Load Dataset

```
import pandas as pd

# Load the dataset into a pandas DataFrame
df = pd.read_csv('Price_Agriculture_commodities_Week.csv')

# Display the first 5 rows of the DataFrame
print("First 5 rows of the DataFrame:")
print(df.head())

print("\nDataFrame Info:")
# Print the concise summary of the DataFrame, including data types
df.info()
```

```
First 5 rows of the DataFrame:
     State District    Market              Commodity      Variety Grade  \
0  Gujarat   Amreli  Damnagar  Bhindi(Ladies Finger)       Bhindi   FAQ
1  Gujarat   Amreli  Damnagar               Brinjal        Other   FAQ
2  Gujarat   Amreli  Damnagar               Cabbage      Cabbage   FAQ
3  Gujarat   Amreli  Damnagar           Cauliflower  Cauliflower   FAQ
4  Gujarat   Amreli  Damnagar      Coriander(Leaves)    Coriander   FAQ

  Arrival_Date  Min Price  Max Price  Modal Price
0   27-07-2023     4100.0     4500.0       4350.0
1   27-07-2023     2200.0     3000.0       2450.0
2   27-07-2023     2350.0     3000.0       2700.0
3   27-07-2023     7000.0     7500.0       7250.0
4   27-07-2023     8400.0     9000.0       8850.0

DataFrame Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23093 entries, 0 to 23092
Data columns (total 10 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   State         23093 non-null  object
 1   District      23093 non-null  object
 2   Market        23093 non-null  object
 3   Commodity     23093 non-null  object
 4   Variety       23093 non-null  object
 5   Grade         23093 non-null  object
 6   Arrival_Date  23093 non-null  object
 7   Min Price     23093 non-null  float64
 8   Max Price     23093 non-null  float64
 9   Modal Price   23093 non-null  float64
dtypes: float64(3), object(7)
memory usage: 1.8+ MB
```

Exploratory Data Analysis (EDA)

```
print("Missing values per column:")
print(df.isnull().sum())

print("\nNumber of duplicate rows:")
print(df.duplicated().sum())
```

```
Missing values per column:
State           0
District        0
Market          0
Commodity       0
Variety         0
Grade           0
Arrival_Date    0
Min Price       0
Max Price       0
Modal Price     0
dtype: int64

Number of duplicate rows:
0
```

```
df['Arrival_Date'] = pd.to_datetime(df['Arrival_Date'], format='%d-%m-%Y')
```

```
df['Year'] = df['Arrival_Date'].dt.year
df['Month'] = df['Arrival_Date'].dt.month
df['DayOfWeek'] = df['Arrival_Date'].dt.dayofweek

print("Converted 'Arrival_Date' and extracted 'Year', 'Month', 'DayOfWeek'.")
print(df[['Arrival_Date', 'Year', 'Month', 'DayOfWeek']].head())
```

```
Converted 'Arrival_Date' and extracted 'Year', 'Month', 'DayOfWeek'.
  Arrival_Date  Year  Month  DayOfWeek
0   2023-07-27  2023      7          3
1   2023-07-27  2023      7          3
2   2023-07-27  2023      7          3
3   2023-07-27  2023      7          3
4   2023-07-27  2023      7          3
```

```
print("Descriptive statistics for numerical price columns:")
print(df[['Min Price', 'Max Price', 'Modal Price']].describe())
```

```
Descriptive statistics for numerical price columns:
           Min Price      Max Price    Modal Price
count   23093.000000   23093.000000   23093.000000
mean     4187.077045    4976.034260    4602.917742
std      5472.783385    6277.308057    5843.822711
min         0.000000       0.000000       0.830000
25%      1750.000000    2000.000000    1955.000000
50%      2725.000000    3400.000000    3000.000000
75%      5000.000000    6000.000000    5500.000000
max    223500.000000  227500.000000  225500.000000
```

```
categorical_cols = ['State', 'District', 'Market', 'Commodity', 'Variety', 'Grade']

print("Analysis of Categorical Columns:")
for col in categorical_cols:
    print(f"\nColumn: {col}")
    print(f"Number of unique values: {df[col].nunique()}")
    print(f"Top 5 most frequent values:\n{df[col].value_counts().head()}")
```

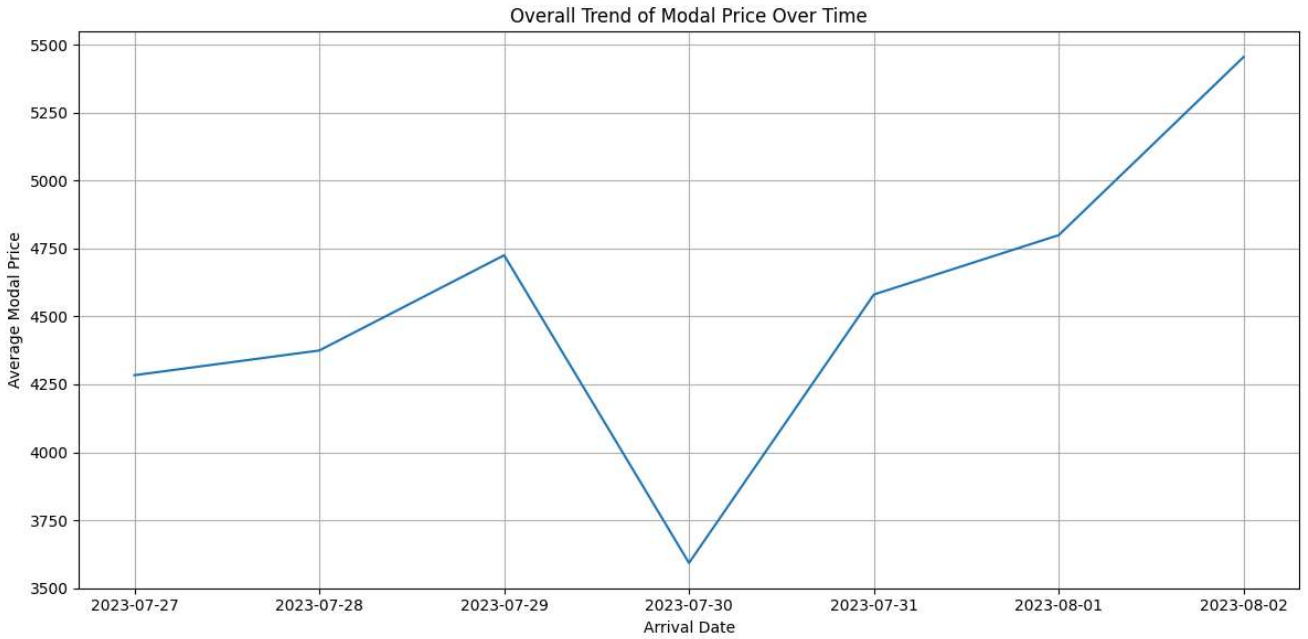
```
Gujarat          1782
Maharashtra      1770
Punjab           1406
Name: count, dtype: int64

Column: District
Number of unique values: 403
Top 5 most frequent values:
District
Pune            588
Ernakulam       402
Kangra          379
Bulandshahar    328
Alappuzha       314
Name: count, dtype: int64

Column: Market
Number of unique values: 1289
Top 5 most frequent values:
Market
Pune                    195
Palampur                185
Pune(Moshi)             175
Jafarganj               149
Kangra(Jaisinghpur)     131
Name: count, dtype: int64

Column: Commodity
Number of unique values: 234
Top 5 most frequent values:
Commodity
Potato                   1205
Onion                    1134
Brinjal                  1033
Green Chilli              931
```

```
Green Chilly     646
Red              506
Bhindi           502
Bottle Gourd     481
Name: count, dtype: int64

Column: Grade
Number of unique values: 4
Top 5 most frequent values:
Grade
FAQ       20500
Medium     2182
Large       371
Small        40
Name: count, dtype: int64
```

```
print("Date Range of the Dataset:")
print(f"Earliest Date: {df['Arrival_Date'].min()}")
print(f"Latest Date: {df['Arrival_Date'].max()}")
```

```
Date Range of the Dataset:
Earliest Date: 2023-07-27 00:00:00
Latest Date: 2023-08-02 00:00:00
```

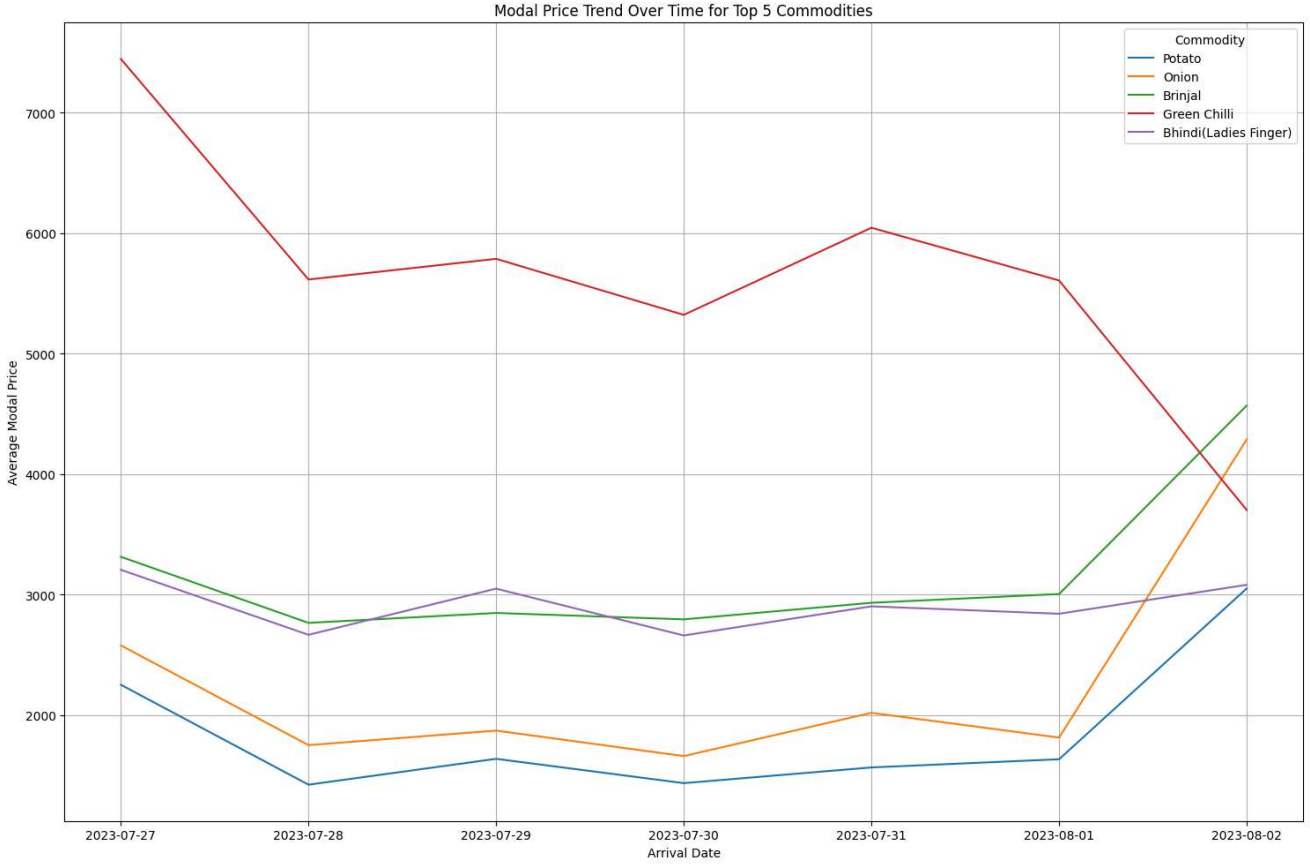
```
import matplotlib.pyplot as plt
import seaborn as sns

# Aggregate 'Modal Price' by 'Arrival_Date'
daily_avg_price = df.groupby('Arrival_Date')['Modal Price'].mean().reset_index()

plt.figure(figsize=(12, 6))
sns.lineplot(x='Arrival_Date', y='Modal Price', data=daily_avg_price)
plt.title('Overall Trend of Modal Price Over Time')
plt.xlabel('Arrival Date')
plt.ylabel('Average Modal Price')
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
top_commodities = df['Commodity'].value_counts().head(5).index.tolist()

print("Top 5 Commodities:", top_commodities)

plt.figure(figsize=(15, 10))
for commodity in top_commodities:
    commodity_df = df[df['Commodity'] == commodity].copy()
    commodity_daily_avg = commodity_df.groupby('Arrival_Date')['Modal Price'].mean().reset_index()
    sns.lineplot(x='Arrival_Date', y='Modal Price', data=commodity_daily_avg, label=commodity)

plt.title('Modal Price Trend Over Time for Top 5 Commodities')
```

```
plt.xlabel('Arrival Date')
plt.ylabel('Average Modal Price')
plt.legend(title='Commodity')
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Top 5 Commodities: ['Potato', 'Onion', 'Brinjal', 'Green Chilli', 'Bhindi(Ladies Finger)']
```



Feature Engineering and Selection Discussion

```
from sklearn.preprocessing import LabelEncoder

categorical_features = ['State', 'District', 'Market', 'Commodity', 'Variety', 'Grade', 'Year', 'Month', 'DayOfWeek']

# Create a copy to avoid SettingWithCopyWarning and to work on processed data
processed_df = df.copy()

# Initialize a dictionary to store label encoders for inverse transformation later if needed
label_encoders = {}

print("Encoding categorical features...")
for feature in categorical_features:
    le = LabelEncoder()
    processed_df[feature] = le.fit_transform(processed_df[feature])
    label_encoders[feature] = le
    print(f"  - Encoded '{feature}'. Number of unique values before: {len(le.classes_)}, after: {processed_df[feature].nunique(

print("Categorical features encoded successfully.")
print("First 5 rows of the DataFrame with encoded categorical features:")
print(processed_df[categorical_features].head())
```

```
Encoding categorical features...
  - Encoded 'State'. Number of unique values before: 27, after: 27
  - Encoded 'District'. Number of unique values before: 403, after: 403
  - Encoded 'Market'. Number of unique values before: 1289, after: 1289
  - Encoded 'Commodity'. Number of unique values before: 234, after: 234
  - Encoded 'Variety'. Number of unique values before: 452, after: 452
  - Encoded 'Grade'. Number of unique values before: 4, after: 4
```

```
  - Encoded 'Year'. Number of unique values before: 1, after: 1
  - Encoded 'Month'. Number of unique values before: 2, after: 2
  - Encoded 'DayOfWeek'. Number of unique values before: 7, after: 7
Categorical features encoded successfully.
First 5 rows of the DataFrame with encoded categorical features:
   State  District  Market  Commodity  Variety  Grade  Year  Month  DayOfWeek
0      6        12     309         22       57      0     0      0          3
1      6        12     309         28      311      0     0      0          3
2      6        12     309         32       81      0     0      0          3
3      6        12     309         40       87      0     0      0          3
4      6        12     309         60      112      0     0      0          3
```

```
from sklearn.preprocessing import MinMaxScaler

numerical_features = ['Min Price', 'Max Price', 'Modal Price']

# Initialize MinMaxScaler
scaler = MinMaxScaler()

print("Scaling numerical features...")
# Fit and transform numerical features and store the scaled data back into processed_df
processed_df[numerical_features] = scaler.fit_transform(processed_df[numerical_features])

print("Numerical features scaled successfully.")
print("First 5 rows of the DataFrame with scaled numerical features:")
print(processed_df[numerical_features].head())
```

```
Scaling numerical features...
Numerical features scaled successfully.
First 5 rows of the DataFrame with scaled numerical features:
   Min Price  Max Price  Modal Price
0   0.018345   0.019780     0.019287
1   0.009843   0.013187     0.010861
2   0.010515   0.013187     0.011970
3   0.031320   0.032967     0.032147
4   0.037584   0.039560     0.039243
```

```
import numpy as np

# Ensure 'Arrival_Date' is in datetime format before sorting
# It was converted at the start of EDA but copied without that conversion in processed_df
processed_df['Arrival_Date'] = pd.to_datetime(df['Arrival_Date'], format='%Y-%m-%d')

# Define all features that will be used for sequence generation
all_features = numerical_features + categorical_features + ['Arrival_Date']

# Create a new DataFrame with only the relevant features, already scaled and encoded
# processed_df already contains these, just re-ordering and ensuring 'Arrival_Date' is correct
processed_df = processed_df[all_features]

# Define grouping columns for sorting
grouping_columns = ['State', 'District', 'Market', 'Commodity', 'Variety', 'Grade']

# Sort the DataFrame by grouping columns and then by 'Arrival_Date'
processed_df = processed_df.sort_values(by=grouping_columns + ['Arrival_Date']).reset_index(drop=True)

print("DataFrame prepared and sorted for sequence generation.")
print("First 5 rows of the sorted DataFrame:")
print(processed_df.head())
print("\nLast 5 rows of the sorted DataFrame:")
print(processed_df.tail())
```

```
DataFrame prepared and sorted for sequence generation.
First 5 rows of the sorted DataFrame:
   Min Price  Max Price  Modal Price  State  District  Market  Commodity  \
0   0.035794   0.043956     0.039908      0       278     243         14
1   0.089485   0.096703     0.093123      0       278     243         23
2   0.049217   0.057143     0.053212      0       278     243         27
3   0.062640   0.070330     0.066515      0       278     243         28
4   0.156600   0.175824     0.155208      0       278     243         92

   Variety  Grade  Year  Month  DayOfWeek Arrival_Date
0      311      2     0      0          0   2023-07-31
1      311      0     0      0          0   2023-07-31
2      311      0     0      0          0   2023-07-31
3      311      0     0      0          0   2023-07-31
```

```
4      311      0     0      0          0   2023-07-31

Last 5 rows of the sorted DataFrame:
       Min Price  Max Price  Modal Price  State  District  Market  Commodity  \
23088   0.015660   0.016264     0.015961     26       384     987        182
23089   0.015660   0.016264     0.015961     26       384     987        182
23090   0.010291   0.010549     0.010418     26       384     987        227
23091   0.010291   0.010549     0.010418     26       384     987        227
23092   0.010291   0.010549     0.010418     26       384     987        227

       Variety  Grade  Year  Month  DayOfWeek Arrival_Date
23088      155      0     0      0          0   2023-07-31
23089      155      0     0      1          1   2023-08-01
23090      256      0     0      0          4   2023-07-28
23091      256      0     0      0          0   2023-07-31
23092      256      0     0      1          1   2023-08-01
```

```
sequence_length = 2 # A small sequence length due to the limited date range

X_sequences = []
y_targets = []
y_target_dates = [] # To store dates corresponding to y_targets for chronological split

# Define the features to be used in X (excluding the target 'Modal Price' and 'Arrival_Date' for X itself)
x_features = [col for col in processed_df.columns if col not in ['Modal Price', 'Arrival_Date']]

# Group by unique identifier columns
for name, group in processed_df.groupby(grouping_columns):
    # Sort group by date to ensure proper sequence generation
    group = group.sort_values(by='Arrival_Date')

    if len(group) > sequence_length:
        for i in range(len(group) - sequence_length):
            # Input sequence (X)
            X_seq = group.iloc[i : i + sequence_length][x_features].values
            X_sequences.append(X_seq)

            # Target value (y) - the Modal Price of the next day
            y_target = group.iloc[i + sequence_length]['Modal Price']
            y_targets.append(y_target)

            # Target date for splitting
            y_target_date = group.iloc[i + sequence_length]['Arrival_Date']
            y_target_dates.append(y_target_date)

print(f"Generated {len(X_sequences)} sequences with a sequence length of {sequence_length}.")
print(f"First X_sequence example (shape: {X_sequences[0].shape}):\n{X_sequences[0]}")
print(f"First y_target example: {y_targets[0]}")
print(f"First y_target_date example: {y_target_dates[0]}")
```

```
Generated 4316 sequences with a sequence length of 2.
First X_sequence example (shape: (2, 11)):
[[2.80492170e-02 3.25978022e-02 1.00000000e+00 8.50000000e+01
  3.02000000e+02 9.80000000e+01 2.56000000e+02 0.00000000e+00
  0.00000000e+00 0.00000000e+00 4.00000000e+00]
 [2.28187919e-02 3.33582418e-02 1.00000000e+00 8.50000000e+01
  3.02000000e+02 9.80000000e+01 2.56000000e+02 0.00000000e+00
  0.00000000e+00 0.00000000e+00 0.00000000e+00]]
First y_target example: 0.03130020389875493
First y_target_date example: 2023-08-01 00:00:00
```

```
X_sequences = np.array(X_sequences)
y_targets = np.array(y_targets)
y_target_dates = np.array(y_target_dates)

# Define the split date
split_date = pd.to_datetime('2023-08-01')

# Create boolean masks for training and testing data
train_mask = y_target_dates < split_date
test_mask = y_target_dates >= split_date

# Split the data into training and testing sets
X_train = X_sequences[train_mask]
y_train = y_targets[train_mask]
X_test = X_sequences[test_mask]
y_test = y_targets[test_mask]
```

```
print(f"Total sequences: {len(X_sequences)}")
print(f"Training sequences: {len(X_train)}")
print(f"Testing sequences: {len(X_test)}")
print(f"X_train shape: {X_train.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_test shape: {y_test.shape}")
```

```
Total sequences: 4316
Training sequences: 1338
Testing sequences: 2978
X_train shape: (1338, 2, 11)
y_train shape: (1338,)
X_test shape: (2978, 2, 11)
y_test shape: (2978,)
```

Build and Train LSTM Model

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

# Get the input shape for the LSTM layer
# X_train shape is (samples, timesteps, features)
input_shape = (X_train.shape[1], X_train.shape[2])

# Define the LSTM model architecture
lstm_model = Sequential([
    LSTM(units=50, activation='relu', input_shape=input_shape, return_sequences=False),
    Dropout(0.2),
    Dense(units=1)
])

# Compile the model
lstm_model.compile(optimizer='adam', loss='mean_squared_error')

# Display model summary
print("LSTM Model Summary:")
lstm_model.summary()

# Train the LSTM model
print("\nTraining LSTM model...")
history_lstm = lstm_model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_test, y_test), verbose=1)

print("LSTM model trained successfully.")
```

```
LSTM Model Summary:
/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199: UserWarning: Do not pass an `input_shape`/`input_dim
  super().__init__(**kwargs)
Model: "sequential"
```

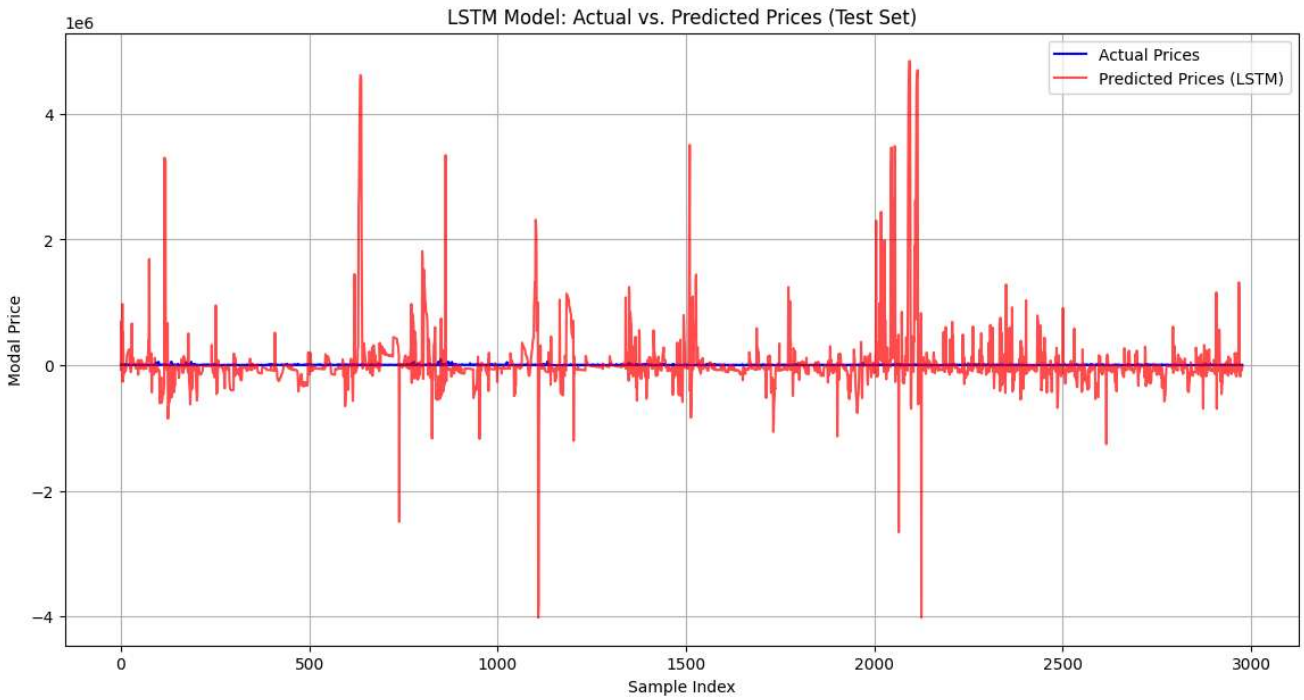
Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 50)	12,400
dropout (Dropout)	(None, 50)	0
dense (Dense)	(None, 1)	51

```
 Total params: 12,451 (48.64 KB)
 Trainable params: 12,451 (48.64 KB)
 Non-trainable params: 0 (0.00 B)

Training LSTM model...
Epoch 1/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - loss: 3663.9565 - val_loss: 262.9333
Epoch 2/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 409.2064 - val_loss: 105.4532
Epoch 3/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 234.7901 - val_loss: 58.0410
Epoch 4/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 179.4039 - val_loss: 37.5403
Epoch 5/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 120.5920 - val_loss: 26.5849
Epoch 6/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 80.3379 - val_loss: 23.9871
Epoch 7/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 57.8170 - val_loss: 21.6887
Epoch 8/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 52.1391 - val_loss: 19.3044
Epoch 9/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 35.7302 - val_loss: 18.6674
Epoch 10/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 36.6473 - val_loss: 18.0005
Epoch 11/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 24.9449 - val_loss: 15.6684
Epoch 12/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 29.2809 - val_loss: 10.9987
Epoch 13/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 25.0844 - val_loss: 8.8187
Epoch 14/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 15.6751 - val_loss: 7.8036
Epoch 15/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 13.3736 - val_loss: 4.9550
Epoch 16/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 7.5599 - val_loss: 4.5537
Epoch 17/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 9.8960 - val_loss: 4.1625
Epoch 18/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 9.1573 - val_loss: 3.2924
Epoch 19/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 9.2874 - val_loss: 3.4646
Epoch 20/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 7.4460 - val_loss: 5.0075
LSTM model trained successfully.
```

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Make predictions on the test set
y_pred_lstm = lstm_model.predict(X_test)

# Inverse transform the predictions and actual values
# We need to create a dummy array for inverse_transform as scaler expects all numerical features
# The numerical_features list contains ['Min Price', 'Max Price', 'Modal Price']
# The 'Modal Price' is the last one, so its index is 2.

dummy_y_pred_lstm = np.zeros((len(y_pred_lstm), len(numerical_features)))
dummy_y_pred_lstm[:, 2] = y_pred_lstm.flatten()
y_pred_lstm_original = scaler.inverse_transform(dummy_y_pred_lstm)[:, 2]

dummy_y_test = np.zeros((len(y_test), len(numerical_features)))
dummy_y_test[:, 2] = y_test.flatten()
y_test_original = scaler.inverse_transform(dummy_y_test)[:, 2]
```

```
# Calculate evaluation metrics
mse_lstm = mean_squared_error(y_test_original, y_pred_lstm_original)
rmse_lstm = np.sqrt(mse_lstm)
mae_lstm = mean_absolute_error(y_test_original, y_pred_lstm_original)
r2_lstm = r2_score(y_test_original, y_pred_lstm_original)

print("\nLSTM Model Evaluation:")
print(f"Mean Squared Error (MSE): {mse_lstm:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse_lstm:.2f}")
print(f"Mean Absolute Error (MAE): {mae_lstm:.2f}")
print(f"R-squared (R2): {r2_lstm:.2f}")

# Visualize predictions vs. actuals
plt.figure(figsize=(14, 7))
plt.plot(y_test_original, label='Actual Prices', color='blue')
plt.plot(y_pred_lstm_original, label='Predicted Prices (LSTM)', color='red', alpha=0.7)
plt.title('LSTM Model: Actual vs. Predicted Prices (Test Set)')
plt.xlabel('Sample Index')
plt.ylabel('Modal Price')
plt.legend()
plt.grid(True)
plt.show()
```

```
94/94 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step

LSTM Model Evaluation:
Mean Squared Error (MSE): 254631647958.29
Root Mean Squared Error (RMSE): 504610.39
Mean Absolute Error (MAE): 201487.52
R-squared (R2): -10664.31
```



Build and Train GRU Model

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dense, Dropout

# Get the input shape for the GRU layer
# X_train shape is (samples, timesteps, features)
input_shape = (X_train.shape[1], X_train.shape[2])

# Define the GRU model architecture
gru_model = Sequential([
```

```
    GRU(units=50, activation='relu', input_shape=input_shape, return_sequences=False),
    Dropout(0.2),
    Dense(units=1)
])

# Compile the model
gru_model.compile(optimizer='adam', loss='mean_squared_error')

# Display model summary
print("GRU Model Summary:")
gru_model.summary()

# Train the GRU model
print("\nTraining GRU model...")
history_gru = gru_model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_test, y_test), verbose=1)

print("GRU model trained successfully.")
```

```
GRU Model Summary:
/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199: UserWarning: Do not pass an `input_shape`/`input_dim
  super().__init__(**kwargs)
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
gru (GRU)	(None, 50)	9,450
dropout_1 (Dropout)	(None, 50)	0
dense_1 (Dense)	(None, 1)	51

```
 Total params: 9,501 (37.11 KB)
 Trainable params: 9,501 (37.11 KB)
 Non-trainable params: 0 (0.00 B)

Training GRU model...
Epoch 1/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 5s 23ms/step - loss: 1919.1123 - val_loss: 99.0216
Epoch 2/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 873.6498 - val_loss: 41.8150
Epoch 3/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 409.0784 - val_loss: 20.4967
Epoch 4/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 237.6502 - val_loss: 19.6517
Epoch 5/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 189.3247 - val_loss: 10.4812
Epoch 6/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 111.2864 - val_loss: 8.7401
Epoch 7/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 85.0025 - val_loss: 4.8752
Epoch 8/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 55.9593 - val_loss: 4.3388
Epoch 9/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 39.9442 - val_loss: 3.2209
Epoch 10/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 25.8857 - val_loss: 2.8578
Epoch 11/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 21.4107 - val_loss: 3.3082
Epoch 12/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 18.4619 - val_loss: 3.2531
Epoch 13/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 13.2851 - val_loss: 1.5217
Epoch 14/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 12.7580 - val_loss: 1.7768
Epoch 15/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 9.7026 - val_loss: 1.7495
Epoch 16/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 10.8264 - val_loss: 1.3865
Epoch 17/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 5.5466 - val_loss: 1.2731
Epoch 18/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 6.4698 - val_loss: 1.5351
Epoch 19/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 5.6272 - val_loss: 1.3482
Epoch 20/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 4.3983 - val_loss: 1.2425
GRU model trained successfully.
```

Build and Train Transformer Model

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Dropout, Layer, GlobalAveragePooling1D, MultiHeadAttention, LayerNormalizat
from tensorflow.keras.models import Model
import numpy as np

# 1. Define MultiHeadSelfAttention class (using Keras's built-in MultiHeadAttention)
class MultiHeadSelfAttention(Layer):
    def __init__(self, embed_dim, num_heads=8, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        if embed_dim % num_heads != 0:
            raise ValueError(
                f"embedding dimension = {embed_dim} should be divisible by number of heads = {num_heads}"
            )
        self.attention_layers = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim // num_heads)
        self.add_norm = LayerNormalization(epsilon=1e-6)

    def call(self, inputs):
        attn_output = self.attention_layers(inputs, inputs, inputs)
        return self.add_norm(inputs + attn_output)

# 2. Define TransformerBlock class
class TransformerBlock(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.ff_dim = ff_dim
        self.rate = rate

        self.att = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim // num_heads)
        self.ffn = tf.keras.Sequential(
            [
                Dense(ff_dim, activation="relu"),
                Dense(embed_dim),
            ]
        )
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(rate)
        self.dropout2 = Dropout(rate)

    def call(self, inputs, training=None):
        # Multi-head self-attention
        attn_output = self.att(inputs, inputs)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)

        # Feed-forward network
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)

# Input shape for the Transformer model
input_shape = (X_train.shape[1], X_train.shape[2])

# Hyperparameters for the Transformer model
embed_dim = X_train.shape[2]  # Embedding dimension, should be equal to the number of features
num_heads = 4                 # Number of attention heads
ff_dim = 32                   # Hidden layer size in feed forward network inside transformer

# Build the Transformer model
inputs = Input(shape=input_shape)
x = inputs

# Add one or more Transformer blocks
x = TransformerBlock(embed_dim, num_heads, ff_dim)(x)
# You can add more blocks for deeper models, e.g., x = TransformerBlock(embed_dim, num_heads, ff_dim)(x)

x = GlobalAveragePooling1D()(x)
x = Dropout(0.2)(x)
outputs = Dense(1)(x)

transformer_model = Model(inputs=inputs, outputs=outputs)
```

```
# Compile the Transformer model
transformer_model.compile(optimizer='adam', loss='mean_squared_error')

# Display model summary
print("Transformer Model Summary:")
transformer_model.summary()

# Train the Transformer model
print("\nTraining Transformer model...")
history_transformer = transformer_model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_test, y_test), ver

print("Transformer model trained successfully.")
```

```
Transformer Model Summary:
Model: "functional_3"
```

Layer (type)	Output Shape	Param #
input_layer_4 (InputLayer)	(None, 2, 11)	0
transformer_block_2 (TransformerBlock)	(None, 2, 11)	1,178
global_average_pooling1d (GlobalAveragePooling1D)	(None, 11)	0
dropout_9 (Dropout)	(None, 11)	0
dense_8 (Dense)	(None, 1)	12

```
 Total params: 1,190 (4.65 KB)
 Trainable params: 1,190 (4.65 KB)
 Non-trainable params: 0 (0.00 B)

Training Transformer model...
Epoch 1/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 6s 15ms/step - loss: 0.6436 - val_loss: 0.0564
Epoch 2/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.1889 - val_loss: 0.0208
Epoch 3/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.1067 - val_loss: 0.0108
Epoch 4/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 0.0604 - val_loss: 0.0061
Epoch 5/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 0.0434 - val_loss: 0.0041
Epoch 6/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0313 - val_loss: 0.0026
Epoch 7/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - loss: 0.0288 - val_loss: 0.0020
Epoch 8/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 0.0191 - val_loss: 0.0018
Epoch 9/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0165 - val_loss: 0.0019
Epoch 10/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0126 - val_loss: 0.0017
Epoch 11/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0108 - val_loss: 0.0017
Epoch 12/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0110 - val_loss: 0.0015
Epoch 13/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0100 - val_loss: 0.0014
Epoch 14/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0091 - val_loss: 0.0013
Epoch 15/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0087 - val_loss: 0.0013
Epoch 16/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0078 - val_loss: 0.0012
Epoch 17/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0075 - val_loss: 0.0015
Epoch 18/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0067 - val_loss: 0.0014
Epoch 19/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0066 - val_loss: 0.0015
Epoch 20/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0064 - val_loss: 0.0014
Transformer model trained successfully.
```

Model Evaluation

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Make predictions on the test set using the Transformer model
y_pred_transformer = transformer_model.predict(X_test)

# Inverse transform the predictions and actual values to their original scale
# We need to create a dummy array for inverse_transform as scaler expects all numerical features
# The 'Modal Price' is the last one in numerical_features, so its index is 2.
dummy_y_pred_transformer = np.zeros((len(y_pred_transformer), len(numerical_features)))
dummy_y_pred_transformer[:, 2] = y_pred_transformer.flatten()
y_pred_transformer_original = scaler.inverse_transform(dummy_y_pred_transformer)[:, 2]

dummy_y_test_transformer = np.zeros((len(y_test), len(numerical_features)))
dummy_y_test_transformer[:, 2] = y_test.flatten()
y_test_original_transformer = scaler.inverse_transform(dummy_y_test_transformer)[:, 2]

# Calculate evaluation metrics for Transformer model
mse_transformer = mean_squared_error(y_test_original_transformer, y_pred_transformer_original)
rmse_transformer = np.sqrt(mse_transformer)
mae_transformer = mean_absolute_error(y_test_original_transformer, y_pred_transformer_original)
r2_transformer = r2_score(y_test_original_transformer, y_pred_transformer_original)

print("\nTransformer Model Evaluation:")
print(f"Mean Squared Error (MSE): {mse_transformer:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse_transformer:.2f}")
print(f"Mean Absolute Error (MAE): {mae_transformer:.2f}")
print(f"R-squared (R2): {r2_transformer:.2f}")

# Visualize predictions vs. actuals for Transformer model
plt.figure(figsize=(14, 7))
plt.plot(y_test_original_transformer, label='Actual Prices', color='blue')
plt.plot(y_pred_transformer_original, label='Predicted Prices (Transformer)', color='purple', alpha=0.7)
plt.title('Transformer Model: Actual vs. Predicted Prices (Test Set)')
plt.xlabel('Sample Index')
plt.ylabel('Modal Price')
plt.legend()
plt.grid(True)
plt.show()
```

```
94/94 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step
```

Model Evaluation Summary and Comparison

```
Transformer Model Evaluation:
Mean Squared Error (MSE): 69738384.85
Root Mean Squared Error (RMSE): 8350.35
Mean Absolute Error (MAE): 4068.57
R-squared (R2): -1.92
```

Below is a summary of the evaluation metrics (Mean Squared Error - MSE, Root Mean Squared Error - RMSE, Mean Absolute Error - MAE, and R-squared - R2) for the LSTM, GRU, and Transformer models on the test set, along with key observations.

LSTM Model Evaluation:

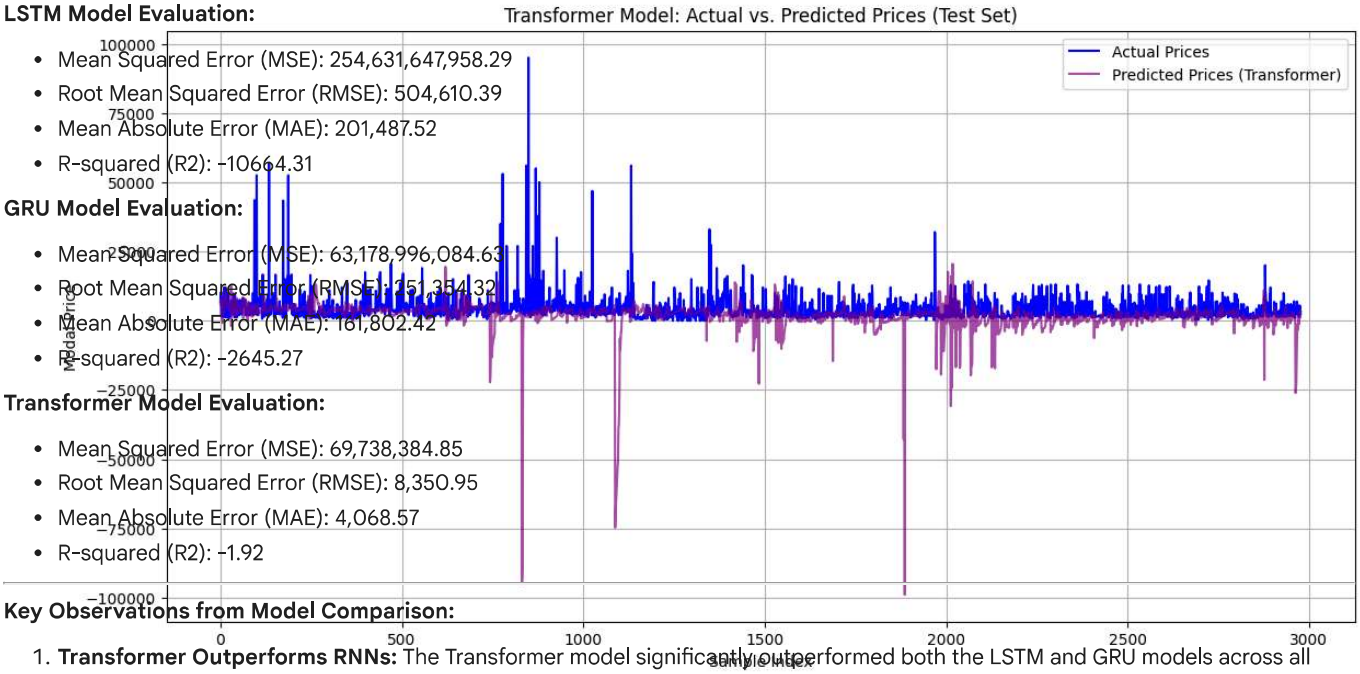
- Mean Squared Error (MSE): 254,631,647,958.29
- Root Mean Squared Error (RMSE): 504,610.39
- Mean Absolute Error (MAE): 201,487.52
- R-squared (R2): -10664.31

GRU Model Evaluation:

- Mean Squared Error (MSE): 63,178,996,084.63
- Root Mean Squared Error (RMSE): 251,354.32
- Mean Absolute Error (MAE): 181,802.42
- R-squared (R2): -2645.27

Transformer Model Evaluation:

- Mean Squared Error (MSE): 69,738,384.85
- Root Mean Squared Error (RMSE): 8,350.95
- Mean Absolute Error (MAE): 4,068.57
- R-squared (R2): -1.92



Key Observations from Model Comparison:

1. **Transformer Outperforms RNNs:** The Transformer model significantly outperformed both the LSTM and GRU models across all metrics (MSE, RMSE, MAE, R2). Its errors are orders of magnitude lower, and its R2 score, while still negative, is substantially closer to zero, indicating it captures some predictive patterns more effectively than the recurrent models.
2. **GRU Better than LSTM:** The GRU model, with its simpler architecture and fewer parameters, performed better than the LSTM model. This suggests that for this specific dataset with a very limited temporal context (sequence length of 2), the added complexity of LSTM's three gates and cell state might have been an overkill or harder to train effectively.
3. **Overall Poor Predictive Power:** Despite the Transformer's relative success, all models still yielded negative R-squared values. An R2 score less than zero indicates that the model performs worse than simply predicting the mean of the target variable. This highlights a fundamental limitation: the extremely short date range of the dataset (only 7 days) provides insufficient historical information for any deep learning model to learn meaningful trends, seasonality, or long-term dependencies essential for accurate time-series price prediction.