Machine learning: Barometric Pressure Prediction Knowledge Documentation

ARIMA model

ARIMA (AutoRegressive Integrated Moving Average) is a powerful time series forecasting model widely used in various fields, including economics, finance, and climate science. It's a combination of Autoregression (AR), Differencing (I), and Moving Average (MA) components. ARIMA is capable of capturing complex patterns and trends in time series data. So what is it typically used for:

Typical Use Cases:

- Financial Forecasting: ARIMA is employed to predict stock prices, exchange rates, and other financial variables.
- Demand Forecasting: ARIMA assists in predicting product demand, allowing better inventory management.
- Climate Prediction: ARIMA is applied to model and forecast temperature, precipitation, and other climatic factors.
- Resource Planning: ARIMA aids in estimating future resource requirements, such as electricity demand.

Strengths:

1. Versatility: ARIMA can handle various time series data with complex patterns.
2. Accurate Forecasting: It excels in short to medium-term forecasting with a limited number of data points.
3. Interpretable: ARIMA's parameters offer meaningful insights into the data dynamics.

Weaknesses:

1. Sensitive to Outliers: ARIMA performance is impacted by outliers in the data.
2. Limited for Long-Term Forecasting: It may not perform well for long-range predictions with a large number of time steps.
3. Manual Selection of Parameters: Choosing ARIMA's order (p, d, q) requires domain knowledge and experimentation.

Using ARIMA

Import library

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
```

Load data

Use pandas,

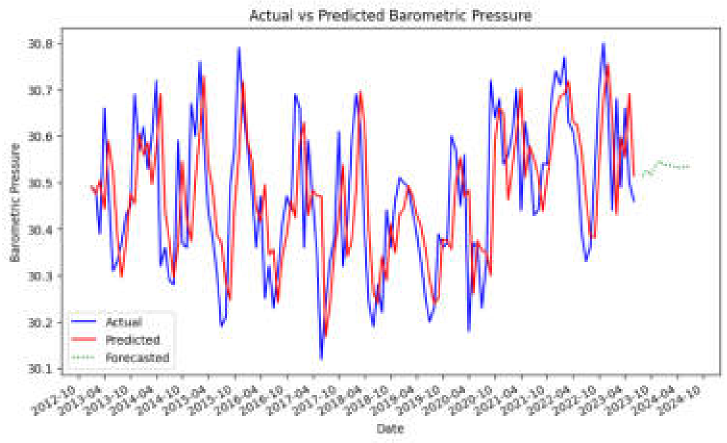
# Load data

df = pd.read_csv('/content/72503014732.csv')

Fitting the model onto data

```
#Training split
train_data = df.iloc[:-int(len(df)*0.2)]
test_data = df.iloc[-int(len(df)*0.2):]
# Fit the ARIMA model
model = ARIMA(train_data, order=(5,1,0))
model_fit = model.fit()
```

Graph



Conclusion

Warnings were displayed of convergence failure. This warning suggests that the maximum likelihood optimization, which is used for estimating model parameters, didn't converge. In other words, the optimization algorithm couldn't find the best-fitting model parameters. It's possible that the model is not well-suited for the data, or the data has certain characteristics that make the optimization challenging.

Albeit, the nature of the data fed was only barometric pressure history from the years 2012 through to today and so in reality many more variables would be needed to make a prediction. Some overfitting is seen too, as the prediction with unseen data did not fare with such low confidence intervals.

Therefore, ARIMA may not capture the underlying patterns in the data effectively. Barometric pressure is influenced by complex atmospheric dynamics and may exhibit non-linear behavior. Other models, such as Long Short-Term Memory (LSTM) and Gradient Boosting, are better suited for time series data with long-term dependencies and non-linear patterns. LSTM can capture sequential patterns, while Gradient Boosting excels at handling complex interactions among features. These models can potentially outperform ARIMA by better capturing the intricate dynamics of barometric pressure, especially when dealing with data that might not align well with ARIMA's assumptions of linearity and stationarity.

Code

```python
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
import matplotlib.dates as mdates
from sklearn.model_selection import train_test_split

# Load data
df = pd.read_csv('/content/72503014732.csv')

# Convert Date-Month to datetime format
df['Date-Month'] = pd.to_datetime(df['Date-Month'], format='%Y-%b')

# Sort by date
df = df.sort_values('Date-Month')

# Set 'Date-Month' as index
df.set_index('Date-Month', inplace=True)
#Training split
train_data = df.iloc[:-int(len(df)*0.2)]
test_data = df.iloc[-int(len(df)*0.2):]
# Fit the ARIMA model
model = ARIMA(train_data, order=(5,1,0))
model_fit = model.fit()

# Make predictions
history = [x for x in df['Barometric Pressure']]
predictions = list()
for t in range(len(df)):
    model = ARIMA(history, order=(5,1,0))
    model_fit = model.fit()
    output = model_fit.forecast()
    pred_value = output[0]
    predictions.append(pred_value)
    obs_value = df['Barometric Pressure'][t]
    history.append(obs_value)

# Plot actual vs predicted values
plt.figure(figsize=(10,6))
plt.plot(df.index, df['Barometric Pressure'].values, color='blue', label='Actual')
plt.plot(df.index, predictions, color='red', label='Predicted')  # Predicted data

# Forecasting for next 12 months
predictions = model_fit.forecast(steps=len(test_data))
forecast_index = pd.date_range(start=df.index[-1], periods=13, freq='M')
plt.plot(forecast_index[1:], forecast, color='green', linestyle='dotted', label='Forecasted')  # Future forecast

plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))
plt.gca().xaxis.set_major_locator(mdates.MonthLocator(interval=6))
plt.legend()
plt.title('Actual vs Predicted Barometric Pressure')
plt.xlabel('Date')
plt.ylabel('Barometric Pressure')
plt.gcf().autofmt_xdate()  # Rotation
plt.show()
```

LSTM RMSE & Gradient Boosting model

LSTM is a specialized type of Recurrent Neural Network (RNN) designed to handle sequential data, such as time series, speech, and natural language. Unlike traditional RNNs, LSTM addresses the vanishing gradient problem, which can hinder the learning of long-term dependencies in data. The vanishing gradient problem occurs when gradients (used to update model parameters) become extremely small, leading to slow or ineffective learning. LSTM introduces a unique memory cell structure with gates (input, output, and forget) that allows it to store and access information over extended time intervals. This makes LSTM proficient in capturing complex patterns and dependencies in sequences, which is essential for time series forecasting, language translation, and other sequential data tasks.

Gradient Boosting:

Gradient Boosting is an ensemble learning technique that creates a strong predictive model by combining multiple weak learners, such as decision trees.(Ensemble learning involves combining multiple individual models (often called "base" or "weak" learners) to create a more powerful and accurate predictive model, known as an "ensemble" model. The idea behind ensemble learning is that by aggregating the predictions of several diverse models, the overall model can be more robust, generalize better, and make more accurate predictions than any individual model). Each weak learner is built to correct the errors of the previous one, making the ensemble model progressively more accurate. This iterative process is carried out by minimizing the model's loss function using gradient descent. By iteratively adding new weak learners, the ensemble gradually learns and adapts to the data's complexity, resulting in a powerful predictive model.

Gradient Boosting is widely used for regression and classification tasks and is known for its excellent performance, robustness to outliers, and ability to handle large datasets.

Use Cases for LSTM and Gradient Boosting:

LSTM:

- Time Series Forecasting: LSTM excels in predicting future values for time series data, such as stock prices, weather patterns, and energy consumption.
- Natural Language Processing (NLP): LSTM is widely employed in tasks like sentiment analysis, language translation, and speech recognition.
- Sequential Data Analysis: It is used in tasks involving sequential patterns, such as gesture recognition and DNA sequence analysis.

Gradient Boosting:

- Regression and Classification: Gradient Boosting is effective for a wide range of regression and classification problems, including house price prediction and image classification.
- Ranking: It is used in search engines and recommendation systems to rank results and suggest personalized content.
- Anomaly Detection: Gradient Boosting helps identify outliers and anomalies in data, such as fraud detection in financial transactions.

Strengths and Weaknesses:

LSTM:

Strengths:

- Long-Term Dependencies: LSTM effectively captures long-term patterns and dependencies in sequential data, making it suitable for time series forecasting.
- Contextual Memory: The memory cell allows LSTM to retain information from earlier time steps, essential for understanding sequential context.
- Flexibility: It can handle sequences of varying lengths, accommodating diverse applications.

Weaknesses:

- Complexity: LSTM models can be computationally expensive to train due to their architecture complexity and may require more data for effective learning.
- Overfitting: Without proper regularization, LSTM models may be prone to overfitting, resulting in poor generalization to new data.

Gradient Boosting:

Strengths:

- High Accuracy: Gradient Boosting produces highly accurate predictive models, often outperforming traditional machine learning algorithms.
- Robustness: It handles outliers and missing data gracefully, reducing the risk of data preprocessing issues.
- Feature Importance: It provides insights into the importance of features, aiding in feature selection and understanding model decisions.

Weaknesses:

- Training Time: Gradient Boosting can be computationally expensive during model training, particularly for large datasets and deep trees.
- Sensitivity to Noise: It might struggle with noisy data, leading to potential overfitting and less reliable predictions.

Using LSTM in code

Importing libraries:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
```

Preparing data into dataframe for training and testing

```
# Assuming 'df' is the DataFrame containing time series data
# Split the data into features (X) and target (y)
X = df['your_feature_column'].values
y = df['Barometric Pressure'].values

# Normalize the data if needed
# (LSTM typically benefits from scaled data)
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X = scaler.fit_transform(X.reshape(-1, 1))
y = scaler.fit_transform(y.reshape(-1, 1))

# Split the data into training and testing sets
# (for simplicity, using a 80-20 split, adjust as needed)
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
```

In the context of MinMaxScaler from Scikit-learn, scaling and transforming refer to the process of normalizing the data to a specific range. The fit_transform method is used to both fit the scaler to the data and then apply the scaling transformation to the data.

Scaling:

Scaling is the process of transforming numerical data to a specific range, typically between 0 and 1. This is important because many machine learning algorithms perform better when the input features are on a similar scale. Scaling prevents features with larger values from dominating the learning process and ensures that all features contribute equally to the model.

Transforming:

In the context of MinMaxScaler, transforming means applying the scaling transformation to the data. The transformation is based on the minimum and maximum values of the data in each feature column.

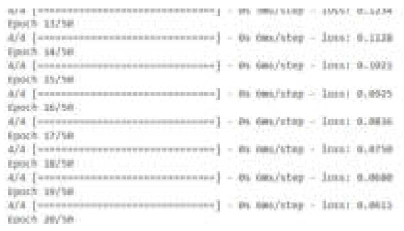
fit_transform for X and y:

fit_transform is a convenience method in Scikit-learn that combines two steps: fitting the scaler to the data (finding the minimum and maximum values) and applying the scaling transformation. When using fit_transform for X and y with MinMaxScaler, each feature column in X and the target column (y) will be independently scaled to the range [0, 1].

<u>Building and training LSTM</u>

```
model = Sequential()
model.add(LSTM(units=50, input_shape=(X_train.shape[1], X_train.shape[2])))
model.add(Dense(units=1))
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=50, batch_size=32)
```

```
Make sure to do enough epochs to lower loss.
```



```
4/4 [==============================] - 0s 6ms/step - loss: 0.1234
Epoch 13/50
4/4 [==============================] - 0s 6ms/step - loss: 0.1128
Epoch 14/50
4/4 [==============================] - 0s 6ms/step - loss: 0.1021
Epoch 15/50
4/4 [==============================] - 0s 6ms/step - loss: 0.0925
Epoch 16/50
4/4 [==============================] - 0s 6ms/step - loss: 0.0836
Epoch 17/50
4/4 [==============================] - 0s 6ms/step - loss: 0.0750
Epoch 18/50
4/4 [==============================] - 0s 6ms/step - loss: 0.0680
Epoch 19/50
4/4 [==============================] - 0s 6ms/step - loss: 0.0613
Epoch 20/50
```

In machine learning, an epoch refers to a single pass through the entire training dataset during model training. The number of epochs is a crucial hyperparameter that impacts model performance and convergence. Too few epochs may lead to underfitting, while too many epochs can cause overfitting. To determine the optimal number of epochs, monitor the model's performance on a validation set during training, and consider early stopping to prevent overfitting by stopping training when the validation loss stops improving or starts increasing. Choosing the right number of epochs is essential for achieving the best model results. A goal of 0.0 onwards is good.

Predicting with LSTM Model:

```
# Assuming you have the test data X_test and y_test
```

```
y_pred = model.predict(X_test)
# Inverse transform the predicted values if you had scaled the data earlier
y_pred = scaler.inverse_transform(y_pred)
```

the inverse is done to present values that are no longer normalized for outputs.

Gradient Boosting libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import GradientBoostingRegressor
```

Preparing data into dataframe for training and testing

```
# Assuming 'df' is the DataFrame containing time series data
# Split the data into features (X) and target (y)
X = df['your_feature_column'].values
y = df['Barometric Pressure'].values

# Split the data into training and testing sets
# (for simplicity, using a 80-20 split, adjust as needed)
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
```

Build and train Gradient Boosting model

```
model = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,
loss='ls')
model.fit(X_train.reshape(-1, 1), y_train)
```

From here, estimator parameter specifies the number of weak learners (individual decision trees) that will be combined to form the ensemble model in Gradient Boosting. Each weak learner tries to correct the errors made by the previous one, and by aggregating their predictions, the ensemble model becomes more accurate and powerful. Increasing the number of estimators can improve the model's performance, but it also increases computational complexity and training time. Optimize it to get enough accuracy, while not spending too much time.

The learning rate parameter controls the contribution of each weak learner to the final ensemble model. It is a small positive value typically set between 0 and 1. A smaller learning rate makes the model learn more slowly, requiring more estimators for accurate predictions. Conversely, a larger learning rate allows the model to learn quickly, but it may lead to overfitting. The learning rate and the number of estimators are interrelated; lower learning rates generally require more estimators to achieve the same level of performance as higher learning rates. I haven’t experimented with tweaking this. Just the estimators in other projects.

Predictions with Gradient Boosting Model

```
# Assuming you have the test data X_test
y_pred = model.predict(X_test.reshape(-1, 1))
```

Initial error of shape sizes

```
index[:-1] shape: (137,)
y_all shape: (125, 1)
lstm_predictions shape: (25, 1)
gb_predictions shape: (25, 1)
```

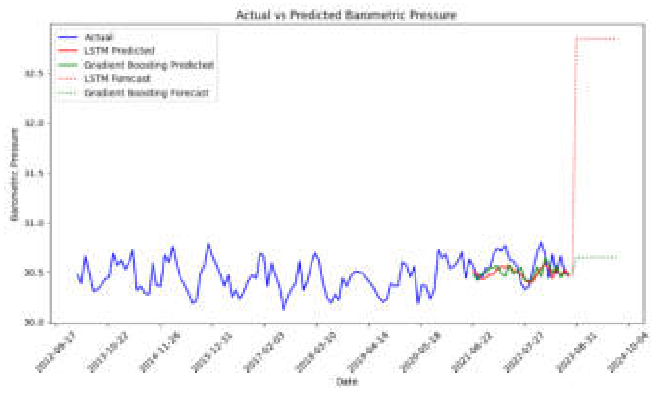
The error encountered indicates a shape mismatch between the x-axis values(time) and the y-axis values (barometric pressure) when plotting. The shapes must match for the plot to be created successfully.
Looking at the printed shapes:

- x axis array shape: (137,): This means the x-axis data has 137 elements.
- y axis array shape: (125, 1): This means the y-axis data has 125 elements, but it is a 2D array with a shape of (125, 1).

To address this, the plotting section was corrected as follows:

1. The index_extended[1:-12] range was used for the actual and predicted values, excluding the last 12 elements to match the lstm_predictions and gb_predictions data.
2. The index_extended[-12:] range was used for the forecasts, including only the last 12 elements to match the lstm_forecasts and gb_forecasts data.
3. The x-axis data index_extended was set to show only 12 months using plt.gca().xaxis.set_major_locator(plt.MaxNLocator(12)).

Graph



As seen, this is better in terms of overfitting, and only the last 20 percent is used to test the data after trained on the first 80 percent. The forcecast for the gradient boosting is very reasonable. Flat lines are typical as a single variable is only being used for forecasting. However, the LSTM prediction spiked uncontrollably to a number never seen historically.

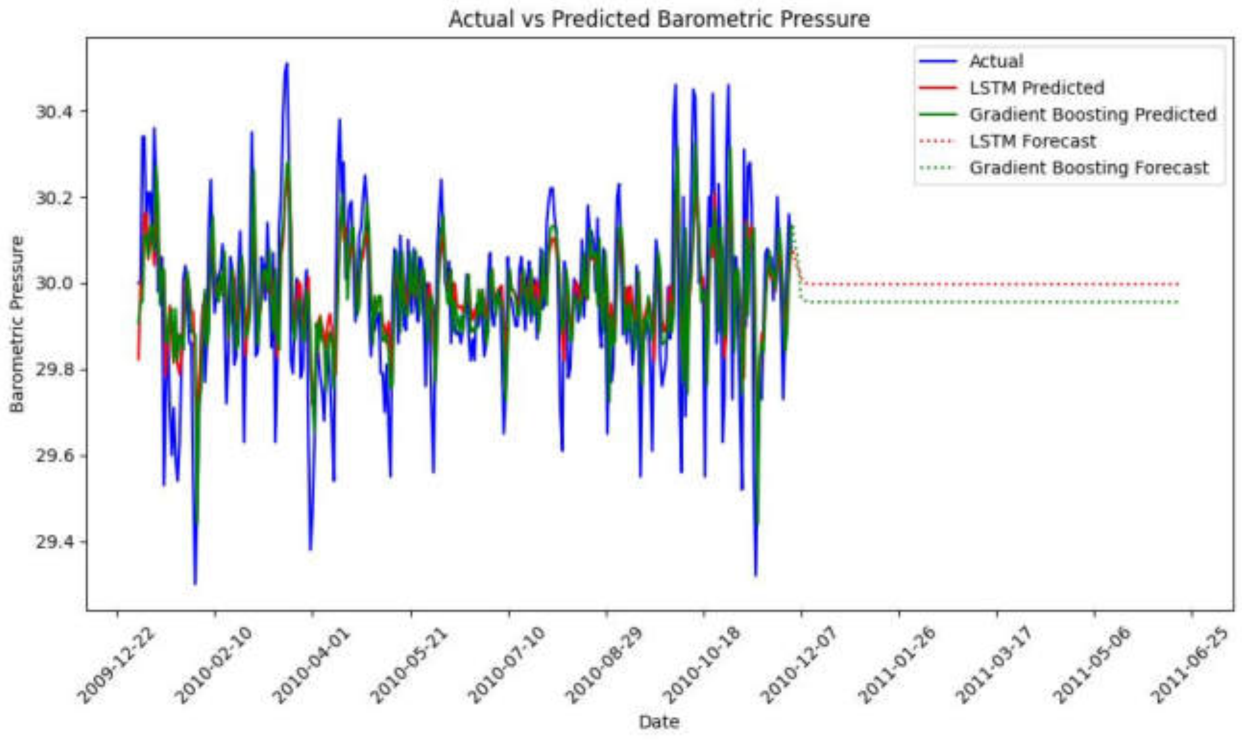
Some research revealed that this spike was caused because of :

1. **Complex Patterns in Data:** LSTM models are capable of capturing complex temporal patterns in the data. If the historical data contains intricate patterns or sudden changes, the LSTM might pick up on these patterns and produce predictions that deviate significantly from the past values.
2. **Overfitting:** If the LSTM model is overfitting the training data, it may memorize the noise or outliers present in the training set. Consequently, during forecasting, it might amplify these noisy patterns, leading to spikes in the predictions.
3. **Limited Training Data:** If the LSTM is trained on a small or limited dataset, it might not generalize well to unseen data during forecasting. In such cases, the model might struggle to capture the underlying patterns and produce unexpected spikes.
4. **Input Representation:** The way input data is prepared can influence LSTM predictions. If the input representation or feature engineering is not capturing all relevant information, the model might generate unrealistic spikes.
5. **Hyperparameters:** The choice of hyperparameters, such as the number of LSTM layers, units per layer, learning rate, or number of epochs, can significantly impact the model's performance. Poorly tuned hyperparameters might lead to overfitting or inadequate generalization.
6. **Outliers or Anomalies:** If there are outliers or anomalies in the data, the LSTM might react strongly to these unusual values, causing spikes in the predictions.

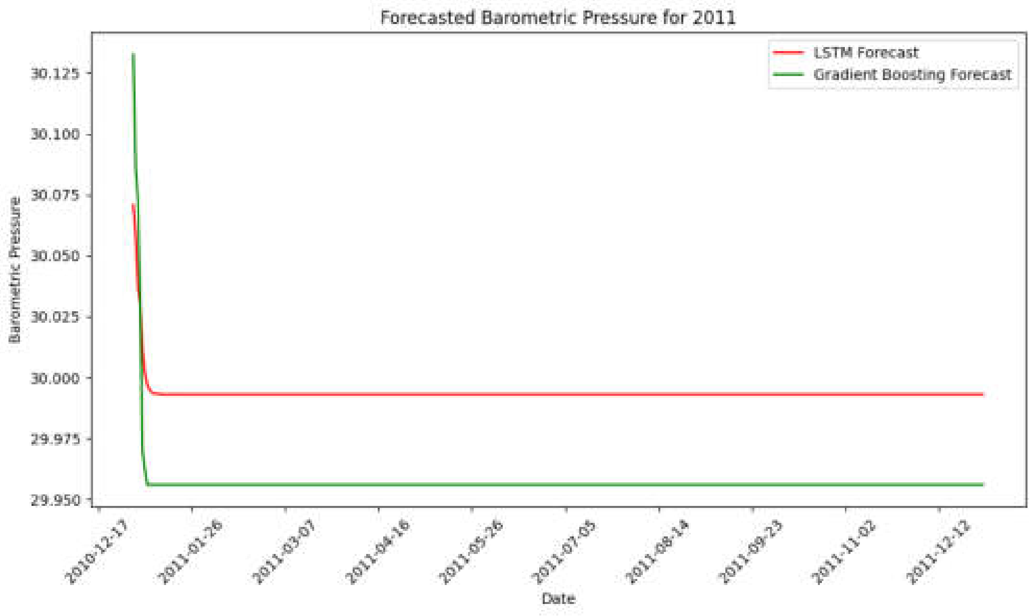
The easiest approach was to feed the model more data, only this time the dataset would not be split up 80-20 for training and testing, but they will be different years altogether. Some public datasets available were used with older files, with 2006 – 2009 used for training files , and 2010 barometric pressures used as the test file.

It is crucial to parse the data and columns correctly! The date formats!!

With more data graph:



A lot better!



It takes the last point and predicts from then on. That is why predicting without data is useless, real time predictions based on rolling in data is best. As seen though, straight plateau estimations are as best it can do with a singular variable dataset. It was found that only 8 epochs maximum were needed with a stable lost off 0.0174 from that point on.

Final code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
```

```python
from tensorflow.keras.layers import LSTM, Dense
from sklearn.ensemble import GradientBoostingRegressor

# Define the list of file names for the training and testing data
train_files = ['2006_barometric_pressure.csv', '2007_barometric_pressure.csv',
               '2008_barometric_pressure.csv', '2009_barometric_pressure.csv']
test_file = '2010_barometric_pressure.csv'

# Function to parse dates
dateparse = lambda dates: pd.to_datetime(dates, format='%Y-%b-%d', errors='coerce')

# Load training data
dfs = []
for file in train_files:
    df = pd.read_csv(f'/content/{file}', parse_dates=['Date-Month'], index_col='Date-Month', date_parser=dateparse)
    df.dropna(inplace=True)
    dfs.append(df)
df_train = pd.concat(dfs).sort_values('Date-Month')

# Load testing data
df_test = pd.read_csv(f'/content/{test_file}', parse_dates=['Date-Month'], index_col='Date-Month', date_parser=dateparse)
df_test.dropna(inplace=True)
df_test = df_test.sort_values('Date-Month')

# Combine training and testing data for preprocessing
df = pd.concat([df_train, df_test])

# Ensure Barometric Pressure is of numeric type
df['Barometric Pressure'] = pd.to_numeric(df['Barometric Pressure'], errors='coerce')
df.dropna(inplace=True)

# Extract features and target variable
X = df.iloc[:-1].values  # Features (all rows except the last)
y = df.iloc[1:].values  # Target variable (all rows except the first)

# Scale the data
scaler = MinMaxScaler(feature_range=(0, 1))
X_scaled = scaler.fit_transform(X)
y_scaled = scaler.transform(y)

# Split the data into training and testing sets
X_train, X_test = X_scaled[:len(df_train) - 1], X_scaled[len(df_train) - 1:]
y_train, y_test = y_scaled[:len(df_train) - 1], y_scaled[len(df_train) - 1:]

# LSTM model
lstm_model = Sequential()
lstm_model.add(LSTM(50, input_shape=(X_train.shape[1], 1)))
lstm_model.add(Dense(1))
lstm_model.compile(loss='mean_squared_error', optimizer='adam')
lstm_model.fit(np.expand_dims(X_train, axis=2), y_train, epochs=20, batch_size=32) #lower epochs for testing

# Predict with LSTM model
lstm_predictions = lstm_model.predict(np.expand_dims(X_test, axis=2))

# Gradient Boosting Model
gb_model = GradientBoostingRegressor()
gb_model.fit(X_train, y_train.ravel())

# Predict with Gradient Boosting Model
gb_predictions = gb_model.predict(X_test)

# Generate forecasts for the next 12 months
last_input = X_test[-1]
lstm_forecasts, gb_forecasts = [], []

for _ in range(200):
    last_input = last_input.reshape((1, -1))

    lstm_forecast = lstm_model.predict(np.expand_dims(last_input, axis=2))
    lstm_forecasts.append(lstm_forecast[0][0])

    gb_forecast = gb_model.predict(last_input)
    gb_forecasts.append(gb_forecast[0])

    last_input = np.roll(last_input, -1)
    last_input[0, -1] = lstm_forecast[0][0] if lstm_forecast[0][0] > gb_forecast[0] else gb_forecast[0]
# Reshape for inverse transform
lstm_predictions = lstm_predictions.reshape(-1, 1)
gb_predictions = gb_predictions.reshape(-1, 1)
lstm_forecasts = np.array(lstm_forecasts).reshape(-1, 1)
gb_forecasts = np.array(gb_forecasts).reshape(-1, 1)
# Inverse transform predictions and forecasts
lstm_predictions_inv = scaler.inverse_transform(lstm_predictions)
gb_predictions_inv = scaler.inverse_transform(gb_predictions)
lstm_forecasts_inv = scaler.inverse_transform(lstm_forecasts)
gb_forecasts_inv = scaler.inverse_transform(gb_forecasts)
# Inverse transform y_test
y_test_inv = scaler.inverse_transform(y_test)
# Create an index for the 2010 data and the next 12 months
index_test = pd.date_range(start=df_test.index.min(), periods=len(y_test) + 200, freq='D')
# Plot actual and predicted values
plt.figure(figsize=(10, 6))
plt.plot(index_test[:len(y_test)], y_test_inv, color='blue', label='Actual')
plt.plot(index_test[:len(y_test)], lstm_predictions_inv, color='red', label='LSTM Predicted')
plt.plot(index_test[:len(y_test)], gb_predictions_inv, color='green', label='Gradient Boosting Predicted')
# Plot the forecasts for the next 12 months
plt.plot(index_test[len(y_test):len(y_test)+200], lstm_forecasts_inv, color='red', linestyle='dotted', label='LSTM Forecast')
plt.plot(index_test[len(y_test):len(y_test)+200], gb_forecasts_inv, color='green', linestyle='dotted', label='Gradient Boosting Forecast')

plt.gca().xaxis.set_major_locator(plt.MaxNLocator(13))  # Set the x-axis to show 13 locations
plt.legend()
plt.title('Actual vs Predicted Barometric Pressure')
plt.xlabel('Date')
plt.ylabel('Barometric Pressure')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```