AI & Finance: Launching Your Journey in LSTM Stock Predictions

Source: https://scribe.rip/@bnhminh_38309/a-beginners-guide-to-predicting-stock-prices-with-lstm-networks-278070252731

In this article, we're going to learn about a neural network model called LSTM that can help us guess the future of stock prices. It's like playing a game where the paths change every day, and we have to guess the right path. We'll look back at the last 60 days of what the stock prices were and use that information to make our guesses. By the end, We'll check how good the program's guesses are and learn how we can make them better.

✍️ Figure 1: Robots Analyzing Finances with Holographic Tools | Credit: DALL·E

Why Guess Stock Prices?

Guessing stock prices is like trying to predict the weather. It's really hard because there are so many things that can change the outcome. But if we can make even a small improvement in our predictions, it can help a lot in making better decisions about buying or selling stocks.

The Challenge

The stock market is influenced by so many things — from big world events to how well companies are doing. This makes predicting stock prices tricky. It's like trying to put together a puzzle without having all the pieces.

How We Use LSTM

LSTM stands for Long Short-Term Memory. It's a type of algorithm that's good at noticing patterns over time. This makes it perfect for looking at past stock prices and guessing what will happen next. For more understanding, go to this post → https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

About The Data

Our analysis focuses on the daily closing prices of the Consumer Discretionary sector (XLY index) from 1998 to 2024, sourced from Yahoo Finance. This sector includes companies whose products are not essential but desired when consumers have discretionary income, such as electronics, leisure, and automobiles. The dataset, saved as `master_data.csv`, serves as our basis for evaluating the LSTM's performance in stock market prediction.

Step 1: Setting Up

First things first, we need to set up our environment. This means getting our computer ready with the tools we'll need. Here's how we do it:

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,
mean_absolute_percentage_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.callbacks import EarlyStopping,
ModelCheckpoint
from tensorflow.keras.preprocessing.sequence import
TimeseriesGenerator
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import math
import random
```

Step 2: Preparing the data

Next, we'll load our dataset and make sure it's ready for the LSTM network. We focus on the closing prices, as they tell us how much stocks were worth at the end of each trading day.

```
filepath = "/kaggle/input/master-data-csv"
data = pd.read_csv(filepath, index_col=0,
parse_dates=True)
data.index = pd.to_datetime(data.index,
utc=True).tz_localize(None)
data = data.iloc[:, 0:1]  # Default to the second
column (index 1) if no column specified.
```

To make sure our experiment can be repeated with the same results, we set a random seed. Setting a random seed works like giving that special code to the computer. It makes sure that every time you run your program, the "random" parts happen in the same way.

```
# Set a random seed for reproducibility
np.random.seed(66)
random.seed(66)
tf.random.set_seed(66)
```

Then, we split our data into parts: one for training our tool, one for checking how well it's learning, and one for testing its predictions. We also change the data a bit (normalize it) so it's easier for the LSTM to understand.

```
# Split the dataset
train_end_year = 2020
valid_start_year = 2021
valid_end_year = 2022
test_start_year = 2023

train_data = data[data.index.year <= train_end_year]
valid_data = data[(data.index.year >= valid_s-
tart_year) & (data.index.year <= valid_end_year)]
test_data = data[data.index.year >= test_start_year]

# Scale the data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_train = scaler.fit_transform(train_data)
scaled_valid = scaler.transform(valid_data)
scaled_test = scaler.transform(test_data)
```

We use Time Series Generator to make sure our data is in the right format for the LSTM:

```
time_steps = 60
batch_size = 64

train_generator = TimeseriesGenerator(scaled_train,
scaled_train, length=time_steps, batch_size=batch_-
size)
valid_generator = TimeseriesGenerator(scaled_valid,
scaled_valid, length=time_steps, batch_size=batch_-
size)
test_generator = TimeseriesGenerator(scaled_test,
scaled_test, length=time_steps,
batch_size=batch_size)
```

These generators help the program learn to predict stock prices by studying 60 days of stock prices in groups of 64 at a time.

Step 3: Building and Training Our LSTM Model

Now, we create our LSTM model. Think of this as building a prediction machine. We tell it how to learn from the data and then let it practice with our training set.

```
# Define the LSTM model architecture
model = Sequential([
    LSTM(128, input_shape=(time_steps, 1), return_se-
quences=True),
    LSTM(64, return_sequences=False),
    Dense(32, activation='relu'),
    Dense(1)
])
model.compile(optimizer='adam',
loss='mean_squared_error')
model.summary()
```

Let's break down our model-building process into simpler terms, like constructing a smart helper to forecast stock prices:

First Layer: This part watches 60 days of stock prices with 128 special sensors (LSTM units), each picking up on different patterns. By setting return_sequences= True, it gives 128 detailed observations for each day, keeping the sequence intact.

Second Layer: With 64 sensors, it digs deeper into what the first layer found, summarizing the key insights. With return_sequences=False, it merges all 60 days' insights into one cohesive summary.

Third Layer: This section, equipped with 32 sensors, refines these insights, focusing on what's really important.

Final Layer: Based on everything it's learned, it predicts tomorrow's stock price.

We use 'adam', a smart learning method, to help our model get better at predicting by learning from past mistakes. The 'mean_squared_error' tells us how accurate our predictions are — the closer to zero, the better.

We train our model with the data, using some smart tricks to make it learn better and faster:

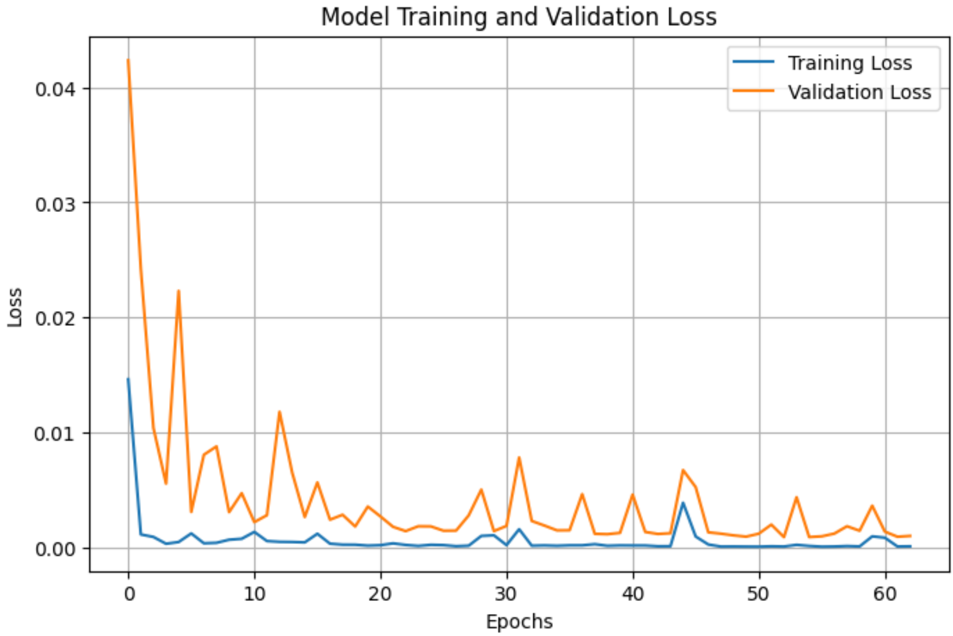
```
# Train the model
epochs = 100
model_path = 'best_model.keras'
callbacks = [
    EarlyStopping(monitor='val_loss', patience=10,
verbose=1),
    ModelCheckpoint(model_path, monitor='val_loss',
save_best_only=True, verbose=1)
]
history = model.fit(train_generator, valida-
tion_data=valid_generator, epochs=epochs, call-
backs=callbacks)
```

When training our model to predict stock prices, we run it through the data 100 times, known as epochs, to enhance its learning. To optimize this process, we employ strategies like EarlyStopping and ModelCheckpoint. EarlyStopping halts training if there's no improvement after 10 tries, preventing unnecessary effort when no progress is being made. ModelCheckpoint, on the other hand, saves the model at its peak performance during training, ensuring we retain the best version regardless of any subsequent declines in accuracy.

Step 4: Evaluating Model Performance

Plotting training history

```
# Plot training history
plt.figure(figsize=(8, 5))
plt.plot(history.history['loss'], label='Training
Loss')
plt.plot(history.history['val_loss'], label='Valida-
tion Loss')
plt.title('Model Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



✍️ Figure 2: Epoch-wise Training and Validation Loss for LSTM Stock Price Prediction Model | Credit: Created by Ryan → https://medium.com/@bnhminh_38309?source=---two_column_layout_sidebar--------------------------------

The graph shows that the machine learning model quickly learned to predict well and kept improving without focusing too much on the specific examples it was trained on (overfitting). Both loss curves remain close throughout the training epochs, demonstrating that the model generalizes well to unseen data. The convergence of training and validation loss suggests that the model has reached an optimal point, and the early stopping mechanism likely ceased training at an appropriate juncture to prevent overfitting. This graph signifies a well-tuned model that strikes a balance between learning the underlying patterns in the data and maintaining the ability to generalize to new data.

```
# Load the best model and evaluate
best_model = tf.keras.models.load_model('best_mod-
el.keras')

# Utility function to predict and inverse transform
predictions and actuals
def predict_and_inverse(generator, model, scaler):
    predictions = model.predict(generator)
    predictions_inverse = scaler.inverse_trans-
form(predictions)

    actuals = []
    for i in range(len(generator)):
        _, y = generator[i]
        actuals.append(y)
    actuals = np.concatenate(actuals, axis=0)
    actuals_inverse = scaler.inverse_transform(actu-
als)

    return predictions_inverse, actuals_inverse

# Evaluate for all three datasets using the best mod-
el
train_predictions_inverse, train_actuals_inverse =
predict_and_inverse(train_generator, best_model,
scaler)
valid_predictions_inverse, valid_actuals_inverse =
predict_and_inverse(valid_generator, best_model,
scaler)
test_predictions_inverse, test_actuals_inverse = pre-
dict_and_inverse(test_generator, best_model, scaler)

# Calculate and print performance metrics for the
test dataset
test_rmse = math.sqrt(mean_squared_error(test_actual-
```

```
s_inverse, test_predictions_inverse))
test_mape = mean_absolute_percentage_error(test_actu-
als_inverse, test_predictions_inverse)

print(f"Test RMSE: {test_rmse}")
print(f"Test MAPE: {test_mape}")

#Test RMSE: 2.881858075912953
#Test MAPE: 0.014588268594683193
```

When we train a model to predict stock prices, we first scale down these prices to make it easier for the model to learn. Think of it as converting all prices into a common scale, like turning different currencies into dollars to compare them easily. After the model makes predictions, these are also in this scaled-down format. To understand these predictions in real-world terms (like actual stock prices), we need to convert them back to their original form, just like exchanging dollars back to local currency. This step is called inverse transformation. It's crucial because it allows us to accurately measure how good the model is by comparing its predictions with the real stock prices using metrics like Root Mean Squared Error - RMSE (how far off our predictions are, on average) and Mean Absolute Percentage Error - MAPE (what percentage of the price our predictions are off by), making sure we're evaluating the model's performance in a way that makes sense in the real world.

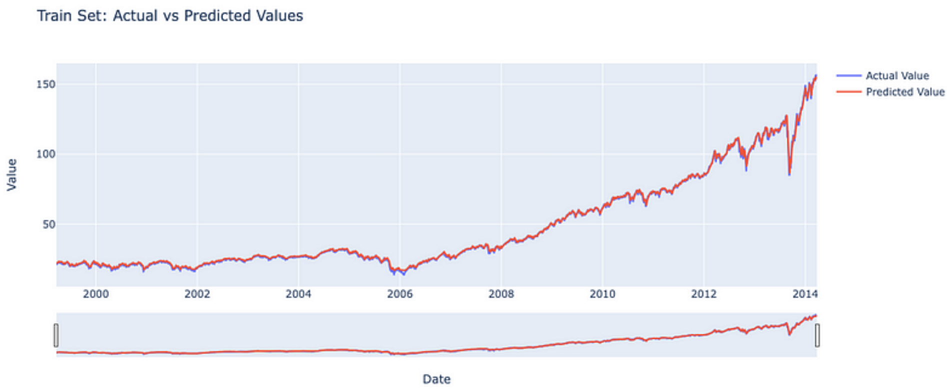
Given the statistical summary of the XLY index, with a mean stock price of about 61.10 and a broad range of values, the Test RMSE of 2.8819 indicates that the model's average prediction error is quite small compared to the overall price scale. This suggests that the predictions are generally close to the actual stock prices. However, in the context of stock prices, which often have small daily changes, a 2% prediction error (Test MAPE of approximately 1.46%)

could be more significant than it first appears. For investors and analysts who track the XLY index, even minor percentage fluctuations can be crucial, as they can influence investment decisions and financial results. Therefore, while the RMSE and MAPE values suggest a high degree of precision, it's essential to consider these errors in light of the usual daily price movements of the stock.

Visualizing Predictions vs. Actual Prices

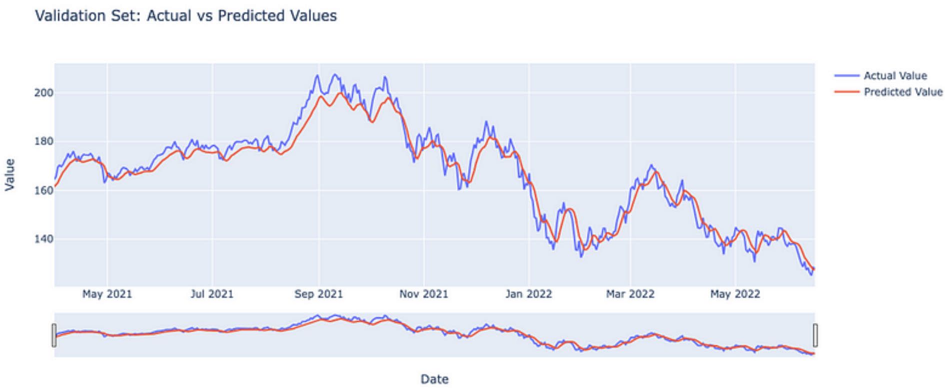
To directly incorporate the generation of date ranges and plotting of results for the training, validation, and test sets, we can use:

```
def plot_prediction_results(data, predictions_in-
verse, actuals_inverse, time_steps, title):
    # Generate date ranges
    dates =
pd.date_range(start=data.index[time_steps], peri-
ods=len(predictions_inverse), freq='D')

    # Plotting
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=dates, y=actuals_in-
verse.flatten(), mode='lines', name='Actual Value'))
    fig.add_trace(go.Scatter(x=dates, y=prediction-
s_inverse.flatten(), mode='lines', name='Predicted
Value'))
    fig.update_layout(title=title,
xaxis_title='Date', yaxis_title='Value', xax-
is_rangeslider_visible=True)
    fig.show()

# Now, we can use the function for all three sets of
data:

plot_prediction_results(train_data, train_prediction-
s_inverse, train_actuals_inverse, 60, "Train Set:
Actual vs Predicted Values")
plot_prediction_results(valid_data, valid_prediction-
s_inverse, valid_actuals_inverse, 60, "Validation
Set: Actual vs Predicted Values")
plot_prediction_results(test_data, test_prediction-
s_inverse, test_actuals_inverse, 60, "Test Set:
Actual vs Predicted Values")
```



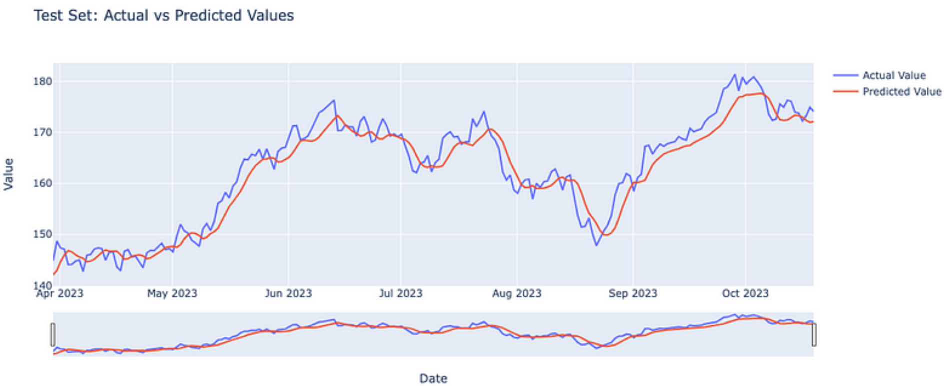
✍️ Figure 3: Comparison of Actual and Predicted Stock Values in Training Dataset | Credit: Created by Ryan → https://medium.com/@bnhminh_38309?source=---two_column_layout_sidebar-----------------------------------

Train Set Graph: This shows the model's performance during the learning phase. The actual and predicted stock values are almost overlapping, which means the model has learned this part of the data very well. It's a tight match, which is what we expect since the model was trained on this data.



✍️ Figure 4: Comparison of Actual and Predicted Stock Values in Validation Dataset | Credit: Created by Ryan → https://medium.com/@bnhminh_38309?source=---two_column_layout_sidebar----------------------------------

Validation Set Graph: Here, we're checking how the model does on data it hasn't learned from but was set aside during training. The lines still follow each other closely, but we see some areas where the model didn't predict the stock's ups and downs perfectly. Overall, though, it's doing a good job of following the trend.



✍️ Figure 5: Comparison of Actual and Predicted Stock Values in Test Dataset | Credit: Created by Ryan → https://medium.com/@bnhminh_38309?source=---two_column_layout_sidebar--------------------------------

Test Set Graph: This is the true test of the model's ability to predict future stock prices. The lines show the actual stock prices versus what the model predicted they would be. The match is still quite good, but there are places where the model misses the mark more than in the train and validation sets. This could be because the test data may have patterns or changes the model hasn't seen before.

The gaps in the test graph could be a sign that the model needs to learn from more examples or different kinds of data to predict these points better. But even with those gaps, the model seems to have a good handle on the overall direction of the stock prices.

Reflections and Next Steps

Our journey into using LSTM networks for predicting stock prices shows us how powerful deep learning can be for finance. However, predicting the stock market is tricky and not always certain because many things can affect stock prices.There's still a lot more to learn and discover in finance and machine learning. We might try incorporating additional data types, experimenting with model configurations, or exploring advanced machine learning techniques.

In short, LSTM networks are a great tool for trying to predict stock prices. By preparing our data, building our model, and checking how well it does, we can get helpful insights for making investment choices. Remember, though, there's always some uncertainty in these predictions. This guide is a starting point, showing the promise of LSTM networks in finance and encouraging us to keep exploring and learning more about predicting stock market trends.

Acknowledgments

Big thanks to Nickolas Discolll → https://scribe.rip/@redeaddiscolll?source=post_page-----5306522f8185-------------------------------- for your inspiring Medium post → https://medium.com/@redeaddiscolll/stock-market-time-series-lstm-failure-5306522f8185, and a special shoutout to my very "close friend", Cato, for his help in reviewing the writing.

Complete Example: LSTM for Stock Price Prediction

Below is a comprehensive script that encapsulates the entire process of predicting stock prices with an LSTM model. This script includes data loading, preprocessing, model building, training, evaluation, and visualization. Feel free to use this as a starting point for your experiments.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,
mean_absolute_percentage_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.callbacks import EarlyStopping,
ModelCheckpoint
from tensorflow.keras.preprocessing.sequence import
TimeseriesGenerator
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import math
import random

class TimeSeriesModel:
    """
    A class to encapsulate the process of loading,
preprocessing, modeling, training,
    and evaluating a time series model using LSTM
neural networks.

    Attributes:
        filepath (str): The path to the CSV file to
be read.
        column_name (str|int, optional): The name or
index of the column to be selected for analysis.
        time_steps (int): The number of time steps to
look back for the LSTM model.
        batch_size (int): Batch size for the genera-
tors.
```

```
        epochs (int): The number of epochs to train
the model.
        model_path (str): The file path to save the
best model during training.
    """
    def __init__(self, filepath, column_index=1,
time_steps=60, batch_size=64, epochs=100,
model_path='best_model.keras'):
        """
        Initializes the TimeSeriesModel with the
specified parameters, including setting the random
seed for reproducibility.
        Note that column_index=1 is the default, as-
suming the first column (index 0) is a datetime index
and the second column (index 1) is the target vari-
able.
        """
        self.filepath = filepath
        self.column_index = column_index  # This is
the change
        self.time_steps = time_steps
        self.batch_size = batch_size
        self.epochs = epochs
        self.model_path = model_path
        np.random.seed(66)
        random.seed(66)
        tf.random.set_seed(66)

    def load_and_preprocess_data(self):
        """
        Load and preprocess time series data from a
CSV file, including handling exceptions for file
reading and data processing.
        """
        try:
            data = pd.read_csv(self.filepath, index_-
```

```
col=0, parse_dates=True)
        except FileNotFoundError:
            raise FileNotFoundError(f"The file at
{self.filepath} was not found.")
        except Exception as e:
            raise Exception(f"An error occurred while
reading the file: {e}")

        try:
            data.index = pd.to_datetime(data.index,
utc=True).tz_localize(None)
            data = data.iloc[:, self.column_index-
1:self.column_index]
        except IndexError:
            raise IndexError(f"Column index {self.-
column_index} is out of bounds for the dataset.")
        except Exception as e:
            raise Exception(f"An error occurred while
processing the data: {e}")

        return data



    def split_data(self, data):
        """
        Split the dataset into training, validation,
and testing sets based on specified years.
        """
        train_data = data[data.index.year <= 2020]
        valid_data = data[(data.index.year >= 2021) &
(data.index.year <= 2022)]
        test_data = data[data.index.year >= 2023]
        return train_data, valid_data, test_data

    def scale_data(self, train_data, valid_data,
test_data):
```

```
        """
        Scale the data using MinMaxScaler.
        """
        self.scaler = MinMaxScaler(feature_range=(0,
1))
        scaled_train = self.scaler.fit_trans-
form(train_data)
        scaled_valid = self.scaler.transform(valid_-
data)
        scaled_test = self.scaler.transform(test_da-
ta)
        return scaled_train, scaled_valid,
scaled_test

    def create_generators(self, scaled_train,
scaled_valid, scaled_test):
        """
        Create TimeseriesGenerators for training,
validation, and testing datasets.
        """
        train_generator = TimeseriesGenera-
tor(scaled_train, scaled_train, length=self.-
time_steps, batch_size=self.batch_size)
        valid_generator = TimeseriesGenera-
tor(scaled_valid, scaled_valid, length=self.-
time_steps, batch_size=self.batch_size)
        test_generator = TimeseriesGenera-
tor(scaled_test, scaled_test, length=self.time_steps,
batch_size=self.batch_size)
        return train_generator, valid_generator,
test_generator

    def define_model(self):
        """
        Define the LSTM model architecture.
        """
```

```
        model = Sequential([
            LSTM(128, input_shape=(self.time_steps,
1), return_sequences=True),
            LSTM(64, return_sequences=False),
            Dense(32, activation='relu'),
            Dense(1)
        ])
        model.compile(optimizer='adam',
loss='mean_squared_error')
        return model

    def train_model(self, model, train_generator,
valid_generator):
        """
        Train the LSTM model with early stopping and
model checkpoint callbacks.
        """
        callbacks = [
            EarlyStopping(monitor='val_loss', pa-
tience=10, verbose=1),
            ModelCheckpoint(self.model_path, moni-
tor='val_loss', save_best_only=True, verbose=1)
        ]
        history = model.fit(train_generator, valida-
tion_data=valid_generator, epochs=self.epochs, call-
backs=callbacks)
        return history

    def evaluate_model(self, model, train_generator,
valid_generator, test_generator):
        """
        Evaluate the trained model on training, vali-
dation, and test datasets.
        """
        def predict_and_inverse(generator):
            predictions = model.predict(generator)
```

```
            predictions_inverse = self.scaler.in-
verse_transform(predictions)  # Use self.scaler
            actuals = np.concatenate([y for _, y in
generator], axis=0)
            actuals_inverse = self.scaler.inverse_-
transform(actuals)
            return predictions_inverse, actuals_in-
verse

        train_predictions_inverse, train_actuals_in-
verse = predict_and_inverse(train_generator)
        valid_predictions_inverse, valid_actuals_in-
verse = predict_and_inverse(valid_generator)
        test_predictions_inverse, test_actuals_in-
verse = predict_and_inverse(test_generator)

        test_rmse =
math.sqrt(mean_squared_error(test_actuals_inverse,
test_predictions_inverse))
        test_mape = mean_absolute_percent-
age_error(test_actuals_inverse, test_predictions_in-
verse)

        print(f"Test RMSE: {test_rmse}")
        print(f"Test MAPE: {test_mape}")

        return {
            'train': (train_predictions_inverse,
train_actuals_inverse),
            'valid': (valid_predictions_inverse,
valid_actuals_inverse),
            'test': (test_predictions_inverse,
test_actuals_inverse)
        }

    def generate_date_ranges(self, actual, predic-
```

```
tions_inverse, time_steps, freq='D'):
        """
        Generate date ranges for datasets based on
the actual data index and the length of predictions.

        Parameters:
        - actual: DataFrame of the actual dataset.
        - predictions_inverse: Inverse transformed
predictions for the dataset.
        - time_steps: Number of time steps used in
the TimeseriesGenerator.
        - freq: Frequency of the dataset, default is
'D' for daily.

        Returns:
        A Pandas DatetimeIndex representing the date
range for the dataset.
        """
        return
pd.date_range(start=actual.index[time_steps], peri-
ods=len(predictions_inverse), freq=freq)

    def plot_results(self, predictions, actuals,
dates, title='Actual vs Predicted Values'):
        """
        Plot the actual vs predicted values using
Plotly.

        Parameters:
        - predictions: Inverse transformed predic-
tions.
        - actuals: Actual values.
        - dates: Date range for plotting.
        - title: Title for the plot.
        """
        fig = go.Figure()
```

```
        fig.add_trace(go.Scatter(x=dates, y=actual-
s.flatten(), mode='lines', name='Actual Value'))
        fig.add_trace(go.Scatter(x=dates, y=predic-
tions.flatten(), mode='lines', name='Predicted
Value'))
        fig.update_layout(title=title,
xaxis_title='Date', yaxis_title='Value', xax-
is_rangeslider_visible=True)
        fig.show()

    def plot_training_history(self, history):
        """
        Plot the training and validation loss over
epochs.

        Parameters:
        - history: A History object returned by the
fit method of models.
        """
        plt.figure(figsize=(8, 5))
        plt.plot(history.history['loss'],
label='Training Loss')
        plt.plot(history.history['val_loss'],
label='Validation Loss')
        plt.title('Model Training and Validation
Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()
        plt.grid(True)
        plt.show()

    def run(self):
        """
        Executes the complete workflow for training
and evaluating the time series model.
```

```
        """
        data = self.load_and_preprocess_data()
        train_data, valid_data, test_data = self.s-
plit_data(data)
        scaled_train, scaled_valid, scaled_test =
self.scale_data(train_data, valid_data, test_data)
        train_generator, valid_generator, test_gener-
ator = self.create_generators(scaled_train, scaled_-
valid, scaled_test)
        model = self.define_model()
        history = self.train_model(model, train_gen-
erator, valid_generator)
        self.plot_training_history(history)  # Use
self to call the method

        best_model = tf.keras.models.load_model(self-
.model_path)  # Use self.model_path
        results = self.evaluate_model(best_model,
train_generator, valid_generator, test_generator)  #
Use self to call the method

        # Generate date ranges for plotting
        train_dates = self.gener-
ate_date_ranges(train_data, results['train'][0],
self.time_steps, freq='D')  # Use self to call the
method
        valid_dates = self.gener-
ate_date_ranges(valid_data, results['valid'][0],
self.time_steps, freq='D')  # Use self to call the
method
        test_dates = self.generate_date_ranges(test_-
data, results['test'][0], self.time_steps, freq='D')
# Use self to call the method

        # Plotting results
        self.plot_results(results['train'][0], re-
```

```
sults['train'][1], train_dates, "Train Set: Actual vs
Predicted Values")  # Use self to call the method
        self.plot_results(results['valid'][0], re-
sults['valid'][1], valid_dates, "Validation Set:
Actual vs Predicted Values")  # Use self to call the
method
        self.plot_results(results['test'][0],
results['test'][1], test_dates, "Test Set: Actual vs
Predicted Values")  # Use self to call the method

# Example usage
if __name__ == "__main__":
    ts_model = TimeSeriesModel(filepath="/kaggle/in-
put/master-data-csv/master_data.csv", column_index=1)
    ts_model.run()
```

✍ Figure 6: Celebrating Financial Success in a Futuristic Cityscape | Credit: DALL-E