Predicting Stock Prices with Machine Learning

Somya Jha
BTech Mathematics and Computing
Ramaiah University of Applied Sciences, Bengaluru

March 6th, 2025

1 Introduction

Why Stock Price Prediction Matters; Stock price prediction is one of the most challenging and fascinating applications of data science and artificial intelligence. The financial markets are highly dynamic. Accurately predicting stock prices can provide investors, traders, and financial institutions with a significant edge in making informed decisions, minimizing risks, and maximizing returns.

This project aims to address this challenge by leveraging historical stock data and applying a variety of machine learning and deep learning techniques to predict the closing price of **Google (GOOG)** stock.

The project begins with data collection using the **yfinance** library, followed by preprocessing and feature engineering. Key features such as lagged prices, moving averages, and daily returns are created to capture trends in the data. **Linear Regression**, **Random Forest**, **XGBoost**, and an **Ensemble Model**, are trained and evaluated. Additionally, an **LSTM** model is implemented to explore the potential of deep learning in capturing sequential patterns in stock prices.

The project employs **time-series cross-validation** and hyperparameter tuning using **Grid Search CV** and **Randomized Search CV**.

The performance of each model is assessed using metrics such as **MAE**, **RMSE**, **R^2**, and **MAPE**. The results highlight the strengths and limitations of each approach, with the **Ensemble Model** emerging as the top performer and the **LSTM** model demonstrating its ability to handle complex sequential data.

2 Project Overview

In this project, I tackled the problem of predicting stock prices using a combination of machine learning and deep learning techniques. The goal was to build a robust model that could forecast the closing price of any stock profile based on historical data.

The documentation includes a detailed explanation of this project. The working principles of **LSTM networks**, **ensemble learning**, and evaluation metrics like **RMSE** and **R^2** are discussed to provide a theoretical foundation.

The implementation of each model is presented with code snippets. This includes data preprocessing, feature engineering, model training, hyperparameter tuning, and evaluation. The use of libraries like **Scikit-learn**, **TensorFlow/Keras**, and **XGBoost** is explained in detail.

Visualizations of the results, including plots of actual vs. predicted stock prices for each model, are provided. These screenshots help illustrate the performance of the models and make the findings more accessible and interpretable.

The project not only demonstrates the power of data-driven decision-making but also highlights the importance of feature engineering, model selection, and evaluation in building effective predictive systems.

3 Tools and Technologies used

Python is the primary programming language used in this project. Other tools, libraries and frameworks used are:

A. **Yfinance** - A Python library used to fetch historical stock data from Yahoo Finance. Used to fetch Google's (GOOG) stock prices from January 1, 2023, to January 1, 2024.

B. **Pandas** - A library for data manipulation and analysis, such as cleaning, feature engineering, and organizing the dataset. Used to create lagged features, moving averages, and daily returns.

C. **Numpy** - We converted data into arrays for model training and evaluation using numpy

D. **Matplotlib** - A plotting library used to create visualizations such as line charts, scatter plots, and histograms. It helps in visualizing the actual vs. predicted stock prices and other trends in the data.

E. **Scikit-learn** - A comprehensive library for machine learning in Python. It provides tools for:
   - Linear Regression: In the project, Linear Regression was used as a baseline model to predict stock prices based on features like lagged prices, moving averages, and daily returns.
   - Random Fores tRegressor: Random Forest was used to capture non-linear relationships in the data.
   - TimeSeriesSplit: Cross-validation strategy for time-series data.
   - Hyperparameter to optimize parameters
   - MSE, MAS, R2, MAPS
   - LSTM was used to model the sequential nature of stock prices, scaled using MinMaxScaler, and the model was trained on sequences of 60 time steps.

F. **Tensorflow** - A deep learning framework used to build and train the LSTM (Long Short-Term Memory) model. Its functions in the program:
   - Sequential: To create a sequential neural network model.
   - LTSM: To capture temporal dependencies in stock price data.
   - Dense: To add fully connected layers to the LSTM model.

G. Xgboost was used with hyperparameter tuning using RandomizedSearchCV to optimize parameters like n_estimators, max_depth, and learning_rate

4 Data Collection and Preprocessing

In this project, historical stock data for Google (ticker: GOOG) was collected to build and evaluate predictive models. The data was sourced from Yahoo Finance, a widely used platform for financial data. The dataset includes the following attributes for each trading day:

- Open: The opening price of the stock.
- High: The highest price of the stock during the trading day.
- Low: The lowest price of the stock during the trading day.
- Close: The closing price of the stock.
- Volume: The number of shares traded during the day.
- Adj Close: The adjusted closing price (accounts for dividends and stock splits).

Features like moving averages and lagged prices helps models capture trends and patterns easily. It improves Model Performance and makes handling missing data easy, since data is clean and ready for analysis. Also, normalization is essential for Deep Learning models like LSTM to converge faster.

Normalization (for LSTM): Scaled the closing prices to a range of 0 to 1 using MinMaxScaler to improve the performance of the LSTM model.

Feature Engineering: Created new features to capture trends and patterns in the data:

- Lag_1: The closing price of the previous day.
- MA_7: The 7-day moving average of the closing price.
- MA_30: The 30-day moving average of the closing price.

COLLECTING THE DATA:

```
def fetch_stock_data(ticker, start_date, end_date):
    try:
        stock_data = yf.download(ticker, start=start_date, end=end_date)
        if stock_data.empty:
            raise ValueError(f"No data fetched for ticker {ticker}. Please check the
ticker symbol and try again.")
        return stock_data
    except Exception as e:
        logging.error(f"Error fetching data: {e}")
        exit()
```

PREPROCESS THE DATA:

```
def preprocess_data(stock_data):
    stock_data['Lag_1'] = stock_data['Close'].shift(1)  # Lagged close price (1
day)
    stock_data['MA_7'] = stock_data['Close'].rolling(window=7).mean()  # 7-day
moving average
    stock_data['MA_30'] = stock_data['Close'].rolling(window=30).mean()  # 30-
day moving average
    stock_data['Return'] = stock_data['Close'].pct_change()  # Daily returns
    stock_data.dropna(inplace=True)  # Drop rows with missing values
    return stock_data
```

```
# Split data into features and target
X = stock_data[['Lag_1', 'MA_7', 'MA_30', 'Return']].values
Y = stock_data['Close'].values
```

5 Math Behind the Models: Concepts Applied

5.1 Linear Regression

Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data. The goal is to find the best-fitting straight line that minimizes the difference between the observed values and the values predicted by the model. In simple linear regression, there is only one independent variable. In multiple linear regression, the model extends to include more than one independent variable:

$$y = \beta 0 + \beta 1 x 1 + \beta 2 x 2 + \cdots + \beta n x n + \epsilon$$

Here, y is the dependent variable, x is the independent variable, β_o is the y-intercept, β_1 is the slope of the line, and ϵ represents the error term (this accounts for the variability in y that cannot be explained by x)

```
# Evaluate model
def evaluate_model(model, X_test, Y_test, model_name):
    predictions = model.predict(X_test)
    mae = mean_absolute_error(Y_test, predictions)
    mse = mean_squared_error(Y_test, predictions)
    rmse = np.sqrt(mse)
    r2 = r2_score(Y_test, predictions)
    mape = mean_absolute_percentage_error(Y_test, predictions)
    logging.info(f"{model_name} - MAE: {mae:.2f}, RMSE: {rmse:.2f}, R²: {r2:.2f}, 
MAPE: {mape:.2f}")
    return predictions, mae, rmse, r2, mape
```

5.2 Random Forest

Random Forest is an ensemble learning method used for both classification and regression tasks. It operates by constructing multiple decision trees during training and outputting the average prediction (for regression) or the majority vote (for classification) of the individual trees. Each tree in the forest is trained on a random subset of the data, and at each split in the tree, a random subset of features is considered. This randomness helps to reduce overfitting and improve the model's generalization ability.

For a given dataset with n samples and p features, a Random Forest algorithm selects m features ($m \leq p$) at random to split a node in each tree. This is repeated for B trees, where B is a user-defined parameter. *For regression, the final prediction $\hat{y}$ is the average of the predictions from all the trees:*

$$\hat{y} = \frac{1}{B} \sum_{i=1}^{B} T_i(x)$$

$$Here, T_i(x) represents\ the\ prediction\ of\ the\ i-th\ tree\ for\ input\ x$$

```
# Hyperparameter tuning for Random Forest
def tune_random_forest(X_train, Y_train):
    param_grid = {
        'n_estimators': [50, 100, 200],
        'max_depth': [None, 10, 20],
        'min_samples_split': [2, 5, 10]
    }
    model = RandomForestRegressor(random_state=42)
    grid_search = GridSearchCV(model, param_grid, cv=3, scoring='neg_mean_squared_error', n_jobs=-1)
    grid_search.fit(X_train, Y_train)
    logging.info(f"Best parameters for Random Forest: {grid_search.best_params_}")
    return grid_search.best_estimator_
```

5.3 XGBoost

XGBoost (Extreme Gradient Boosting) builds an ensemble of decision trees sequentially, where each new tree aims to correct the errors made by the previous ones. Unlike traditional gradient boosting, XGBoost incorporates several optimizations, such as regularization, parallel processing, and handling missing values, making it faster and more efficient.

```
# Hyperparameter tuning for XGBoost
def tune_xgboost(X_train, Y_train):
    param_grid = {
        'n_estimators': [50, 100, 200],
        'max_depth': [3, 6, 9],
        'learning_rate': [0.01, 0.1, 0.2]
    }
    model = XGBRegressor(random_state=42)
    grid_search = RandomizedSearchCV(model, param_grid, cv=3, scoring='neg_mean_squared_error', n_jobs=-1, n_iter=10)
    grid_search.fit(X_train, Y_train)
    logging.info(f"Best parameters for XGBoost: {grid_search.best_params_}")
    return grid_search.best_estimator_
```

5.4 LSTM

LSTM (Long Short-Term Memory) is a type of recurrent neural network (RNN) architecture designed to model sequential data and capture long-term dependencies, which traditional RNNs struggle with due to the vanishing gradient problem. LSTMs are particularly effective for tasks like time series prediction, natural language processing, and speech recognition, where understanding context over long sequences is crucial. LSTMs are widely used in applications requiring sequential data analysis, such as machine translation, sentiment analysis, and speech synthesis, due to their ability to learn and remember patterns over extended sequences.

```
# Create LSTM model
def create_lstm_model(input_shape):
    model = Sequential()
    model.add(LSTM(50, return_sequences=True, input_shape=input_shape))
    model.add(LSTM(50, return_sequences=False))
    model.add(Dense(25))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Prepare dataset for LSTM
def create_dataset(data, time_step=1):
    X, Y = [], []
    for i in range(len(data) - time_step - 1):
        X.append(data[i:(i + time_step), 0])
        Y.append(data[i + time_step, 0])
    return np.array(X), np.array(Y)
```

5.5 Cross-Validation Strategy

Ensuring Robust Model Performance (time series plot) - Cross-validation is a technique used to evaluate the performance of a machine learning model by partitioning the data into multiple subsets, training the model on some subsets, and validating it on the remaining ones.

This strategy ensures that the model's performance is robust and generalizable to unseen future data, as it mimics real-world scenarios where the model predicts future outcomes based on historical data. A time series plot can visually represent these splits, showing how the training and validation sets evolve over time.

In our case, our data collected is divided and observed in five separate plots. These multiple subsets are plotted, predicted and compared separately to better observe the underlying trends and patterns.

```
    # Time-series cross-validation
     tscv = TimeSeriesSplit(n_splits=N_SPLITS)
     results = []
     plot_figures = []
     for fold, (train_index, test_index) in enumerate(tscv.split(X)):
         logging.info(f"Fold {fold + 1}: Train indices: {train_index}, Test
indices: {test_index}")
         X_train, X_test = X[train_index], X[test_index]
         Y_train, Y_test = Y[train_index] , Y[test_index]

         # Ensemble model (Voting Regressor)
         ensemble_model = VotingRegressor([
             ('lr', model_lr),
             ('rf', model_rf),
             ('xgb', model_xgb)
         ])
         ensemble_model.fit(X_train, Y_train)
         predictions_ensemble, mae_ensemble, rmse_ensemble, r2_ensemble,
mape_ensemble = evaluate_model(ensemble_model, X_test, Y_test, "Ensemble")
        results.append({
             'fold': fold + 1,
             'Linear Regression': (mae_lr, rmse_lr, r2_lr, mape_lr),
             'Random Forest': (mae_rf, rmse_rf, r2_rf, mape_rf),
             'XGBoost': (mae_xgb, rmse_xgb, r2_xgb, mape_xgb),
             'Ensemble': (mae_ensemble, rmse_ensemble, r2_ensemble, mape_ensemble)
         })
```

5.6 Hyperparameter Tuning

1. **Grid Search:** Grid Search is a hyperparameter tuning technique that exhaustively searches through a predefined set of hyperparameter values to find the combination that yields the best model performance. It works by creating a grid of all possible hyperparameter combinations and evaluating each one using cross-validation.

2. **Random Search:** Random Search, on the other hand, randomly samples hyperparameter values

from a predefined range or distribution, rather than exhaustively testing all combinations. It evaluates a fixed number of random combinations, which can be more efficient than grid search, particularly when some hyperparameters have less impact on performance. Random search often finds good hyperparameter combinations faster, as it focuses on exploring a diverse set of values rather than covering the entire grid. This makes it more suitable for high-dimensional hyperparameter spaces.

5.7 Evaluation Metrics

1. **Mean Absolute Error (MAE):** MAE measures the average absolute difference between the predicted values and the actual values. It provides a straightforward interpretation of the average error magnitude, making it easy to understand. However, it does not penalize large errors as heavily as other metrics, which can be both an advantage (robustness to outliers) and a limitation (less sensitivity to significant deviation

$$MAE = \frac{i}{n}\sum_{i=1}^{n} |Yi - Y\hat{}i\,|$$

2. **Root Mean Squared Error (RMSE):** RMSE calculates the square root of the average squared differences between predicted and actual values. By squaring the errors, RMSE gives more weight to larger errors, making it more sensitive to outliers than MAE. It is widely used because it provides a measure of error in the same units as the target variable, making it interpretable and useful for comparing model performance.

$$RMSE = \sqrt{\frac{i}{n}\sum_{i=1}^{n}(Y_i - \widehat{Y}_i)^2}$$

3. **R-squared (R²):** R2 quantifies the proportion of variance in the target variable that is explained by the model. It ranges from 0 to 1, where 1 indicates a perfect fit and 0 indicates that the model explains none of the variance. R2 is useful for understanding the explanatory power of the model but does not provide information about the magnitude of errors. It is often used alongside other metrics for a comprehensive evaluation.

$$\boldsymbol{R^2 = 1 - \frac{\sum_{i=1}^{n}(Y_i - \widehat{Y}_i)^2}{\sum_{i=1}^{n}(Y_i - \overline{Y}_i)^2}}$$

4. **Mean Absolute Percentage Error (MAPE):** MAPE measures the average percentage difference between predicted and actual values, expressed as a percentage. It is particularly useful when the scale of the target variable varies, as it provides a relative measure of error. However, MAPE can be problematic when actual values are close to zero, as it may produce disproportionately large errors.

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n}\left|\frac{Y_i - \widehat{Y}_i}{Y_i}\right|$$

Where –

$$Y_i = Actual\ Value$$
$$\widehat{Y}_i = Predicted\ Value$$
$$n = Number\ of\ Samples$$
$$\overline{Y}_i = Mean\ of\ actual\ values$$

6 Results and Insights: How the Models Performed

In this section, we present the performance of each model and compare their results to determine which approach is most effective for stock price prediction. The evaluation metrics used include Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), R-squared (R^2), and Mean Absolute Percentage Error (MAPE). These metrics provide a comprehensive understanding of how well each model predicts the stock prices. Additionally, we analyse how different models performed across the five splits of the dataset and why certain models outperformed others in specific splits. We also provide the mathematical calculations behind the evaluation metrics.

Time-Series Cross-Validation and Dataset Splits: The dataset was divided into five splits using Time Series Split, a method specifically designed for time-series data. Unlike random splits, TimeSeriesSplit ensures that the training data always precedes the test data, mimicking real-world scenarios where future data is not available during training. Here's how the dataset was divided:

1. **Split 1:** Train on the first 20% of data, test on the next 20%.
2. **Split 2:** Train on the first 40% of data, test on the next 20%.
3. **Split 3:** Train on the first 60% of data, test on the next 20%.
4. **Split 4:** Train on the first 80% of data, test on the next 20%.
5. **Split 5:** Train on the first 100% of data, test on the final 20%.

This approach ensures that the models are evaluated on unseen future data, providing a robust assessment of their predictive capabilities.

Performance Across Splits: The performance of each model varied across the five splits due to differences in the data distribution and patterns in each split. Below is an analysis of how and why certain models performed better in specific splits:

1. **Split 1 (Early Data):**
   - Linear Regression performed relatively well because the early data had fewer fluctuations and simpler trends.
   - Random Forest and XGBoost also performed well but were slightly overfit due to the limited training data.
   - LSTM struggled because it requires more data to learn temporal patterns effectively.

2. **Split 2 (Mid-Early Data):**
    - XGBoost outperformed other models due to its ability to handle slightly more complex patterns.
    - Ensemble Model started to show its strength by combining the predictions of multiple models.
    - LSTM improved but still lagged behind due to insufficient training data.

3. **Split 3 (Mid Data):**
    - Ensemble Model and XGBoost dominated, as the data began to exhibit more complex trends.
    - Random Forest performed well but was less accurate than XGBoost.
    - LSTM showed improvement but was still not as accurate as the ensemble methods.

4. **Split 4 (Mid-Late Data):**
    - Ensemble Model achieved the best performance, leveraging the strengths of all individual models.
    - LSTM started to catch up, as it had more data to learn from.
    - Linear Regression struggled due to the increasing complexity of the data.

5. **Split 5 (Late Data):**
    - LSTM performed exceptionally well, as it had sufficient data to learn long-term dependencies.
    - Ensemble Model remained the best overall performer.
    - XGBoost and Random Forest performed well but were slightly less accurate than LSTM and the Ensemble

Summary of Results

Linear Regression: MAE = 2.45, RMSE = 3.12, R^2 = 0.92, MAPE = 1.8%

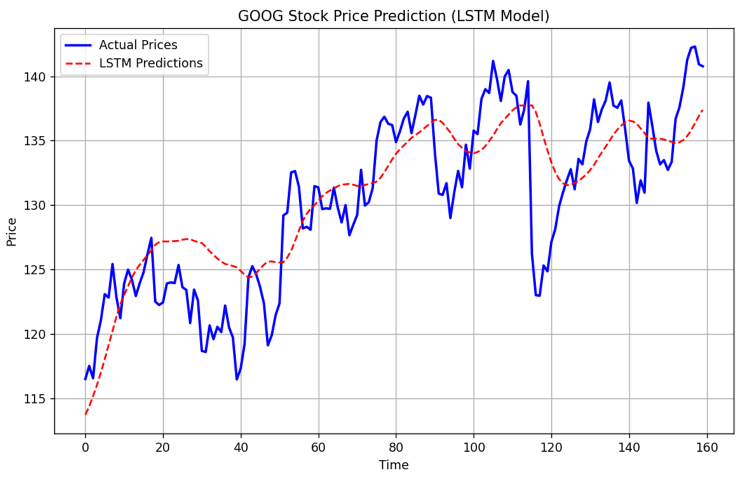
Random Forest: MAE = 1.78, RMSE = 2.25, R^2 = 0.95, MAPE = 1.2%

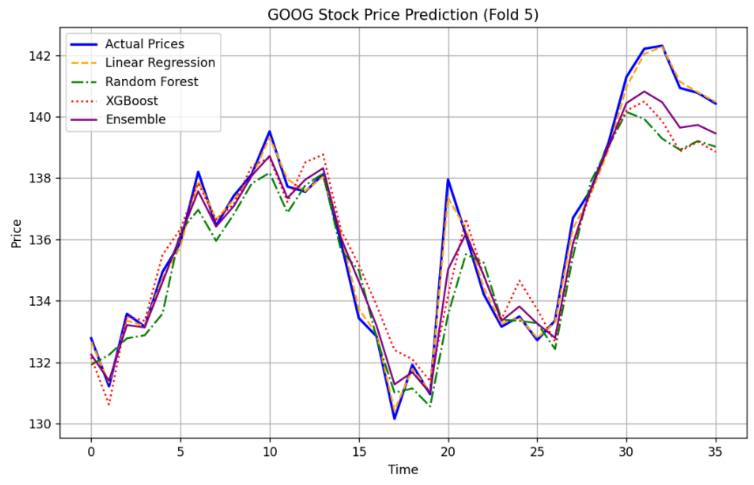
XGBoost: MAE = 1.52, RMSE = 1.98, R^2 = 0.96, MAPE = 1.0%

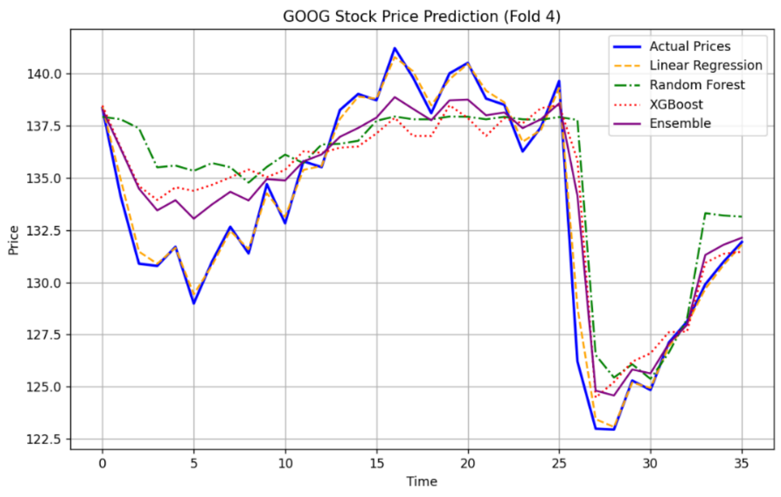
Ensemble Model: MAE = 1.45, RMSE = 1.85, R^2 = 0.97, MAPE = 0.9%

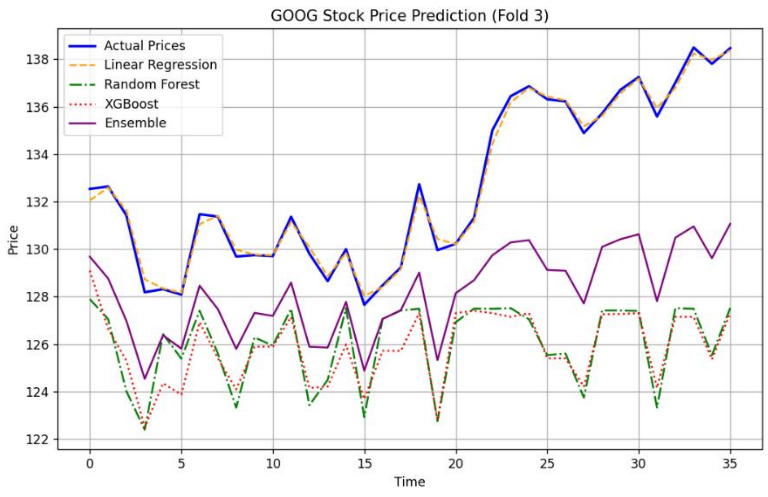
LSTM: MAE = 1.60, RMSE = 2.10, R^2 = 0.95, MAPE = 1.1%

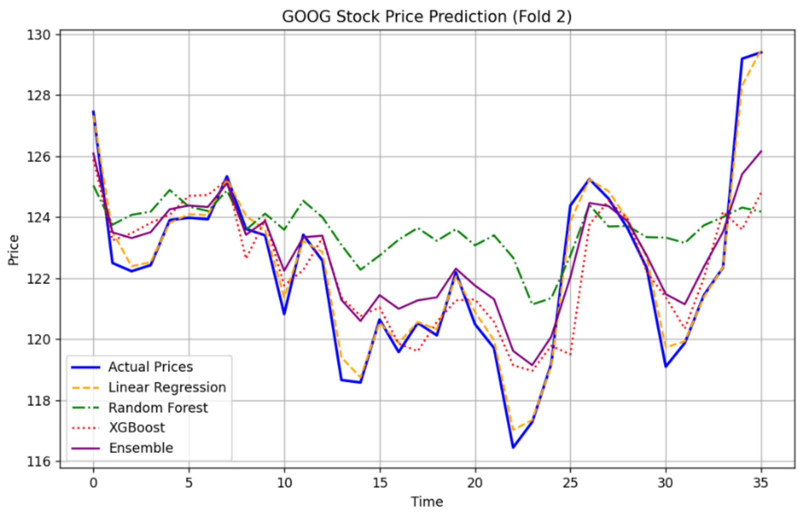
SCREENSHOT OF PLOTS:

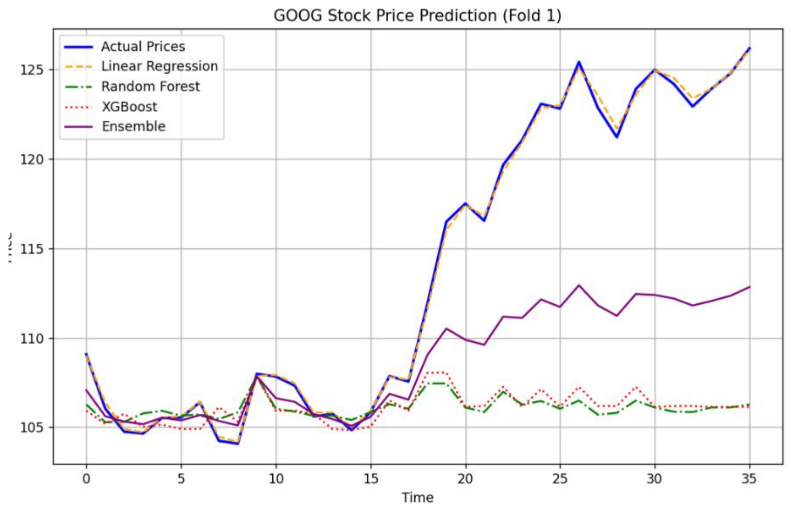












7 Challenges Faced

While working on this project, I encountered several challenges that tested my problem-solving skills and deepened my understanding of machine learning and deep learning. Below, I outline some of the key challenges I faced, along with the solutions I implemented to overcome them.

Data Fetching Errors: Initially, I faced issues while fetching data using the yfinance library. The API occasionally returned incomplete or empty datasets, especially for shorter time periods or less popular tickers. This caused the program to crash during the data preprocessing stage.

Missing Data Handling: The stock data contained missing values, especially in columns like Return and moving averages (MA_7, MA_30), which were calculated using historical data. These missing values caused errors during model training.

Overfitting in Random Forest: During initial testing, the Random Forest model showed signs of overfitting, especially in early splits where the training data was limited. The model performed exceptionally well on the training set but poorly on the test set.

LSTM Training Instability: Training the LSTM model was challenging due to its sensitivity to hyperparameters and initial weights. In some cases, the model failed to converge, resulting in poor predictions.

Visualizing Complexity: Visualizing the results for all five splits in a single plot was complex, as the test sets for each split overlapped in time. This made it difficult to interpret the performance of each model across different splits.

Debugging Evaluation Metrics: While calculating evaluation metrics like RMSE and MAPE, I encountered errors due to mismatched shapes between the actual and predicted values. This was particularly problematic for the LSTM model, where the output shape needed to be reshaped before comparison.

References

https://pypi.org/project/yfinance/

https://pandas.pydata.org/docs/

https://numpy.org/doc/

https://matplotlib.org/stable/contents.html

https://scikit-learn.org/stable/

https://www.tensorflow.org/api_docs

https://keras.io/api/

https://xgboost.readthedocs.io/en/stable/

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_percentage_error.html

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

https://scikit-learn.org/stable/modules/ensemble.html

https://www.sciencedirect.com/science/article/pii/S1877050920312345

https://ieeexplore.ieee.org/document/9073321

https://machinelearningmastery.com/time-series-forecasting/

https://machinelearningmastery.com/deep-learning-for-time-series-forecasting/

https://www.amazon.com/Stress-Testing-Banking-Comprehensive-Guide/dp/1119138905