ↪input_shape=(n_steps, n_features)))

# Add the second layer (stacked LSTM)
model_mult.add(LSTM(units=64, return_sequences=True))

# Add the third layer (stacked LSTM)
model_mult.add(LSTM(units=64))

# Add the output layer
model_mult.add(Dense(units=n_steps_out))

# Compile the model
model_mult.compile(optimizer='adam', loss='mse',␣
 ↪metrics=['mean_absolute_error'])

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

[ ]:
```
#early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.00001,␣
 ↪patience=5, mode='min', verbose=1)
history_mult = model_mult.fit(x_train_mult, y_train_mult, epochs=50,␣
 ↪batch_size=32, verbose=1, validation_data=(x_test_mult, y_test_mult))
```

[ ]:
```
plt.figure(figsize=(12, 8))
plt.plot(history_mult.history["loss"])
plt.plot(history_mult.history["mean_absolute_error"])
plt.legend(['Mean Squared Error','Mean Absolute Error'])
plt.title("Losses")
plt.xlabel("epochs")
plt.ylabel("loss")
plt.show()
```

[ ]:
```

```

[ ]:
```
# Get Prediction
predictions_mult = model_mult.predict(x_test_mult)
```

[ ]:
```
#inverse predictions scaling

```

```
# Inverse transform the predictions
predictions_mult = scaler.inverse_transform(predictions_mult)

# The predictions will now have the shape (n_samples, n_features)

# Print the predictions
#print(predictions_temp)
```

[ ]:
```
#inverse y_test_temp scaling
y_test_mult= scaler.inverse_transform(y_test_mult)
#print(y_test_temp)
```

[ ]:
```
RMSE = np.sqrt(np.mean( y_test_mult - predictions_mult )**2).round(2)
print("The RMSE is",RMSE)
```

[ ]:
```
preds_acts_mult = pd.DataFrame(data={'Predictions':predictions_mult.
↪flatten(), 'Actuals':y_test_mult.flatten()})
preds_acts_mult
```

[ ]:
```
plt.figure(figsize = (16, 6))
plt.plot(preds_acts_mult['Predictions'], alpha = 0.7)
plt.plot(preds_acts_mult['Actuals'], alpha = 0.7)
plt.legend(['Predictions', 'Actuals'])
# Add a title to the plot
plt.title("Predicted vs Actual (Time Step = 7)")

# Label the x-axis
plt.xlabel("Time")

# Label the y-axis
plt.ylabel("Degrees Celsius ")
plt.show()
```

8. Arima Model

[24]:
```
import pmdarima
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import numpy as np
import matplotlib.pyplot as plt
```

[48]:
```
### load data
my_data_1 = pd.read_csv('DailyDelhiClimateTest.csv', header = 0)
```

```
my_data_2 = pd.read_csv('DailyDelhiClimateTrain.csv', header = 0)



# Concatenate the dataframes vertically
combined_data = pd.concat([my_data_1, my_data_2], axis=0)

# Reset the index of the combined dataframe
combined_data.reset_index(drop=True, inplace=True)
data= combined_data

data.head()
```

[48]:

	date	meantemp	humidity	wind_speed	meanpressure
0	2017-01-01	15.913043	85.869565	2.743478	59.000000
1	2017-01-02	18.500000	77.222222	2.894444	1018.277778
2	2017-01-03	17.111111	81.888889	4.016667	1018.333333
3	2017-01-04	18.700000	70.050000	4.545000	1015.700000
4	2017-01-05	18.388889	74.944444	3.300000	1014.333333

[49]:
```
df= data
df['date'] = pd.to_datetime(df['date'])
df.set_index('date', inplace= True)
data=(df.iloc[:, 0:1] )
print(data.shape)

scaler = MinMaxScaler(feature_range= (0, 1))
scaled_data = scaler.fit_transform(np.array(data))
train_size = int(len(data) * 0.80)
test_size = len(data) - train_size
print("Train Size :",train_size,"Test Size :",test_size)
train_data = scaled_data[0:train_size, :]
train_data.shape
```

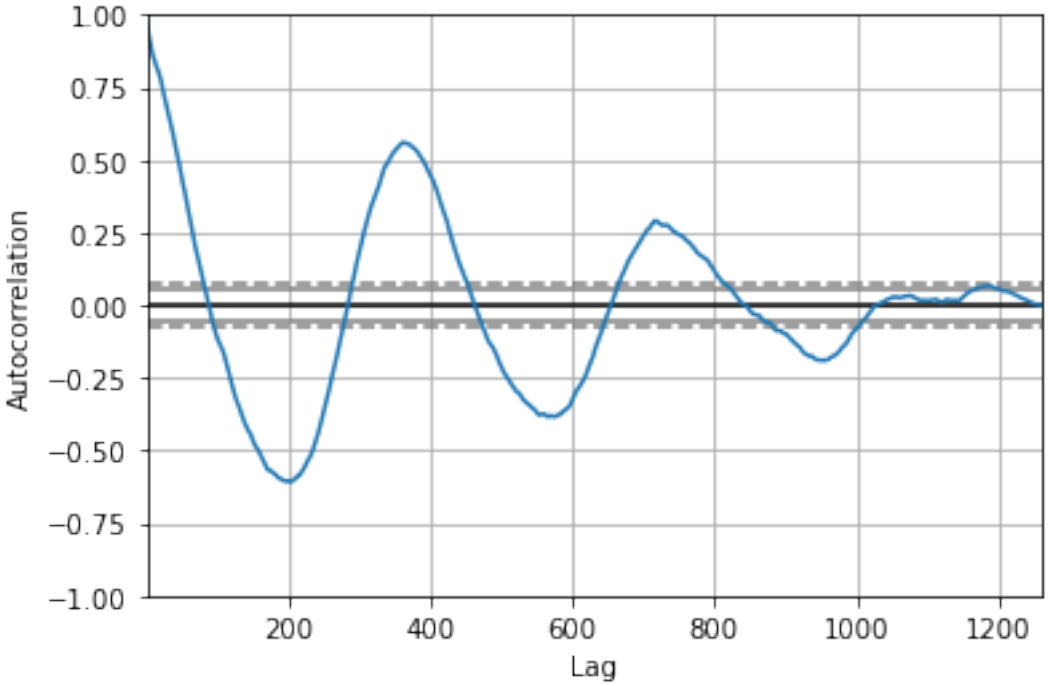
```
(1576, 1)
Train Size : 1260 Test Size : 316
```

[49]:
```
(1260, 1)
```

[50]:
```
test_data =  scaled_data[train_size:, :]
```

[51]:
```
# train_temp
#y_test_temp.shape
from pandas.plotting import autocorrelation_plot
```

```
autocorrelation_plot(train_data)
plt.show()
```



[25]:
```
stepwise_model = pmdarima.arima.auto_arima(train_data, start_p=1,␣
↪start_q=1,
                           max_p=20, max_q=20, m=12,
                           start_P=0, seasonal=True,
                           d=1, D=1, trace=True,
                           error_action='ignore',
                           suppress_warnings=True,
                           stepwise=True)
print(stepwise_model.aic())
```

[52]:
```
future_forecast = stepwise_model.predict(n_periods=test_data.shape[0])
# This returns an array of predictions:
print(future_forecast)
```

[53]:
```
future_forecast = scaler.inverse_transform(future_forecast.reshape(316,␣
↪1))
test_data = scaler.inverse_transform(test_data)
```

[55]:
```
preds_acts = pd.DataFrame(data={'Predictions':future_forecast.flatten(),␣
↪'Actuals':test_data.flatten()})
preds_acts
```

[55]:

	Predictions	Actuals
0	23.387578	21.000000
1	23.681532	21.000000
2	23.835261	21.428571
3	21.908141	21.687500
4	20.365696	22.562500
..	...	...
311	71.630285	17.217391
312	71.571754	15.238095
313	72.276046	14.095238
314	72.947579	15.052632
315	71.333900	10.000000

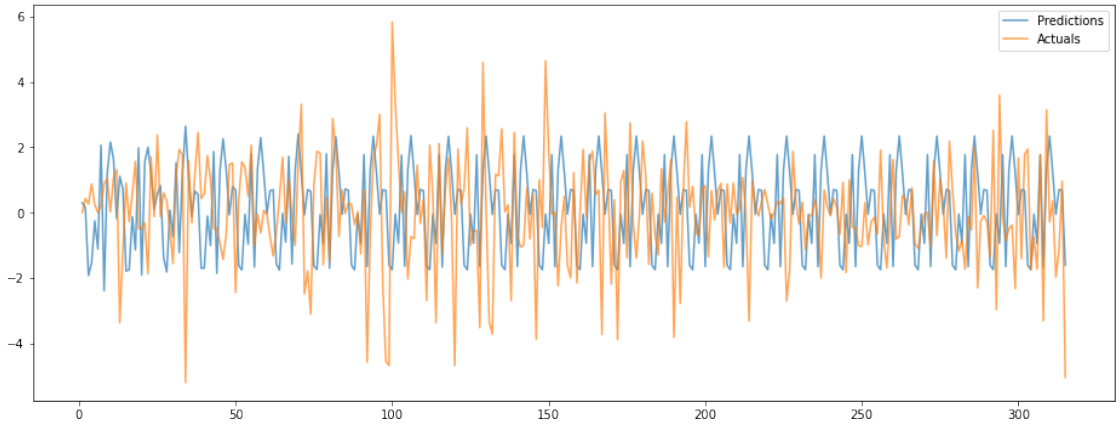
[316 rows x 2 columns]

[58]:
```
diff_preds_acts = preds_acts.diff()
```

[60]:
```
RMSE = np.sqrt(np.mean( diff_preds_acts['Predictions'] -␣
↪diff_preds_acts['Actuals'] )**2).round(2)
print("The RMSE is",RMSE)
```

```
The RMSE is 0.19
```

[59]:
```
plt.figure(figsize = (16, 6))
plt.plot(diff_preds_acts['Predictions'], alpha = 0.7)
plt.plot(diff_preds_acts['Actuals'], alpha = 0.7)
plt.legend(['Predictions', 'Actuals'])
plt.show()
```



Email address: mrcqian@ucdavis.edu

Email address: leylin@ucdavis.edu

Email address: lyewang@ucdavis.edu

Email address: efbasa@ucdavis.edu