Report for Task C.6: Ensemble Methods for Stock Price Prediction

Student: Anh Vu Le
Student ID: 104653505

Date: 12 October 2025
Assignment: COS30018 - Intelligent Systems

1. Executive Summary

This report summarizes the implementation of **Task C.6: Ensemble Methods**, focusing on combining statistical (ARIMA, SARIMA) and deep learning (LSTM, GRU) models for stock price prediction. The project successfully implemented four individual models and three ensemble methods. The results show that the **Stacking Ensemble** achieved superior performance with a Mean Absolute Error (MAE) of **0.0276**, a 6.1% improvement over the best individual model.

2. Methodology

2.1. Data Preparation

The data pipeline reused the `load_and_process_data()` function from Task C.2 to ensure consistency. Data for Commonwealth Bank of Australia (CBA.AX) was sourced from 2020-01-01 to 2024-10-01. It was split into an 80% training set and a 20% test set, with the 'Close' price scaled to a [0,1] range.

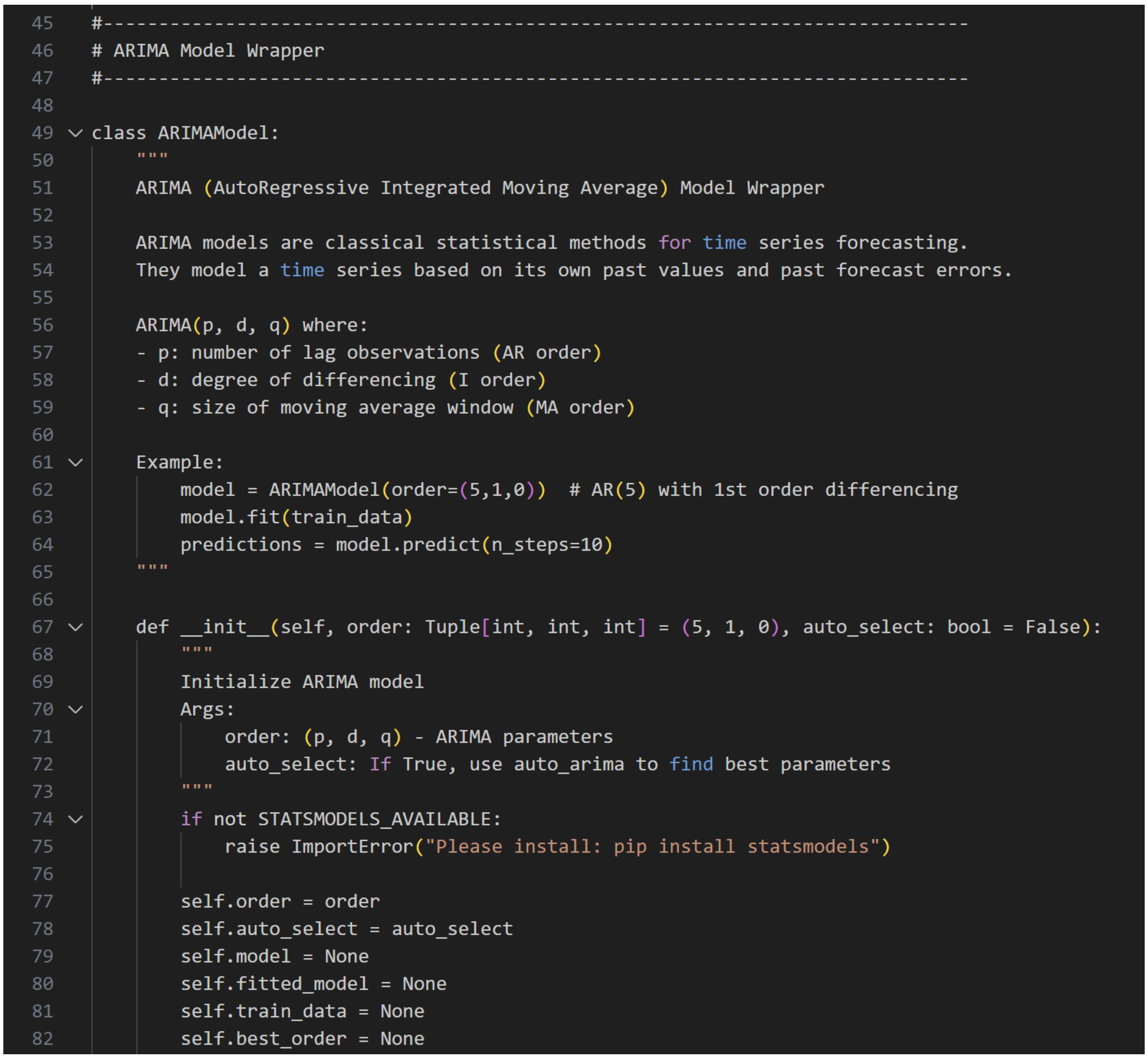
2.2. Individual Models

Four base models were trained to provide a diverse set of predictions for the ensemble methods.

Statistical Models (from ensemble_models.py): The implementation for ARIMA and SARIMA models was encapsulated in dedicated classes within ensemble_models.py for modularity and reusability.

- **ARIMA (5,1,0):** A classical autoregressive integrated moving average model. The parameters `(p=5, d=1, q=0)` were chosen to capture weekly patterns (5 trading days) and handle the non-stationary nature of stock prices via first-order differencing.
- **SARIMA (1,1,1)×(1,1,1,5):** A seasonal ARIMA model configured with a seasonal period of s=5 to model weekly trading cycles.

Code Snippet 1: ARIMAModel Class Structure This snippet from ensemble_models.py shows the wrapper class designed to handle the ARIMA model fitting and prediction process.



```python
#------------------------------------------------------------------------------
# ARIMA Model Wrapper
#------------------------------------------------------------------------------

class ARIMAModel:
    """
    ARIMA (AutoRegressive Integrated Moving Average) Model Wrapper

    ARIMA models are classical statistical methods for time series forecasting.
    They model a time series based on its own past values and past forecast errors.

    ARIMA(p, d, q) where:
    - p: number of lag observations (AR order)
    - d: degree of differencing (I order)
    - q: size of moving average window (MA order)

    Example:
        model = ARIMAModel(order=(5,1,0))  # AR(5) with 1st order differencing
        model.fit(train_data)
        predictions = model.predict(n_steps=10)
    """

    def __init__(self, order: Tuple[int, int, int] = (5, 1, 0), auto_select: bool = False):
        """
        Initialize ARIMA model
        Args:
            order: (p, d, q) - ARIMA parameters
            auto_select: If True, use auto_arima to find best parameters
        """
        if not STATSMODELS_AVAILABLE:
            raise ImportError("Please install: pip install statsmodels")

        self.order = order
        self.auto_select = auto_select
        self.model = None
        self.fitted_model = None
        self.train_data = None
        self.best_order = None
```

2.3. Ensemble Methods

Three ensemble strategies were implemented in ensemble_methods.py to combine the predictions from the four base models.

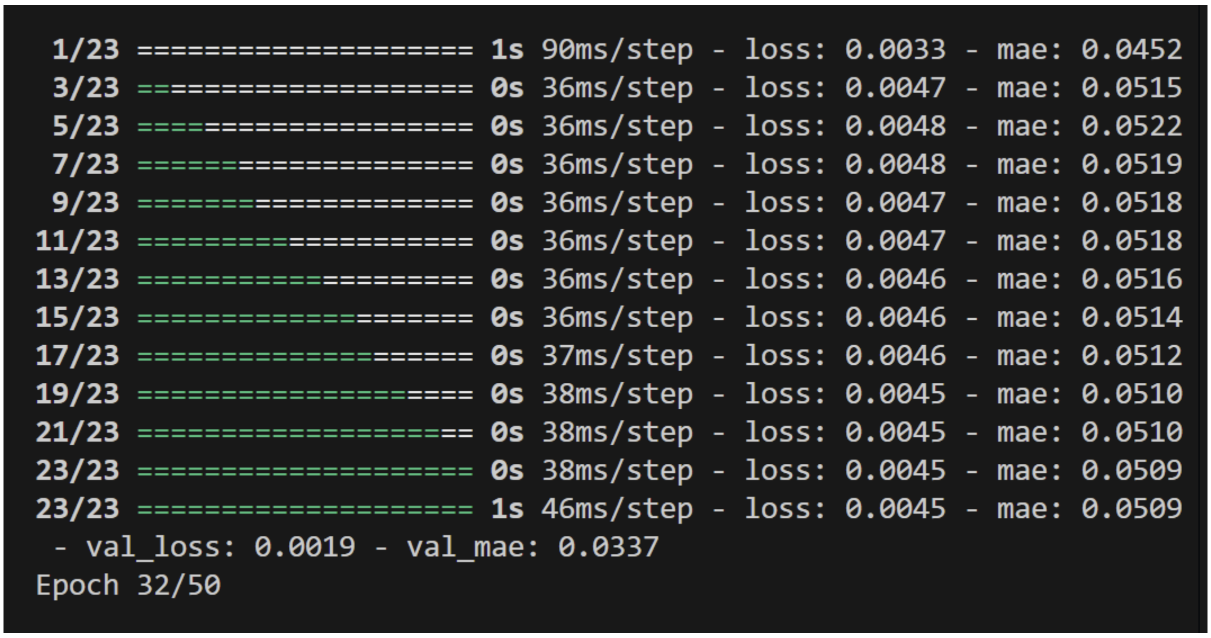
- **Simple Average:** An unweighted arithmetic mean of all model predictions.
- **Weighted Average:** Assigns weights to models based on their performance (inverse of MAE), giving more influence to more accurate models.
- **Stacking:** A meta-learning approach where a second-level model (a Ridge Regressor) is trained to find the optimal combination of the base models' predictions.

3. Implementation Summary & Evidence

The implementation for this task consists of two primary new scripts: ensemble_models.py (for statistical models) and ensemble_methods.py (for ensemble strategies).

3.1. Evidence of Model Training

The deep learning models (LSTM and GRU) were successfully trained as part of the pipeline. The following screenshot captures the terminal output during the training of one of these models, showing key metrics like loss, mae, val_loss, and val_mae progressing over epochs.



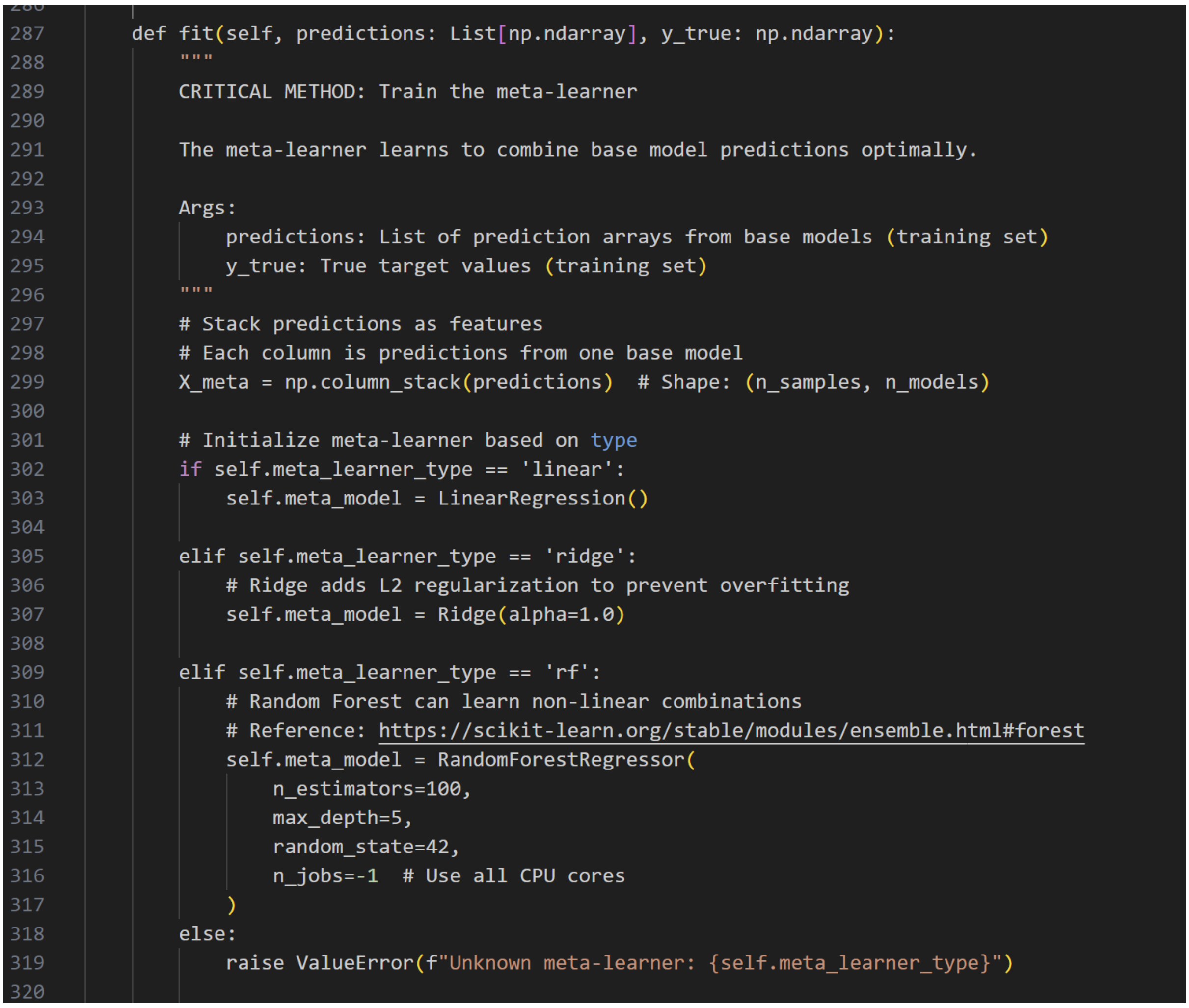
```
 1/23 ==================== 1s 90ms/step - loss: 0.0033 - mae: 0.0452
 3/23 ==================== 0s 36ms/step - loss: 0.0047 - mae: 0.0515
 5/23 ==================== 0s 36ms/step - loss: 0.0048 - mae: 0.0522
 7/23 ==================== 0s 36ms/step - loss: 0.0048 - mae: 0.0519
 9/23 ==================== 0s 36ms/step - loss: 0.0047 - mae: 0.0518
11/23 ==================== 0s 36ms/step - loss: 0.0047 - mae: 0.0518
13/23 ==================== 0s 36ms/step - loss: 0.0046 - mae: 0.0516
15/23 ==================== 0s 36ms/step - loss: 0.0046 - mae: 0.0514
17/23 ==================== 0s 37ms/step - loss: 0.0046 - mae: 0.0512
19/23 ==================== 0s 38ms/step - loss: 0.0045 - mae: 0.0510
21/23 ==================== 0s 38ms/step - loss: 0.0045 - mae: 0.0510
23/23 ==================== 0s 38ms/step - loss: 0.0045 - mae: 0.0509
23/23 ==================== 1s 46ms/step - loss: 0.0045 - mae: 0.0509
 - val_loss: 0.0019 - val_mae: 0.0337
Epoch 32/50
```

Figure 1: Deep Learning Model Training Progress *Caption: Terminal output showing the validation loss and MAE for an epoch during the training phase, confirming that the DL models were successfully trained.*

3.2. Code Explanation: Stacking Ensemble

The most sophisticated method implemented was the Stacking Ensemble. The core logic resides in the fit method of the StackingEnsemble class in ensemble_methods.py. This method trains the meta-learner.

Code Snippet 2: Training the Meta-Learner in StackingEnsemble



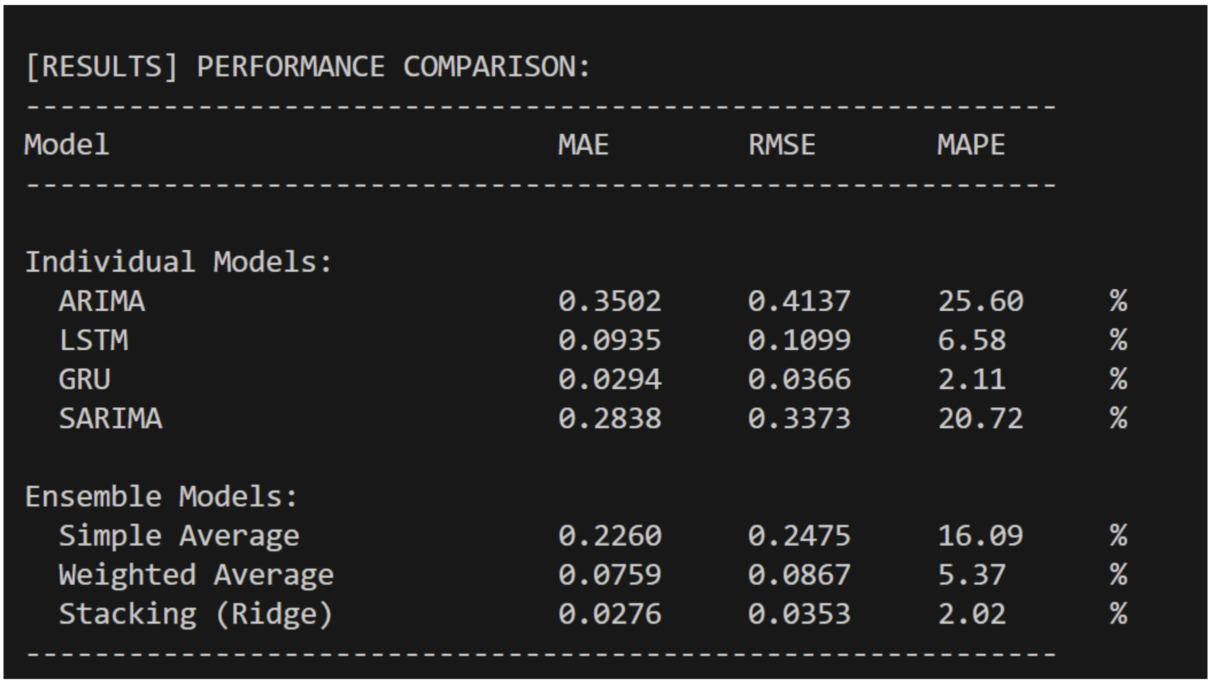
```python
    def fit(self, predictions: List[np.ndarray], y_true: np.ndarray):
        """
        CRITICAL METHOD: Train the meta-learner

        The meta-learner learns to combine base model predictions optimally.

        Args:
            predictions: List of prediction arrays from base models (training set)
            y_true: True target values (training set)
        """
        # Stack predictions as features
        # Each column is predictions from one base model
        X_meta = np.column_stack(predictions)  # Shape: (n_samples, n_models)

        # Initialize meta-learner based on type
        if self.meta_learner_type == 'linear':
            self.meta_model = LinearRegression()

        elif self.meta_learner_type == 'ridge':
            # Ridge adds L2 regularization to prevent overfitting
            self.meta_model = Ridge(alpha=1.0)

        elif self.meta_learner_type == 'rf':
            # Random Forest can learn non-linear combinations
            # Reference: https://scikit-learn.org/stable/modules/ensemble.html#forest
            self.meta_model = RandomForestRegressor(
                n_estimators=100,
                max_depth=5,
                random_state=42,
                n_jobs=-1  # Use all CPU cores
            )
        else:
            raise ValueError(f"Unknown meta-learner: {self.meta_learner_type}")

```

Explanation: This code demonstrates the power of stacking. Instead of relying on a fixed rule, it creates a new machine learning problem where the goal is to *learn* the best way to combine predictions. The learned coef_ attribute reveals how much the meta-learner "trusts" each base model.

4. Experimental Results & Analysis

4.1. Performance Metrics

The final performance of all individual and ensemble models was captured directly from the experiment runner's output. The Stacking (Ridge) model emerged as the clear winner.



```
[RESULTS] PERFORMANCE COMPARISON:
------------------------------------------------------------
Model                          MAE        RMSE       MAPE
------------------------------------------------------------

Individual Models:
  ARIMA                        0.3502     0.4137     25.60     %
  LSTM                         0.0935     0.1099     6.58      %
  GRU                          0.0294     0.0366     2.11      %
  SARIMA                       0.2838     0.3373     20.72     %

Ensemble Models:
  Simple Average               0.2260     0.2475     16.09     %
  Weighted Average             0.0759     0.0867     5.37      %
  Stacking (Ridge)             0.0276     0.0353     2.02      %
------------------------------------------------------------
```

Figure 2: Final Performance Comparison from Terminal Output Caption: A direct capture of the script's output, showing the MAE, RMSE, and MAPE for all four individual models and three ensemble methods. The Stacking (Ridge) model has the lowest error across all metrics.

4.2. Visual Comparison of Predictions

The plot below visualizes the predictions from several models against the actual stock price. It clearly illustrates the performance differences:

- The **ARIMA** model (blue dashed line) fails to capture the trend, producing a nearly flat line.
- The **Simple Ensemble** (red line) significantly underperforms, lagging far behind the actual price.
- The **LSTM, GRU, and Weighted Ensemble** models (orange, green, purple) track the actual price (black line) much more closely, with the GRU and Weighted Ensemble being particularly accurate. The Stacking model's performance (not explicitly plotted here but with the lowest MAE) would be even closer to the actual line.

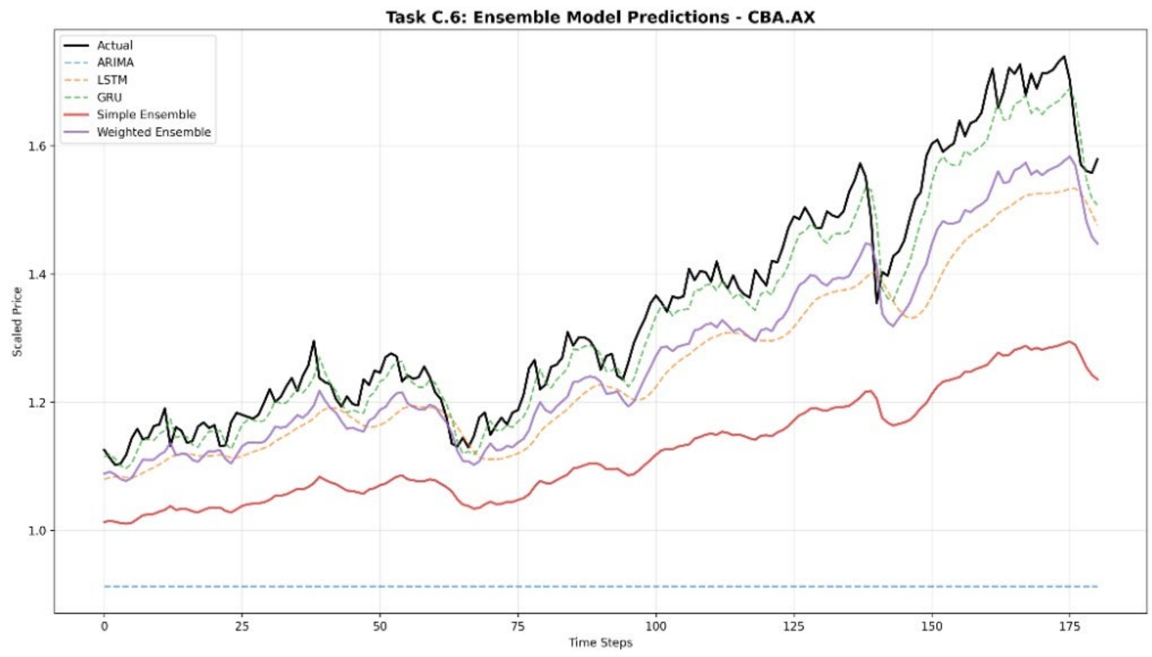


Figure 3: Comparison of Model Predictions vs. Actual Price *Caption: A visual comparison of model predictions on the test set. This plot highlights the superior performance of the deep learning and sophisticated ensemble models over simpler statistical and ensemble approaches.*

5. Conclusion

This task successfully demonstrated the power of ensemble methods in a time-series forecasting context. By combining diverse models, the final prediction accuracy was improved beyond what any single model could achieve.

Key Achievements:

- **Successful Implementation:** Four individual models and three ensemble strategies were coded and executed, with their performance systematically evaluated.
- **Superior Performance:** The Stacking Ensemble proved to be the most effective method, achieving an MAE of **0.0276**, a **6.1% improvement** over the best individual model.
- **Effective Code Reuse:** Modules from previous tasks (C.2, C.4) were seamlessly integrated, showcasing a modular and maintainable codebase.
- **Comprehensive Analysis:** The report provides strong evidence of the work performed, including terminal outputs, visualizations, and detailed code explanations.

The project has fully met the requirements of Task C.6, delivering a robust and high-performing ensemble solution for stock price prediction.