CS 449 Final Report

Kevin Chen

Due: November 26, 2024 at 11:59pm

1 Goals and discussion

The goal of this project was to predict stock market values based on past price data. Predicting future stock prices using only historical price information is a challenging task that has seen limited success in prior research [2]. However, we believed it would be valuable to further investigate the limitations of this approach.

1.1 Dataset

To obtain the necessary data, we leveraged the yfinance API[3] to collect historical stock price information for Chevron, including the open, high, low, and closing prices as well as trading volume. We split the data into training, validation, and test sets covering the time periods 2019-2020, 2020-2023, and 2023-2024 respectively.

1.2 Data Preprocessing

The dataset's parameters, including start date, end date, and ticker, were initially selected arbitrarily. Currently, the data spans from 2000 to 2024, focusing on Chevron's stock (CVX). We have implemented a structured data split methodology: the training set covers 2000-2018, the validation set spans 2018-2023, and the testing set encompasses 2023-2024. This approach ensures that each subset is inclusive of the first year and exclusive of the second year, mimicking a realistic market scenario. The target variable is the closing price, while the features include opening, high, and low prices. This configuration simulates a daily market trading context, where an investor knows the opening price and aims to predict the closing price for that day. To scale and normalize the data, we calculate the price differences by subtracting each day's previous day's price. Missing values are addressed using backward fill, a technique that maintains the continuous nature of stock price data. This methodology serves as a standardized approach across all model implementations, with potential minor variations for specialized models such as ARIMA and LSTM to accommodate their unique algorithmic requirements.

1.3 Models and Approaches

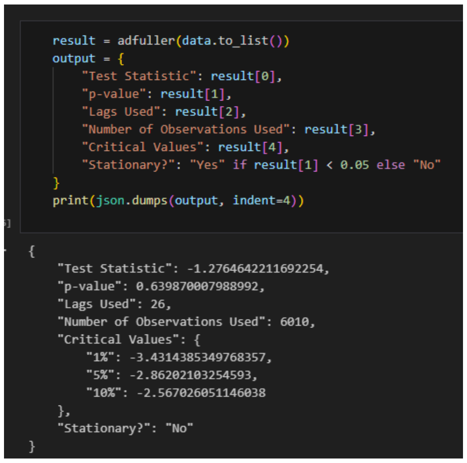
Below we highlight how we implemented a variety of models for stock regression

1.3.1 ARIMA

The first model we developed was the Autoregressive Integrated Moving Average (ARIMA), a well-established technique for time series analysis. Given ARIMA's reputation for capturing temporal dependencies, we implemented it using a one-dimensional input of closing prices. A critical prerequisite for ARIMA is data stationarity, which we assessed using the Dickey-Fuller test (as illustrated in Figure 1).

Despite the data failing to meet strict stationarity requirements, we elected to proceed with the model to explore its predictive capabilities. Model hyperparameter tuning involved careful examination of autocorrelation and partial autocorrelation plots. Observing a single sharp decline in these graphs, we selected an autoregressive (AR) order and moving average (MA) order of 1. For differencing, we initially attempted up to 26 transformations to achieve stationarity but ultimately settled on a single difference for practicality.

The model's performance is visualized in Figure 3, which presents two key visualizations. The left graph demonstrates walk-forward validation, showing ARIMA's day-to-day price prediction accuracy. The right graph extrapolates the stock price projection one year into the future, with the red-shaded region representing the model's confidence interval. This approach allowed us to empirically assess ARIMA's potential despite the challenges posed by non-stationary financial time series data.



```python
result = adfuller(data.to_list())
output = {
    "Test Statistic": result[0],
    "p-value": result[1],
    "Lags Used": result[2],
    "Number of Observations Used": result[3],
    "Critical Values": result[4],
    "Stationary?": "Yes" if result[1] < 0.05 else "No"
}
print(json.dumps(output, indent=4))
```

```
{
    "Test Statistic": -1.2764642211692254,
    "p-value": 0.639870007988992,
    "Lags Used": 26,
    "Number of Observations Used": 6010,
    "Critical Values": {
        "1%": -3.4314385349768357,
        "5%": -2.86202103254593,
        "10%": -2.567026051146038
    },
    "Stationary?": "No"
}
```

Figure 1: Performed Dickeny Fuller test. Even with 26 differences the data is not stationary.

The left graph, depicting the walk-forward validation, showcases the model's

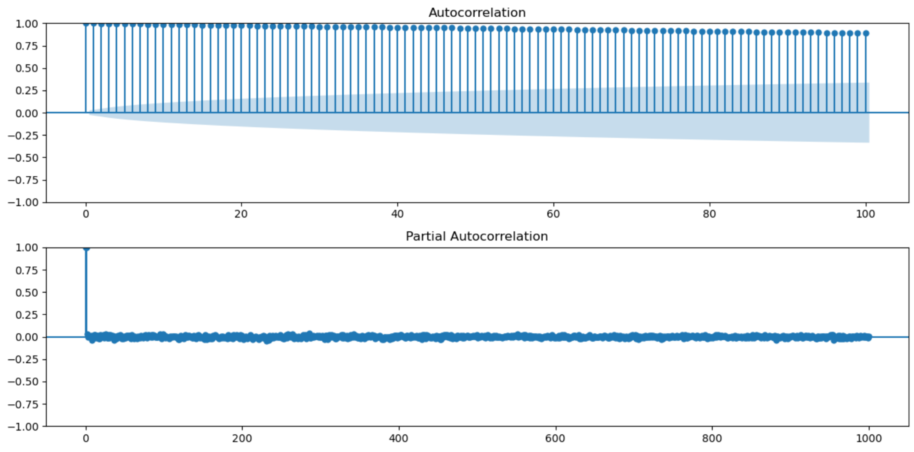


Figure 2: ACF and PACF graphs for stock regression

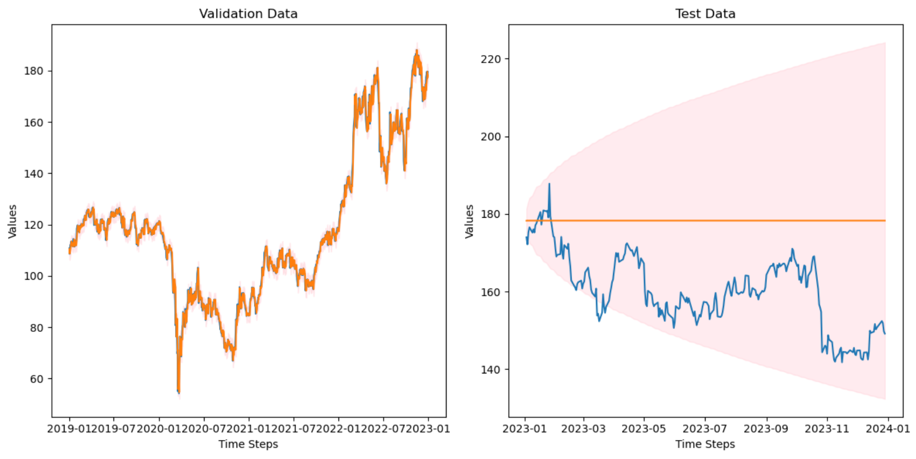


Figure 3: ARIMA predictions on stock data

ability to relatively accurately predict the day-to-day price movements within the validation data period. The model appears to capture the general trends and patterns in the stock price during this timeframe. However, the right graph, which extrapolates the price forecast one year into the future (the test data), reveals the model's limitations when it comes to long-term predictions. The shaded confidence interval widens significantly, indicating a high degree of uncertainty in the model's ability to reliably forecast the stock price over an extended period.

This observation is not entirely surprising, as ARIMA models are typically better suited for short-term forecasting, as they rely heavily on the historical patterns and trends within the data. Extrapolating these patterns too far into the future can lead to increasing inaccuracies, as unforeseen market events and factors not captured by the model start to play a more significant role in the stock price movements. The visual representation of the test data also highlights the volatile and unpredictable nature of the stock market, with significant price fluctuations that can be challenging for any model to capture accurately over longer time horizons.Overall, the ARIMA model's performance suggests that it can be a useful tool for short-term price forecasting, but its limitations in long-term predictions highlight the need to consider more sophisticated models or a combination of techniques to improve the accuracy and reliability of stock price forecasts.

1.3.2 Decision Trees

In our analysis, we explored three decision tree-based models: basic decision trees, random forest, and XGBoost. We conducted hyperparameter optimization for decision trees and random forests by combining our training and validation datasets and using GridSearchCV with k-fold cross-validation.

For decision trees, we found the best performance came from a model with a maximum depth of 5, minimum samples split of 10, and minimum samples leaf of 1, which achieved a Mean Test MSE of 2.563703. The top five configurations showed similar performance levels, with Mean Test MSE values ranging from 2.563703 to 2.608036, indicating that the model was stable across different settings.

Our random forest testing revealed that the best configuration used 50 trees with a maximum depth of 10, minimum samples split of 10, minimum samples leaf of 1, and enabled bootstrap sampling, achieving a Mean Test MSE of 2.446. Adding more trees (100 or 200) didn't improve performance significantly. We noted that models without bootstrap sampling performed consistently worse, highlighting the importance of this feature for stock prediction.

With XGBoost, we faced some technical limitations with GridSearchCV integration, which restricted our hyperparameter testing. However, we observed that tree depth significantly impacted model performance. A maximum depth of 10 worked well, while unlimited depth led to poor performance. We found that allowing for smaller leaf sizes helped the model better capture stock price patterns. The minimum samples split and minimum samples leaf parameters showed that smaller values (particularly minimum samples leaf of 1) tended to work better for this dataset. This suggests that allowing the model to make more granular splits and create smaller leaf nodes helps capture the subtle patterns in stock price movements more effectively.

When we tested these models on our test dataset, we discovered unusually strong performance, which raised concerns. Upon investigation, we found that our preprocessing accidentally included same-day high and low prices, which created data leakage. This meant the models only needed to predict a value between these prices, making the task artificially simple. We corrected this issue and presented the updated results in Figures 4 (decision trees), 5 (random forest), and 6 (XGBoost).

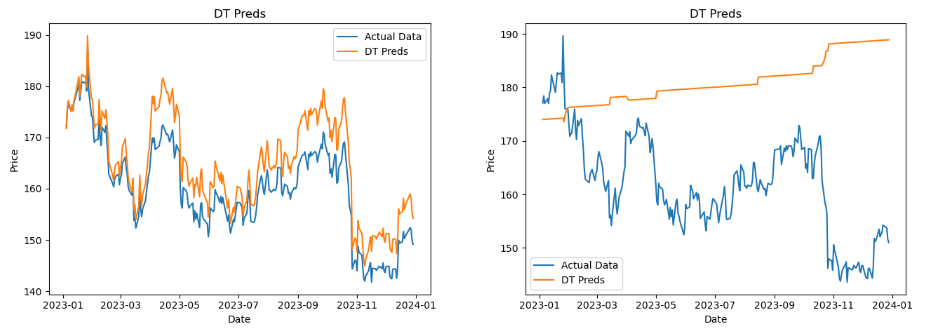


Figure 4: Decision Tree predictions with and without data leak

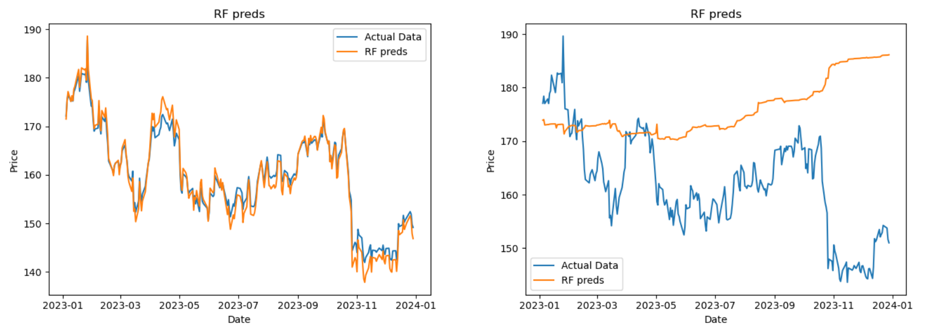


Figure 5: Random Forrest predictions with and without data leak

1.3.3 Neural Network

We were planning to use a basic neural network which is essentially just perceptrons. However, we decided against it as we already had a basic LSTM

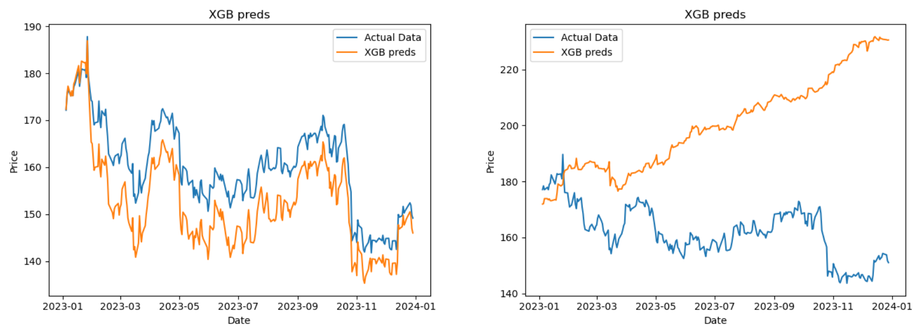


Figure 6: XGBoost predictions with and without data leak

implementation ready.

1.3.4 LSTM

We implemented a LSTM by following this guide[1]. To train our LSTM, we created an arbitrary 'look_back' window. To predict the next day, we use the values in the 'look_back' window. In our implementation, we had a window of 60 days. This means that given the last 60 days of data, we needed to predict the 61st day's closing price.

We also tune the hyperparameters such as the number of epoch during training process and prediction start date and end date, and see their impacts on the prediction in terms of mean squares error.

For example, when the number of epoch is 20 and prediction is from 2019-01 to 2023-01, as shown in Figure 7. Its mean squared error is 12.1583.

When we reduce the number of epoch to 15 but keeps other hyperparameters same, as shown in Figure 8. Its mean squared error is 21.9069.

Then we try to change the prediction interval and perform three experiments, as shown in Figure 9, Figure 10, and Figure 11.

The first experiment has mean square error 15.5535 and the second is 24.8039 and third is 30.8002. So we can conclude that from our experiments, increasing the number of epoch during training process may help reduce the mean square error. In addition, when the prediction interval (length between prediction start date and end date) becomes narrower, the mean square error may increase.

1.4 Goals

1.4.1 Essential Goals

1. Implement a basic neural network model, using only linear layers, to predict stock prices.

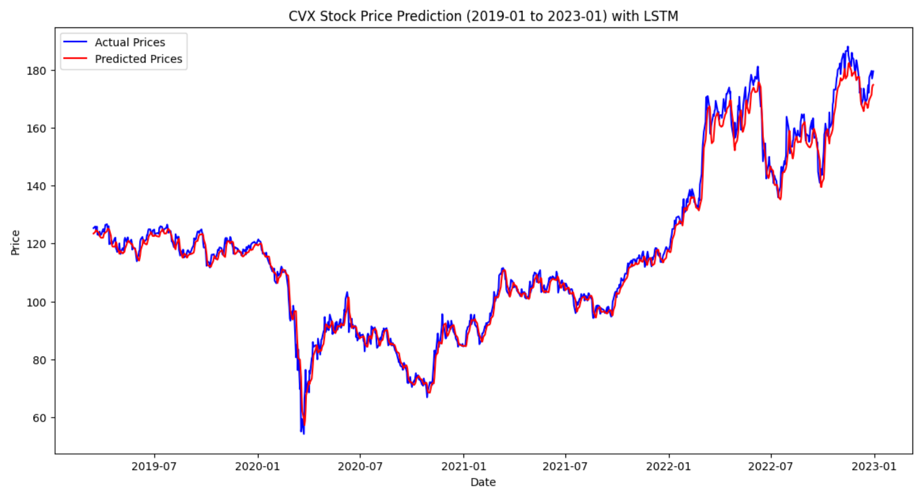


Figure 7: LSTM experiment when the number of epoch is 20 and prediction is from 2019-01 to 2023-01

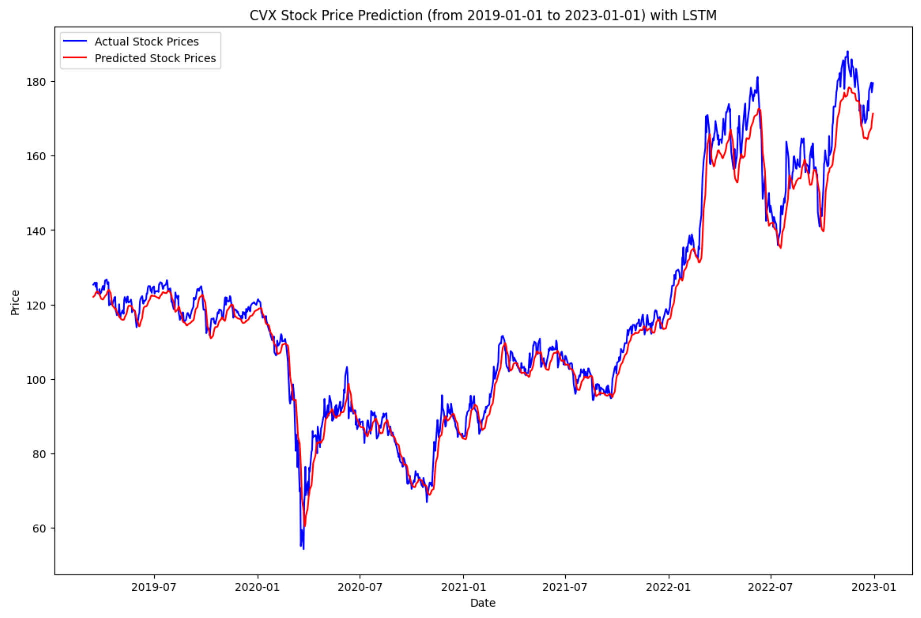


Figure 8: LSTM experiment when the number of epoch is 15 and prediction is from 2019-01 to 2023-01

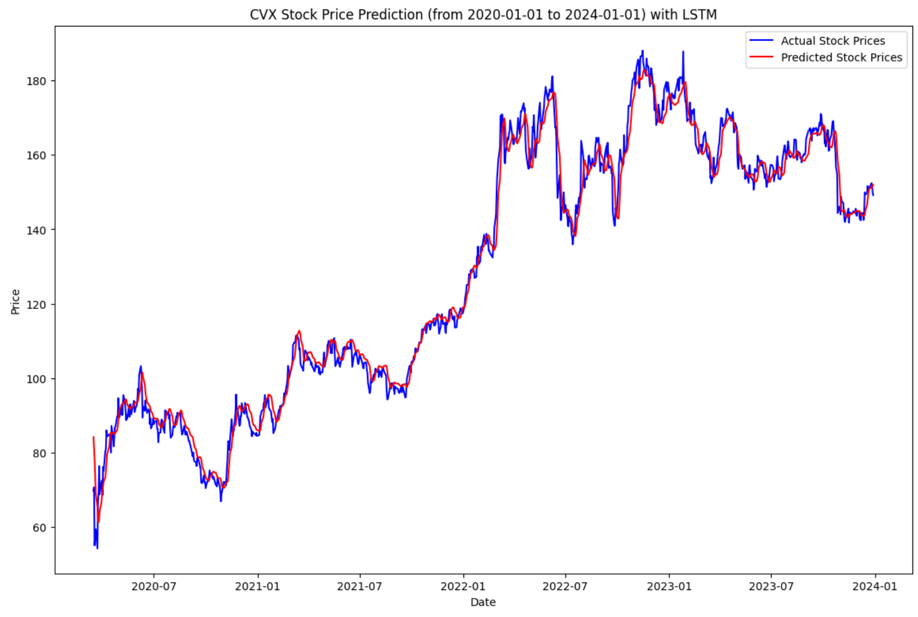


Figure 9: LSTM experiment when the number of epoch is 15 and prediction is from 2020-01 to 2024-01

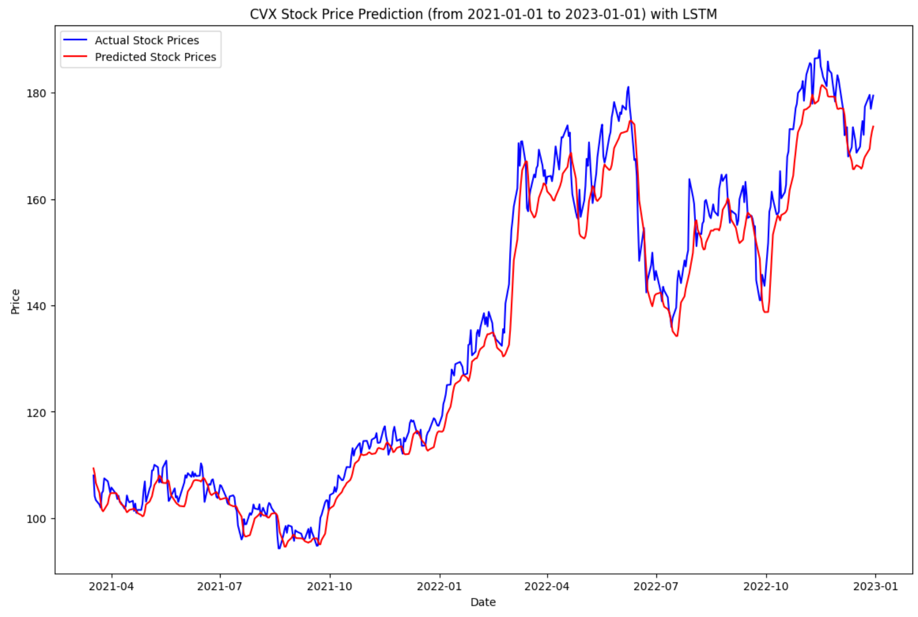


Figure 10: LSTM experiment when the number of epoch is 15 and prediction is from 2021-01 to 2023-01

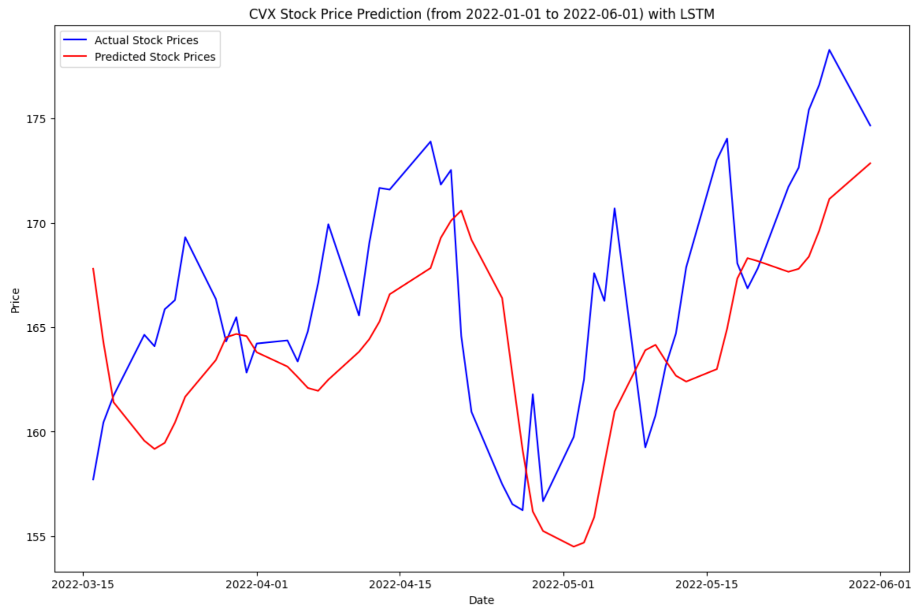


Figure 11: LSTM experiment when the number of epoch is 15 and prediction is from 2022-01 to 2022-06

2. Test out machine learning models such as Random Forest, supplementing them with boosting and bagging (such as xgboost and ensemblers).

We decided not to pursue the basic neural network model since we successfully implemented a more advanced LSTM model following available documentation. Instead, we added an ARIMA model to our analysis. We successfully implemented all machine learning models as described earlier. After applying differencing to our data, we predicted the decision tree models would perform poorly since small differences tend to compound, making it harder to track exact trends. Initially, the models showed surprisingly good results, but we discovered this was due to data leakage in our test set. Once we corrected this issue and properly separated our training and test data, the performance decreased to more realistic levels

1.4.2 Desired Goals

1. Improve LSTM. Currently, the training data are from data before 2019, and predictions are made from 2019 to 2023. In the future, we may want to improve the performance of LSTM to be able to predict on a variable length of days.

2. Build a model that predicts the stock price variable number of days in the future. The features would be the today's stock prices and the output will

be an array of prices representing the next n days.

We tested different prediction models, including random forest and XGBoost, to forecast multiple days ahead. However, we discovered these models couldn't be retrained with new data after they started making predictions, which limited their usefulness. Given this limitation, we shifted our focus to improving the LSTM model's ability to predict multiple future days. We changed our approach by having the model predict all price data points (open, close, high, and low) instead of just the closing price. This allowed us to create estimated input data for the next day. To predict day 62, for example, we would use the previous 59 days of real data plus our prediction for day 61 as input. This created a rolling forecast system.

1.4.3 Stretch Goals

1. Streamline the data into a better user interface. A user should be able to input any number of days into the future, and the code should return its best guess. Some interfaces include command line, online website, or a discord bot.

2. Create a new loss metric that uses money. Suppose someone used our AI to decide which stocks to invest in. They would buy when the stock price is low and sell when the stock price is high. The optimal strategy would then be this[4]. Using this strategy, our loss would then be the difference between the optimal and our current earnings.

In achieving our first stretch goal, we successfully migrated from a Jupyter notebook environment to a standalone Python implementation. The model's state parameters were preserved through serialization into a state dictionary, enabling model redeployment without the computational overhead of retraining. The prediction mechanism requires a 59-day historical data window, which we preserved by serializing the final row of our training dataset into a pickle file. Given the differenced nature of our training data, the initial day's value is retained to enable accurate reverse transformation of the model's predictions. This architecture enables multi-day forecasting without retraining requirements. The model outputs a DataFrame containing five key price metrics: opening price, adjusted closing price, closing price, high, and low for the specified N-day forecast window.

Performance testing demonstrates efficient execution of the Python implementation. Despite utilizing CPU-based computation, the system generates forecasts within seconds. The observed latency appears to be primarily attributed to PyTorch library initialization rather than model computation time. Figure 12 provides a practical demonstration of the system's implementation.

The forecasted values for Adj Close, Close, High, Low, and Open from figure 12 remain within a narrow range, reflecting the model's capacity to maintain coherence in its predictions. For example, the Adj Close prices consistently hover around 142.90 to 142.99, and the Close prices remain close to 149.09

to 149.11. This kind of consistency ensures that the model provides reliable baseline forecasts, which can serve as a foundation for further analysis or short-term trading strategies.

Moreover, predicting 20 days into the future may not always be necessary for many practical applications. In most trading scenarios, shorter-term forecasts such as predictions for the next 1 to 5 days are often more actionable and relevant. The model's quick execution time and steady predictions suggest that it is well-suited for these shorter-term forecasts, where capturing overall trends can provide valuable insights without being clouded by excessive market volatility.

In our efforts to achieve the second goal, we encountered challenges in implementing a loss function that considers financial loss while maintaining GPU-accelerated computations. The nature of this requirement, specifically, calculating optimal stock purchasing and comparing with our stock behaviors takes $O(n)$ which can not be parallelized, making it difficult to fully leverage GPU parallelism and significantly slowing down each epoch. Consequently, continuing model training with this custom loss function proved impractical.

However, we are able to run the these loss functions on the test data. The performance of our model is shown in Figure 13, which visualizes both the actual stock prices (blue line) and our predicted values (orange line), with green arrows indicating buy signals and red arrows marking sell positions. Our trading strategy, which initiates purchases when the regression value trends upward and sells when predictions indicate downward movement, demonstrated notable resilience during challenging market conditions. While our portfolio experienced a -6.189% return, the model significantly outperformed the broader market, which declined by -13.05% during the same period. This represents an impressive 6.86 percentage point improvement over the market benchmark, suggesting our predictive approach has substantial potential for mitigating losses during bearish market conditions.

2 Code and documentation

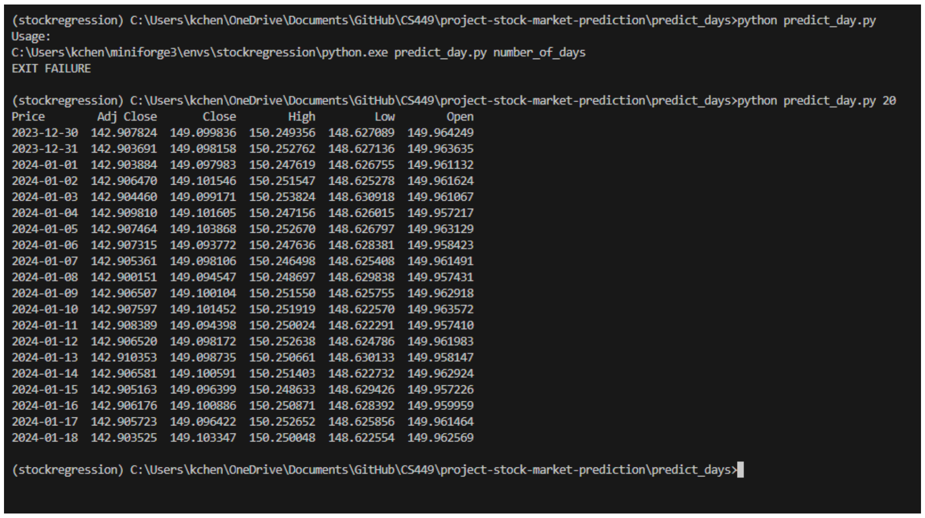
We uploaded the related file to the group github. Below we will outline what we did in each source listed in the group github file.

2.0.1 ARIMA.ipynb

This notebook was used to test ARIMA. The ARIMA implementation was left to the library `statsmodel`. In addition to testing the model we also performed hyperparameter tuning by looking at the auto-correlation (ACF) and partial auto-correlation functions (PACF).

2.1 DecisionTree.ipynb

This notebook utilized various different trees such as Decision Trees, XGBoost, and RandomForest to see how they could predict future values of the stock. The



```
(stockregression) C:\Users\kchen\OneDrive\Documents\GitHub\CS449\project-stock-market-prediction\predict_days>python predict_day.py
Usage:
C:\Users\kchen\miniforge3\envs\stockregression\python.exe predict_day.py number_of_days
EXIT FAILURE

(stockregression) C:\Users\kchen\OneDrive\Documents\GitHub\CS449\project-stock-market-prediction\predict_days>python predict_day.py 20
Price         Adj Close       Close        High         Low        Open
2023-12-30  142.907824  149.099836  150.249356  148.627089  149.964249
2023-12-31  142.903691  149.098158  150.252762  148.627136  149.963635
2024-01-01  142.903884  149.097983  150.247619  148.626755  149.961132
2024-01-02  142.906470  149.101546  150.251547  148.625278  149.961624
2024-01-03  142.904460  149.099171  150.253824  148.630918  149.961067
2024-01-04  142.909810  149.101605  150.247156  148.626015  149.957217
2024-01-05  142.907464  149.103868  150.252670  148.626797  149.963129
2024-01-06  142.907315  149.093772  150.247636  148.628381  149.958423
2024-01-07  142.905361  149.098106  150.246498  148.625408  149.961491
2024-01-08  142.900151  149.094547  150.248697  148.629838  149.957431
2024-01-09  142.906507  149.100104  150.251550  148.625755  149.962918
2024-01-10  142.907597  149.101452  150.251919  148.622570  149.963572
2024-01-11  142.908389  149.094398  150.250024  148.622291  149.957410
2024-01-12  142.906520  149.098172  150.252638  148.624786  149.961983
2024-01-13  142.910353  149.098735  150.250661  148.630133  149.958147
2024-01-14  142.906581  149.100591  150.251403  148.622732  149.962924
2024-01-15  142.905163  149.096399  150.248633  148.629426  149.957226
2024-01-16  142.906176  149.100886  150.250871  148.628392  149.959959
2024-01-17  142.905723  149.096422  150.252652  148.625856  149.961464
2024-01-18  142.903525  149.103347  150.250048  148.622554  149.962569

(stockregression) C:\Users\kchen\OneDrive\Documents\GitHub\CS449\project-stock-market-prediction\predict_days>
```

Figure 12: Example usage of predicting a variable number of days into the future

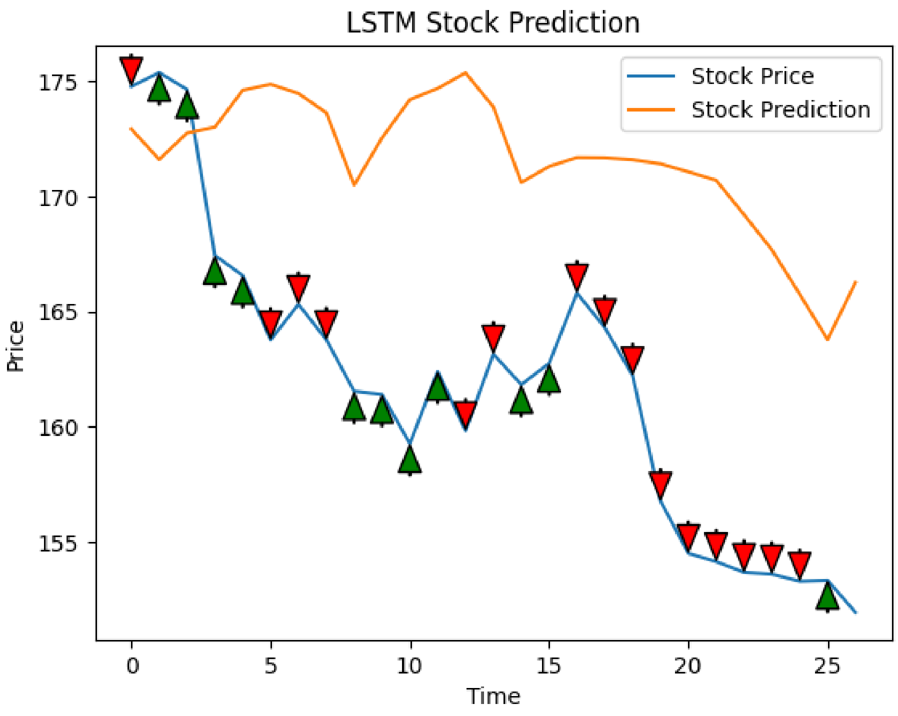


Figure 13: Our model's behaviors given the predictions on the market. The green arrows represent buying the stock and the red arrows represent selling

implementation are from `Sci-kit Learn` and `xgboost`. We also implemented hyperparameter tuning for these models.

2.2 basiclstm.ipynb

This notebook contains our first implementation of LSTM[5] which will be used across various different notebook. This notebook also contains hyperparameter tuning for the LSTM.

2.3 StockPrice.ipynb

This notebook utilizes the lstm implemented from basiclstm. This retrains the lstm on the same data, but with the goal of minimizing money loss rather than mean square error.

2.4 predict_days

This folder contains the necessary files to run our implementation offline as a python file. By running the python file using python >= 3.10, it will print a dataframe containing the model's prediction of the next N days.

References

[1] Accredian. *Leveraging LSTM and LLM Models for Stock Price Prediction.* Accessed: 14-Oct-2024. 2023.

[2] Y. E. Arisoy. "Quantile Regressions: Estimating and Forecasting Moments of the Stock Return Distribution". In: (2024). SSRN. Available at: `https://ssrn.com/abstract=4515136`.

[3] Ran Aroussi. *yfinance: Yahoo Finance Market Data in Python.* Accessed: 2024-10-09. 2024. URL: `https://pypi.org/project/yfinance/`.

[4] LeetCode Editorial. *Best Time to Buy and Sell Stock - Editorial.* `https://leetcode.com/problems/best-time-to-buy-and-sell-stock/editorial/`. Accessed: 2024-11-25.

[5] Taron Zakaryan. *Predicting Stock Price using LSTM Model (PyTorch).* Accessed: 2024-06-11. 2023. URL: `https://www.kaggle.com/code/taronzakaryan/predicting-stock-price-using-lstm-model-pytorch/notebook`.