Stock Price Prediction Using News Data and Deep Learning

Step by step process to use deep learning Model for stock price prediction

Pranjal Saxena · Following
Published in Level Up Coding
13 min read · 6 days ago

Listen Share More

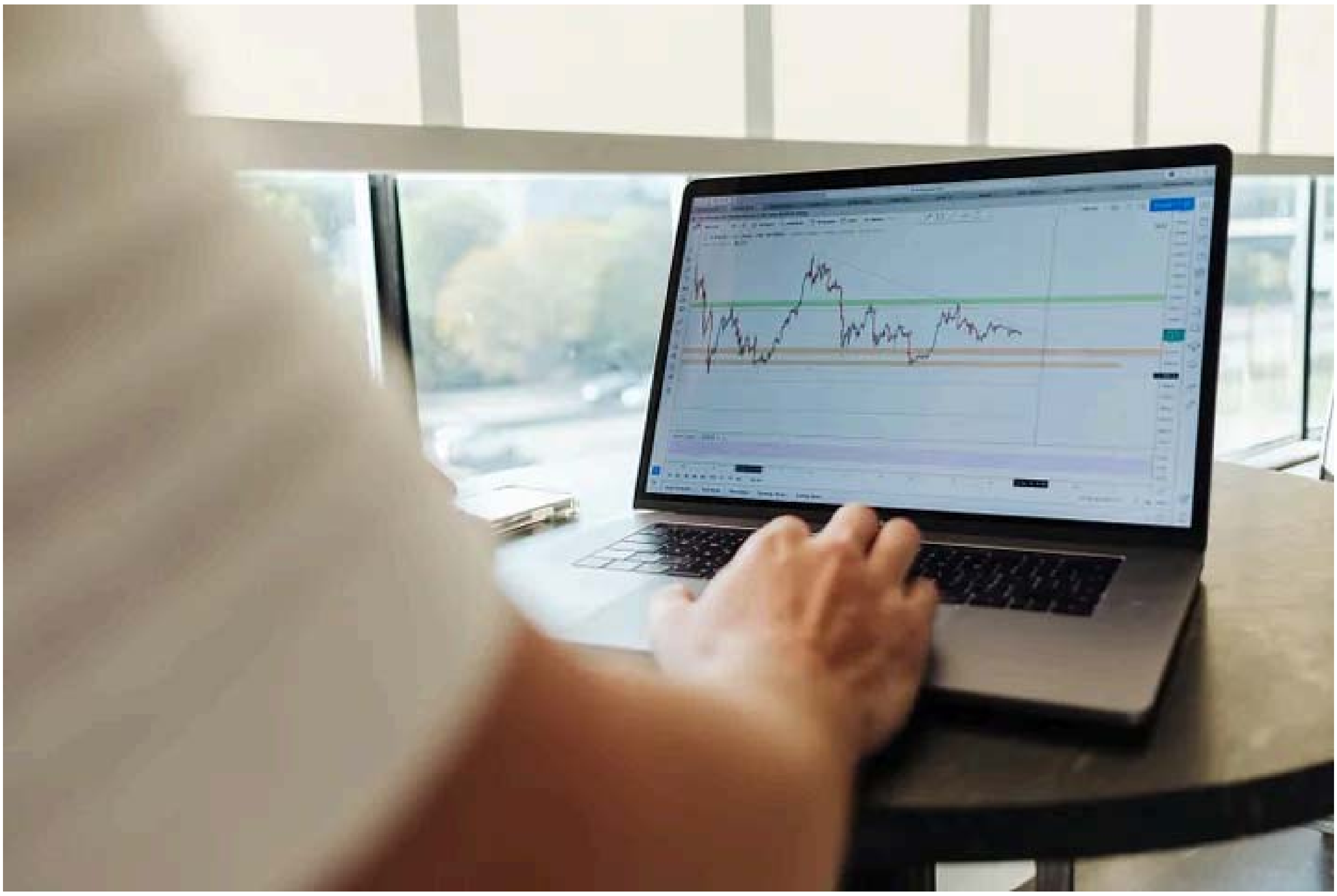


Photo by Jason Briscoe on Unsplash

Investors and financial analysts continually seek cutting-edge tools to predict stock market trends with high accuracy. This endeavor not only boosts investment returns but also mitigates financial risks. In the realm of stock price prediction, the advent of deep learning technologies has marked a revolutionary leap. Among these technologies, Long Short-Term Memory (LSTM) models stand out due to their ability to remember long-term dependencies, a crucial factor in the volatile domain of stock prices.

LSTM models have carved a niche for themselves, demonstrating robustness in handling time-series data like stock prices, which are influenced by a multitude of factors including past performance and current events. By learning from the sequence of data points without losing the 'memory' of earlier inputs, LSTMs offer a profound advantage in predictive accuracy over traditional models.

To fuel these sophisticated models, high-quality, real-time data is paramount. Here, we are going to use EODHD, a real-time price data provider with a solid reputation among data providers in this market. Its real-time API provides US Stock Market price feeds with an impressive 50ms delay.

Throughout this article, we will explore the integration of LSTM with the robust data retrieval capabilities of the EODHD API. We'll cover the essential processes of fetching and preparing data, feature engineering, normalizing inputs, and constructing a predictive model to forecast stock prices. By the end, you'll gain insights into harnessing these advanced tools in your financial analyses, potentially transforming your approach to market predictions.

Advantages of EODHD API in Financial Data Analysis

EODHD API emerges as a pivotal tool in the financial data analysis realm, offering an expansive suite of features designed to cater to the nuanced needs of investors, analysts, and developers. Its robust, powerful, and reliable data foundation supports a variety of financial data needs, from end-of-day and live data to fundamentals and technical indicators.

Comprehensive Data Access

The APIs provide a comprehensive solution for accessing Fundamental, Historical, Intraday, Live and Real-time data (with a minimal delay) covering worldwide stocks, ETFs, bonds, forex pair, crypto, financial news and many more.

Real-Time Data Via WebSockets

For those requiring the utmost immediacy, EODHD's Real-Time Data API, utilizing WebSockets, facilitates sub-50ms latency data transmission. This is especially crucial for US stocks, where the API caters to pre-market and post-market hours, offering flexibility and depth in data analysis.

Financial News API

The Financial News API stands out by allowing users to filter company news by date, type, and tickers, providing an extra layer of depth to market analysis. This feature is indispensable for understanding market sentiment and making informed decisions based on the latest developments.

User-Friendly and Flexible

EODHD is designed with user experience in mind, offering easy-to-use JSON and CSV APIs for seamless integration into financial analysis workflows. Additionally, the API provides flexible pricing plans starting from $19.99, with no long-term obligations, catering to the varying needs and budgets of its users. Quick start guides, sample codes, and ready-to-go solutions for platforms like MS Excel and Google Sheets further enhance its accessibility and ease of use.

Fetching Stock and News Data

To predict stock prices accurately, combining real-time stock data with relevant news articles is crucial. This dual approach enables analysts to understand how market sentiment and real-world events influence stock movements. Below, we outline a Python-based method to fetch and combine these data types provided by EODHD.

Setting Up Your Python Environment

First, ensure your Python environment is set up with the necessary libraries. You'll need `requests` for API calls and `pandas` for data manipulation. Install them using pip if you haven't already:

```
!pip install tensorflow numpy pandas
```

Fetching the Data

You'll require an API key from EODHD to access their data. Once you have your key, you can use the following functions to fetch stock and news data. Here's how you can structure these functions:

```
import pandas as pd
import requests
from datetime import datetime, timedelta

def fetch_stock_data(symbol, start_date, end_date, api_key):
    url = f"https://eodhistoricaldata.com/api/eod/{symbol}"
    params = {
        "from": start_date,
        "to": end_date,
        "fmt": "json",
        "api_token": api_key
    }
    response = requests.get(url, params=params)
    if response.status_code == 200:
        data = response.json()
        df = pd.DataFrame(data)
        df['date'] = pd.to_datetime(df['date']).dt.date
        return df
    else:
        print("Failed to fetch stock data:", response.status_code)
        return pd.DataFrame()  # Return an empty DataFrame in case of failure



def fetch_news_data(symbol, start_date, end_date, api_key):
    all_news = []
    current_end_date = end_date

    while True:
        url = "https://eodhistoricaldata.com/api/news"
        params = {
            "s": symbol,
            "from": start_date,
            "to": current_end_date,
            "api_token": api_key,
            "limit": 1000,
        }
        response = requests.get(url, params=params)
        if response.status_code == 200:
            data = response.json()
            if not data:
                break  # Break the loop if no data is returned
            all_news.extend(data[::-1])  # Reverse the data to maintain chronol
            # Assuming the news data is sorted by date, get the date of the fir
            first_news_date = data[-1]['date']
            first_news_date_obj = datetime.strptime(first_news_date.split("T")[
            # If the first news date is the start date, break the loop
            if first_news_date_obj <= datetime.strptime(start_date, "%Y-%m-%d")
```

```
                break
            # Set the next end date to the day before the first news date in th
            next_end_date_obj = first_news_date_obj - timedelta(days=1)
            current_end_date = next_end_date_obj.strftime("%Y-%m-%d")
        else:
            print("Failed to fetch news data:", response.status_code)
            break  # Exit the loop in case of failure

    df = pd.DataFrame(all_news)
    df['date'] = pd.to_datetime(df['date']).dt.date  # Convert to date to match
    return df


api_key = 'your_api_key_here'  # Replace with your actual API key
symbol = "AAPL.US"  # Example symbol
start_date = "2019-01-01"
end_date = "2024-04-08"

# Fetch stock data
stock_df = fetch_stock_data(symbol, start_date, end_date, api_key)
stock_df['date'] = pd.to_datetime(stock_df['date'])

# Fetch news data
news_df = fetch_news_data(symbol, start_date, end_date, api_key)
news_df["date"] = pd.to_datetime(news_df["date"])
news_df = news_df.sort_values(by="date")

news_df['polarity'] = news_df['sentiment'].apply(lambda x: x['polarity'] if x i

# Group by date and calculate average polarity
avg_polarity_df = news_df.groupby('date')['polarity'].mean().reset_index()

# Ensure the 'date' column in both DataFrames is of the same type
avg_polarity_df['date'] = pd.to_datetime(avg_polarity_df['date'])
```

Combining the Data

Once you have both datasets, the next step is to integrate them. This can be done based on the timestamps available in the news articles and the stock price entries to align insights contextually.

```
combined_df = pd.merge(stock_df, avg_polarity_df, on='date', how='left')

# Fill NaN values in polarity with 0 (neutral sentiment)
combined_df['polarity'].fillna(0, inplace=True)
```

This basic merging by date assumes both datasets include a ‘date’ column formatted similarly. This method aligns stock prices with corresponding news articles by dates, providing a comprehensive dataset to analyze the impact of news on stock prices.

This setup allows financial analysts to harness both quantitative stock data and qualitative news information. By doing so, they can gain a fuller understanding of market dynamics and make more informed predictions. Remember, the accuracy of your predictions can significantly improve by considering both numerical data and narrative context, which reveals the market sentiment.

Be sure to replace `’your_api_key_here’` with your actual EODHD API key and adjust the column names in the merge function as necessary, depending on the exact structure of your data frames. This approach highlights how integrating different data sources can enhance analytical capabilities in financial contexts.

Feature Engineering

Effective feature engineering is crucial for enhancing the predictive power of machine learning models in stock price prediction. By creating new features from existing data, you can uncover patterns that might not be immediately obvious but have significant predictive value. Here’s a detailed guide on developing features that can help improve the accuracy of a financial model using Python.

Moving Averages

Moving averages smooth out price data by creating a constantly updated average price, which can be useful for identifying trends. Here’s how you can calculate the 7-day and 30-day moving averages:

```
# Calculate the short-term 7-day moving average
combined_df['7_day_MA'] = combined_df['close'].rolling(window=7).mean()

# Calculate the long-term 30-day moving average
combined_df['30_day_MA'] = combined_df['close'].rolling(window=30).mean()
```

Daily Percentage Change

This feature measures the percentage change in price from one day to the next, providing insights into daily volatility:

```
# Calculate daily percentage change
combined_df['daily_pct_change'] = combined_df['close'].pct_change() * 100
```

MACD and Signal Line

The Moving Average Convergence Divergence (MACD) is a trend-following momentum indicator that shows the relationship between two moving averages of a security’s price. The Signal Line is the exponential moving average of the MACD:

```
# Calculate MACD
exp1 = combined_df['close'].ewm(span=12, adjust=False).mean()
exp2 = combined_df['close'].ewm(span=26, adjust=False).mean()
combined_df['MACD'] = exp1 - exp2
combined_df['Signal_Line'] = combined_df['MACD'].ewm(span=9, adjust=False).mear
```

Relative Strength Index (RSI)

RSI is a momentum indicator that measures the magnitude of recent price changes to evaluate overbought or oversold conditions:

```
def calculate_RSI(series, period=14):
    delta = series.diff(1)
    gain = (delta.where(delta > 0, 0)).rolling(window=period).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()

    RS = gain / loss
    return 100 - (100 / (1 + RS))

combined_df['RSI'] = calculate_RSI(combined_df['close'])
```

Volume Weighted Average Price (VWAP)

VWAP is the average price a security has traded at throughout the day, based on both volume and price. It is important for understanding the relative strength of market trends:

```
combined_df['VWAP'] = (combined_df['volume'] * (combined_df['high'] + combined_
```

Sentiment Analysis

If you are integrating news data, computing a 7-day sentiment average based on the polarity scores of news articles could provide insights into how public sentiment is affecting stock prices:

```
# Average sentiment score over the past 7 days
combined_df['7_day_sentiment'] = combined_df['polarity'].rolling(window=7).mear
```

These features provide a more granular view of the market conditions, potentially leading to more accurate predictions. They encapsulate both technical indicators and fundamental analysis aspects, ensuring a comprehensive approach to stock price forecasting. Integrating these engineered features into your model can significantly enhance its predictive capabilities, enabling it to recognize complex patterns in stock price movements.

Normalization

Proper data preprocessing is pivotal in enhancing the performance of machine learning models, particularly in the field of financial analytics. Standard scaling, or Z-score normalization, is one such technique that readjusts the features to a common scale by removing the mean and scaling to unit variance. This section details the use of a Python function to standardize features, ensuring that the model interprets them equally, without bias to the scale of original data.

Purpose of Standard Scaling

The main objective of standard scaling is to transform data to have a mean of zero and a standard deviation of one. This transformation is crucial because features often vary widely in magnitudes, units, and range, and models can behave unexpectedly if this variance is not accounted for. By normalizing the data, each feature contributes equally to the ability of the model to learn, which is particularly important in algorithms that are sensitive to the input scale such as SVMs and k-nearest neighbors.

Implementing of Standard Scaling

Here is a function for standard scaling that you can integrate into your data

```
mean_close = np.mean(combined_df['close'])
std_close = np.std(combined_df['close'])

def preprocess_data(data):
    # Normalize features except for the 'date' column
    for column in data.columns:
        if column != 'date':
            data[column] = (data[column] - np.mean(data[column])) / np.std(data
    return data

def reverse_normalize(value, mean, std):
    return value * std + mean
```

Applying the Standard Scaling Function

This function iterates through each column in a DataFrame, excluding ‘date’ because date is a categorical attribute that doesn’t require scaling. For each numerical feature, the function subtracts the mean and divides by the standard deviation, aligning all features to a z-distribution with a mean of zero and standard deviation of one:

Normalization Process: For each non-date column, the function calculates the mean and standard deviation, then applies the z-score formula to standardize the values.
Benefits: This approach not only facilitates a fair comparison across different features but also enhances the algorithm’s convergence during training, as features are weighted equally.

Splitting the Data

Properly segmenting your dataset into training and validation sets is a crucial step in preparing for effective model training. This process not only helps in assessing the model’s performance but also ensures that it generalizes well on unseen data.

The Importance of Data Splitting

Splitting the data serves as a critical check against overfitting. Overfitting occurs when a model performs exceptionally well on its training data but poorly on new, unseen data. By setting aside a portion of the data as a validation set, you provide a fair means of evaluating the model's performance during and after the training process.

How to Split the Data

Typically, data is divided into training and validation sets, and sometimes a third subset, the test set, is used for a final evaluation. Here's a simple way to split your data using Scikit-learn's `train_test_split` function:

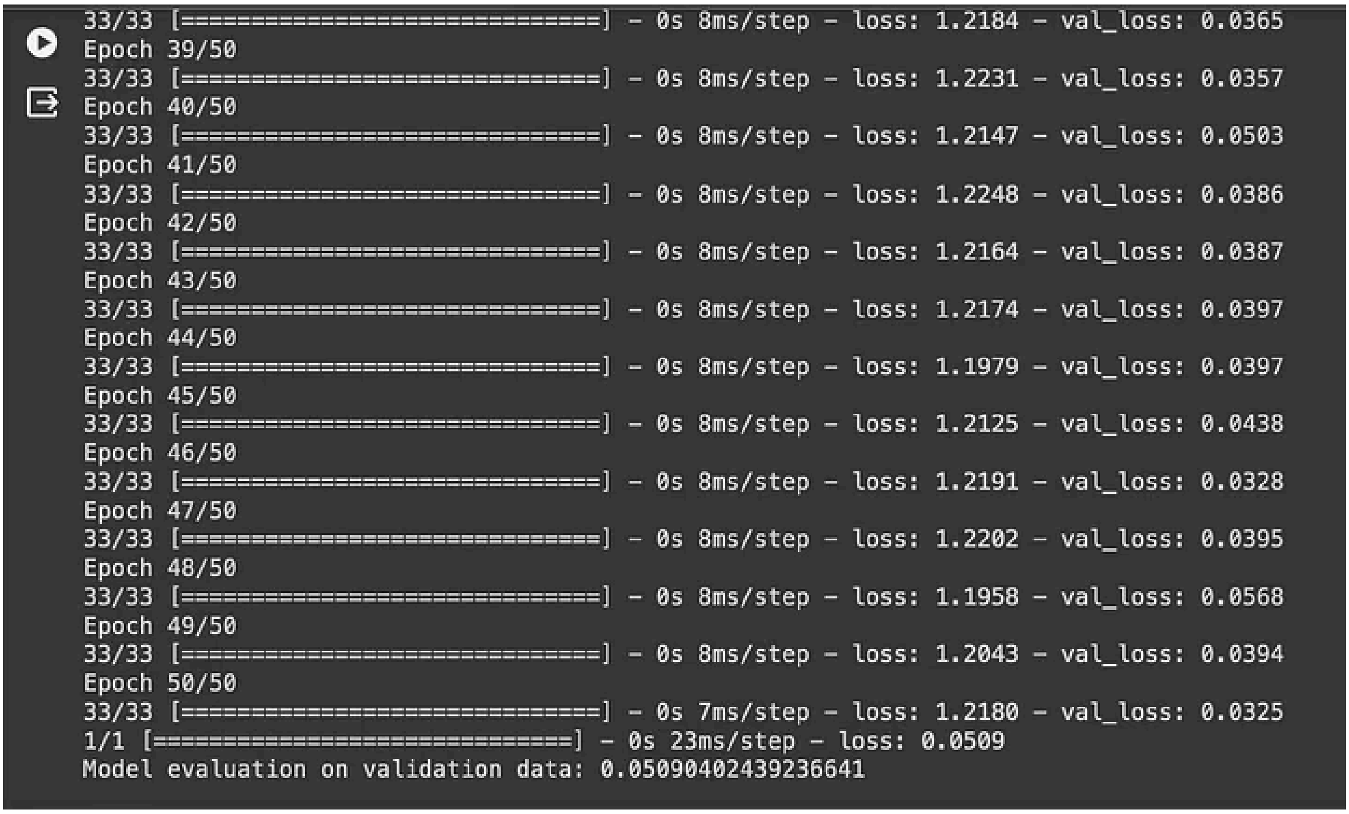
```
def split_data(data, train_end_date='2024-03-31', val_start_date='2024-04-01',
    train_data = data[data['date'] <= train_end_date].drop(columns=['date'])
    val_data = data[(data['date'] >= val_start_date) & (data['date'] <= val_enc
    return train_data, val_data

# Split data
train_data, val_data = split_data(processed_data)

# Split into features and labels
train_features = train_data.drop(columns=['close']).values.reshape((train_data.
train_labels = train_data['close'].values
val_features = val_data.drop(columns=['close']).values.reshape((val_data.shape[
val_labels = val_data['close'].values
```

Model Building and Training

Creating a robust predictive model is a pivotal phase in any machine learning project, especially for tasks like stock price forecasting where precision is paramount. Here we outline the construction of a deep learning model using LSTM (Long Short-Term Memory) layers, well-suited for time-series data such as stock prices.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, Dense, TimeDistributed, Dropou

def create_tft_model(input_shape, output_size):
    inputs = Input(shape=input_shape)
    lstm_out = LSTM(128, return_sequences=True)(inputs)
    bn_out = BatchNormalization()(lstm_out)
    lstm_out2 = LSTM(64, return_sequences=True)(bn_out)
    dropout_out = Dropout(0.3)(lstm_out2)
    dense_out = TimeDistributed(Dense(64))(dropout_out)
    activation_out = LeakyReLU(alpha=0.1)(dense_out)
    dense_out2 = TimeDistributed(Dense(32, activation='relu'))(activation_out)
    outputs = Dense(output_size)(dense_out2)

    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam', loss='mse')
    return model

# Create and train the model
model = create_tft_model(input_shape=(train_features.shape[1], train_features.s
train_model(model, train_features, train_labels, epochs=50)
```

The model provided utilizes the Keras library, a part of TensorFlow, which simplifies the creation of deep learning models.



```
33/33 [==============================] - 0s 8ms/step - loss: 1.2184 - val_loss: 0.0365
Epoch 39/50
33/33 [==============================] - 0s 8ms/step - loss: 1.2231 - val_loss: 0.0357
Epoch 40/50
33/33 [==============================] - 0s 8ms/step - loss: 1.2147 - val_loss: 0.0503
Epoch 41/50
33/33 [==============================] - 0s 8ms/step - loss: 1.2248 - val_loss: 0.0386
Epoch 42/50
33/33 [==============================] - 0s 8ms/step - loss: 1.2164 - val_loss: 0.0387
Epoch 43/50
33/33 [==============================] - 0s 8ms/step - loss: 1.2174 - val_loss: 0.0397
Epoch 44/50
33/33 [==============================] - 0s 8ms/step - loss: 1.1979 - val_loss: 0.0397
Epoch 45/50
33/33 [==============================] - 0s 8ms/step - loss: 1.2125 - val_loss: 0.0438
Epoch 46/50
33/33 [==============================] - 0s 8ms/step - loss: 1.2191 - val_loss: 0.0328
Epoch 47/50
33/33 [==============================] - 0s 8ms/step - loss: 1.2202 - val_loss: 0.0395
Epoch 48/50
33/33 [==============================] - 0s 8ms/step - loss: 1.1958 - val_loss: 0.0568
Epoch 49/50
33/33 [==============================] - 0s 8ms/step - loss: 1.2043 - val_loss: 0.0394
Epoch 50/50
33/33 [==============================] - 0s 7ms/step - loss: 1.2180 - val_loss: 0.0325
1/1 [==============================] - 0s 23ms/step - loss: 0.0509
Model evaluation on validation data: 0.05090402439236641
```

Model Training

Key Components Explained:

- Input Layer: This is where you define the shape of the input data that the model will expect.
- LSTM Layers: These are used to process time-series data, capturing long-term dependencies in data sequences.

- Batch Normalization: This layer normalizes the activations from the previous layer, which can improve the learning process by reducing internal covariate shift.
- Dropout: A regularization technique where input and recurrent connections to LSTM units are randomly excluded from each update cycle, which helps in preventing overfitting.
- TimeDistributed Dense Layer: Applies a Dense (fully connected) operation to every temporal slice of an input. This is useful for making frame-wise predictions, as you might need in a time-series analysis.
- LeakyReLU: An activation function that allows a small gradient when the unit is not active.
- Output Layer: Outputs the final prediction. The dimension is set by `output_size`, which should match the dimensionality of your target variable.

Future Stock Price Prediction

Once a predictive model is trained, the next logical step is to use it to forecast future stock prices. This step not only tests the model's effectiveness but also showcases its practical application in financial analysis. Here's how you can approach making predictions with your LSTM model, particularly focusing on using rolling price data.

Forecasting with the LSTM Model

The trained LSTM model you've prepared can now be employed to predict future stock prices based on historical data. The key to effective forecasting in financial markets is leveraging the model's ability to understand and predict patterns from time-series data. Here's a general approach to making predictions:

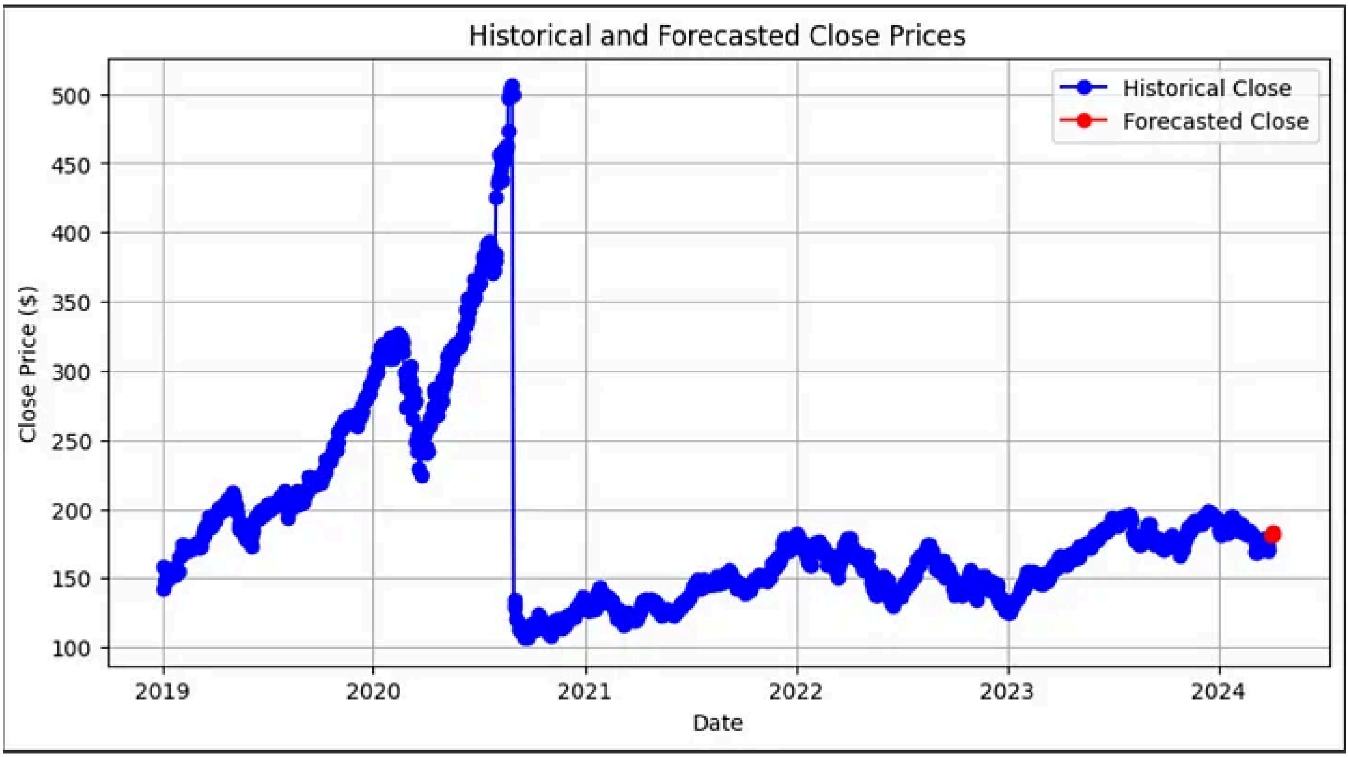
Using Rolling Prices for Continuous Forecasting

To forecast next day's prices continuously, you can use a technique known as rolling prediction, where the model uses its own predictions as part of the input for future predictions. This approach simulates a real-world scenario where each new prediction updates with the latest available data.

Here's a simple example of how to implement rolling predictions:

```
import numpy as np
```

```
def rolling_forecast(model, initial_input, steps=7):
    # Assuming initial_input is (6, 1, 14) and you need the last input to start
    current_input = initial_input[-1:, :, :].copy()  # Use only the last timest
    predictions = []

    for _ in range(steps):
        # Predict the next step
        next_point = model.predict(current_input)

        # Append the prediction to the output
        predictions.append(next_point.flatten()[0])  # Flatten if necessary, de

        # Update the current input for the next prediction
        # Update all features as needed; here, we simply copy over the last pre
        current_input[0, 0, -1] = next_point  # Update the last feature with th

    return np.array(predictions)

initial_input = val_features[-1:, :, :]  # This should be shaped as (1, 1, 14)

# Forecast the next 7 days using the rolling forecast function
next_7_days = rolling_forecast(model, initial_input, steps=7)
original_val_predictions = reverse_normalize(next_7_days, mean_close, std_close

print("Forecasted prices from April 8 to April 14, 2024:", original_val_predict
```



Making predictions with your LSTM model involves not just applying the model to new data but also strategically using its outputs to forecast further into the future.

By employing a rolling prediction method, the model can continually update its forecasts based on the latest predicted values, which mimics the dynamic and continuously changing nature of stock markets.

This method of using rolling prices enhances the model's utility by keeping its forecasts updated, potentially increasing the accuracy and relevance of its predictions in real-time financial decision-making. Properly managing these predictions and continuously validating and adjusting the model's inputs and structure are essential for maintaining its effectiveness over time.

Conclusion

Throughout this article, we've explored the robust potential of using an LSTM model combined with the comprehensive data provided by the EOD Historical Data (EODHD) API for predicting stock prices. This powerful integration leverages the deep learning capabilities of LSTM networks and the extensive, accurate financial datasets from EODHD, creating a sophisticated tool for financial analysts and investors.

For financial professionals looking to refine their predictive models or those just beginning to integrate machine learning into their analysis toolkit, the combination of LSTM models and the EODHD API offers a promising avenue. By adopting this advanced approach, analysts can gain a deeper understanding of market dynamics and develop more nuanced, informed trading strategies. Therefore, we encourage you to explore this integration further for your financial analysis needs, harnessing the power of advanced analytics to navigate and profit in the financial markets more effectively.

This synergy between cutting-edge machine learning techniques and robust financial data platforms like EODHD not only pushes the boundaries of what's possible in financial forecasting but also democratizes access to sophisticated tools that were once only available to large financial institutions. By embracing these technologies, you can stay ahead in the competitive world of stock trading.

Python Stock Price Prediction Eod Stock Market Api Lstm Stock Market

Following

Written by Pranjal Saxena

2.5K Followers · Writer for Level Up Coding

Senior Data Scientist | NLP Expert | Connect with me : https://linktr.ee/pranjalai

More from Pranjal Saxena and Level Up Coding

Pranjal Saxena in DataDrivenInvestor

Stock Volatility Prediction Using OpenAI and Python

Step by step guide to predict stock volatility using OpenAI

10 min read · Mar 12, 2024

287 3

Alexander Nguyen in Level Up Coding

Why I Keep Failing Candidates During Google Interviews…

They don't meet the bar.

4 min read · Apr 13, 2023

5.2K 157

Sean Maxwell in Level Up Coding

Why I left object-oriented programming in full-stack TypeScript

And what I use as an alternative…

6 min read · Mar 24, 2024

517 16

Pranjal Saxena in DataDrivenInvestor

Top Stock Market APIs You Must Be Aware of

The Top Stock Market APIs Revealed for 2024

7 min read · Feb 6, 2024

135 1

See all from Pranjal Saxena

See all from Level Up Coding

Recommended from Medium

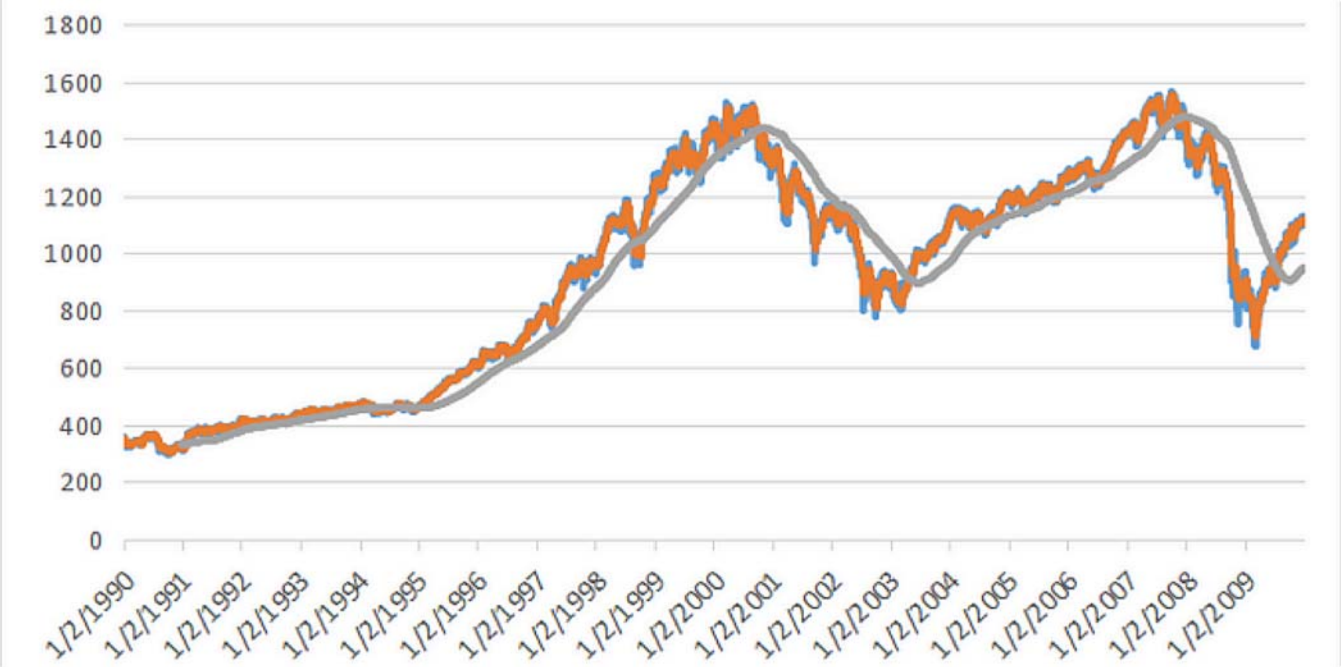
EODHD APIs

Excel Stock Portfolio — How To Build?

Free Template Inside

6 min read · Apr 10, 2024

128 3



Ayrat Murtazin in DataDrivenInvestor

Citadel's Strategy Anyone Can Use — Moving Averages tell more than Strategists

In the investing world, accurately predicting macro market regimes, such as economic downturns, crisis turning points, or the duration of...

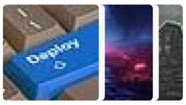
5 min read · Apr 9, 2024

30 1

Lists

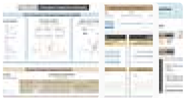
Coding & Development

11 stories · 565 saves



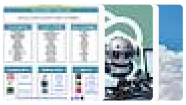
Predictive Modeling w/ Python

20 stories · 1105 saves



Practical Guides to Machine Learning

10 stories · 1319 saves



ChatGPT

21 stories · 577 saves

Modular
M
Mojo
One language for all AI programming
Fabian Tschopp - talk at SINTEF - 17th of January 2024

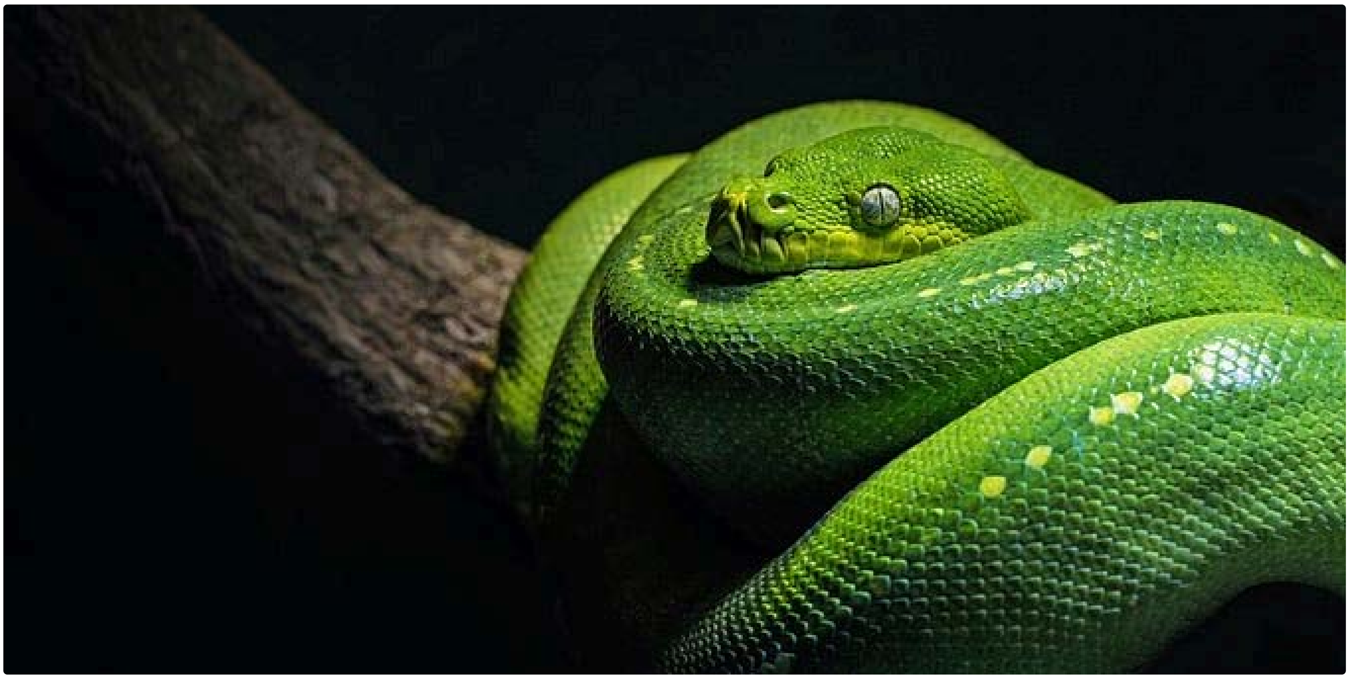
Dylan Cooper in Stackademic

Mojo, 90,000 Times Faster Than Python, Finally Open Sourced!

On March 29, 2024, Modular Inc. announced the open sourcing of the core components of Mojo.

✦ · 10 min read · Apr 8, 2024

2.6K 21



Marco Peixeiro in Towards Data Science

iTransformer: The Latest Breakthrough in Time Series Forecasting

Discover the architecture of iTransformer and apply the model in a small experiment using Python.

✦ · 9 min read · Apr 9, 2024

549 8

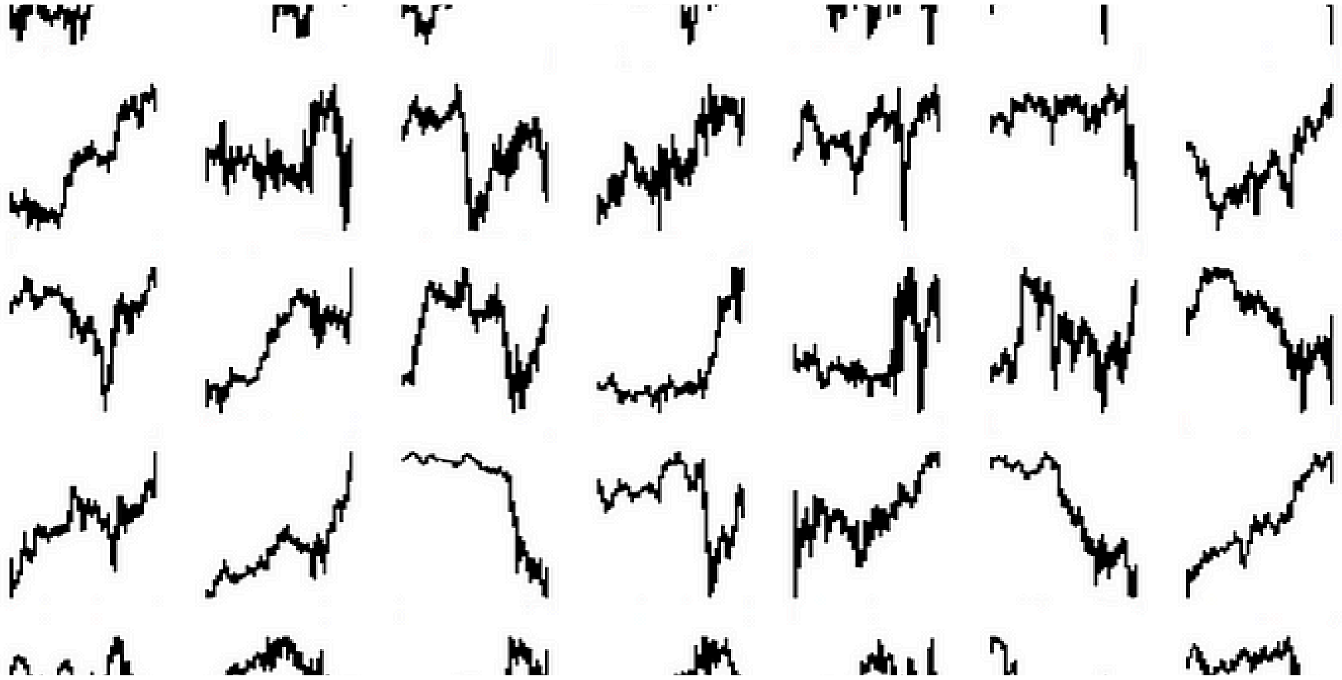
Sze Zhong LIM in Data And Beyond

Mastering Exploratory Data Analysis (EDA): Everything You Need To Know

A systematic approach to EDA your data and prep it for machine learning.

18 min read · Apr 6, 2024

327 3



Andrew Kreimer

MarketNet: Computer Vision Applications in Financial Markets

We’ve got WordNet and ImageNet, it’s time for a MarketNet. As traders we follow various asset classes across multiple time-frames. The…

22 min read · Mar 22, 2024

212 2

See more recommendations