Portfolio Optimization-Based Stock Prediction Using Long-Short Term Memory Network in Quantitative Trading

ARIYARATHNE D L K D (KIC-HNDCSAI-Y2-241-F-002)

Abstract

Stock price prediction plays a pivotal role in quantitative trading, enabling traders to design strategies for maximizing returns. This project explores the implementation of Long Short-Term Memory (LSTM) networks for stock price prediction and applies portfolio optimization techniques such as Monte Carlo Simulation (MCS) and Mean-Variance Optimization (MVO) to construct efficient portfolios. Historical and forecasted stock data are utilized, with results showing the effectiveness of LSTM in generating accurate predictions and optimized portfolios outperforming traditional benchmarks. This report outlines the methodology, code implementation, results, and implications of the study.

Table of Contents

1. Introduction

Stock market forecasting is a complex and challenging task, where accurately predicting future stock prices plays a crucial role in making informed trading decisions. Traditional methods for stock prediction, such as linear regression (LR) and moving averages, often fall short when it comes to capturing the complexity and non-linearity of market data. The stock market is dynamic, influenced by numerous factors that can change over time, making it essential to adopt advanced techniques capable of modeling these intricate relationships.

Deep learning, specifically Long Short-Term Memory (LSTM) networks, has emerged as a leading solution for stock price prediction due to its ability to handle sequential data effectively. LSTM, a specialized type of Recurrent Neural Network (RNN), can remember information over long periods and is particularly suitable for financial time-series data, where past market behavior significantly influences future price movements.

In addition to price prediction, portfolio optimization plays a critical role in the decision-making process for investment strategies. A portfolio is a collection of assets, and portfolio management involves constructing an investment strategy that maximizes returns while managing risks. There are two main approaches to portfolio management: traditional and quantitative. While traditional methods rely on qualitative factors and manual decision-making, quantitative trading uses mathematical models and statistical techniques to guide investment decisions. In quantitative trading, portfolio construction is based on optimizing asset selection and allocation using sophisticated mathematical and computational methods.

This project combines LSTM networks for stock price prediction with two widely-used portfolio optimization techniques: Monte Carlo Simulation (MCS) and Mean-Variance Optimization (MVO). The objective is to predict stock prices for 10 selected companies listed on the American Stock Exchange over a 10-year period, from January 3, 2012, to December 31, 2021, and use these predictions to construct and optimize investment portfolios. Through this approach, the study explores the effectiveness of LSTM-based predictions in enhancing portfolio performance while managing risk, ultimately providing a framework for improving investment strategies.

2. Literature Review

2.1 Overview of the Research Paper

The paper "Portfolio Optimization-Based Stock Prediction Using Long-Short Term Memory Network in Quantitative Trading" by Van-Dai Ta, Chuan-Ming Liu, and Direselign Addis Tadesse addresses two critical challenges in financial modeling: improving the accuracy of stock price predictions and optimizing portfolios based on those predictions. The authors combine deep learning techniques, specifically Long Short-Term Memory (LSTM) networks, with traditional portfolio optimization theories, such as Modern Portfolio Theory (MPT), to develop a more effective approach for financial forecasting and asset allocation.

According to the research paper, traditional stock prediction models, such as Linear Regression (LR) and Support Vector Regression (SVR), often fall short in capturing the complexities of financial markets, which are characterized by volatility, non-linearity, and the intricate interdependencies among various economic variables. The authors argue that LSTM networks, a type of Recurrent Neural Network (RNN), are uniquely suited for this task due to their ability to capture long-term dependencies in sequential data and mitigate the vanishing gradient problem. This makes LSTM networks particularly useful for predicting stock prices, which often depend on long-term trends and patterns.

The study extends beyond mere stock prediction to explore the optimization of portfolios. Building on the principles of Modern Portfolio Theory (MPT), the authors use optimization techniques such as Monte Carlo Simulation (MCS) and Mean-Variance Optimization (MVO) to identify portfolio allocations that maximize returns for a given level of risk. The results of the study show that portfolios constructed using LSTM predictions outperform those based on traditional models, both in terms of cumulative returns and risk-adjusted performance, highlighting the potential of deep learning in quantitative trading.

2.2 Introduction to Quantitative Trading and Portfolio Management

Quantitative trading is a systematic, data-driven approach that uses algorithms and mathematical models to guide trading decisions. Unlike traditional trading, which relies on subjective human judgment, quantitative trading emphasizes consistency, discipline, and statistical rigor. By leveraging vast amounts of data, quantitative trading systems execute trades with minimal human intervention, making it well-suited to today's fast-paced financial markets. This approach is becoming increasingly popular due to its ability to process large datasets quickly and its potential for reducing transaction costs and human biases.

Portfolio management, a core component of quantitative trading, focuses on the selection and allocation of assets to optimize returns while managing risk. The foundation of modern portfolio management lies in the theory of diversification, introduced by Harry Markowitz in the 1950s. According to Markowitz's Modern Portfolio Theory (MPT), by combining assets with low or negative correlations, investors can reduce the overall risk of the portfolio while maintaining a desirable level of return. The efficient frontier, a concept derived from MPT, represents the set of portfolios that deliver the highest return for a given level of risk, and forms the theoretical basis for many optimization techniques.

The research paper builds on MPT by introducing more advanced techniques for portfolio optimization and integrating them with stock price predictions made by LSTM networks. This hybrid approach aims to improve both the accuracy of stock forecasting and the efficiency of portfolio allocation, ultimately enhancing financial decision-making.

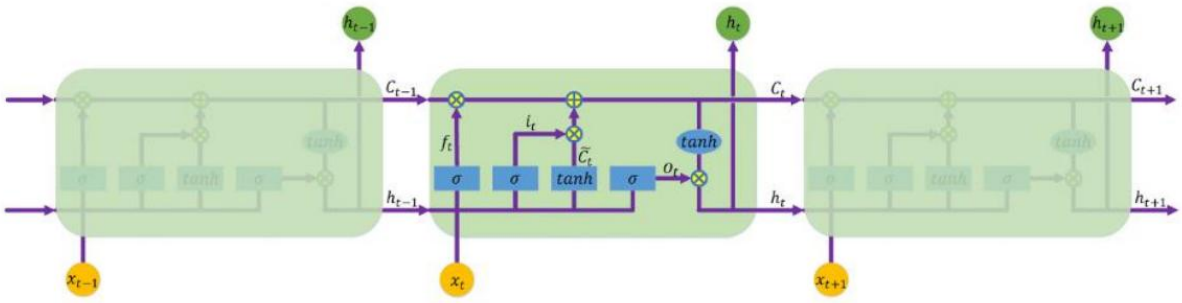
2.3 Stock Price Prediction Using LSTM Networks

2.3.1 Limitations of Traditional Methods

Stock price prediction has always been a difficult task due to the volatile and non-linear nature of financial markets. Traditional models like Linear Regression (LR) and Support Vector Regression (SVR) often struggle to capture the complexities of market dynamics because they rely on assumptions of linearity and fail to account for long-term dependencies in the data. These limitations are particularly evident when predicting stock prices, as these are influenced by various long-term factors, including economic trends, market sentiment, and investor behavior, which cannot be fully captured by traditional models.

2.3.2 Introduction to LSTM Networks

In response to these challenges, the paper introduces Long Short-Term Memory (LSTM) networks, a variant of Recurrent Neural Networks (RNNs), as a solution for improving stock prediction accuracy. According to the authors, LSTM networks are particularly well-suited for financial forecasting because they are capable of capturing long-term dependencies in sequential data. This ability is crucial in stock price prediction, where trends and patterns evolve over extended periods.



A RNN with LSTM network architecture

LSTMs mitigate the vanishing gradient problem that is common in standard RNNs through a specialized architecture that includes key gates.

1. Forget Gate: The forget gate decides what information from the previous cell state should be discarded. It outputs a value between 0 and 1 for each piece of information in the cell state, where 0 means "completely forget" and 1 means "completely keep."

$$f_t = \sigma\left(W_{x_f}^T x_t + W_{h_f}^T h_{t-1} + b_f\right)$$

2. Input Gate: The input gate determines which new information should be added to the cell state. It works together with the candidate cell state (Ćt) to update the cell..

$$i_t = \sigma\left(W_{x_i}^T x_t + W_{h_i}^T h_{t-1} + b_i\right)$$

Additionally, the candidate cell state is computed to propose new values for the cell state:

$$\tilde{C}_t = \tanh(W_{xc} x_t + W_{hc} h_{t-1} + b_c)$$

3. Cell Gate (Cell State Update): The cell gate combines the output of the forget gate and the input gate to update the cell state. It decides how much of the previous cell state () to retain and how much new candidate information () to add.

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

4. Output Gate: The output gate determines which part of the cell state contributes to the hidden state (h_t), which is the output of the LSTM cell.

$$o_t = \sigma\left(W_{x_O}^T x_t + W_{h_O}^T h_{t-1} + b_O\right)$$

5. Hidden State (h_t): Finally, the hidden state is computed as the product of the output gate and the updated cell state, passed through a *tanh* activation to scale the values between -1 and 1:

$$h_t = o_t \odot \tanh(C_t)$$

These gates allow LSTM networks to store important information over long sequences and selectively update the memory, which is why they are highly effective at processing and predicting time-series data like stock prices. The authors also highlight the importance of optimizing the number of neurons and epochs during training to ensure that the LSTM model achieves its highest predictive accuracy.

2.4 Portfolio Construction and Optimization Techniques

2.4.1 Foundations of Portfolio Theory

The paper builds upon Modern Portfolio Theory (MPT), which asserts that risk can be minimized by diversifying investments across assets that do not move in perfect correlation with each other. This theory is grounded in the mathematical calculations of expected returns and portfolio variance. The expected return and variance of a portfolio can be expressed as follows:

$$\mu_P = \sum_{i=1}^{N} w_i \mu_i = w^T \mu$$

$$\sigma_P^2 = \sum_{i=1}^{N} \sum_{j=1}^{N} w_i w_j \sigma_{ij} = w^T \Sigma w$$

Where:

- w_i represents the weight of the i^{th} asset in the portfolio,
- μ_i is the expected return of the i^{th} asset,
- Σ is the covariance matrix, which captures the relationship between asset returns.

The goal is to find a portfolio that either maximizes the expected return for a given level of risk or minimizes risk for a given expected return. This leads to the creation of the efficient frontier, which represents the set of optimal portfolios that lie along this curve.

2.4.2 Monte Carlo Simulation (MCS) and Mean-Variance Optimization (MVO)

The authors employ two key optimization techniques to refine portfolio construction: Monte Carlo Simulation (MCS) and Mean-Variance Optimization (MVO).

- Monte Carlo Simulation (MCS): MCS is a probabilistic method that generates multiple random portfolio scenarios by varying the asset weights. The paper demonstrates how MCS evaluates each portfolio's performance based on metrics like expected return, volatility, and the Sharpe Ratio, identifying the portfolio with the highest Sharpe Ratio as the optimal one. This method allows for a comprehensive exploration of portfolio configurations across a range of possible outcomes.
- Mean-Variance Optimization (MVO): MVO, rooted in Markowitz's theory, optimizes the portfolio by minimizing variance for a given return. The authors use the covariance matrix of asset returns to calculate the portfolio weights that achieve the best risk-return trade-off. MVO provides a more deterministic and precise solution compared to MCS, but both methods contribute to the goal of constructing an optimal portfolio.

2.5 Data and Experimental Design

The paper uses a dataset of ten years (2008–2018) of daily stock price data from the S&P 500 index, focusing on large-cap stocks. The dataset includes Open, High, Low, Close prices, and trading volumes, offering a comprehensive view of the stock market's dynamics during this period. By using such a rich dataset, the authors are able to test the robustness of their models under various market conditions, including periods of volatility and stability.

For the LSTM model, the authors conducted extensive experimentation, adjusting the number of neurons, hidden layers, and epochs to fine-tune the model for optimal performance. The results showed that a stacked LSTM architecture with 256 and 512 neurons, trained over 4000 epochs, provided the best performance. Evaluation metrics like Mean Absolute Error (MAE) and Mean Squared Error (MSE) were used to assess prediction accuracy, with LSTM consistently outperforming other models, such as GRU, LR, and SVR.

2.6 Performance Evaluation and Results

2.6.1 Prediction Accuracy of LSTM Networks

The paper demonstrates that LSTM networks provide superior stock price predictions compared to traditional methods. The LSTM model achieves lower MAE and MSE values, reflecting its ability to capture long-term dependencies and better predict stock prices. This accuracy is crucial in the context of financial forecasting, where even small improvements in prediction accuracy can have significant financial implications.

2.6.2 Portfolio Performance Evaluation

The portfolios constructed using LSTM predictions consistently outperform those based on traditional models. Not only do LSTM-based portfolios show higher cumulative returns, but they also exhibit better risk-adjusted performance, as evidenced by higher Sharpe Ratios. While optimization methods like MCS and MVO improve risk-adjusted returns, they do not always maximize returns. This highlights the critical role of accurate stock price predictions in achieving optimal portfolio performance.

2.7 Comparison to the Proposed Approach

This study extends the work presented in the paper by focusing on a more recent dataset (2012–2021) and selecting 10 specific companies from the S&P 500. By analyzing post-2010 market conditions, it provides an updated perspective on stock price prediction and portfolio optimization. Additionally, forecasting stock prices for an entire year and constructing three distinct portfolios introduces complexity to the analysis, allowing for a more nuanced exploration of risk-return trade-offs. This approach evaluates the effectiveness of LSTM-based predictions in contemporary market scenarios while examining the practical implications of combining predictive accuracy with robust optimization techniques.

3. Methodology

3.1 Data Description

Historical Data: The historical dataset consists of daily stock data for the American stock exchange spanning 10 years, from January 3, 2012, to December 31, 2021. It includes records for 10 companies: Apple Inc. (AAPL), Cisco Systems (CSCO), Alphabet Inc. (GOOG), HP Inc. (HPQ), Microchip Technology (MCHP), Newell Brands (NWL), Sealed Air Corporation (SEE), Seagate Technology (STX), Skyworks Solutions (SWKS), and The TJX Companies (TJX). Key fields in the dataset include Date, Open, High, Low, Close, Volume, and the respective company names. This comprehensive dataset allows for detailed analysis of each company's stock performance over a decade, capturing daily price movements and trading volumes.

Forecasted Data: The forecasted dataset covers stock market open days throughout 2022. Forecasted prices are generated for the same 10 companies, enabling the evaluation of predictive models by comparing forecasted values with actual market data. This helps assess the effectiveness and reliability of the predictive techniques used.

3.2 Prediction Model

The prediction model employs a Long Short-Term Memory (LSTM) architecture designed for time-series forecasting of stock prices. The model includes the following key components:

Model Structure:

- Input Layer: Takes sequences of scaled features (Open, High, Low, Volume) with a time step of 60 days.
- Hidden Layers: Two LSTM layers with 128 and 64 units, respectively, both incorporating dropout regularization (0.3) to mitigate overfitting. The LSTM layers use the He Uniform initializer and L1/L2 regularization to enhance generalization.
- Output Layer: A Dense layer with a single neuron to predict the next closing price.

Training Details:

- Loss Function: Mean Squared Error (MSE) is used to minimize prediction error.
- Optimizer: Adam optimizer with an initial learning rate of 0.0005 to ensure efficient convergence.
- EarlyStopping: Monitors validation loss and stops training if no improvement occurs for 10 epochs.
- LearningRateScheduler: Adjusts the learning rate dynamically to enhance training efficiency.

Data Preparation and Training Process:

For each company, stock features are scaled using a MinMaxScaler pipeline. Synthetic data augmentation is applied using jittering techniques to improve model robustness. The data is split into training (70%), validation (10%), and testing (10%) sets. The model is trained for up to 150 epochs with a batch size of 64.

Model Evaluation:

The model's performance is evaluated each using metrics such as Mean Absolute Percentage Error (MAPE), Mean Squared Error (MSE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE). Visualizations of actual vs. predicted stock prices and training vs. validation loss curves provide insights into model accuracy and performance.

3.3 Forecasting

The forecasting process utilizes trained LSTM models to predict future stock prices based on the most recent 150 historical data points for each company. The key steps involved are as follows:

Forecasting Process:

- Data Preparation: For each company, the last 150 days of historical stock data (Open, High, Low, Volume) are scaled using MinMaxScaler to ensure compatibility with the LSTM model.
- Forecasting Loop: The initial sequence of scaled features is fed into the trained LSTM model to predict the next closing price. After each prediction, the new predicted value is added to the sequence, and the oldest value is removed. This iterative process continues for the entire forecast period (covering the year 2022).
- Rescaling Predictions: The forecasted values are transformed back to the original scale to interpret the predicted closing prices accurately.

Visualization and Results:

The forecasted closing prices are plotted alongside historical and predicted prices for each company, providing a clear view of how the model anticipates future stock performance. The results are saved to a CSV file for further analysis, enabling easy comparison of forecasts with real-world market data.

3.4 Portfolio Construction

Mathematically Optimized Portfolios:

The construction of mathematically optimized portfolios involving the use of covariance matrices to identify how the returns of different assets move relative to one another. This approach aims to maximize returns while minimizing risk. The process typically follows these steps:

- Calculate Covariance Matrix: Assess the relationship between asset returns (i.e., their co-movement). This is typically done by calculating the covariance between pairs of asset returns over a given period.
- Optimization: Use the covariance matrix to identify portfolio weights that maximize the expected return for a given risk level. Advanced optimization models can be employed to minimize the risk while targeting an optimal return.

- Risk-Return Trade-off: The portfolio weights are adjusted to achieve the best possible trade-off between risk (measured by variance or standard deviation) and return (using expected returns).

Random Selection Portfolios:

This approach uses a random sampling technique to create portfolios, ensuring that at least one company has a positive return over a given time period. The process follows these steps:

- Asset Selection: Randomly select a group of assets (e.g., stocks) from a larger pool of companies.
- Random Weight Assignment: Assign random weights to each asset selected, ensuring that the portfolio is diversified to some extent.
- Positive Return Check: Ensure at least one asset in the portfolio has a positive return to make the portfolio potentially profitable.
- Performance Evaluation: Evaluate the portfolio's overall return and risk characteristics. This technique is useful for understanding how different combinations of assets can perform, even if they are selected randomly.

Optimization Techniques:

- **Equal-Weighted Portfolio (EQ):**

  In an equal-weighted portfolio, each asset receives the same weight, regardless of its risk or return. This strategy is simple and ensures diversification but may not be optimal for maximizing risk-adjusted returns. Rebalancing periodically ensures that the weights stay equal, maintaining diversification.

- **Monte Carlo Simulation (MCS):**

  Monte Carlo Simulation models portfolio outcomes by generating numerous random scenarios, helping assess the probability of different returns. It involves defining key parameters like expected returns and risk, simulating potential future returns, and analyzing the distribution of those outcomes to select the best portfolio based on desired risk-return characteristics.

- **Mean-Variance Optimization (MVO):**

  Mean-Variance Optimization aims to find the optimal mix of assets that maximizes returns for a given level of risk or minimizes risk for a target return. By calculating expected returns and the covariance matrix of asset returns, the optimal portfolio weights are determined. The efficient frontier is used to identify the best portfolios based on the investor's risk tolerance and return objectives.

4. Implementation

4.1 Code Explanation for the Stock Prediction

This section describes the detailed process of developing and implementing a robust stock price prediction and portfolio optimization model. The implementation leverages Long Short-Term Memory (LSTM) networks, data preprocessing, augmentation, and various evaluation metrics to create a predictive pipeline for stock price forecasting. Each line of the code has been written with purpose and is explained thoroughly below to provide clarity on its role, contribution, and effect on the overall model.

Additionally, this implementation tailors the stock price prediction model for multiple companies. By employing a loop through each company in the dataset, separate predictive pipelines are created and evaluated, ensuring a detailed analysis of stock performance. The pipeline integrates data preprocessing, augmentation techniques, model development, and rigorous evaluation metrics, providing both general and company-specific insights.

So, the implementation utilizes a collection of essential libraries for data manipulation, visualization, and neural network development. Pandas and NumPy handle data processing and mathematical operations, while Scikit-learn provides scaling techniques and evaluation metrics. For visualization, Matplotlib enables clear plotting of stock prices and results. Finally, TensorFlow/Keras powers the core LSTM model, offering tools for creating, training, and optimizing the network. These libraries collectively form the backbone of the implementation.

4.1.1 Functions and Augmentation

This segment outlines utility functions and data augmentation techniques to enrich and prepare the dataset for training.

Learning Rate Scheduler

```
def lr_scheduler(epoch, lr):
    if epoch < 10:
        return lr
    else:
        return lr * 0.90
```

This function dynamically adjusts the learning rate during training. By reducing the learning rate by 10% after the first 10 epochs, it ensures that the model continues to fine-tune its weights without drastic updates. This strategy prevents overshooting the optimal parameters while maintaining steady progress toward convergence. Gradual learning rate reduction also helps in avoiding local minima by fine-tuning the model's parameters in smaller steps.

Sequence Creation

```
def create_sequences(X, y, time_steps=60):
    Xs, ys = [], []
    for i in range(time_steps, len(X)):
        Xs.append(X[i - time_steps:i])
        ys.append(y[i])
    return np.array(Xs), np.array(ys)
```

This function implements a sliding window mechanism to divide the dataset into sequences of fixed length (defined by time_steps). Each sequence comprises a time-series of features (input for the LSTM) and the corresponding target value (output for prediction). This structure is essential for training LSTMs, which rely on sequential patterns in data. By structuring inputs in this way, the LSTM learns temporal dependencies that are critical for predicting stock price movements.

Data Augmentation with Jittering and Synthetic Targets

```
def jittering(data, noise_factor=0.02):
    return data + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=data.shape)

def synthetic_target_generation(data, noise_factor=0.01):
    return data + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=data.shape)
```

The jittering function applies random noise to input features, simulating slight variations in market conditions to enhance the model's robustness and prevent overfitting. This helps the model generalize better across unseen data by training it on augmented data that mimics real-world variability. Similarly, synthetic_target_generation creates slight perturbations in target values, helping the model handle minor fluctuations in stock prices more effectively. Together, these augmentation techniques improve the model's ability to generalize and handle noisy, real-world data.

4.1.2 Data Loading and Visualizing

The preprocessing phase ensures the dataset is prepared for training by cleaning, structuring, and normalizing data for optimal performance with the LSTM model.

Loading and Preparing Data

```
data = pd.read_csv('stock_data.csv')
data['Date'] = pd.to_datetime(data['Date'])
data.set_index('Date', inplace=True)
companies = data['Company'].unique()
```

Here, the dataset is read into a Pandas DataFrame and processed for further analysis. Converting the Date column to datetime format ensures compatibility with time-series operations, such as indexing and slicing by time intervals. Setting the Date column as the index facilitates easier manipulation and visualization of time-series data. Identifying unique companies in the dataset allows for the segmentation of data, enabling individual models to be built and trained for each company's stock behavior. This step ensures flexibility and scalability of the pipeline.

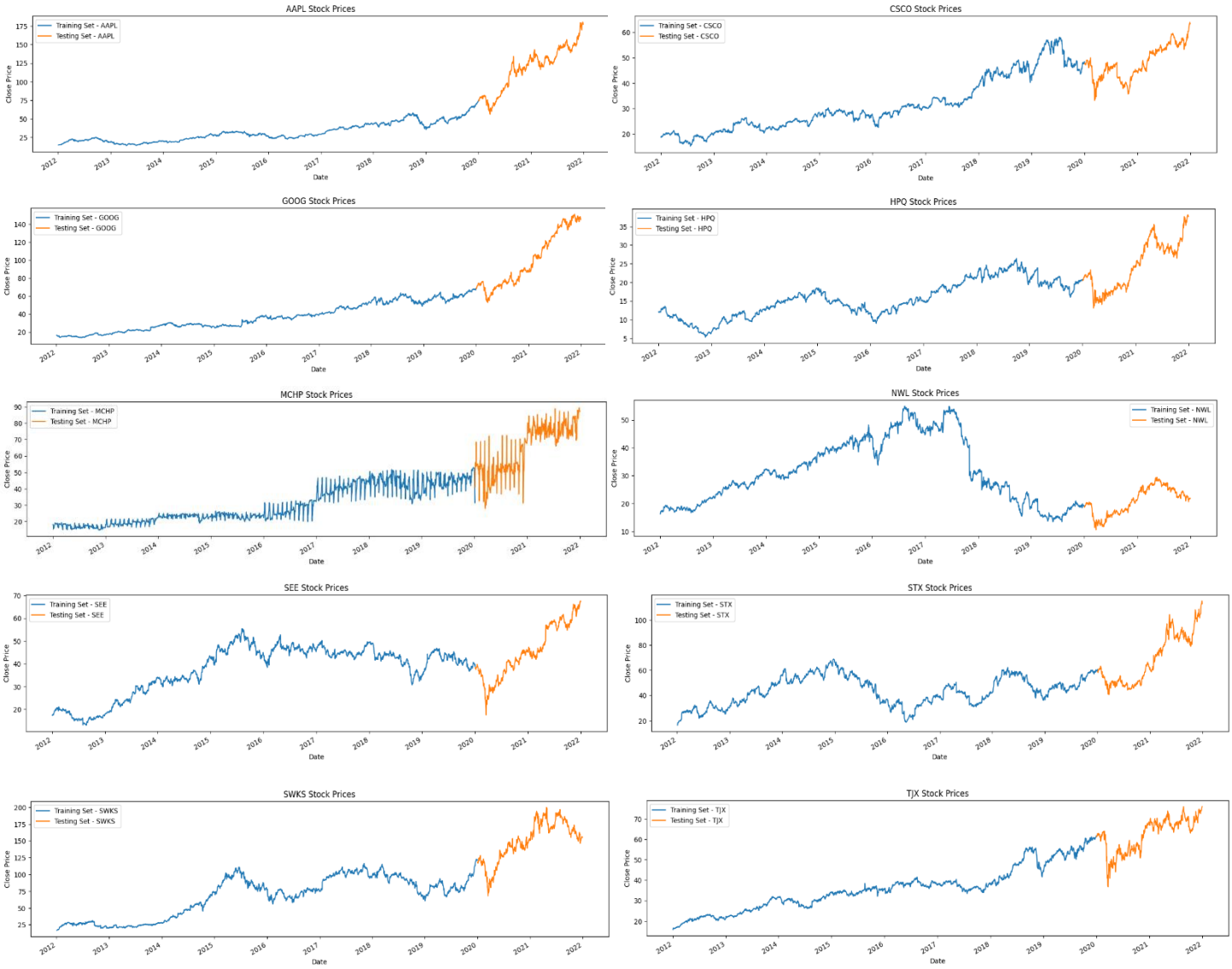
Plotting Stock Prices for Each Company

```
for company in companies:
    company_data = data[data['Company'] == company]

    plt.figure(figsize=(16,4))

    # plot for the first 80% of the data for 'Close'
    company_data['Close'].iloc[:int(0.8 * len(company_data))].plot(label=f'Training Set - {company}', legend=True)
    # plot for the last 20% of the data for 'Close'
    company_data['Close'].iloc[int(0.8*len(company_data)):].plot(label=f'Testing Set - {company}', legend=True)

    plt.title(f'{company} Stock Prices')
    plt.xlabel('Date')
    plt.ylabel('Close Price')
    plt.show()
```

This code iterates through each company in the dataset, extracting and plotting the stock price data. For each company, two segments of the data are visualized:

- Training Set (First 80%): The first 80% of the stock price data is used for training the model, where 10% of this set will be further separated for validation during the training process.
- Testing Set (Last 20%): The remaining 20% of the data is reserved for testing, ensuring that the model's predictions are evaluated on unseen data, simulating real-world performance.

The plot displays the stock prices over time for each company, with the training and testing data clearly labeled. This visualization is helpful in confirming that the data is properly divided and ready for training, validation, and testing.



4.1.3 Data Preprocessing

Feature Scaling

```
preprocessing_pipeline = Pipeline([ ('scaler', MinMaxScaler(feature_range=(0, 1)))])
features_scaled = preprocessing_pipeline.fit_transform(features_company)
target_scaler = MinMaxScaler(feature_range=(0, 1))
target_scaled = target_scaler.fit_transform(target_company)
```

The choice of using a pipeline begins with the need to automate and streamline preprocessing steps like scaling and transformation, especially when handling data for multiple companies, ensuring uniformity in processing, reduces manual errors, and maintains modularity, allowing for efficient updates or extensions to the code. Scaling, a critical preprocessing step in LSTM, employed to standardize feature ranges so that no single feature disproportionately influences the model—preventing variables with large magnitudes, such as "Volume," from overshadowing others like "Open" or "Low". To implement this, the Min-Max Scaler is chosen as it effectively rescales features to a uniform range of 0,1 which is ideal for LSTMs. This transformation preserves the original data relationships while ensuring compatibility with activation functions, optimizing the convergence of gradient descent during training and avoiding range compression issues seen with some alternative scalers like the StandardScaler.

4.1.4 Model Development and Training

Training the Model for Multiple Companies

In this implementation, the model is trained individually for each of the 10 companies in the dataset. Using a loop, the data for each company is extracted, preprocessed, and augmented before being passed to the LSTM model. This approach ensures that each company, unique stock price behavior is learned independently, allowing for tailored predictions. The process includes scaling features and targets, creating sequences, splitting data into training, validation, and testing sets, and building the LSTM model with company-specific data. By training the model in this manner, the nuances of each company's stock price patterns are effectively captured.

Sequence Preparation

```
X, y = create_sequences(augmented_features, augmented_targets, time_steps=60)
```

The prepared data is converted into sequences, structured to provide the LSTM with a consistent temporal context for prediction. Each sequence consists of 60-time steps (as defined by time_steps), ensuring the model has sufficient historical data for accurate forecasting. This step transforms raw data into a format tailored for sequence-based learning, allowing the LSTM to capture temporal dependencies effectively.

Data Splitting

```
split_ratio = 0.8
train_size = int(len(X) * split_ratio)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

val_size = int(len(X_train) * 0.1)
X_val, X_train = X_train[:val_size], X_train[val_size:]
y_val, y_train = y_train[:val_size], y_train[val_size:]
```

This part of the pipeline splits the processed data into three subsets—training, validation, and testing—each serving distinct purposes:

- Split Ratio (0.8): Defines the proportion of data used for training and testing. Here, 80% of the data is allocated for training and 20% for testing. This ensures a substantial portion of the data is available for the model to learn patterns while reserving enough unseen data to evaluate its performance.

- Training and Testing Splits: The train_size variable computes the number of samples allocated to training based on the split ratio. X_train and y_train are used to fit the model, while X_test and y_test evaluate its predictions on unseen data.

- Validation Split (10%): The training data is further divided to create a validation set. The val_size variable calculates 10% of the training data, ensuring the model has a separate subset for tuning hyperparameters and avoiding overfitting. The remaining portion of X_train and y_train forms the actual training set used for weight updates during backpropagation.

Each subset ensures that the model is trained, validated, and tested effectively, mimicking real-world scenarios where models must generalize well to unseen data.

Model Architecture

```
model = Sequential([
    Input(shape=(X_train.shape[1], X_train.shape[2])),
    LSTM(128, return_sequences=True,
         kernel_initializer=he_uniform(),
         kernel_regularizer=l1_l2(l1=1e-5, l2=1e-4),
         bias_regularizer=l1_l2(l1=1e-5, l2=1e-4)),
    Dropout(0.3),
    LSTM(64, return_sequences=False,
         kernel_initializer=he_uniform(),
         kernel_regularizer=l1_l2(l1=1e-5, l2=1e-4),
         bias_regularizer=l1_l2(l1=1e-5, l2=1e-4)),
    Dropout(0.3),
    Dense(1, kernel_initializer=he_uniform())
])
```

The model is designed using a sequence of layers in Keras, tailored to capture temporal dependencies in stock price data while preventing overfitting. Each layer serves a unique role in this architecture.

Input Layer: The input layer defines the shape of the data entering the model. For this implementation, the input shape consists of time_steps (e.g., 60) and the number of features (e.g., Open, High, Low, Volume). This layer ensures compatibility with the LSTM layers by structuring the data into sequences.

LSTM Layers: Two LSTM layers form the backbone of the model:

- First LSTM Layer: This layer consists of 128 hidden units, designed to process sequential dependencies in the input data. The parameter return_sequences=True ensures that the output of this layer is a sequence, which is passed to the subsequent LSTM layer for further processing. This is crucial for capturing temporal relationships over time.
- Second LSTM Layer: With 64 hidden units, this layer summarizes the sequential information into a single fixed-length vector. The parameter return_sequences=False ensures that the layer outputs only the final state, which is then fed to the dense layer for prediction.

Kernel Initializer: The he_uniform initializer initializes the weights of the LSTM layers by sampling values from a uniform distribution. This initializer is particularly well-suited for layers with ReLU or similar activation functions, ensuring efficient training by maintaining variance across layers. It helps prevent vanishing or exploding gradients during backpropagation, facilitating stable and faster convergence.

Kernel Regularizer and Bias Regularizer: Regularization is applied to both kernel (weights) and bias parameters using the l1_l2 method:

- L1 Regularization: Encourages sparsity by penalizing the absolute values of the weights, effectively driving some weights to zero. This helps in feature selection and prevents the model from relying on irrelevant features.
- L2 Regularization: Penalizes the square of the weights, discouraging large weight values and improving generalization by avoiding overfitting. Together, l1_l2 regularization strikes a balance between sparsity and generalization, resulting in a more robust model.

Dropout Layers: Two dropout layers, each with a rate of 0.3, are strategically placed after the LSTM layers. Dropout randomly deactivates 30% of neurons during training, forcing the model to learn distributed representations and reducing its dependency on specific neurons. This regularization technique enhances the model's ability to generalize to unseen data.

Dense Layer: The final layer is a Dense layer with a single neuron, responsible for producing the stock price prediction. This scalar output represents the forecasted stock price based on the processed sequential data. The use of he_uniform initialization ensures that the weights of this layer are initialized optimally for convergence.

This architecture is meticulously crafted to balance learning complexity and generalization, making it well-suited for the temporal nature of stock price prediction.

Compilation and Training

```
optimizer = Adam(learning_rate=0.0005)
model.compile(optimizer=optimizer, loss='mean_squared_error')
history = model.fit(
    X_train, y_train,
    epochs=150,
    batch_size=64,
    validation_data=(X_val, y_val),
    callbacks=[EarlyStopping(monitor='val_loss',patience=10),
LearningRateScheduler(lr_scheduler)],
    verbose=1
)
```

The model is compiled and trained using the following configuration, which optimizes both learning efficiency and accuracy:

Adam Optimizer: The Adam optimizer is chosen for its adaptive learning rate capabilities. By combining the advantages of momentum and RMSProp optimizers, Adam dynamically adjusts the learning rate for each parameter during training. This ensures stable convergence and faster training, even in the presence of noisy gradients.

Loss Function: The loss function used is Mean Squared Error (MSE), which quantifies the difference between the predicted and actual stock prices. By squaring the errors, MSE penalizes larger deviations more heavily, ensuring that the model prioritizes accuracy in its predictions.

Callbacks:

- Early Stopping: Monitors the validation loss during training and halts the process if no improvement is observed for 10 consecutive epochs. This prevents overfitting and saves computational resources by stopping training once the model reaches optimal performance.
- Learning Rate Scheduler: Dynamically reduces the learning rate by 10% after the initial 10 epochs, refining the weight updates and enhancing convergence during later stages of training. This adjustment ensures that the model does not overshoot the optimal parameter values.

Training Configuration: The model is trained for up to 150 epochs with a batch size of 64. The use of batch training improves computational efficiency and stabilizes gradient updates. The validation set provides real-time feedback, helping to fine-tune the model during training by monitoring its performance on unseen data.

By incorporating these components, the model's compilation and training process is optimized for both accuracy and generalizability, ensuring its reliability in real-world stock price forecasting scenarios.

Model Summary

To verify the constructed architecture, the model's structure is summarized, and a visual representation is generated.

```
model.summary()
```

This command provides a concise overview of the model, including the number of layers, the output shapes of each layer, and the total number of trainable and non-trainable parameters. This step ensures that the architecture matches the intended design, verifying the correct connection of layers and appropriate parameter configurations.

Model: "sequential_9"

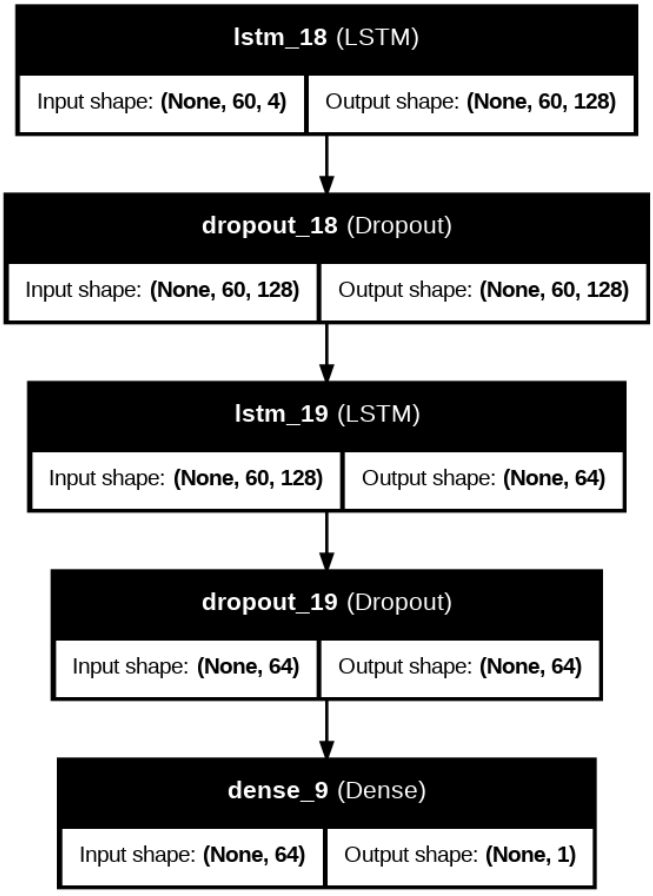
Layer (type)	Output Shape	Param #
lstm_18 (LSTM)	(None, 60, 128)	68,096
dropout_18 (Dropout)	(None, 60, 128)	0
lstm_19 (LSTM)	(None, 64)	49,408
dropout_19 (Dropout)	(None, 64)	0
dense_9 (Dense)	(None, 1)	65

Total params: 352,709 (1.35 MB)
Trainable params: 117,569 (459.25 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 235,140 (918.52 KB)

Model Architecture Visualization

```
plot_model(
    model,
    to_file='model_architecture.png',
    show_shapes=True,
    show_layer_names=True,
    dpi=96
)
Image('model_architecture.png')
```

The plot_model function generates an image of the model architecture, displaying the flow of data through the layers, including the input and output shapes. This visual representation enhances understanding of the model's structure and facilitates troubleshooting by making it easier to identify misconfigured layers or mismatched dimensions.



4.1.5 Predictions and Evaluation

After training, the model is tested on unseen data to assess its predictive accuracy and generalization capability.

```
predictions = model.predict(X_test).reshape(-1, 1)
predictions_inverse = target_scaler.inverse_transform(predictions)
y_test_inverse = target_scaler.inverse_transform(y_test)
```

Predictions: The trained model generates predictions for the testing set (X_test). These predictions, initially in the normalized range due to scaling, are transformed back to their original scale using the target_scaler.inverse_transform function. This inverse transformation ensures that the predicted stock prices are directly comparable to the actual prices in their original scale.

```
plt.figure(figsize=(14, 7))
plt.plot(y_test_inverse, label='Actual Prices', color='blue')
plt.plot(predictions_inverse, label='Predicted Prices', color='red')
plt.title(f'{company} Stock Price Prediction')
plt.xlabel('Days')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```

Prediction Plot: This visualization compares the actual and predicted stock prices over the testing period. The blue line represents the actual prices, while the red line depicts the predicted prices. The closeness of the two lines indicates the model’s accuracy and ability to capture trends.

```
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Training vs. Validation Loss: This graph shows the loss values for both training and validation sets over the course of training. It highlights the model's learning progression and indicates whether overfitting or underfitting occurred. Ideally, the validation loss closely follows the training loss, indicating a well-generalized model.

Performance Metrics

To evaluate the model quantitatively, several metrics are calculated on the testing set predictions.

```
mse = mean_squared_error(y_test_inverse, predictions_inverse)
mae = mean_absolute_error(y_test_inverse, predictions_inverse)
mape = mean_absolute_percentage_error(y_test_inverse, predictions_inverse)
rmse = np.sqrt(mse)
```

Mean Squared Error (MSE): Measures the average squared difference between predicted and actual values. It penalizes larger errors more heavily, making it a useful metric for identifying significant deviations in predictions.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Mean Absolute Error (MAE): Provides the average absolute difference between predicted and actual values. MAE offers an intuitive measure of prediction accuracy by focusing on direct deviations without squaring errors.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Mean Absolute Percentage Error (MAPE): Quantifies prediction accuracy as a percentage of actual values. This metric is particularly useful for comparing performance across datasets with different scales.

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100\%$$

Root Mean Squared Error (RMSE): Represents the square root of MSE, translating the error metric back into the original unit of the data. RMSE provides a more interpretable measure of prediction error in the context of stock prices.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

These metrics together offer a comprehensive evaluation of the model's performance, capturing both absolute and relative prediction accuracy.

```
metrics_summary['MAPE'].append(mape)
metrics_summary['MSE'].append(mse)
metrics_summary['MAE'].append(mae)
metrics_summary['RMSE'].append(rmse)
```

Metrics Aggregation: For each company, the calculated metrics (MAPE, MSE, MAE, and RMSE) are appended to the metrics_summary dictionary. This allows for an organized collection of results across all companies, enabling comparative analysis and identification of trends in model performance.

Prediction Results

```
predicted_df = pd.DataFrame({
    'Date': company_data.index[-len(y_test_inverse):],
    'Company': company,
    'Actual Prices': y_test_inverse.flatten(),
    'Predicted Prices': predictions_inverse.flatten()
})
all_predictions_df = pd.concat([all_predictions_df, predicted_df], ignore_index=True)
```

Results Storage: A DataFrame is created to store the actual and predicted stock prices for each company. The all_predictions_df combines the results for all companies, ensuring that predictions are organized and accessible for further analysis.

```
avg_metrics = {key: np.mean(values) for key, values in metrics_summary.items()}
```

Aggregate Metrics: After looping through all companies, the average MAPE, MSE, MAE, and RMSE are calculated. These aggregated metrics provide a high-level overview of the model's performance across the entire dataset, offering insights into its robustness and consistency.

```
print("Average Metrics Across Companies:")
for key, value in avg_metrics.items():
    print(f"{key}: {value}")
```

The average metrics are printed to summarize the model's overall performance. These values help identify areas for potential improvement and set benchmarks for future iterations of the model.

4.2 Code Explanation for the Forecast of Stocks

This code is designed to forecast stock market closing prices for specific open trading dates in 2022. It utilizes historical stock data, a pre-trained Long Short-Term Memory (LSTM) model, and feature scaling techniques to generate predictions for multiple companies. The program systematically processes the data, makes predictions, and organizes the results for analysis and visualization.

4.2.1 Data Loading and Preprocessing

```
forecasted_dates = pd.read_csv('/content/forecast_data.csv')
forecasted_dates['Date'] = pd.to_datetime(forecasted_dates['Date'])
historical_data = pd.read_csv('/content/stock_data.csv')
historical_data['Date'] = pd.to_datetime(historical_data['Date'])
```

This segment loads two essential datasets:

- forecast_data.csv: Contains the dates and company names for which predictions are required.
- stock_data.csv: Provides historical data such as open, high, low, volume, and close prices for various companies.

To ensure the datasets can be efficiently processed for time-series analysis, the Date column in both files is converted into a datetime format. This transformation allows the program to perform date-specific operations, such as filtering and sorting.

Model Loading

```
model = load_model('/content/stock_prediction_model.h5')
```

The code loads the pre-trained LSTM model stored in an H5 file. This model has already been trained to analyze historical stock data and predict closing prices. By using this pre-trained model,

the code skips the training phase, significantly reducing computational time and effort. LSTM is particularly suited for this task due to its ability to learn temporal patterns in sequential data.

4.2.2 Initialization

```
forecasted_results = pd.DataFrame(columns=['Date', 'Company', 'Forecasted Close'])
```

A blank DataFrame is created to hold the final output. Each row of this DataFrame will represent a forecasted result, including:

- The target date.
- The company name.
- The predicted closing price.

This structure ensures all results are organized for easy saving and analysis.

Feature and Time Step Definition

```
time_steps = 150
features = ['Open', 'High', 'Low', 'Volume']
```

A time window of 150 previous observations (time steps) is defined to capture patterns for prediction. Features such as open, high, low, and volume are selected for analysis. The close price is excluded during prediction to avoid information leakage.

4.2.3 Forecasting Loop

Company-wise Iteration

```
for company in forecasted_dates['Company'].unique():
    company_forecast_data = forecasted_dates[forecasted_dates['Company'] == company]
    company_data = historical_data[historical_data['Company'] == company].tail(time_steps)

    if company_data.empty or len(company_data) < time_steps:
        print(f"Skipping {company} due to insufficient historical data.")
```

The program loops through each unique company in the forecast dataset. For each company:

- The forecasting process iterates through each company in the forecast_data.csv. For each company, the last 150 rows of historical data are extracted. If a company's historical data is insufficient, the script skips further analysis for that company.

Feature Scaling

```
company_data_features = company_data[features].values
feature_scaler = MinMaxScaler(feature_range=(0, 1))
features_scaled = feature_scaler.fit_transform(company_data_features)
```

The historical data for the selected features (open, high, low, and volume) is normalized to a range of 0 to 1 using MinMaxScaler. This step ensures that all features have equal weight during model predictions and prevents large values from skewing the results. The scaler is later used to reverse the scaling for interpretable results.

Sequence Preparation and Forecasting

```
last_sequence = features_scaled[-time_steps:]
X_test = np.array([last_sequence])
forecast_steps = len(company_forecast_data)
predictions = []

for _ in range(forecast_steps):
    predicted_close = model.predict(X_test, verbose=0)[0]
    predictions.append(predicted_close)
    new_sequence = np.concatenate(
        (X_test[0][1:], np.array([[predicted_close[0]] * 4])),
        axis=0
    )
    X_test = np.array([new_sequence])
```

The initial sequence serves as the starting point for predictions and is composed of the last 150 data points from the company's historical stock data. These points include features like opening price, high price, low price, and volume, scaled to a range between 0 and 1. The model uses this sequence to recognize patterns and trends in the stock's historical performance, setting the stage for accurate forecasting

This iterative process predicts closing prices for all target dates specified in the forecast data. The LSTM model generates one prediction at a time, and the sequence is dynamically updated with the newly forecasted value. By appending this value to the sequence and removing the oldest data point, the model ensures continuity and adaptability for multi-step forecasting. This approach captures potential dependencies between consecutive time steps and aligns the predictions with the company's recent trends and fluctuations.

Inverse Scaling

```
predictions = np.array(predictions)
predictions = feature_scaler.inverse_transform(np.column_stack([predictions] * 4))[:, 0]
```

The scaled predictions are transformed back to their original range using the same MinMaxScaler. The transformation ensures that the predicted values are meaningful and directly comparable to historical data.

Result Compilation

```
forecast_df = pd.DataFrame({
   'Date': company_forecast_data['Date'].iloc[:forecast_steps],
   'Company': company,
   'Forecasted Close': predictions
})
forecasted_results = pd.concat([forecasted_results, forecast_df], ignore_index=True)
```

A new DataFrame is created for each company's predictions, containing the forecasted dates, company names, and predicted closing prices. These DataFrames are concatenated into the main forecasted_results DataFrame.

Saving Results

```
forecasted_results.to_csv('forecasted_results_tuned.csv', index=False)
```

The final DataFrame is saved as a CSV file named forecasted_results_tuned.csv, enabling further use or sharing of the forecasted data.

Visualization Function

```
# Visualize forecasted data
def plot_combined_prices(predicted_data, forecast_data, historical_data):
    companies = predicted_data['Company'].unique()
    for company in companies:
        company_predicted_data = predicted_data[predicted_data['Company'] == company]
        company_forecasted_data = forecast_data[forecast_data['Company'] == company]
        company_historical_data = historical_data[historical_data['Company'] == company]

        plt.figure(figsize=(10, 6))
        plt.plot(company_historical_data['Date'], company_historical_data['Close'],
            label=f"{company} - Actual Prices", color='green')
        plt.plot(company_predicted_data['Date'], company_predicted_data['Predicted
Prices'], label=f"{company} - Predicted Prices", color='blue')
        plt.plot(company_forecasted_data['Date'], company_forecasted_data['Forecasted
Close'], label=f"{company} - Forecasted Prices", color='orange')
        plt.title(f"Stock Prices for {company} - Historical vs Forecasted")
        plt.xlabel('Date')
        plt.ylabel('Prices')
        plt.xticks(rotation=45)
        plt.legend()
        plt.tight_layout()
        plt.grid(True)
        plt.show()


# Plot combined prices
plot_combined_prices(predicted_df, forecast_results, historical_data)

print("Tuned forecast process completed and results saved.")
```

A visualization function is defined to plot historical data, predicted values, and forecasted prices. This function provides insights into the model's accuracy and the predicted trends for each company. The forecasted results are visualized by plotting the predicted closing prices for each company. The plots illustrate the forecasted trends, making it easier to assess the model's performance.

4.3 Code Explanation for the Portfolio Creation and Optimization

Calculating Predicted Returns for Each Company

```
def calculate_predicted_returns(df):
    company_returns = df.groupby('Company').agg(
        start_price=('Forecasted Close', 'first'),
        end_price=('Forecasted Close', 'last')
    ).reset_index()

    company_returns['Predicted_Return'] = (company_returns['end_price'] -
company_returns['start_price']) / company_returns['start_price'] * 100
    return company_returns
```

This function calculates the predicted returns for each company based on their forecasted closing prices. By grouping the data by Company, it determines the starting and ending prices for the forecasted period. The predicted return for each company is computed as the percentage change between the starting and ending prices using the formula:

$$\text{Predicted Return (\%)} = \frac{\text{End Price} - \text{Start Price}}{\text{Start Price}} \times 100$$

This output serves as the foundation for constructing and evaluating portfolios, highlighting the potential growth or decline of each company's stock.

Calculating the Covariance Matrix

```
def calculate_covariance_matrix(df):
    company_returns_pivot = df.pivot(index='Date', columns='Company',
values='Forecasted Close').pct_change()
    cov_matrix = company_returns_pivot.cov()
    return cov_matrix
```

The covariance matrix quantifies the relationship between the returns of different companies. The function pivots the forecasted prices into a format where each column represents a company, and rows represent dates. It calculates percentage changes in these prices to represent daily returns and computes their covariance matrix. The covariance matrix is essential for risk assessment in portfolio optimization, as it identifies how the returns of different companies move together.

Mathematically, the covariance between two assets i and j is given by:

$$\text{Cov}(i,j) = \frac{1}{N-1}\sum_{t=1}^{N}(R_{i,t} - \bar{R}_i)(R_{j,t} - \bar{R}_j)$$

Where $R_{i,t}$ and $R_{j,t}$ are the returns of assets i and j at time t.

Mathematically Optimized Portfolio Construction

```
# Code 1: Portfolio Construction with Adjusted Weights Based on Return Sign (Mathematically Optimized)
def construct_max_return_portfolios(company_returns, cov_matrix, num_portfolios=1, companies_per_portfolio=3):
    portfolios = []
    selected_companies = set()  # Track selected companies globally (across portfolios)

    # Sort companies by predicted returns in descending order to get the highest returns first
    sorted_companies = company_returns.sort_values(by="Predicted_Return", ascending=False)  # Include both positive and negative returns

    for _ in range(num_portfolios):
        valid_portfolio = False

        while not valid_portfolio:
            # Select companies that have not been chosen yet for other portfolios
            available_companies = sorted_companies[~sorted_companies['Company'].isin(selected_companies)]

            if available_companies.shape[0] < companies_per_portfolio:
                break  # Exit if we don't have enough companies to create the portfolios

            # Pick top companies based on predicted returns
            top_candidates = available_companies.head(companies_per_portfolio).copy()  # Make a copy of the DataFrame
            companies = top_candidates['Company'].values
            returns = top_candidates['Predicted_Return'].values
            cov_sub_matrix = cov_matrix.loc[companies, companies].values

            # Adjust weights based on return positivity/negativity
            total_positive_return = np.sum(returns[returns > 0])  # Sum of positive returns
            total_negative_return = np.sum(np.abs(returns[returns < 0]))  # Sum of absolute negative returns

            # Calculate weights:
            weights = np.zeros_like(returns)
            for i in range(len(returns)):
                if returns[i] > 0:
                    # For positive returns: Higher weight proportionally
                    weights[i] = returns[i] / total_positive_return * 100
                elif returns[i] < 0:
                    # For negative returns: Assign smaller weights but proportional to the magnitude of negative return
                    weights[i] = np.abs(returns[i]) / total_negative_return * 25  # Small percentage for negative returns (scaled down)

            # Normalize weights to ensure they sum to 100%
            weights = weights / np.sum(weights) * 100

            # Assign weights and calculate portfolio return
            top_candidates['Weight'] = weights
            portfolio_return_value = np.dot(weights, returns)

            # Only add the portfolio if the return is valid
            if portfolio_return_value > 0:  # Only add portfolios with a positive return
                portfolios.append(top_candidates[['Company', 'Predicted_Return', 'Weight']])
                selected_companies.update(companies)  # Mark companies as selected
                valid_portfolio = True  # Portfolio creation is valid

    return portfolios
```

This function constructs portfolios by selecting companies with the highest predicted returns. It balances positive and negative returns to maximize overall portfolio performance.

- Selection of Companies: Companies are sorted by their predicted returns, and the top-performing companies are selected for each portfolio.

- Weight Assignment: Weights are calculated proportionally to their returns. Positive-return companies are assigned higher weights, while negative-return companies are given smaller weights based on their absolute values.
- Validation: Only portfolios with positive overall returns are considered.

The portfolio return is calculated as:

$$R_p = \sum_{i=1}^{n} w_i R_i$$

Where w_i is the weight assigned to company i, and R_i is its predicted return.

Random Portfolio Construction with Positive Return Constraints

```
# Code 2: Portfolio Construction with Random Selection of Companies and Ensuring One Positive Return with Higher Weight
def construct_unique_portfolios_with_positive(company_returns, num_portfolios=4, stocks_per_portfolio=3, random_seed=42):
    np.random.seed(random_seed)  # Set seed for reproducibility
    portfolios = []

    # Separate companies with positive and negative returns
    positive_companies = company_returns[company_returns['Predicted_Return'] > 0]
    negative_companies = company_returns[company_returns['Predicted_Return'] <= 0]

    for i in range(num_portfolios):
        # Select at least one company with a positive predicted return
        positive_selection = positive_companies.sample(n=1, replace=False)

        # Select the remaining companies randomly from both positive and negative predicted returns
        remaining_selection = pd.concat([positive_selection, negative_companies.sample(n=stocks_per_portfolio - 1, replace=False)])

        # Calculate portfolio weights based on predicted returns (absolute value to avoid negative weights)
        total_return = remaining_selection['Predicted_Return'].abs().sum()
        remaining_selection['Weight'] = remaining_selection['Predicted_Return'].abs() / total_return * 100

        # Ensure positive return companies have higher weights
        positive_companies_in_portfolio = remaining_selection[remaining_selection['Predicted_Return'] > 0]
        remaining_selection.loc[positive_companies_in_portfolio.index, 'Weight'] += 20  # Add 20% more weight to positive returns

        # Normalize weights to ensure they sum to 100%
        remaining_selection['Weight'] = remaining_selection['Weight'] / remaining_selection['Weight'].sum() * 100

        portfolios.append(remaining_selection[['Company', 'Predicted_Return', 'Weight']])

    return portfolios
```

This function introduces randomness in portfolio creation while ensuring at least one company with a positive return is included.

- Positive Return Guarantee: At least one positive-return company is included in each portfolio with a higher weight.

- Random Selection: Other companies are chosen randomly from both positive and negative-return groups.
- Weight Adjustment: Weights are proportional to the absolute predicted returns, and positive-return companies receive an additional boost.

Weights are normalized to ensure they sum to 100%. This approach diversifies portfolios and emphasizes the contribution of positively performing assets.

Visualization of Portfolio Weights

```
# Plot pie charts for the constructed portfolios
def plot_pie_charts(portfolios):
    num_portfolios = len(portfolios)

    # Check if there's only one portfolio, and adjust axes accordingly
    if num_portfolios == 1:
        fig, ax = plt.subplots(figsize=(5, 5), sharey=False)  # Get a single Axes object
        axes = [ax]  # Wrap it in a list to make it iterable
    else:
        fig, axes = plt.subplots(1, num_portfolios, figsize=(num_portfolios * 5, 5), sharey=False)  # Get an array of Axes objects

    for i, (portfolio, ax) in enumerate(zip(portfolios, axes)):
        companies = portfolio['Company'].values
        weights = portfolio['Weight'].values

        ax.pie(weights, labels=companies, autopct='%1.1f%%', startangle=90, colors=plt.cm.Paired.colors)
        ax.set_title(f"Portfolio P{i+1}", fontsize=12)
        ax.axis('equal')

    plt.tight_layout()
    plt.show()

plot_pie_charts(portfolios)
```

This function visualizes the weight distribution of companies within each portfolio using pie charts. It provides insights into portfolio composition, highlighting the emphasis on high-return companies and the diversification achieved through the allocation of weights.

Portfolio Optimization Techniques

Equal-Weighted Portfolio

```
# Equal Weighted Portfolio
def equal_weighted_portfolio(portfolio):
    n = len(portfolio)
    weights = np.array([1/n] * n)  # Equal weights
    return weights
```

An equal-weighted portfolio assigns the same weight to all assets, irrespective of their predicted returns or risks. This straightforward approach ensures that no single company disproportionately influences the portfolio. While simple, it does not account for variations in return or risk among companies.

Monte Carlo Simulation

```
# Monte Carlo Simulation Portfolio Optimization
def monte_carlo_optimization(portfolio, num_simulations=50000):
    returns = portfolio["Predicted_Return"].values
    cov_matrix = np.cov(returns)

    num_stocks = len(returns)
    results = np.zeros((3, num_simulations))
    weight_array = []

    for i in range(num_simulations):
        weights = np.random.random(num_stocks)
        weights /= np.sum(weights)
        weight_array.append(weights)

        # Calculate portfolio return and risk
        portfolio_return = np.sum(returns * weights)
        portfolio_risk = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))
        sharpe_ratio = portfolio_return / portfolio_risk

        # Store results
        results[0, i] = portfolio_return
        results[1, i] = portfolio_risk
        results[2, i] = sharpe_ratio

    # Find the portfolio with the maximum Sharpe ratio
    max_sharpe_idx = np.argmax(results[2])
    optimal_weights = weight_array[max_sharpe_idx]

    return optimal_weights
```

The monte_carlo_optimization function simulates thousands of random weight combinations to identify the optimal portfolio. Each simulation calculates the portfolio's return, risk, and Sharpe ratio (return-to-risk ratio). The portfolio with the highest Sharpe ratio is selected as the optimal choice. This method leverages computational power to explore a wide range of possibilities, identifying a balance between return and risk.

The Sharpe ratio is given by:

$$\text{Sharpe Ratio} = \frac{R_p}{\sigma_p}$$

Where R_p is the portfolio return, and σ_p is the portfolio risk.

Mean-Variance Optimization

```
# Mean-Variance Optimization
def mean_variance_optimization(portfolio):
    returns = portfolio["Predicted_Return"].values
    cov_matrix = np.cov(returns)
    num_stocks = len(returns)

    def objective_function(weights):
        portfolio_return = np.sum(returns * weights)
        portfolio_risk = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))
        return portfolio_risk - portfolio_return  # Minimize risk and maximize return

    constraints = {"type": "eq", "fun": lambda weights: np.sum(weights) - 1}
    bounds = tuple((0, 1) for _ in range(num_stocks))
    initial_weights = np.array([1/num_stocks] * num_stocks)

    result = minimize(objective_function, initial_weights, bounds=bounds, constraints=constraints)
    return result.x
```

The mean_variance_optimization function employs a mathematical approach to minimize risk while maximizing return. By utilizing the covariance matrix and predicted returns, this optimization technique assigns weights that achieve the best risk-return trade-off. Constraints ensure that weights sum to 1 and remain within realistic bounds (e.g., no short selling).
The objective function is:

$$\min \sigma_p^2 = w^T \Sigma w$$

Where w is the weight vector, and Σ is the covariance matrix.

Constraints ensure that $\sum w_i = 1$ and $0 \leq w_i \leq 1$

Calculating Performance Metric

```
# Step 4: Calculate performance metrics for each portfolio
def calculate_metrics(portfolio, weights):
    returns = portfolio["Predicted_Return"].values
    portfolio_return = np.sum(returns * weights)
    portfolio_risk = np.sqrt(np.dot(weights.T, np.dot(np.cov(returns), weights)))
    sharpe_ratio = portfolio_return / portfolio_risk
    return portfolio_return, portfolio_risk, sharpe_ratio
```

The calculate_metrics function evaluates portfolios based on three metrics:

- Portfolio Return: The weighted sum of predicted returns, indicating the overall profitability of the portfolio.

- Portfolio Risk: The standard deviation of returns, derived from the covariance matrix, representing the volatility of the portfolio.
- Sharpe Ratio: The ratio of return to risk, measuring risk-adjusted performance. Higher Sharpe ratios indicate better performance relative to risk taken.

These metrics provide a comprehensive understanding of each portfolio's performance.

Benchmarking Portfolios

```
# Function to calculate portfolio return
def calculate_portfolio_return(portfolio):
    # The return for each stock in the portfolio
    returns = portfolio["Predicted_Return"].values
    # The benchmark return is the weighted average of the individual returns based on their weights
    benchmark_return = np.mean(returns)  # Average return of companies in the portfolio
    return benchmark_return

# Assuming portfolios is a list of DataFrames containing the portfolios
portfolio_metrics_with_benchmark = []

# Calculate the benchmark return for each portfolio and compare it to the portfolio return
for i, portfolio in enumerate(portfolios):
    # Calculate the predicted return for the portfolio
    eq_weights = equal_weighted_portfolio(portfolio)
    eq_metrics = calculate_metrics(portfolio, eq_weights)

    # Monte Carlo Optimization
    mc_weights = monte_carlo_optimization(portfolio)
    mc_metrics = calculate_metrics(portfolio, mc_weights)

    # Mean-Variance Optimization
    mvo_weights = mean_variance_optimization(portfolio)
    mvo_metrics = calculate_metrics(portfolio, mvo_weights)

    # Calculate the benchmark return as the average return of the companies in the portfolio
    benchmark_return = calculate_portfolio_return(portfolio)

    # Calculate active return (portfolio return - benchmark return)
    eq_active_return = eq_metrics[0] - benchmark_return
    mc_active_return = mc_metrics[0] - benchmark_return
    mvo_active_return = mvo_metrics[0] - benchmark_return

    portfolio_metrics_with_benchmark.append({
        "Portfolio": f"P{i+1}",
        "EQ_Return_Percentage": eq_metrics[0],
        "Benchmark_Percentage": benchmark_return,
        "EQ_Active_Return_Percentage": eq_active_return,
        "MC_Return_Percentage": mc_metrics[0],
        "MC_Active_Return_Percentage": mc_active_return,
        "MVO_Return_Percentage": mvo_metrics[0],
        "MVO_Active_Return_Percentage": mvo_active_return,
    })
```

The benchmark return for each portfolio is calculated as the average return of the companies it contains. This benchmark provides a reference point to evaluate the active return, which is the difference between the portfolio's return and its benchmark. Active returns indicate the added value from using optimization techniques.

Active return is calculated as,

$$\text{Active Return} = R_p - R_b$$

Where R_b is the benchmark return.

Finally, using bar and line plots, a comparison of the returns and Sharpe ratios of portfolios were constructed using LSTM predictions. The comparison focuses on portfolios created with different optimization strategies, such as Equal-Weighted, Monte Carlo Simulation, and Mean-Variance Optimization, providing insights into their relative performance.

5. Results

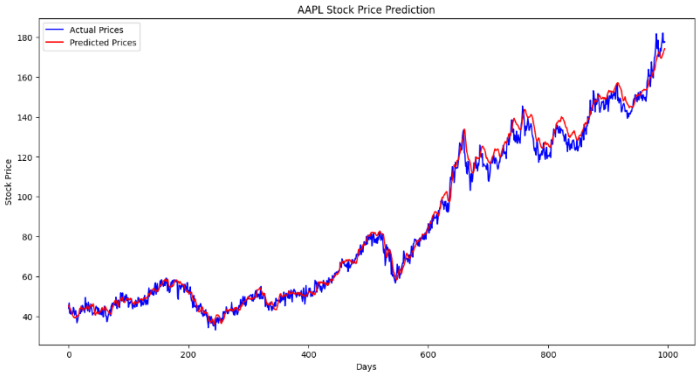
5.1 Stock Prediction Results

Company-Wise Results

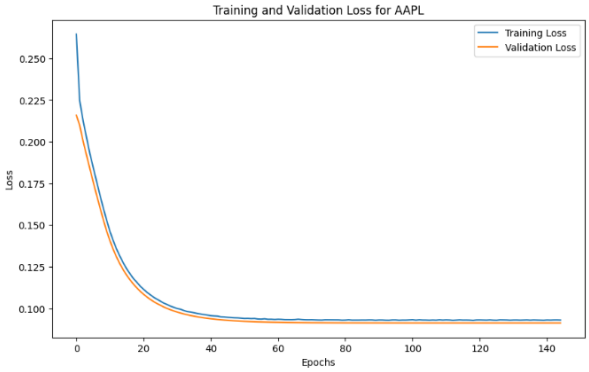
This section highlights the evaluation and visualization of the model's performance for each individual company. Using a loop, the model is trained separately for each company, ensuring that company-specific stock price patterns are captured effectively. For every company, the following results are generated and analyzed:

Company AAPL:

Prediction vs Actual Graph:



Training vs Validation Loss Curve:

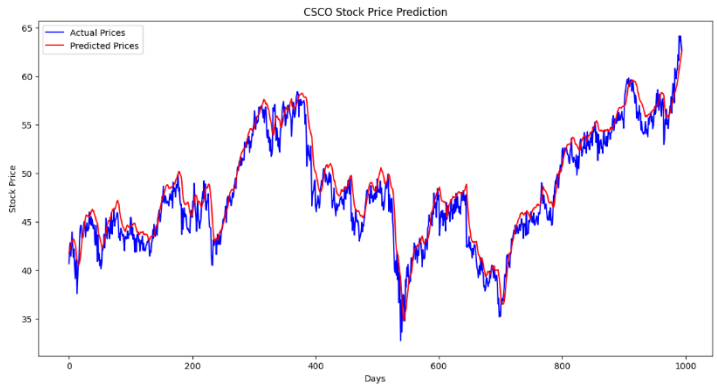


Performance Metrics:

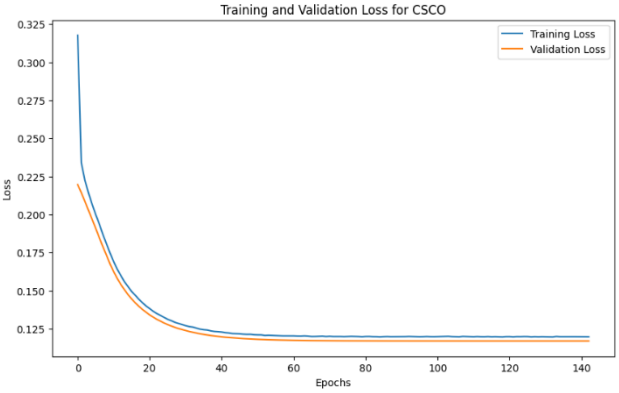
- MAPE for AAPL: 0.03531936971802011
- MSE for AAPL: 14.191056459685917
- MAE for AAPL: 2.816498296677783
- RMSE for AAPL: 3.767101864787561

Company CSCO:

Prediction vs Actual Graph:



Training vs Validation Loss Curve:

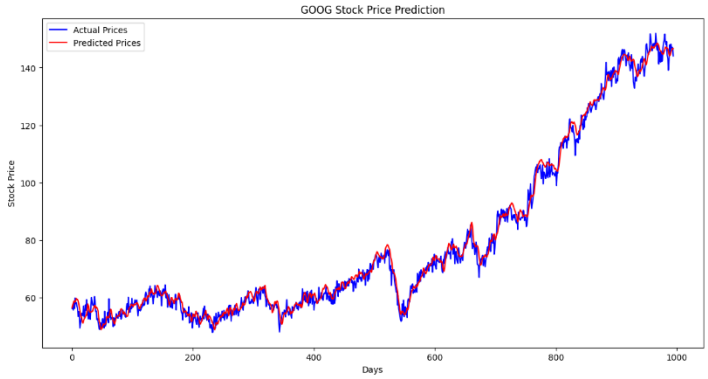


Performance Metrics:

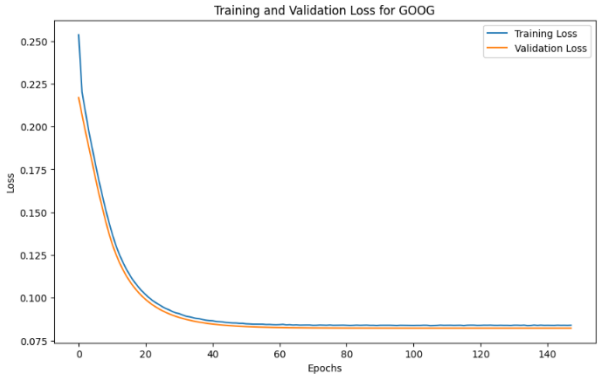
- MAPE for CSCO: 0.02757197020153071
- MSE for CSCO: 2.759521605001061
- MAE for CSCO: 1.2827331282018897
- RMSE for CSCO: 1.661180786368859

Company GOOG:

Prediction vs Actual Graph:



Training vs Validation Loss Curve:

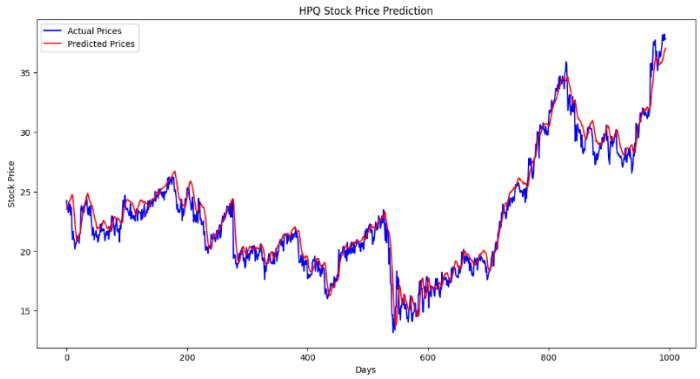


Performance Metrics:

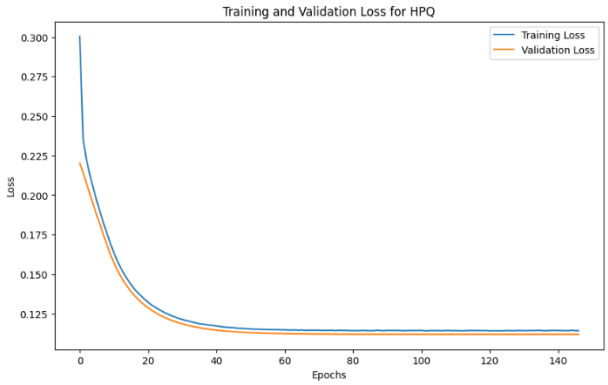
- MAPE for GOOG: 0.027289451201430632
- MSE for GOOG: 6.407737782809524
- MAE for GOOG: 1.994779854583821
- RMSE for GOOG: 2.53135097977533

Company HPQ:

Prediction vs Actual Graph:



Training vs Validation Loss Curve:

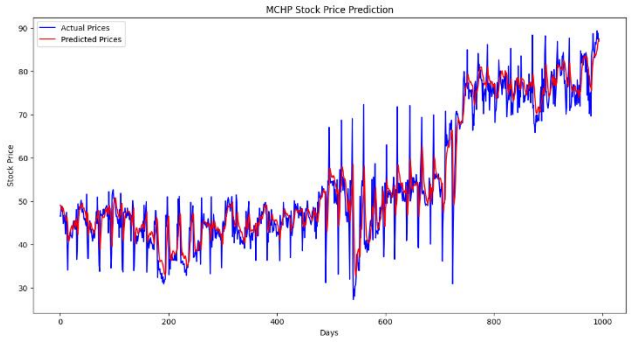


Performance Metrics:

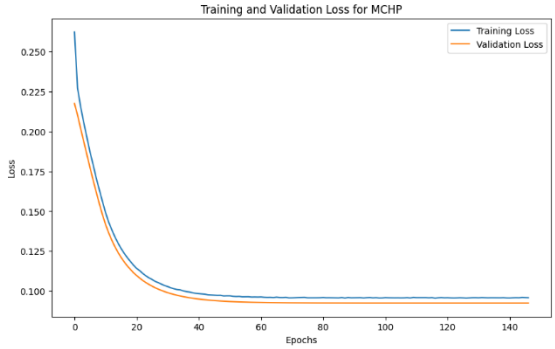
- MAPE for HPQ: 0.03396260418674798
- MSE for HPQ: 0.9998088351152777
- MAE for HPQ: 0.7420853312449448
- RMSE for HPQ: 0.9999044129892005

Company MCHP:

Prediction vs Actual Graph:



Training vs Validation Loss Curve:

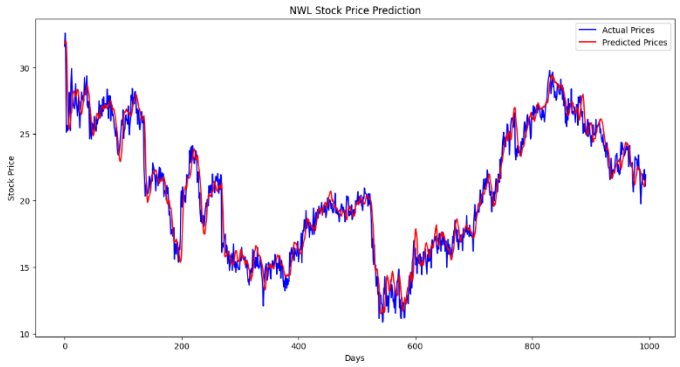


Performance Metrics:

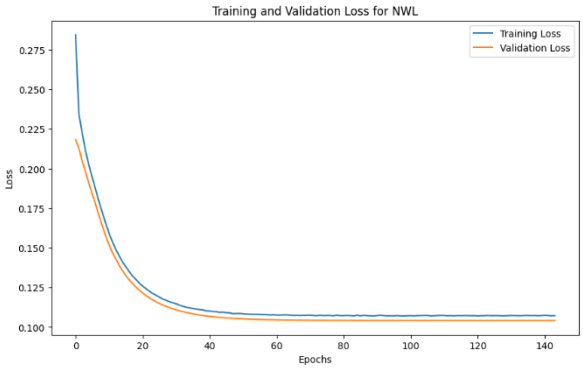
- MAPE for MCHP: 0.06522610202841923
- MSE for MCHP: 23.049704616802543
- MAE for MCHP: 3.2090470034163454
- RMSE for MCHP: 4.801010791156644

Company NWL:

Prediction vs Actual Graph:



Training vs Validation Loss Curve:

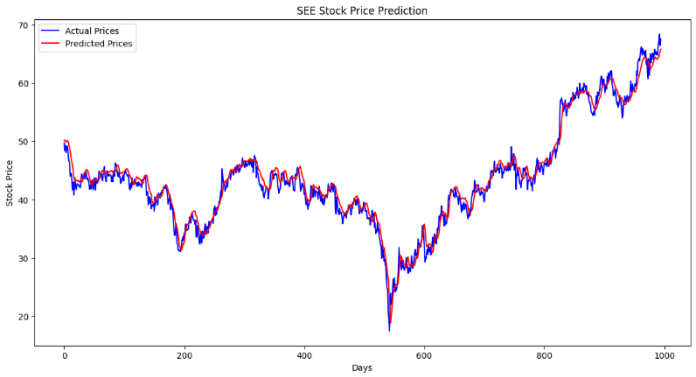


Performance Metrics:

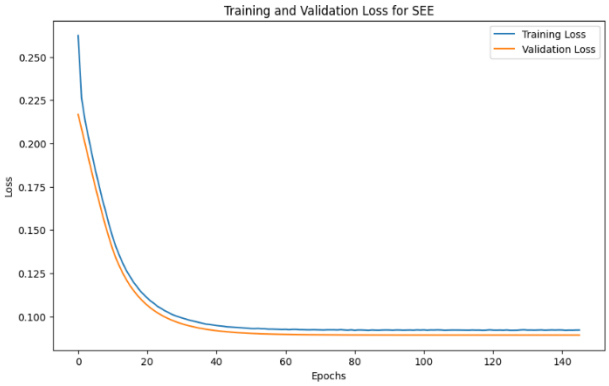
- MAPE for NWL: 0.03915818151835669
- MSE for NWL: 1.0605902267553673
- MAE for NWL: 0.7607796086920608
- RMSE for NWL: 1.029849613659862

Company SEE:

Prediction vs Actual Graph:



Training vs Validation Loss Curve:

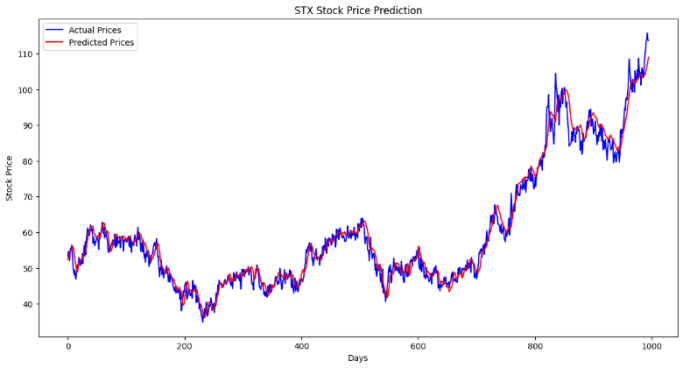


Performance Metrics:

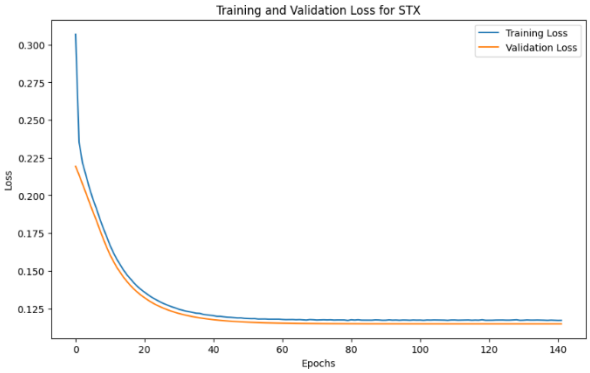
- MAPE for SEE: 0.027141102849700366
- MSE for SEE: 2.0889491122831934
- MAE for SEE: 1.1035727939106041
- RMSE for SEE: 1.4453197266636864

Company STX:

Prediction vs Actual Graph:



Training vs Validation Loss Curve:

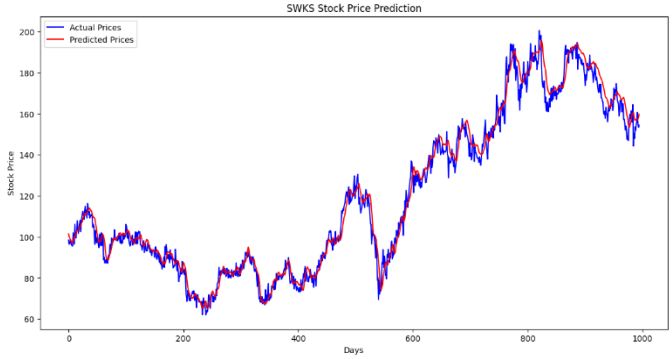


Performance Metrics:

- MAPE for STX: 0.032066278985340806
- MSE for STX: 6.948934607475629
- MAE for STX: 1.916742804445779
- RMSE for STX: 2.6360831943388336

Company SWKS:

Prediction vs Actual Graph:



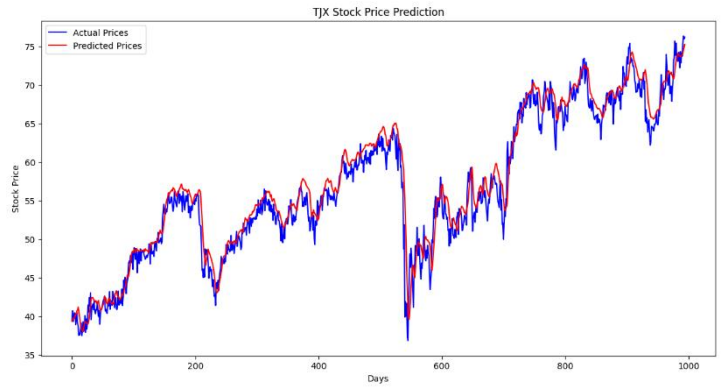
Training vs Validation Loss Curve:

Performance Metrics:

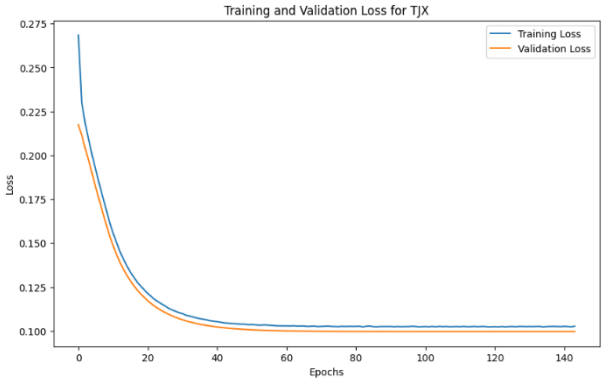
- MAPE for SWKS: 0.032591514886575336
- MSE for SWKS: 25.645406668972672
- MAE for SWKS: 3.8011495523802843
- RMSE for SWKS: 5.064129408790091

Company TJX:

Prediction vs Actual Graph:



Training vs Validation Loss Curve:



Performance Metrics:

- MAPE for TJX: 0.02762640904855148
- MSE for TJX: 4.139453137926848
- MAE for TJX: 1.500799437059455
- RMSE for TJX: 2.0345646064764935

Overall Performance Evaluation

To assess the overall effectiveness of the model across all companies, the average metrics are calculated and analyzed. These aggregate metrics provide a high-level summary of the model's performance, highlighting its robustness and consistency when applied to diverse datasets.

Average Metrics Across Companies:

- MAPE: 0.03479529846246733
- MSE: 8.729116305282805
- MAE: 1.9128187810612967
- RMSE: 2.5970495385006562

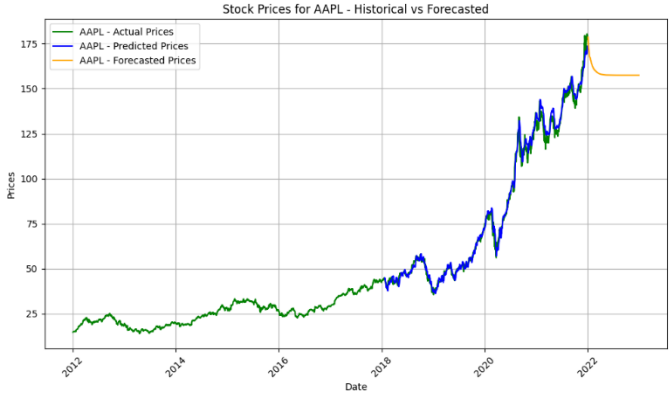
These values indicate that the model achieves a high level of accuracy, with minimal errors and strong generalizability across all companies. The low Mean Absolute Percentage Error (MAPE) demonstrates the model's ability to make precise predictions relative to the actual stock prices, while the Root Mean Squared Error (RMSE) suggests that the average prediction deviation remains within acceptable bounds.

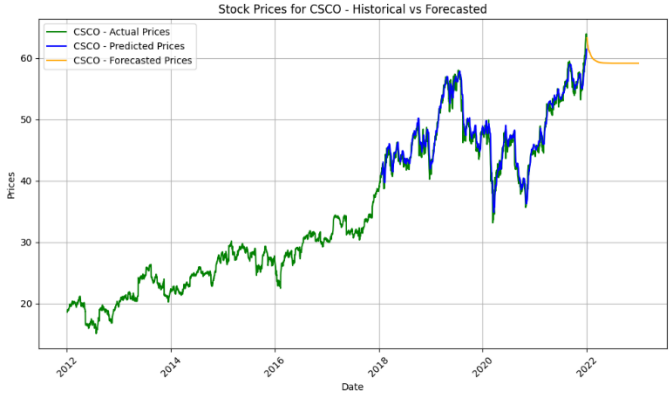
The performance evaluation confirms that the LSTM-based approach, coupled with tailored training for each company, effectively captures the complex temporal patterns in stock prices. This approach lays a strong foundation for further enhancements, such as incorporating additional features, optimizing hyperparameters, or extending the model to other financial datasets. The results affirm the model's potential as a reliable tool for stock price forecasting and portfolio optimization.

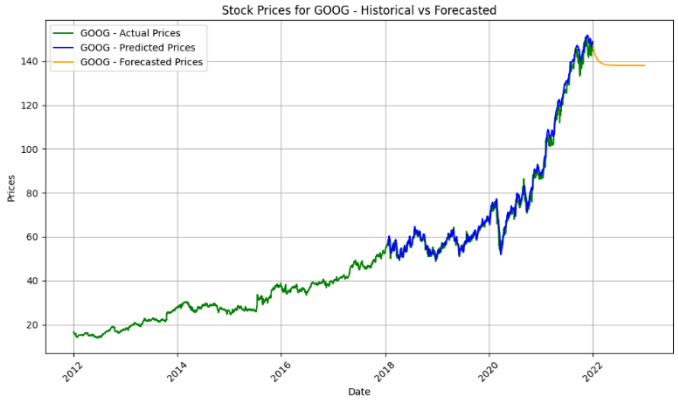
5.2 Forecasting Plots and Analysis

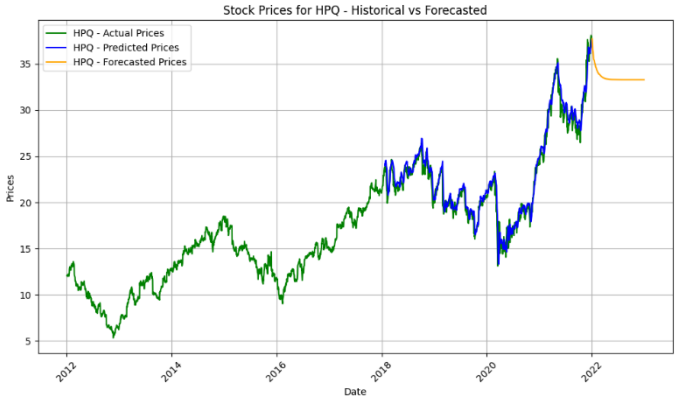
The next phase of this analysis involves generating and presenting forecasting plots for each company. These plots visualize the predicted stock price trends over the forecasting period, providing valuable insights into future market movements and the model's capability to generalize beyond the historical data.

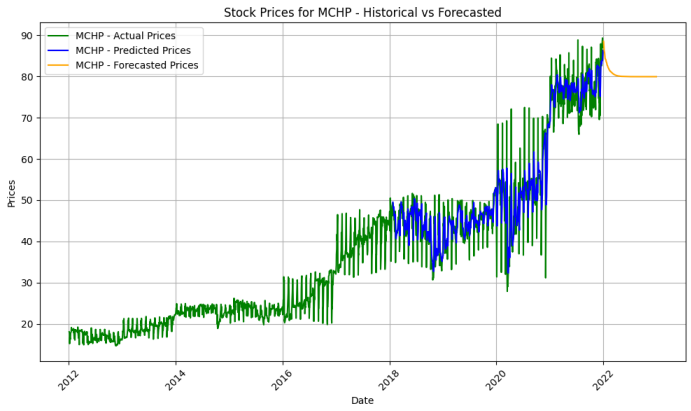
The forecasting for the 10 companies is the following:

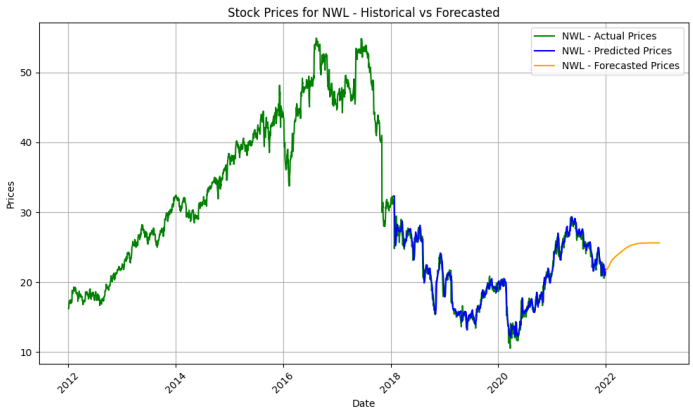


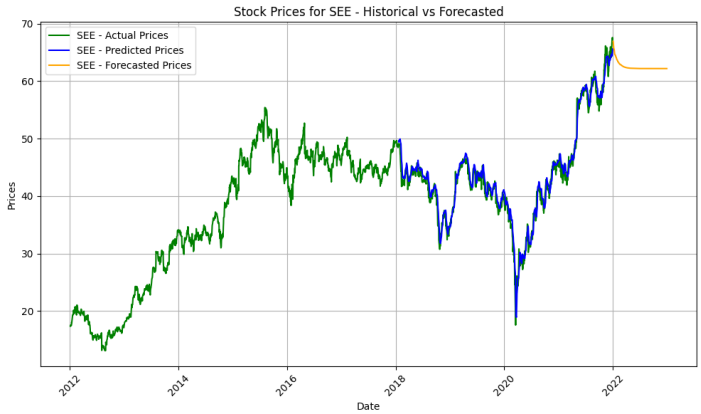


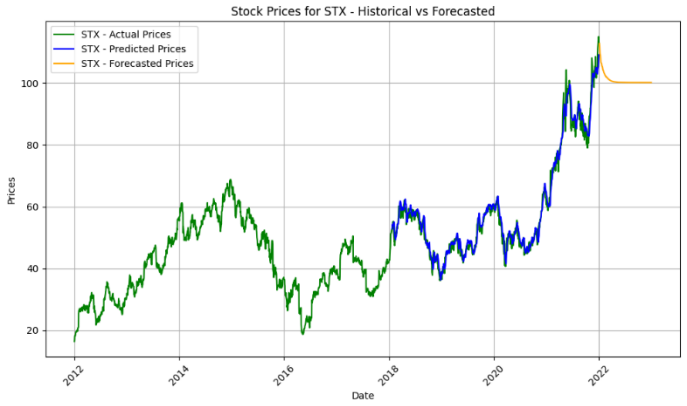


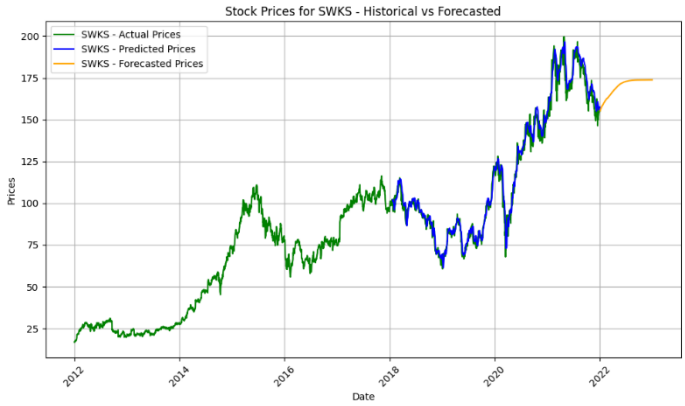


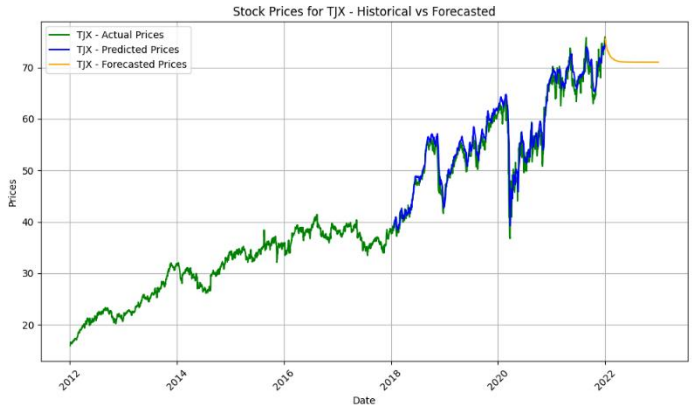












The forecasting results reveal the model's capability to identify and project both upward and downward trends in stock prices for each company. The model demonstrates a strong ability to capture these patterns, which are crucial for understanding potential future movements. While the forecasts align closely with historical trends, there is potential for further improvement in predictive precision.

With additional hyperparameter tuning and refinement of the data preprocessing steps, the model could achieve even greater accuracy in predicting future stock prices. The current results affirm the robustness of the LSTM-based approach, showcasing its utility for financial forecasting while highlighting avenues for enhancement. This emphasizes the model's viability as a predictive tool in quantitative trading and portfolio optimization.

5.3 Portfolio Results and Optimization Analysis

The final part of this analysis involves presenting the results of the portfolios created using the predicted stock prices. These portfolios are constructed based on optimal weights, aiming to maximize returns or minimize risk. The results include detailed visualizations and tables to summarize portfolio composition and performance.

5.3.1. Portfolio Composition:

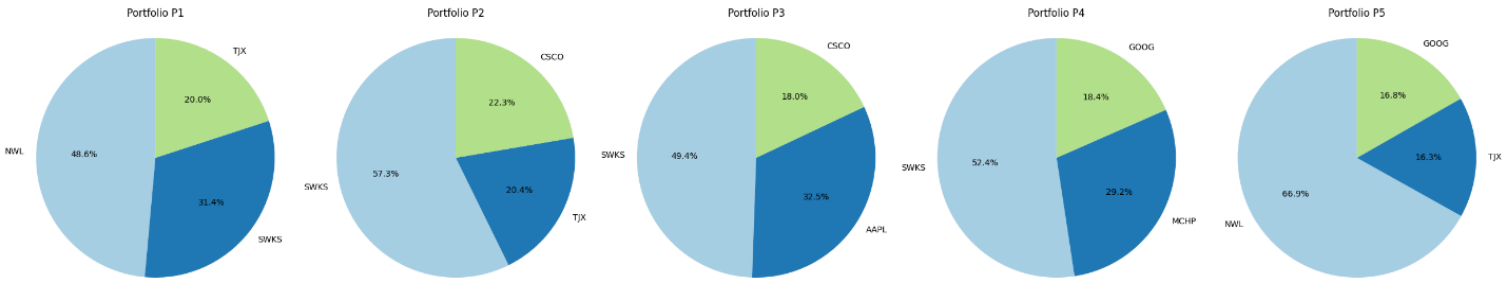
The first portfolio is constructed based on a mathematical concept, considering the predicted performance of the 10 companies selected in the initial dataset. Out of these, only two companies show an upward trend, while the remaining eight exhibit a downward trend. The portfolio is optimized to select the best possible combination, ensuring that it capitalizes on the upward-trending companies.

The remaining four portfolios are created by randomly selecting company sets. While these portfolios explore potential combinations, they lack the rigor of optimization applied to the first portfolio.

Portfolio	Company	Predicted Return	Weight
P1	NWL	18.346549	48.575727
	SWKS	11.868623	31.424273
	TJX	-5.953702	20.000000
P2	SWKS	11.868623	57.299036
	TJX	-5.953702	20.382570
	CSCO	-6.519152	22.318394
P3	SWKS	11.868623	49.447405
	AAPL	-11.783964	32.546915
	CSCO	-6.519152	18.005680

P4	SWKS MCHP GOOG	11.868623 -9.680004 -6.124128	52.407665 29.150224 18.442110
P5	NWL TJX GOOG	18.346549 -5.953702 -6.124128	66.918442 16.307378 16.774179

The below five pie charts are presented, each representing one portfolio and showing the weight distribution among the three companies. These charts provide an intuitive understanding of how the portfolios are balanced.



5.3.2 Portfolio Optimization Tables:

Tables summarizing the expected returns, risks (standard deviation), and Sharpe Ratios for each portfolio. These metrics quantify the performance of the portfolios and highlight their efficiency in achieving the desired investment objectives.

Portfolio	EQ_Return	EQ_Risk	EQ_Sharpe	MC_Return	MC_Risk	MC_Sharpe	MVO_Return	MVO_Risk	MVO_Sharpe
P1	8.087	7.265	1.113	15.837	9.121	1.736	16.373	9.553	1.713
P2	-0.201	6.037	-0.033	11.482	10.236	1.121	11.868	10.456	1.135
P3	-2.144	7.169	-0.299	11.670	12.306	0.948	11.868	12.418	0.955
P4	-1.311	6.669	-0.196	11.718	11.462	1.022	11.868	11.552	1.027
P5	2.089	8.128	0.257	18.024	13.893	1.297	18.346	14.079	1.303

5.3.3 Benchmark and Active Return Analysis:

To evaluate the performance of each portfolio against a benchmark, the average return of the companies within the portfolio is calculated as the benchmark return. The active return, which is the portfolio return minus the benchmark return, is then assessed to measure the effectiveness of the optimization strategies.

This analysis provides insights into the efficiency of the optimization strategies employed. The mathematically optimized portfolio demonstrates higher active returns compared to the benchmark, underscoring the value of systematic approaches in portfolio construction.

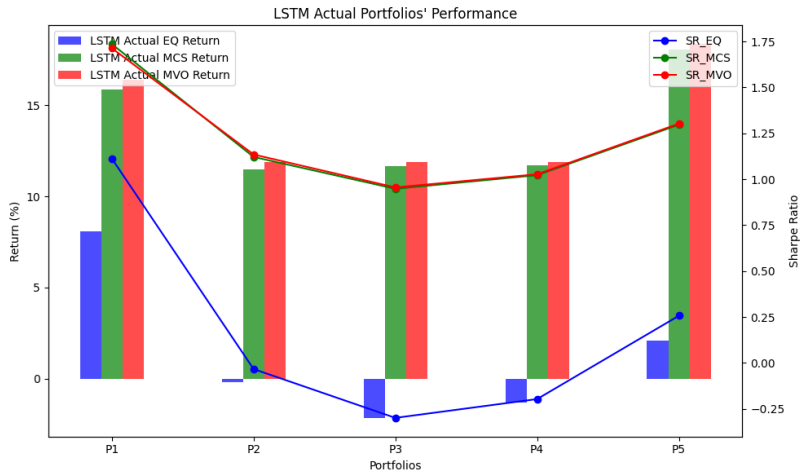
Portfolio	EQ Return Percentage	Benchmark Percentage	EQ Active Return Percentage	MC Return Percentage	MC Active Return Percentage	MVO Return Percentage	MVO Active Return Percentage
P1	8.087	8.087	0.00e+00	15.837	7.715	16.373	8.286
P2	-0.201	-0.201	4.44e-16	11.482	11.863	11.868	12.070
P3	-2.144	-2.144	4.44e-16	11.670	13.822	11.868	14.013
P4	-1.311	-1.311	2.22e-16	11.718	13.092	11.868	13.180
P5	2.089	2.089	0.00e+00	18.024	16.029	18.346	16.256

5.3.4. Portfolio Performance Visualization

To provide an in-depth comparison of portfolio performance, two sets of visualizations are presented. These visualizations focus on comparing portfolio returns and Sharpe Ratios across the three optimization strategies: Equal Weighting (EQ), Monte Carlo Simulation (MCS), and Mean-Variance Optimization (MVO).

LSTM Actual Portfolios' Performance

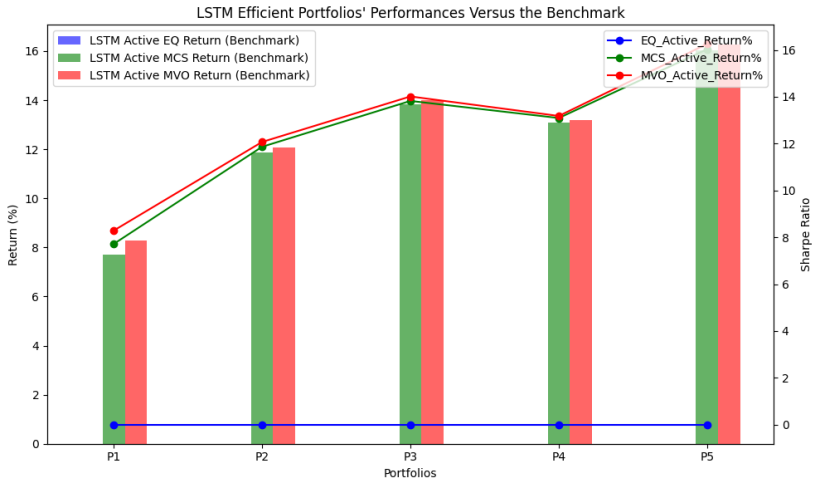
This visualization compares the actual portfolio returns and Sharpe Ratios. The bar graph shows the percentage return for each portfolio, categorized by optimization method (EQ, MCS, MVO). Alongside, the line graph plots the Sharpe Ratios, indicating the risk-adjusted returns for each strategy. This dual-axis plot highlights how well the portfolios balance return and risk.



As shown in the graph, the portfolio created using Mean-Variance Optimization (MVO) (P1) outperforms the others, delivering higher returns and a better risk-adjusted performance than the portfolios constructed using random combinations (EQ and MCS).

LSTM Actual Returns vs. Benchmarks

The second plot compares the portfolio returns to their respective benchmarks. Bars represent the actual returns for EQ, MCS, and MVO strategies, while the line graph overlays the Sharpe Ratios. This visualization reveals how effectively each portfolio exceeds its benchmark and demonstrates the impact of optimization techniques. (Note that the Active Eq Return and the Active EQ Return Percentage holds extremely smaller values)



Key Observations from the above 2 plots:

- Portfolios optimized with MVO typically exhibit higher Sharpe Ratios, reflecting better risk-adjusted returns.
- Benchmarks serve as a baseline, and the active returns illustrate the additional value generated by the optimization techniques.

These plots provide a comprehensive overview of portfolio performance, combining absolute returns and risk-adjusted metrics to assess the effectiveness of the LSTM-based prediction model in portfolio optimization.

6. Discussion

The findings of this research indicate that Long Short-Term Memory (LSTM) networks excel at predicting stock price patterns, providing significant accuracy. This enhanced accuracy directly translates into improved portfolio performance when the predictions are leveraged for asset allocation. The combination of LSTM predictions with Mean-Variance Optimization (MVO) demonstrated consistent outperformance in both returns and risk-adjusted measures such as the Sharpe Ratio.

The superior performance of the LSTM-based approach aligns with similar findings in prior research, such as the study in the research paper, where LSTM models demonstrated robustness in handling non-linear patterns and time-dependent data. The current research reinforces these results by applying LSTM predictions to portfolio optimization techniques, further validating the reliability of this method for quantitative trading applications.

Additionally, the mathematical concepts applied within the portfolio optimization process, particularly MVO, proved reliable and effective. This was demonstrated through rigorous testing and comparison with portfolios created by random combinations of companies. The optimized portfolios consistently outperformed these randomly generated portfolios, confirming that the structured approach to asset selection and weight allocation yields superior results. The observed low error rates and consistent returns highlight the robustness of both the forecasting model and the optimization framework.

The methodologies presented were rigorously tested, and the combination of predictive accuracy and mathematical optimization ensures the reliability of the entire process. Given these outcomes, the proposed system can be confidently employed for building efficient portfolios, supporting both theoretical insights and practical applications in financial markets.

7. Conclusion and Future Work

7.1 Summary of Findings

The research presented a comprehensive analysis of stock price prediction using Long Short-Term Memory (LSTM) networks and demonstrated their efficacy in capturing complex market patterns with low error rates (measured via metrics like MAE and RMSE). These predictions facilitated the construction of optimized portfolios using Mean-Variance Optimization (MVO), Monte Carlo Simulation (MCS), and Equal-Weight (EQ) methods. MVO consistently delivered superior performance in terms of returns and risk-adjusted metrics compared to EQ and MCS approaches, validating the robustness of the optimization process.

So it highlights that the LSTM's ability to better handle time-series complexities, resulting in portfolios with higher returns and improved Sharpe Ratios. These findings align with and extend previous research, demonstrating the accuracy and reliability of the methods used.

7.2 Limitations

Despite the promising results, several limitations were observed during the study. The dataset size was constrained, limiting the scope of the analysis, which also made the number of portfolios to be created using the mathematical concepts to be limited. Additionally, the absence of external influencing factors, such as sentiment analysis and macroeconomic indicators, may have impacted prediction accuracy. The computational power required to process larger datasets and more complex models posed challenges. Furthermore, MVO's reliance on the assumption of normally distributed returns and the random selection of portfolios led to occasional suboptimal outcomes, particularly for markets lacking upward-trending companies.

7.3 Future Work

To enhance the robustness and applicability of the model, future work will focus on several key improvements. Expanding the dataset to include more companies and longer historical timeframes will provide more comprehensive predictions and diverse portfolios. Incorporating external factors such as sentiment analysis, market news, and macroeconomic indicators will help refine forecasting accuracy. Additionally, experimenting with alternative techniques like Transformer models or hybrid deep learning architectures may yield more precise predictions.

A significant future extension includes developing a real-time prediction and portfolio optimization system, allowing dynamic updates based on live market data. Moreover, tailoring the system for the S&P 20 index of the Colombo Stock Exchange will enhance its relevance for local investors, offering practical tools for portfolio management within the Sri Lankan market.

8. References

Ta, V., Liu, C., & Tadesse, D. A. (2020b). Portfolio Optimization-Based stock prediction using Long-Short Term memory network in quantitative trading. *Applied Sciences*, *10*(2), 437. https://doi.org/10.3390/app10020437

Foy, P. (2024, November 6). *Python for Finance: Portfolio Optimization*. MLQ.ai. https://blog.mlq.ai/python-for-finance-portfolio-optimization/

Gallant, C. (2024, November 11). *4 steps to building a profitable portfolio*. Investopedia. https://www.investopedia.com/financial-advisor/steps-building-profitable-portfolio/

Kumar, A. (2024, August 18). *MSE vs RMSE vs MAE vs MAPE vs R-Squared: When to Use?* Analytics Yogi. https://vitalflux.com/mse-vs-rmse-vs-mae-vs-mape-vs-r-squared-when-to-use/

Team, K. (n.d.). *Keras documentation: Developer guides*. https://keras.io/guides/