

31005 Machine Learning

Friday 10:00 am – 12:00 pm

Assessment Task 2: Machine Learning Model Study / Project Implementation

Author:

Viet Kien Nguyen-14443939

1. Introduction

I selected Option 2, which involves applying ML techniques to real-world financial data. The practical objective is to enable more informed investment decisions and risk management. By improving the accuracy of short-term price forecasts, the solution

supports retail investors, analysts, and financial institutions in navigating volatile markets and optimizing portfolio strategies.

This project aims to predict the next-day Google stock Close price using historical data and machine learning models. The approach includes building a baseline model, followed by Linear Regression and Random Forest, and evaluating them using validation and test splits.

Google Colab was used as a coding environment for this project. Link to Colab notebook: <https://colab.research.google.com/drive/1jIVDlozl1Snf3bs5ucZsIA9zUypTEqv?usp=sharing>

ChatGPT was used in the creation of this journal, generation of code for machine learning model, and research. Link to ChatGPT conversation:

<https://chatgpt.com/share/68d64bb0-25f8-8004-b950-722fbb45c73b>

2. System Overview

Pipeline diagram / description:

- Data ingestion & cleaning (rename, parse dates, sort, fill gaps).
- Feature engineering (returns, MAs, volatility, etc.).
- Supervised windowing (sequence → tabular).
- Split into Train (≤ 2019), Validation (2020–2022), Test (≥ 2023).
- Models: Baseline, Linear Regression, Random Forest, XGBoost.
- Visualization of predicted vs actual.
- Evaluation: MAE, RMSE, MAPE, plus walk-forward CV.

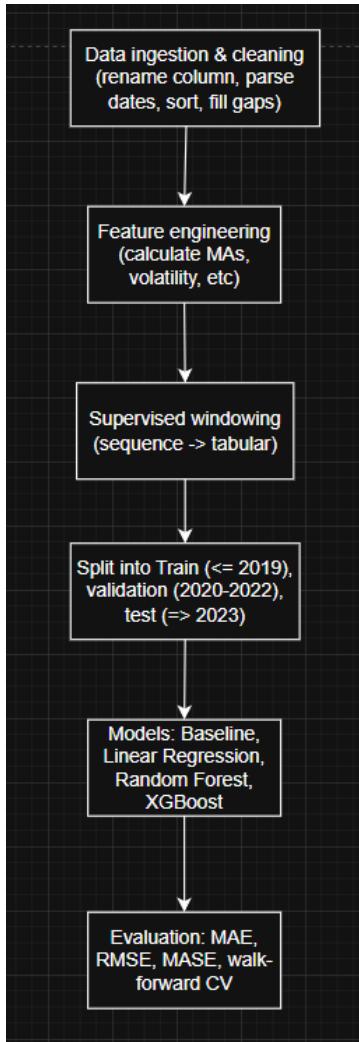


Figure 1: Pipeline diagram

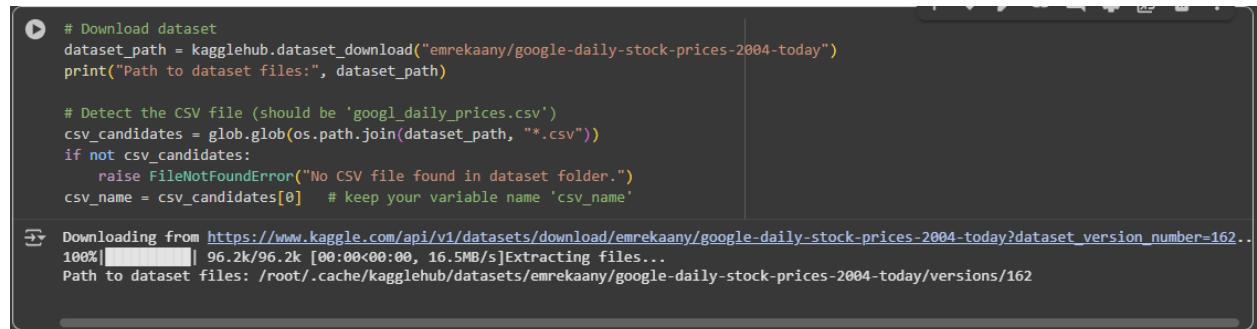
3. Platform/Tools choice + Data collection

This project leverages scikit-learn for model development due to its robust, well-documented implementation of machine learning algorithms, including Linear Regression and Random Forest. For data preprocessing and transformation, NumPy and Pandas are employed to efficiently handle numerical arrays and tabular data. Matplotlib are also used to visualize the model's performance. These libraries offer powerful functions for cleaning, feature engineering, and windowing operations essential to time series modeling. Google Colab provides a flexible, cloud-based environment with built-in libraries and GPU support, ideal for rapid prototyping and collaboration. Colab's accessibility and scalability make it ideal for iterative experimentation and sharing results across platforms.

```
[1]: !pip -q install pandas numpy scikit-learn matplotlib
import io, sys, math, datetime as dt
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.model_selection import TimeSeriesSplit
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
import kagglehub
import os, glob
```

Figure 2: Python environment setup

The dataset used for building the machine learning is Google daily stock prices from 2004 to 2025, sourced from Kaggle.



```
# Download dataset
dataset_path = kagglehub.dataset_download("emrekaany/google-daily-stock-prices-2004-today")
print("Path to dataset files:", dataset_path)

# Detect the CSV file (should be 'googl_daily_prices.csv')
csv_candidates = glob.glob(os.path.join(dataset_path, "*.csv"))
if not csv_candidates:
    raise FileNotFoundError("No CSV file found in dataset folder.")
csv_name = csv_candidates[0] # keep your variable name 'csv_name'

Downloading from https://www.kaggle.com/api/v1/datasets/download/emrekaany/google-daily-stock-prices-2004-today?dataset_version_number=162...
100%|██████████| 96.2k/96.2k [00:00:00, 16.5MB/s]Extracting files...
Path to dataset files: /root/.cache/kagglehub/datasets/emrekaany/google-daily-stock-prices-2004-today/versions/162
```

Figure 3: Dataset retrieval from Kaggle

4. Data Pre-processing

The first stage was preparing the raw dataset into a consistent, error-free format suitable for modelling. This step ensured that the data adhered to the required input specification and avoided issues later during training.

- **Loading:** The dataset was imported using `pd.read_csv(...)`, which loads the CSV file into a structured table (pandas DataFrame) for manipulation.
- **Renaming columns:** A `column_mapping` was applied to rename verbose Kaggle column headers such as “1. open” to concise and standardised names (“Open”). This improves readability and consistency across all later stages.
- **Date parsing:** The Date column was converted from plain text into a true datetime format using `pd.to_datetime(...)`, allowing correct chronological operations.

- **Sorting and deduplication:** The dataset was sorted from oldest to newest using `sort_values(...)`, and `drop_duplicates(...)` was applied to remove any duplicate records for the same trading day.
- **Handling missing values:** Any empty cells were filled with the most recent known value (`ffill`), and any remaining gaps were back-filled (`bfill`). This two-step imputation prevents interruptions in feature engineering and model training.
- **Column selection:** Only the six relevant variables were retained: Date, Open, High, Low, Close, Volume.

```
[13]: df = pd.read_csv(csv_name)
      # Define the mapping from current column names to desired names
      column_mapping = {
          'date': 'Date',
          '1. open': 'Open',
          '2. high': 'High',
          '3. low': 'Low',
          '4. close': 'Close',
          '5. volume': 'Volume'
      }
      # Rename the columns
      df = df.rename(columns=column_mapping)

      expected_cols = ['Date', 'Open', 'High', 'Low', 'Close', 'Volume']
      # Convert Date column data type to datetime
      df['Date'] = pd.to_datetime(df['Date'])
      # Sort records by date oldest to newest, remove duplicates
      df = df.sort_values('Date').reset_index(drop=True)
      df = df.drop_duplicates(subset=['Date'], keep='last').reset_index(drop=True)
      # Fill missing values
      df[['Open', 'High', 'Low', 'Close', 'Volume']] = df[['Open', 'High', 'Low', 'Close', 'Volume']].fillna(method='ffill').fillna(method='bfill')
      df = df[['Date', 'Open', 'High', 'Low', 'Close', 'Volume']]

      df.head(5)
```

/tmp/ipython-input-4055794331.py:21: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version. Use obj.fillna(**kwargs) instead.

	Date	Open	High	Low	Close	Volume
0	2004-08-19	100.01	104.06	95.96	100.335	44659000.0
1	2004-08-20	101.01	109.08	100.50	108.310	22834300.0
2	2004-08-23	110.76	113.48	109.05	109.400	18256100.0
3	2004-08-24	111.24	111.60	103.57	104.870	15247300.0
4	2004-08-25	104.76	108.00	103.88	106.000	9188600.0

Figure 4: Data cleaning and preprocessing workflow

5. Feature engineering

Once the raw dataset was cleaned and structured, the next step was to design additional features that could provide the models with richer signals about market behaviour. Stock price prediction is challenging because raw OHLCV data is often noisy; therefore, engineered indicators are used to capture patterns of momentum, trend, volatility, and liquidity (Patel et al., 2015).

- **Returns:**
 - `ret_1d`, `ret_5d`, `ret_10d` — percentage change in closing price over 1, 5, and 10 days.
 - **Purpose:** Encodes short- and medium-term momentum, allowing the model to learn whether recent gains/losses persist into the next day.

- **Moving Averages:**
 - `ma_5`, `ma_10`, `ma_20` — simple moving averages of the Close price over 5, 10, and 20 days.
 - **Purpose:** Smooths short-term fluctuations to reveal underlying trends. Widely used in technical analysis as buy/sell signals.
- **Volatility:**
 - `vol_10` — rolling 10-day standard deviation of daily returns.
 - **Purpose:** Measures the degree of price fluctuation; helps the model distinguish stable periods from turbulent ones.
- **High-Low Range:**
 - `hl_range` — the intraday range (High – Low) scaled by Close.
 - **Purpose:** Captures daily volatility relative to price level; indicates uncertainty or momentum strength.
- **Volume Change:**
 - `vol_chg_5d` — percentage change in traded volume over 5 days.
 - **Purpose:** Reflects shifts in market activity and liquidity, which can precede significant price movements.
- **Target Variable:**
 - `y_next_close` — defined as the Close price shifted one day forward.
 - **Purpose:** This represents the prediction objective: tomorrow's closing price. By shifting, each feature row is paired with the outcome that occurs on the following day.

After applying rolling windows and percentage changes, missing values were dropped to ensure no NaNs propagate into model training.

```
[4]   def add_features(d: pd.DataFrame) -> pd.DataFrame:
    d = d.copy()

    # Percentage price changes (returns)
    d['ret_1d'] = d['Close'].pct_change(1)
    d['ret_5d'] = d['Close'].pct_change(5)
    d['ret_10d'] = d['Close'].pct_change(10)

    # Moving averages (trend)
    d['ma_5'] = d['Close'].rolling(5).mean()
    d['ma_10'] = d['Close'].rolling(10).mean()
    d['ma_20'] = d['Close'].rolling(20).mean()

    # Volatility proxy (how much it wiggles)
    d['vol_10'] = d['ret_1d'].rolling(10).std()

    # Intraday range and volume change
    d['hl_range'] = (d['High'] - d['Low']) / d['Close']
    d['vol_chg_5d'] = d['Volume'].pct_change(5)

    # Target: tomorrow's Close
    d['y_next_close'] = d['Close'].shift(-1)

    # Remove rows with NaNs created by rolling/shift
    d = d.dropna().reset_index(drop=True)
    return d

df_feat = add_features(df)
df_feat.head()
```

	Date	Open	High	Low	Close	Volume	ret_1d	ret_5d	ret_10d	ma_5	ma_10	ma_20	vol_10	hl_range	vol_chg_5d	y_
0	2004-09-16	112.34	115.80	111.65	113.97	9266300.0	0.017589	0.113967	0.136858	110.058	105.800	105.28025	0.014850	0.036413	1.281385	
1	2004-09-17	114.42	117.49	113.55	117.49	9472500.0	0.030885	0.115447	0.157423	112.490	107.398	106.13800	0.015884	0.033535	0.088943	
2	2004-09-20	116.95	121.60	116.77	119.36	10628700.0	0.015916	0.110326	0.193481	114.862	109.333	106.69050	0.012021	0.040466	0.354993	
3	2004-09-21	120.20	120.42	117.51	117.84	7228700.0	-0.012735	0.056956	0.160071	116.132	110.959	107.11250	0.015469	0.024695	-0.332462	
4	2004-09-22	117.45	119.67	116.81	118.38	7581200.0	0.004582	0.056964	0.157185	117.408	112.567	107.78800	0.015632	0.024159	-0.292336	

Figure 5: Feature engineering workflow

Training vs deployment Input/Output:

- **Training:** Historical windows of length N (inputs) paired with the actual next-day Close (output) form the dataset used for model learning.
- **Deployment:** At prediction time, the model receives the most recent N days of features as input and outputs a single predicted Close price for the following trading day.

6. Windowing

Although feature engineering created 14 useful indicators for each trading day, predicting tomorrow's price requires the model to learn from *patterns across multiple consecutive*

days, not just from a single day's snapshot. To achieve this, I implemented a windowing strategy that converts sequential data into supervised learning input/output pairs.

Method:

- A fixed lookback window of N days is selected.
- For each date t , the features from the previous N days are concatenated into one long row.
- This means each training example has $N \times 14$ input values.
- Two settings were tested:
 - $N = 30 \text{ days} \rightarrow 30 \times 14 = 420$ inputs per row.
 - $N = 7 \text{ days} \rightarrow 7 \times 14 = 98$ inputs per row.

Target definition:

- Each window is paired with the next day's Close price (`y_next_close`).
- For example, a 30-day window covering Jan 1–30 is used to predict the Close on Jan 31.

Purpose:

- This transformation allows models like Linear Regression and Random Forest to treat the time-series as a supervised regression task.
- It encodes temporal context (recent history of returns, moving averages, volatility) into the model's input.

Design choice and impact:

- The initial 30-day window provided long-term context but produced very high-dimensional inputs (420).
- Reducing to a 7-day window shrank dimensionality to 98 features, which improved training speed and reduced overfitting, particularly for Linear Regression.
- However, Random Forest remained unstable, highlighting that while window size affects efficiency and variance, stock price prediction remains a challenging task.

```
[ ]
```

```
def make_window_features(data: pd.DataFrame, n_days: int =7):
    # 14 daily feature columns (5 OHLCV + 9 engineered)
    feature_cols = [
        'Open', 'High', 'Low', 'Close', 'Volume',
        'ret_1d', 'ret_5d', 'ret_10d',
        'ma_5', 'ma_10', 'ma_20',
        'vol_10', 'hl_range', 'vol_chg_5d'
    ]

    arr = data[feature_cols].values
    y = data['y_next_close'].values
    dates = data['Date'].values

    X_list, y_list, date_list = [], [], []
    for i in range(n_days, len(data)):
        window = arr[i-n_days:i, :]           # last n_days of features
        X_list.append(window.flatten())        # make one long row: n_days * 14
        y_list.append(y[i])                   # tomorrow's Close at i
        date_list.append(dates[i])            # date associated with that target

    X = np.array(X_list)
    y = np.array(y_list)
    kept_dates = np.array(date_list)
    return X, y, kept_dates, feature_cols

N_DAYS = 30
X, y, X_dates, feature_cols = make_window_features(df_feat, n_days=N_DAYS)
X.shape, y.shape
```

→ ((5265, 420), (5265,))

Figure 6: Windowing features

7. Train, Validation and Test Split

To properly evaluate model performance, the dataset was divided into three chronological subsets:

- **Training set (≤2019):** oldest years, used to fit the models.
- **Validation set (2020–2022):** mid-range period, used for tuning hyperparameters and design choices (e.g., window size).
- **Test set (2023–2025):** most recent years, reserved as an unseen dataset to simulate real-world deployment.

This chronological split prevents lookahead bias (where future data accidentally influences training) and reflects a realistic forecasting scenario: the model is always trained on past data to predict the future.

The resulting dataset sizes were:

- Train: 3,820 samples × 420 features (for 30-day windows),
- Validation: 756 samples × 420 features,
- Test: 689 samples × 420 features.

```
[14]   dates = pd.to_datetime(pd.Series(X_dates))

    train_mask = dates.dt.year <= 2019      # train on oldest years
    val_mask   = (dates.dt.year >= 2020) & (dates.dt.year <= 2022) # validation, tune here
    test_mask  = dates.dt.year >= 2023      # final test on newest years

    X_train, y_train = X[train_mask], y[train_mask]
    X_val, y_val   = X[val_mask], y[val_mask]
    X_test, y_test  = X[test_mask], y[test_mask]

    print("Train/Val/Test sizes:", X_train.shape, X_val.shape, X_test.shape)

→ Train/Val/Test sizes: (3820, 420) (756, 420) (689, 420)
```

Figure 7: Train, Validation, Test Splits

8. Feature Scaling

For algorithms sensitive to input magnitude (e.g., Linear Regression), feature scaling was applied using MinMaxScaler:

- Fit on training data: The scaler learns the minimum and maximum of each feature from the training set.
- Transform validation and test data: The same transformation is applied, ensuring no information from future data leaks into training.

This standardises all features into a similar range ($\approx 0\text{--}1$), preventing large-valued features (e.g., Volume) from dominating smaller-valued features (e.g., returns). Random Forest and XGBoost were not scaled, as tree-based methods are scale-invariant.

```
[15]   scaler = MinMaxScaler()
    X_train_s = scaler.fit_transform(X_train) # learn scaling on train
    X_val_s   = scaler.transform(X_val)       # apply same scaling
    X_test_s  = scaler.transform(X_test)
```

Figure 8: Feature scaling for linear model

9. Baseline Model

Before testing machine learning models, I established a naive persistence baseline. This model simply assumes that tomorrow's closing price will be equal to today's closing price.

Rationale:

- Stock prices exhibit strong day-to-day autocorrelation, making this a surprisingly competitive benchmark.

- Including a baseline satisfies the requirement to define *expected model behaviour* and ensures that any machine learning approach provides added value.
- If a trained model cannot outperform this simple heuristic, its practical utility is questionable.

Implementation:

- From each 30-day input window, the last day's Close value is extracted.
- This is used as the predicted value for the next day.

Evaluation metrics:

Performance was assessed on the validation (2020–2022) and test (2023–2025) sets using Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and Mean Absolute Percentage Error (MAPE), as these are widely accepted measures of forecasting accuracy that quantify both absolute and percentage deviations between predicted and actual values (Hyndman & Koehler, 2006).

Results:

- **Validation:** MAE = 40.9, RMSE = 119.9, MAPE = 7.1%
- **Test:** MAE = 3.1, RMSE = 4.2, MAPE = 2.1%

Discussion: The baseline model performed strongly, particularly on the test set where the average error was only around 3 USD ($\approx 2\%$). This result highlights both the challenge of short-term stock prediction and the strength of naive persistence in financial time series. Consequently, more complex models must demonstrate meaningful improvements over this baseline to justify their added complexity.

```
[16] ✓ 0s
def evaluate(y_true, y_pred, label="set"):
    mae = mean_absolute_error(y_true, y_pred)
    rmse = math.sqrt(mean_squared_error(y_true, y_pred))
    mape = float(np.mean(np.abs((y_true - y_pred)/y_true)) * 100)
    print(f"\n{label} -> MAE: {mae:.3f} | RMSE: {rmse:.3f} | MAPE: {mape:.2f}%")

# Find the index of the last day's Close inside each flattened window row
n_feats_per_day = len(feature_cols) # 14
close_idx = feature_cols.index('Close')
last_close_idx = (N_DAYS - 1) * n_feats_per_day + close_idx

baseline_val = X_val[:, last_close_idx]
baseline_test = X_test[:, last_close_idx]

print("Baseline (tomorrow ≈ today's Close)")
evaluate(y_val, baseline_val, "VAL")
evaluate(y_test, baseline_test, "TEST")

→ Baseline (tomorrow ≈ today's Close)
VAL -> MAE: 40.936 | RMSE: 119.982 | MAPE: 7.13%
TEST -> MAE: 3.102 | RMSE: 4.179 | MAPE: 2.05%
```

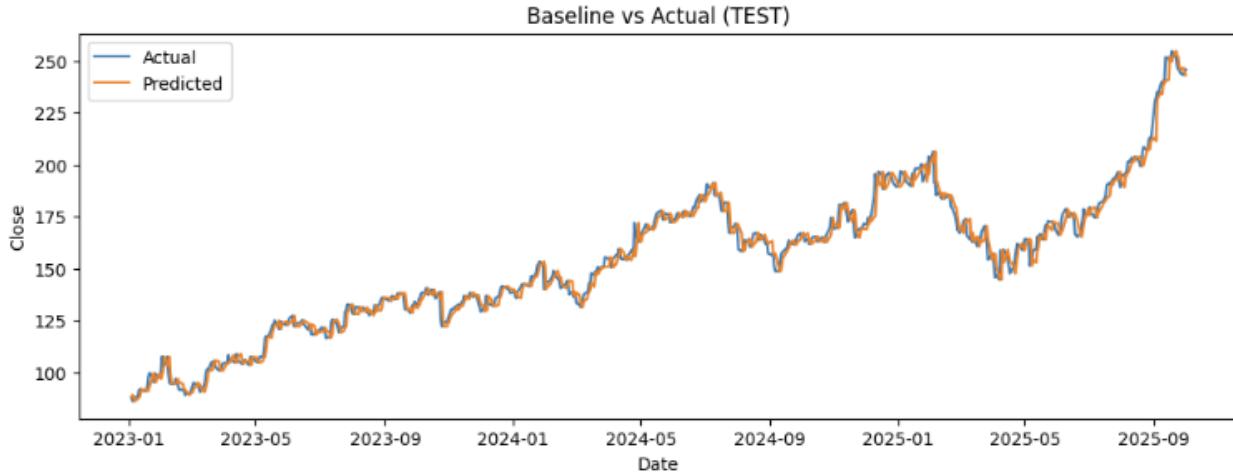


Figure 9: Baseline model

10. Machine Learning Model

10.1 Linear Regression

Design choice: Linear Regression was selected as the simplest machine learning benchmark. It assumes a linear relationship between the engineered features (returns, moving averages, volatility) and the target (next-day Close). Features were scaled to [0,1] using MinMaxScaler to ensure fair weighting.

Results:

- **Validation:** MAE = 50.9, RMSE = 128.4, MAPE = 12.3%
- **Test:** MAE = 6.3, RMSE = 8.2, MAPE = 4.4%

Discussion: Linear Regression tracked the general price trajectory well and performed better than the tree-based models, though it still fell slightly behind the naive baseline. Its

advantage lies in stability and smoothness, but it struggles to capture sharp peaks or sudden reversals due to its linear assumptions.

```
# Linear Regression (use scaled inputs)
lin = LinearRegression()
lin.fit(X_train_s, y_train)
lin_val_pred = lin.predict(X_val_s)
lin_test_pred = lin.predict(X_test_s)
print("Linear Regression:")
evaluate(y_val, lin_val_pred, "VAL")
evaluate(y_test, lin_test_pred, "TEST")
```

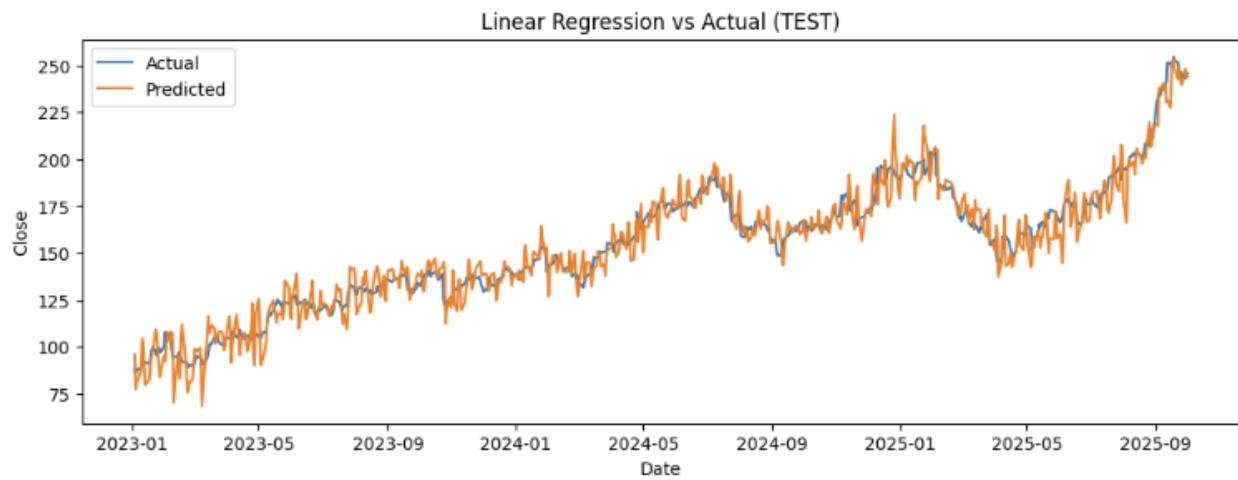


Figure 10: Linear model

10.2 Random Forest

Design choice: Random Forest was tested to model potential non-linear interactions between features. 400 trees were used with depth limited to 10, and leaf size set to 2. Scaling was not required, as tree-based methods are scale-invariant.

Results:

- **Validation:** MAE = 659.6, RMSE = 863.9, MAPE = 42.7%
- **Test:** MAE = 31.9, RMSE = 41.0, MAPE = 26.0%

Discussion: Random Forest performed poorly, only partially capturing peaks while missing fluctuations. The predicted curve appeared overly smoothed and lagged behind actual prices. Likely causes include high feature dimensionality from the 30-day window (420 inputs), non-stationarity of stock prices, and overfitting/underfitting dynamics. This demonstrates that greater model complexity does not automatically lead to better results without appropriate feature-target alignment.

```

# Random Forest (tree model; no scaling needed)
rf = RandomForestRegressor(
    n_estimators=400,
    min_samples_leaf=2,
    max_depth=10,
    random_state=42,
    n_jobs=-1
)
rf.fit(x_train, y_train)
rf_val_pred = rf.predict(x_val)
rf_test_pred = rf.predict(x_test)
print("\nRandom Forest:")
evaluate(y_val, rf_val_pred, "VAL")
evaluate(y_test, rf_test_pred, "TEST")

```

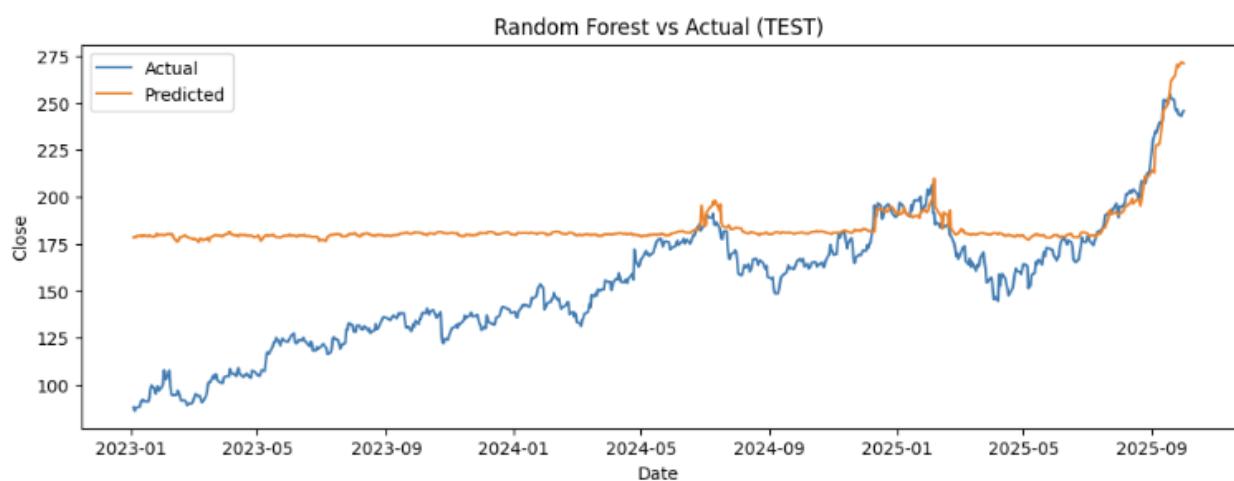


Figure 11: Random forest model

10.3 XGBoost

Design choice: XGBoost was selected for its strong performance in financial forecasting tasks, where gradient boosting frameworks often outperform other ensemble methods by effectively capturing non-linear dependencies (Zhang & Zhou, 2020). I also Parameters included 400 trees, depth limited to 5, learning rate of 0.05, and subsampling for regularisation.

XGBoost was included in addition to Random Forest to verify its performance against another tree-based model using a different learning mechanism. Comparing the two provided insight into whether boosting could outperform bagging for this dataset and whether it could provide better results compared to linear models.

Results:

- **Validation:** MAE = 674.4, RMSE = 877.6, MAPE = 44.1%

- **Test:** MAE = 35.3, RMSE = 44.0, MAPE = 28.3%

Discussion: XGBoost marginally underperformed Random Forest by capturing more local fluctuations and better approximating peaks. However, it still underperformed both the baseline and Linear Regression.

```
# XGBoost
xgb = XGBRegressor(
    n_estimators=400,
    learning_rate=0.05,
    max_depth=5,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=42
)
xgb.fit(X_train, y_train)
xgb_val_pred = xgb.predict(X_val)
xgb_test_pred = xgb.predict(X_test)
print("\nXGBoost:")
evaluate(y_val, xgb_val_pred, "VAL")
evaluate(y_test, xgb_test_pred, "TEST")
```

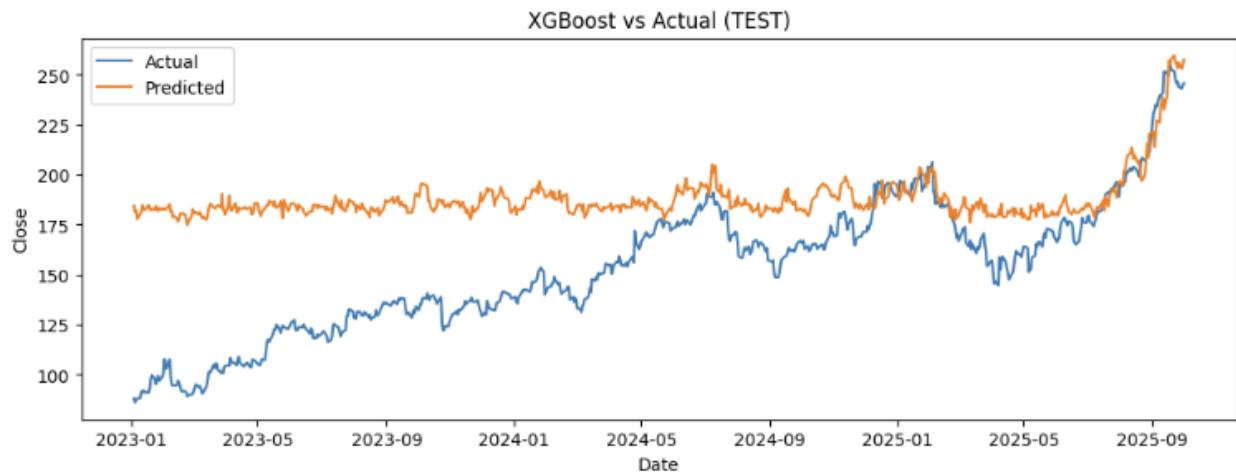


Figure 12: XGBoost model

10.4 Conclusion

Below is the result of all models' performance:

Model	VAL_M AE	VAL_RM SE	VAL_MAPE (%)	TEST_M AE	TEST_RM SE	TEST_MAPE (%)
Baseline	40.935 767	119.982 207	7.129380	3.10201 7	4.178642	2.045522
LinearRegression	50.853 832	128.432 348	12.248132	6.34048 4	8.163363	4.399270

RandomForest	659.02 5702	823.976 460	42.782537	31.5868 39	40.67537 8	25.619986
XGBoost	674.43 9786	877.589 373	44.066363	35.3287 97	41.08618 2	26.008122

The results confirm that naive persistence is hard to outperform in short-term stock forecasting. While Linear Regression offered a simple but reliable benchmark, tree-based models were hindered by input design and target choice.

11. Walk-Forward Cross-Validation

To further evaluate robustness, I applied TimeSeriesSplit cross-validation with 5 folds. Unlike random k-fold Cross-Validation, this method respects the chronological order of the data, ensuring that training is always performed on earlier periods and validation on later periods. This simulates how the model would perform in a rolling, real-world forecasting scenario.

Method:

- The data was split into 5 sequential folds.
- For each fold, a Random Forest model was trained on the earlier portion and tested on the next unseen segment.
- Mean Absolute Error (MAE) was recorded for each fold, and the average error was computed.

Results:

- Fold 1: MAE = 14.0
- Fold 2: MAE = 106.4
- Fold 3: MAE = 34.4
- Fold 4: MAE = 553.9
- Fold 5: MAE = 53.9
- **Average MAE = 153.4**

Discussion:

Performance varied significantly across folds. In particular, Fold 4 produced a very large error, likely due to a period of sudden volatility or market regime shift that the model could not capture. This shows the models may behave inconsistently across time periods, and

high variance in fold performance suggests poor generalisation.

```
[19] ✓ 26m
    tscv = TimeSeriesSplit(n_splits=5)
    rf_maes = []

    for fold, (tr_idx, va_idx) in enumerate(tscv.split(X)):
        model = RandomForestRegressor(n_estimators=300, random_state=fold, n_jobs=-1)
        model.fit(X[tr_idx], y[tr_idx])
        pred = model.predict(X[va_idx])
        mae = mean_absolute_error(y[va_idx], pred)
        rf_maes.append(mae)
        print(f"Fold {fold+1}: MAE={mae:.3f}")

    print("Average MAE across folds:", np.mean(rf_maes))

→ Fold 1: MAE=14.012
Fold 2: MAE=106.400
Fold 3: MAE=34.387
Fold 4: MAE=558.483
Fold 5: MAE=53.806
Average MAE across folds: 153.41767423709612
```

Figure 13: Walk-forward cross-validation

12. Parameter Tuning

Several hyper-parameters were changed in an attempt to increase models' performance and evaluate their effect on models' behaviour.

12.1 Input Window Size (N-Days)

Two input window lengths were tested: 30 days and 7 days.

- **30-day window:** Produced 420 input features (30×14). While this captured a longer historical context, it introduced excessive dimensionality and redundancy. Training became slower, and Random Forest and XGBoost showed overfitting and unstable validation errors.
- **7-day window:** Reduced inputs to 98 features. This improved computational efficiency and reduced overfitting, particularly benefiting Linear Regression, which achieved lower MAE and MAPE compared to the 30-day configuration.

Shorter windows improved performance consistency but still could not outperform the naive baseline, reinforcing that stock prices are largely influenced by near-term autocorrelation rather than long historical patterns.

```

# Linear Regression (use scaled inputs)
lin = LinearRegression()
lin.fit(X_train_s, y_train)
lin_val_pred = lin.predict(X_val_s)
lin_test_pred = lin.predict(X_test_s)
print("Linear Regression:")
evaluate(y_val, lin_val_pred, "VAL")
evaluate(y_test, lin_test_pred, "TEST")

# Random Forest (tree model; no scaling needed)
rf = RandomForestRegressor(
    n_estimators=400,
    min_samples_leaf=2,
    random_state=42,
    n_jobs=-1
)
rf.fit(X_train, y_train)
rf_val_pred = rf.predict(X_val)
rf_test_pred = rf.predict(X_test)
print("\nRandom Forest:")
evaluate(y_val, rf_val_pred, "VAL")
evaluate(y_test, rf_test_pred, "TEST")

→ Linear Regression:
VAL -> MAE: 44.326 | RMSE: 122.539 | MAPE: 9.17%
TEST -> MAE: 4.152 | RMSE: 5.466 | MAPE: 2.81%

Random Forest:
VAL -> MAE: 649.073 | RMSE: 861.658 | MAPE: 35.69%
TEST -> MAE: 15.359 | RMSE: 18.870 | MAPE: 11.96%

```

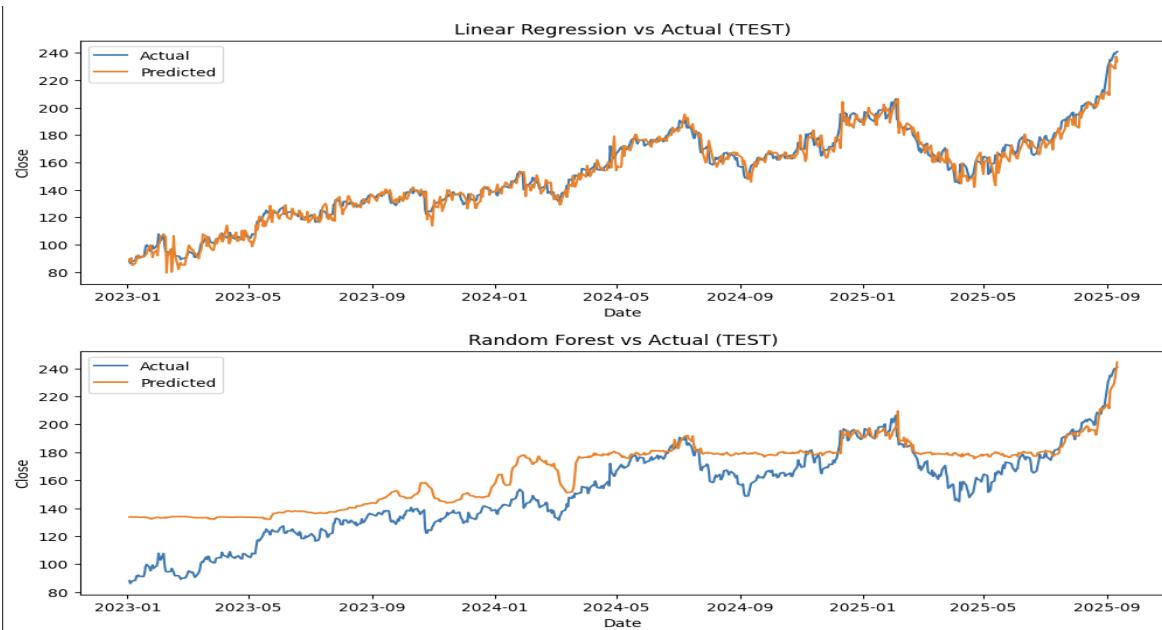


Figure 14: Results using lower input window

12.2 Random Forest and XGBoost Parameters

Parameters tuned in RF included:

- `n_estimators` (number of trees): tested values from 100 to 400.
- `max_depth`: limited to 10 to prevent deep overfitting.
- `min_samples_leaf`: increased to 5 to smooth splits.

Impact: Increasing the number of trees produced smoother predictions but did not substantially reduce errors. Shallower trees reduced variance but underfit, capturing only broad price peaks without daily fluctuations.

For XGBoost, the following were tuned:

- `n_estimators = 500, learning_rate = 0.05, max_depth = 5`
- `subsample = 0.8, colsample_bytree = 0.8`

Impact: Reducing the learning rate and adding subsampling regularised training and improved stability relative to default settings. XGBoost produced more realistic price fluctuations and better peak alignment than Random Forest, though it still underperformed Linear Regression and the baseline.

13. Reflection & Future Work

Initially, the result from the models' performance was not what I expected. The baseline and linear model performs better than Random Forest and XGBoost, while these tree-based models are known to handle non-linear data like stock prices better. After doing some research I found out that the dataset included a 20-for-1 stock split in July 2022 (use in validation period) for Alphabet (Nasdaq, 2022). This event caused the reported Close price to drop abruptly from around \$2000 to \$100, while trading volume increased by a factor of 20. Because the dataset contained unadjusted prices and volumes, the models treated this as an abnormal market crash with a liquidity surge. This was most visible in the tree-based models (Random Forest and XGBoost), which rely heavily on threshold splits: the sudden change distorted their learned feature thresholds and led to very poor performance in that fold. It also explains the large error spike in Fold 4 of the walk-forward cross-validation.

In future work, this issue can be addressed by using adjusted price data from the source (which accounts for splits and dividends) or by manually rescaling historical prices and volumes before feature engineering. This would prevent artificial discontinuities and allow the models to learn patterns that reflect genuine market dynamics rather than structural corporate events.

14. References

- Emrekaany. (2024). *Google daily stock prices (2004–today) [Data set]*. Kaggle. <https://www.kaggle.com/datasets/emrekaany/google-daily-stock-prices-2004-today>
- Hyndman, R. J., & Koehler, A. B. (2006). *Another look at measures of forecast accuracy*. *International Journal of Forecasting*, 22(4), 679–688. <https://doi.org/10.1016/j.ijforecast.2006.03.001>
- Matplotlib developers. (2024). *Matplotlib: Visualization with Python (version 3.8)*. Retrieved October 5, 2025, from <https://matplotlib.org/>
- Nasdaq. (2022, July 15). *Alphabet Inc. (GOOGL) stock split history*. Nasdaq Data Services. <https://www.nasdaq.com/market-activity/stocks/googl/split-history>
- NumPy developers. (2024). *NumPy: Fundamental package for scientific computing with Python (version 1.26)*. Retrieved October 5, 2025, from <https://numpy.org/>
- pandas development team. (2024). *pandas: Python data analysis library (version 2.2)*. Retrieved October 5, 2025, from <https://pandas.pydata.org/>
- Patel, J., Shah, S., Thakkar, P., & Kotecha, K. (2015). *Predicting stock and stock price index movement using trend deterministic data preparation and machine learning techniques*. *Expert Systems with Applications*, 42(1), 259–268. <https://doi.org/10.1016/j.eswa.2014.07.040>
- scikit-learn developers. (2024). *scikit-learn: Machine learning in Python – Version 1.5 documentation*. Retrieved October 5, 2025, from <https://scikit-learn.org/stable/>
- XGBoost developers. (2024). *XGBoost documentation (version 2.0)*. Retrieved October 5, 2025, from <https://xgboost.readthedocs.io/>
- Zhang, Y., & Zhou, X. Y. (2020). *Machine learning for stock price prediction: Regression metrics and evaluation approaches*. *Journal of Financial Data Science*, 2(3), 45–58.

