May 1, 2024

C964: Computer Science Capstone

Wesley Lancaster

Part A: Project Proposal Plan

This document covers all the necessary details regarding the AI Stock Helper program, which will support employees by providing data-based trading suggestions.

Project Summary

- The Problem: Human investors are subject to bias and emotional decision-making, resulting in stock market losses.
- Employees at Dated Investors, Inc. are tasked with selecting stocks that hopefully will grow in value. They serve as the company's backbone, making decisions that benefit or harm the company and its clients. These employees will benefit significantly from an impartial program that uses the latest technology to recommend stock trades.
- While existing tools provide our employees calculations, none utilize machine learning. Such a program could be a “second mind” to aid decision-making.
- Deliverables:
    - Program takes a stock ticker, trains an AI model on the stock, and returns a recommendation
    - Models are saved, retrieved, and updated from a database
    - The UI is added to the application
    - The application is deployed to an online platform and is accessible by a link
- The expected outcome of this project is that by using a machine learning tool, investors can be aided in making impartial and rational decisions. Investors will feel more comfortable trading when the tool agrees and reconsider their decision when it disagrees.

Data Summary

- Data will be collected using the yfinance API during runtime.
- During development, ten years of daily stock prices will be pulled from the API. The application will preprocess the “Closing” prices from each day and split the data into a larger training set from the beginning of the history and a smaller testing set. Once trained to reasonably predict stock prices, the model will be saved alongside its ticker symbol, output, and last update date. After deployment, the application will attempt to update the model with new data every time the ticker is requested.
- The API can return all financial data from Yahoo Finance, which is more than enough data to train AI models. Though the API is robust, the numpy library will aid in preprocessing the data.
- All data available in yfinance is freely available to the public. However, the program should not be trained on any stock where insider trading may be an issue. Additionally, the details regarding the application can only be shared on a need-to-know basis. Once deployed, employees are not to share links, data, or screenshots of the program with persons outside the company.

Implementation

The CRISP-DM methodology will ensure the project starts well-defined and on a trajectory to success:

Business Understanding	Investors need reliable, timely tools to help them make stock market decisions. An AI software tool will be created that receives a ticker symbol, updates its model to reflect current data, and provides a suggestion for the investor on whether to buy or sell that stock.
Data Understanding	Data will be collected using the yfinance API during runtime. The software will preprocess the "Close" prices from each day for the AI and train to examine trends and patterns while mitigating irregularities. Once trained initially, it can be updated using new data as needed. The models will be updated whenever their last updated date doesn't match the previous market close.
Data Preparation	Ten years of daily stock prices will be used.
Modeling	Train models using a majority of the dataset.
Evaluation	Test models on the remaining minority of the dataset.
Deployment	Build an application with a database and UI around the models.
Monitoring	Models attempt to automatically update every time a user submits a request. Models will be continually tested, fine-tuned, and improved.

Project implementation plan:

1. **Jupyter Notebooks:** The implementation will start in Jupyter Notebooks for convenience in cross-editing the same document, running portions of code in real-time, and displaying results. Objectives of this part of the implementation are:
    a. The model will be trained and tested on ten years of data
    b. A Mean-Squared Error goal will be determined to ensure the model's accuracy is sufficient
    c. A standard for training a model that produces consistent results across any ticker
    d. The model will predict future prices
    e. The model will be updatable when new data is available
    f. Saving and loading a model from a database
2. **Flask App:** The code will be transferred to a Python file from Jupyter Notebooks, and the application will be developed in Visual Studio Code. The objectives of this portion are:
    a. Build a simple Flask application around the model file that runs locally
    b. The application provides tickers in a drop-down menu and provides a simple text result
    c. The application can load, update, and save models.
3. **Deployment:** The Flask application will be deployed to Heroku and accessible by link.
4. **Maintenance:** The application will be assessed and continuously tested. This is also the point where plans can be made to make the program more robust, such as including more prescriptive features, testing its accuracy, and creating a plan to improve accuracy.
5. **Outcome:** The expected outcome is that the application will be able to give a conservative prediction that our employees can rely on for a grounded insight into the future stock's price.

Timeline

Deliverable	Duration	Projected start date	Anticipated end date
Project definition & planning	4 Days	6/3	6/7
AI model coding	5 Days	6/10	6/14
Database creation	2 Days	6/17	6/18
UI creation	1 Day	6/19	6/19
Combining elements into one app	2 Days	6/20	6/21
Deployment to Heroku	1 Day	6/24	6/24

Evaluation Plan

Reviews and testing will be conducted during each step to verify that the project is being built correctly. A review of the requirements before starting a deliverable will ensure that the deliverable meets all the project needs, and a review after the deliverable will double-check its completeness. Testing will be conducted as appropriate for the deliverable – some deliverables, such as model coding, allow testing to be done before beginning, during, and after completion. Others can only be tested after completion. The rule of thumb here will be to start testing as early as possible, with every step tested thoroughly by the end of the deliverable. With a focus on reviewing before and after each deliverable and prioritizing testing, this project will come together seamlessly without needing to backtrack to solve issues.

The model itself will need to be tested as well. Fortunately, as the model is trained and tested, it returns a Mean Squared Error (MSE), a feature built-in by the project libraries. This MSE is a floating-point number that helps the user understand how close the trained model is getting to actual data. While the model is being developed, a general goal for each model's MSE will be established to ensure consistency.

Once the application is finished, our team will conduct user acceptance testing, utilizing the Asset Management team so our fellow employees can try out the program. Their feedback will be recorded to determine the validity of the application. Once deployed, the program will undergo field testing to evaluate its performance. This will help us determine if the application has met the needs of our project.

Resources and Costs

Resource	Phase	Cost
Hardware – work computers (already provided)	All	Free
Software – SQLite, Jupyter Notebook, Visual Studio Code, Google Colab, several dependencies	Development, Deployment, Maintenance	Free
Development Work Hours – 75	Development, Deployment	$3600
Heroku – Hosting platform	Deployment, Maintenance	$25/month
Maintenance Work Hours – 10 per month	Maintenance	$475/month

Part B: Application

AIStockHelper Web App

http://wesslancaster.com:5000/

This is the completed application that users may access via a link.

Jupyter Notebook Code Demo

https://colab.research.google.com/drive/1z96VjkJXcIOQ6KdNjEPjhmxKKfLd7FLH?usp=sharing

This is an excellent resource for understanding the code. It shows every facet of building a model as well as an explanation. It begins by creating an outdated model and performs an update later. It also tests SQL and data processes vigorously.

Github

https://github.com/wessbl/wbl-aistocks

Please note that the installation is extensive due to the number and size of packages the project requires, so I recommend only using this resource to review the code that makes the application. Additionally, updates will continue to be posted in the future. Currently, these are the main files of the program:

- app.py: Contains the code for launching the application and serves as the controller
- model/lstm_model.py: The LSTM model file. Most of the back-end processing is done in this one file, including database management, model updating, and training. That being said, the Jupyter Notebook was prepared as a walkthrough of the code to demonstrate how it works.

Part C: Post-implementation Report

Business Requirements Document

Company Name	**Date**
Dated Investors, Inc.	6/2/24
Project Name	**Created By**
AIStockHelper	Wess Lancaster

Executive Summary	• This project will utilize AI to help investors trade in the stock market • It will result in an online application and recommend buying or selling a particular stock
Business Objectives	• Use the latest technology to predict stock prices • User will submit a ticker and receive a recommendation • Project will deploy on June 24th
Needs Statement	• AI is proving to be valuable cutting-edge technology in our industry • Our employees need an additional resource that can help them decide on stock market trades • This project will be a “second mind” for investors to help them grow value in the stock market
Project Scope	• Create a web app that takes a ticker and returns a recommendation • Development will cost $3600, $500 per month afterward in platform and maintenance costs • Deliverables: Project planning, AI model, Database, UI, Flask App, Deployment
Requirements	• Initial Development: Google Colab, Jupyter Notebooks • Mid-Development: Visual Studio Code, expansive dependencies including Flask and AI libraries • Deployment: AWS EC2 server instance, PuTTY

Solution Summary

Trading in the stock market is risky, and human error can result in costly mistakes. Our employees needed better tools to make balanced decisions. It was decided to utilize AI to help our employees trade more effectively. I designed a plan to create a tool to give a prediction for a selected stock ticker that is accessible online.

Utilizing AI, I created software to take a ticker and predict an increase or decrease in that stock's price. This means that our employees will have the confidence to buy a stock that the AI suggests will increase in value. Our employees can also run the software daily to see if that stock will lose value, helping them determine when to exit the investment appropriately. If the software employee desires to purchase a stock but the AI asserts it will lose value, the employee will need to consider the timing of the investment and tighten up their strategy. Regardless, the tool will encourage rational thinking and keep employees grounded.

Development began in Jupyter Notebooks, where I determined how to build the model. All the primary functions were written there, from modeling stock and making predictions to generating visualizations to database management. This proved to be an excellent resource throughout future stages of development. Once finished, I transferred the code to Visual Studio Code to turn it into a Flask application and deployed it on an Amazon Web Service (AWS) EC2 server. While much more hands-on, deploying with AWS allowed a virtual environment on their server, which gives the application much more freedom in the dependencies and versions it uses. Further, the cost is comparable to Heroku, the original deployment route.

The original hypothesis of the project was that a machine learning program would be able to use stock price history to provide an adequate recommendation relatively quickly. The question I'm led to ask is, how quickly? In a real-world setting, the app responds in less than 5 seconds on average, though it tends to take just over 20 seconds when it is updated. While I've been humbly reminded that predicting stock prices requires extensive calculations, I am pleased to accept that my hypothesis was correct. Though we are used to getting immediate answers in the digital era, having a good recommendation to purchase a stock in less than 30 seconds is a significant feat. And while accuracy is not a factor during this stage, I am impressed with the results I've seen. It tends to mirror another popular stock tool, the moving average, except it can also guess where the stock is heading in the future. If nothing else, the application has the potential to be highly accurate with future development.

Data Summary

All data is accessed via the yfinance API during runtime, using the yf.download() function to get ten years of stock prices. From there, only the closing prices are kept, and the numbers are scaled down for training. The scalar that transformed the data is kept for future use. Long Short-Term Memory (LSTM) models are trained on the data over several epochs to become acquainted with its patterns and velocities. Afterward, the scalar performs a reverse transformation on the original stock price and the model predictions of the stock price. All data is used in matplotlib functions to display the data.

No data was collected during the design phase because the software is the method of obtaining the data. Data was collected during development to train the models initially and during maintenance to

update the models on new data. Updates involve getting all prices from the same start date through the most recent close price, then training the models on the latest price over three epochs.

Over the following pages, I've included screenshots of the data as it is processed in preparation to be fed to an LSTM.

```
import yfinance as yf
stock_data = yf.download('AAPL', start='2014-01-01', end='2023-12-31')
stock_data
```

```
[*********************100%%**********************]  1 of 1 completed
```

Date	Open	High	Low	Close	Adj Close	Volume
2014-01-02	19.845715	19.893929	19.715000	19.754642	17.273230	234684800
2014-01-03	19.745001	19.775000	19.301071	19.320715	16.893812	392467600
2014-01-06	19.194643	19.528570	19.057142	19.426071	16.985929	412610800
2014-01-07	19.440001	19.498571	19.211430	19.287144	16.864454	317209200
2014-01-08	19.243214	19.484285	19.238930	19.409286	16.971252	258529600
...	...	...	...	...	...	...
2023-12-22	195.179993	195.410004	192.970001	193.600006	193.091385	37122800
2023-12-26	193.610001	193.889999	192.830002	193.050003	192.542816	28919300
2023-12-27	192.490005	193.500000	191.089996	193.149994	192.642548	48087700
2023-12-28	194.139999	194.660004	193.169998	193.580002	193.071426	34049900
2023-12-29	193.899994	194.399994	191.729996	192.529999	192.024185	42628800

2516 rows × 6 columns

Figure 1: The raw data from yfinance

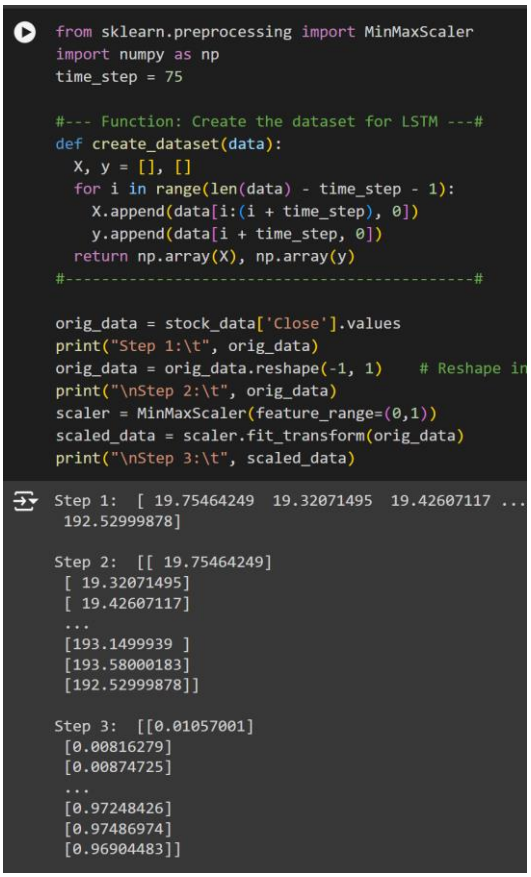

Figure 2: Step 1: All but 'Close' prices are filtered. Step 2: The data is put into a multidimensional array. Step 3: The data is scaled to numbers between -1 and 1

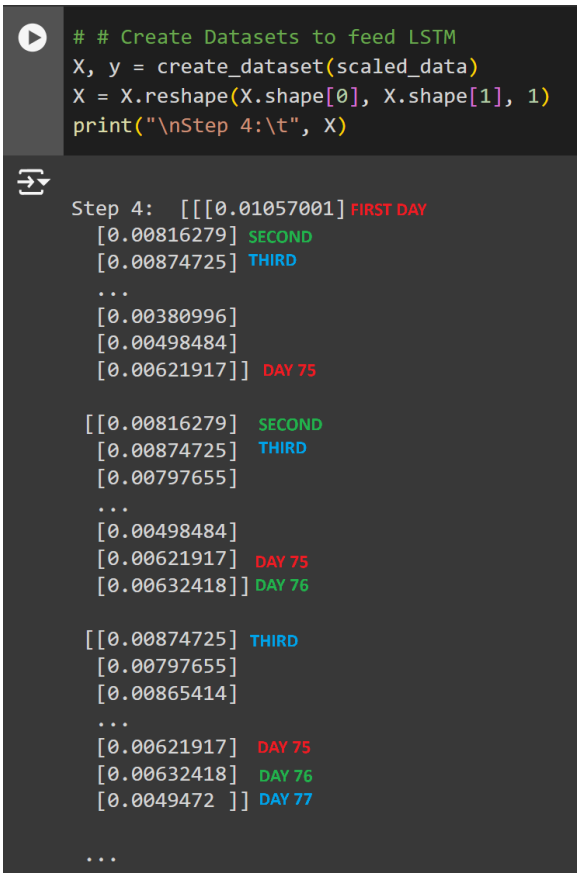

Figure 3: For each day in the dataset, 75 days are added to an array starting with that day and ending 75 days later.

That last image of the processed input data is one of the best ways to understand how an LSTM works step-by-step. The machine takes data in 75 pieces at a time and tries to predict what the 76th piece will look like. For example, it takes days 1-75 to predict day 76, then days 2-76 to predict 77, and so on. The remarkable thing is that the output is continuous, so you see a smooth line on a larger scale instead of a jumble of points.

Machine Learning

- **What:** A Long Short-Term Memory model (LSTM) is a Recurrent Neural Network, a collection of nodes that use logic to make decisions. They utilize regression, a way for computers to determine patterns from raw data, and output a continuous value, such as temperatures over time. This makes them a natural choice for evaluating stocks and predicting future prices.
- **How:** The model was trained with a supervised learning technique, where it was given historical prices and asked to predict future ones. This is seen in my examples using model.predict(days), which is used both to mirror known data and make a prediction for future prices. Specifically, I built the model to monitor different time values over 1, 5, and 75 days to model daily changes and roughly capture weekly and seasonal cycles, as the stock market only trades five days per week.

Fortunately, all the heavy lifting was done by utilizing package libraries such as keras and tensorflow.

- **Why:** Each stock has a pattern of highs and lows with wild fluctuations in between. I chose an LSTM because they can predict by identifying such patterns over a specified period. This makes LSTMs well-suited for predicting stock prices using only the history of recent prices.

Here is a sample of the code used to generate the models. Note that lstm_unit = 75, and epochs should equal 20 or more to achieve a Mean Squared Error (MSE) less than 0.0003.

```
#--- Function: Train a new or existing model ---#
def train_model(model):
  # Build and compile the LSTM, if needed
  if (model == None):
    model = Sequential()
    model.add(LSTM(lstm_unit, return_sequences=True,
        input_shape=(time_step, 1)))
    model.add(LSTM(lstm_unit))
    model.add(Dense(5)) # Capture weekly cycles, 5 trading days/week
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mean_squared_error')

  # Train model
  model.fit(X, y, epochs=epochs, batch_size=batch)
  if verbose: print("Model is trained!")
  return model
#-----------------------------------------------#
```

The models didn't only need to be trained; they also needed to return a recommendation to buy or sell the stock. To do this, Keras' LSTM.predict() method was used on a loop to predict a specified number of days in the future. Seventy-five days of preprocessed and scaled data must be given before the first day can be predicted. After that, the prediction is added to the array, and the first day is removed so the next day can be predicted, and so on. Here is the method:

```
    #--- Function: Predict price over future given days ---#
def predict(days):
  multi_day_data = scaled_data[-time_step:]
  X_predict = multi_day_data[:time_step].reshape(1, time_step, 1)

  # Create an array with the first index at last known close price
  prediction = []
  prediction.append(orig_data[len(orig_data) - 1][0])

  # Predict the next 5 days' prices iteratively
  print("days:\t", days)    # TODO remove debug print
  for i in range(days):
```

```
      # Predict the next day's price
      scaled_price = model.predict(X_predict)
      actual_price = scaler.inverse_transform(scaled_price.reshape(-1, 1))
      prediction.append(actual_price[0, 0])

      # Update the input sequence for the next prediction
      multi_day_data = np.append(multi_day_data, scaled_price)
      X_predict = multi_day_data[-time_step:].reshape(1, time_step, 1)

  return prediction
#-------------------------------------------------------#
```

The returned prediction is used for the Model Prediction image type below. Now that we've received the prediction, we can also decide what it means for the user. The stock should be bought if the last predicted price equals or exceeds the previous closing price. Otherwise, the stock will decrease in value and should be sold (or not bought at all). For the curious, I decided to buy the stock if it were to stay the same because assets that merely hold their value are sometimes valued by investors, such as so-called "park my cash" investments. Additionally, I added a percentage to help the end user understand the numbers behind the prediction.

Here is the buy_or_sell method:

```
#--- Function: Determine whether to buy or sell stock ---#
def buy_or_sell(prediction):
  last_price = orig_data[len(orig_data) - 1][0]
  last_predicted = prediction[len(prediction) - 1]
  percent = (last_predicted * 100 / last_price)

  if last_price <= last_predicted:
    percent = percent - 100
    percent = str(f"{percent:.2f}")
    return "Buy: AIStockHelper says this stock will go up in value by "
        + percent + "%."
  else:
    percent = 100 - percent
    percent = str(f"{percent:.2f}")
    return "Sell: AIStockHelper says this stock will go down in value by "
        + percent + "%."
#-------------------------------------------------------#
```

Validation

To validate the LSTM's training, I divided the data into training and testing sets, where the model was trained on the first 80% and tested on the last 20% of the closing prices of a stock over the last ten

years. I followed the MSE values as it trained to monitor its progress and visually inspected its predictions. I established a general goal of an MSE value of 3.0e-04 or lower, a feat that is quickly achievable through careful tuning of the LSTM.

There was also a point when future stock prices would always jump way up, another phase where they would take a dramatic plunge. I carefully monitored its predictions and tweaked how it learned to ensure its results were realistic and consistent with the pattern of its stock.

Addressing the program's accuracy in lay terms requires an explanation of MSE. MSE measures the average squared difference between the actual and the predicted values. This seemingly incongruous metric is more understandable if you convert it to a percentage by determining the Root Mean Squared Error (RMSE) and dividing it by the mean of actual prices. I included a block of code that does precisely this in Jupyter Notebooks and found an error percentage of 4.98%, as seen below. This result varies widely each time a model is trained but tends to stay around 5-6%.

```
[37] ### Get the percentage accuracy of the program as it is after updating (20 epochs)

     # Get actual and predicted values over the same range
     actual = scaled_data[(time_step + 1):]
     mirror = mirror_data(model, False)

     # Calculate MSE & RMSE
     mse = np.mean((actual - mirror)**2)
     rmse = np.sqrt(mse)

     # Mean of actual values
     mean_actual = np.mean(actual)

     # Percentage error
     percentage_error = (rmse / mean_actual) * 100

     print(f'MSE: {mse:.4f}')
     print(f'Percentage Error: {percentage_error:.2f}%')

     81/81 [==============================] - 2s 27ms/step
     MSE: 0.0003
     Percentage Error: 4.98%

Above: MSE, RMSE, and an error percentage of the model
```

Figure 4: Code and output that calculates the MSE, RMSE, and percentage error of the LSTM model

There were many revisions to the original plan. My first choice was to create a Support Vector Machine since they are good with regression problems. Unfortunately, the first generation of these was utterly inferior to the first LSTM I developed, with the prediction nearly cutting the price by a third. Meanwhile, the first generation of LSTM was intriguing – while it also predicted the stock would plummet, it at least traced a line downward that was more realistic. But even more surprising was that it mirrored the actual price closely, following every market trend. The LSTM interested me and seemed to suit my needs better.

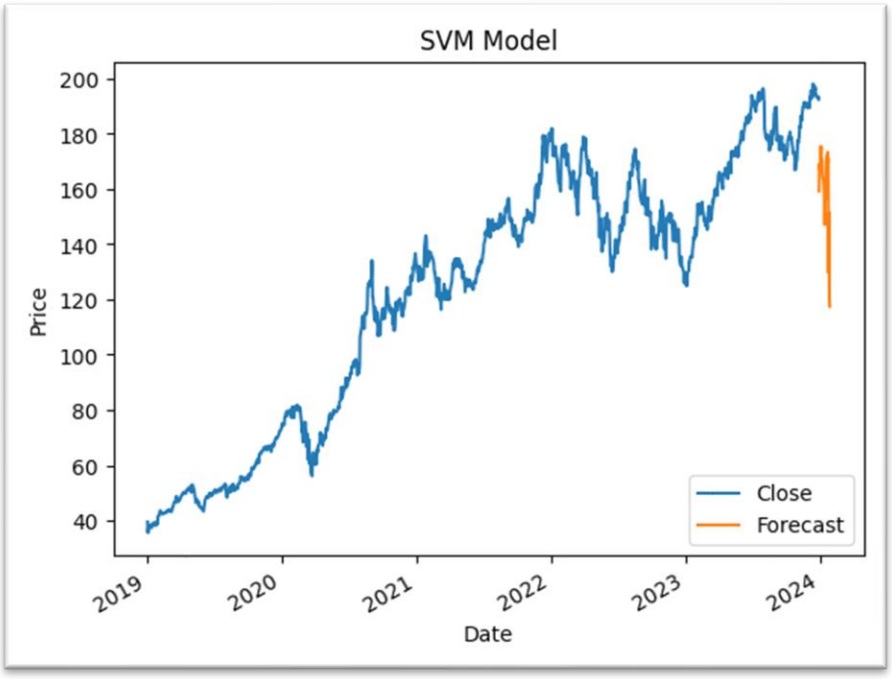

Figure 5: My first generation SVM model on AAPL stock

Figure 6: My first generation LSTM model on AAPL stock, zoomed in

To make the LSTM more accurate, I changed how the models were trained. Below is the code behind the first-generation model, as well as a node diagram of the model generated by the code:

```
# Build the LSTM model
model = Sequential()
model.add(LSTM(64, return_sequences=True, input_shape=(time_step, 1)))
model.add(LSTM(64))
model.add(Dense(64))
model.add(Dense(1))
```

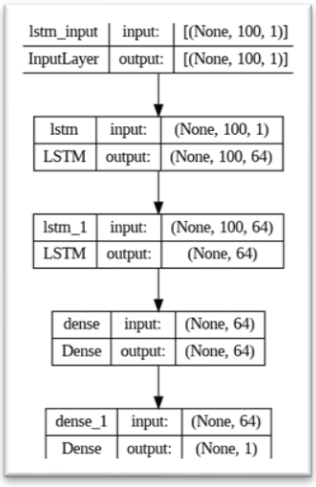

Figure 7: A node diagram of my first LSTM model

As discussed before, I changed these values to fit the weekly and seasonal fluctuations of the stock market, and the node diagram for the final version is provided in the next section. Additionally, I shrank the time_step from 100 in the first generation to 75 in the last and doubled the number of epochs. These changes had a dramatic impact on the stability of the predictions as well as the way the model mirrored actual data.

Once I got the final version of the model, I made the Flask app around it and deployed it. I found that wait times for training and image generation exploded when running the application on a remote server – it was time to optimize. I had the models train on ten epochs at the beginning, and ten epochs were used every time new data was available, but I found the wait times were longer than one minute. This did not meet one of my objectives for the project: to provide recommendations quickly. I lowered the epochs to three and monitored the MSE to ensure it did not drop. I had seen before that adding new data can reduce the model's accuracy. I formulated a worst-case scenario: what if only one employee wants to check on a stock ticker and does so only once per month? And what if there was only one epoch to train every time? I was surprised to find that my initial assumption was incorrect – even with a month between each update, each epoch will gradually bring the MSE down:

```
Day: 0
37/37 [==============================] - 9s 115ms/step - loss: 0.0114
Day:  20
38/38 [==============================] - 5s 135ms/step - loss: 5.3308e-04
Day:  40
38/38 [==============================] - 5s 133ms/step - loss: 4.4114e-04
Day:  60
39/39 [==============================] - 6s 143ms/step - loss: 4.3765e-04
Day:  80
39/39 [==============================] - 5s 121ms/step - loss: 4.2353e-04
Day:  100
39/39 [==============================] - 5s 128ms/step - loss: 4.1113e-04
Day:  120
40/40 [==============================] - 6s 139ms/step - loss: 3.8522e-04
Day:  140
40/40 [==============================] - 4s 110ms/step - loss: 4.0709e-04
Day:  160
40/40 [==============================] - 6s 149ms/step - loss: 3.6203e-04
Day:  180
41/41 [==============================] - 5s 123ms/step - loss: 3.4746e-04
```

Figure 8: One epoch every month (20 days) – its decreasing MSE is in the right column

The best way to handle this within the scope of the project would be to train all the models heavily up-front and only provide a few epochs to learn from new data. I set the models to train on 20 epochs and update on 3, and my evaluation suggested it was the right move. I saw an occasional dip in accuracy and suspected it would still roughly have an MSE value of 3e-04, even after weeks of neglect. To ensure this, the MSE of the models will be routinely checked during the maintenance phase. Here is a sample of the MSE values after epoch 20 and 3 updates for three days afterward:

```
Epoch 20/20
40/40 [==============================] - 5s 120ms/step - loss: 2.7427e-04
Model is trained!
Day:  2555
Epoch 1/3
40/40 [==============================] - 5s 119ms/step - loss: 2.8225e-04
Epoch 2/3
40/40 [==============================] - 6s 151ms/step - loss: 2.6378e-04
Epoch 3/3
40/40 [==============================] - 5s 119ms/step - loss: 2.9094e-04
Day:  2556
Epoch 1/3
40/40 [==============================] - 6s 155ms/step - loss: 2.7517e-04
Epoch 2/3
40/40 [==============================] - 5s 120ms/step - loss: 3.1594e-04
Epoch 3/3
40/40 [==============================] - 6s 159ms/step - loss: 2.7803e-04
Day:  2557
Epoch 1/3
40/40 [==============================] - 5s 117ms/step - loss: 2.9041e-04
Epoch 2/3
40/40 [==============================] - 6s 150ms/step - loss: 2.5269e-04
Epoch 3/3
40/40 [==============================] - 5s 126ms/step - loss: 2.4669e-04
```

Figure 9: Testing the effect of a single epoch per close price, starting with 20 epochs

Visualizations

While the web app generates two images in real time, all three types of visualizations are accessible from the Jupyter Notebooks link. Here is an explanation of the three types of images seen in the notebook:

1. Model Nodes: This image shows the different nodes of the LSTM over the values of the time sequence. These nodes are responsible for capturing the behavior of the stock with the layers provided: 1, 5, 75.

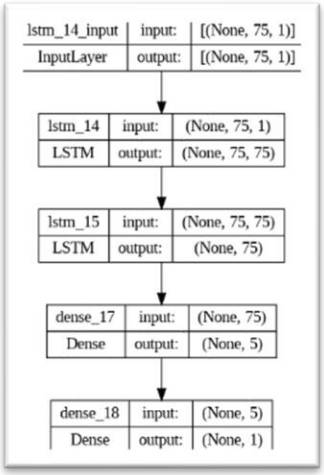

Figure 10: A node diagram of my final LSTM model

2. Model Overlay: This type of image shows how the model's predictions compare to real-world data. After training, I had the model predict all the stock prices after the first 75 days. I then overlayed its prediction (orange) over the actual closing stock prices (blue). This is a simple way to see how well the model mirrors the data.

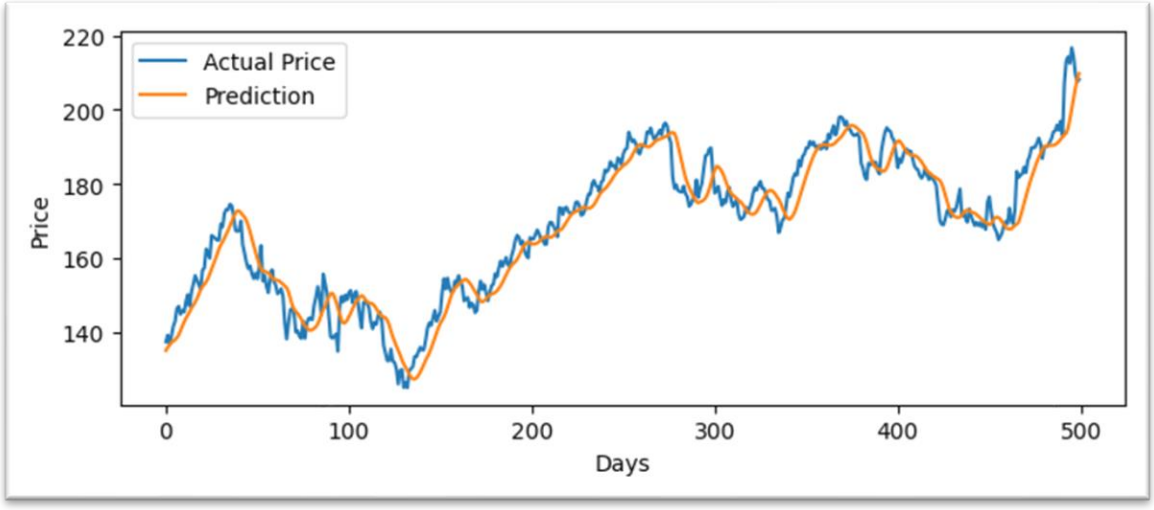

Figure 11: A sample of a Model Overlay for META stock – 6/27/24

3. Future Prediction: This image shows the most recent stock data, followed by the model's prediction for future prices.

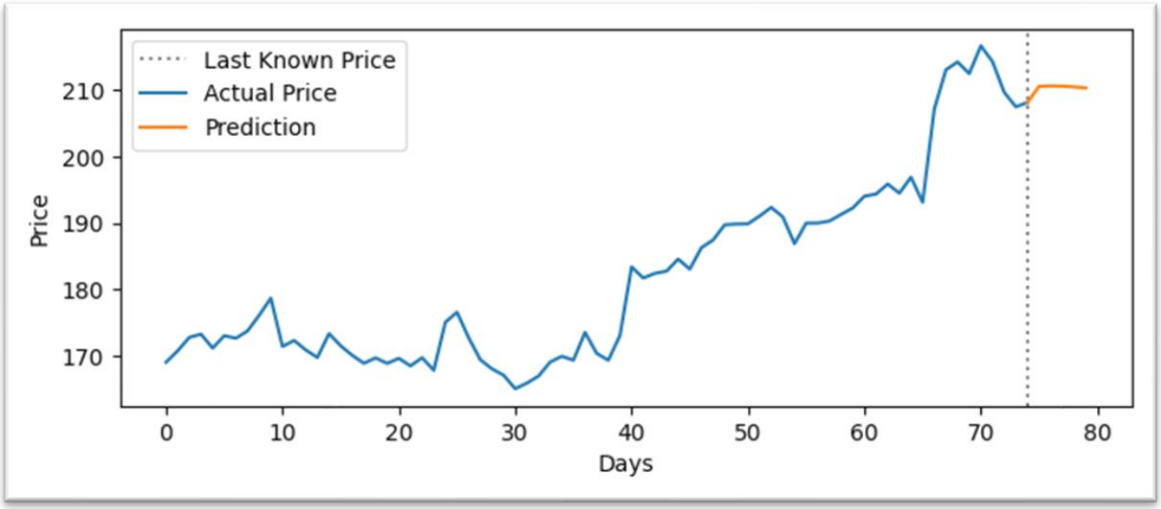

Figure 12: A sample of a Future Prediction for META stock – 6/27/24

While the screenshot above is nearly a flat line, the model's predictions vary greatly depending on the overall context of the stock. The next page shows a screenshot of the web app, and it is an understandably bumpy prediction of Amazon's stock price after its record high.

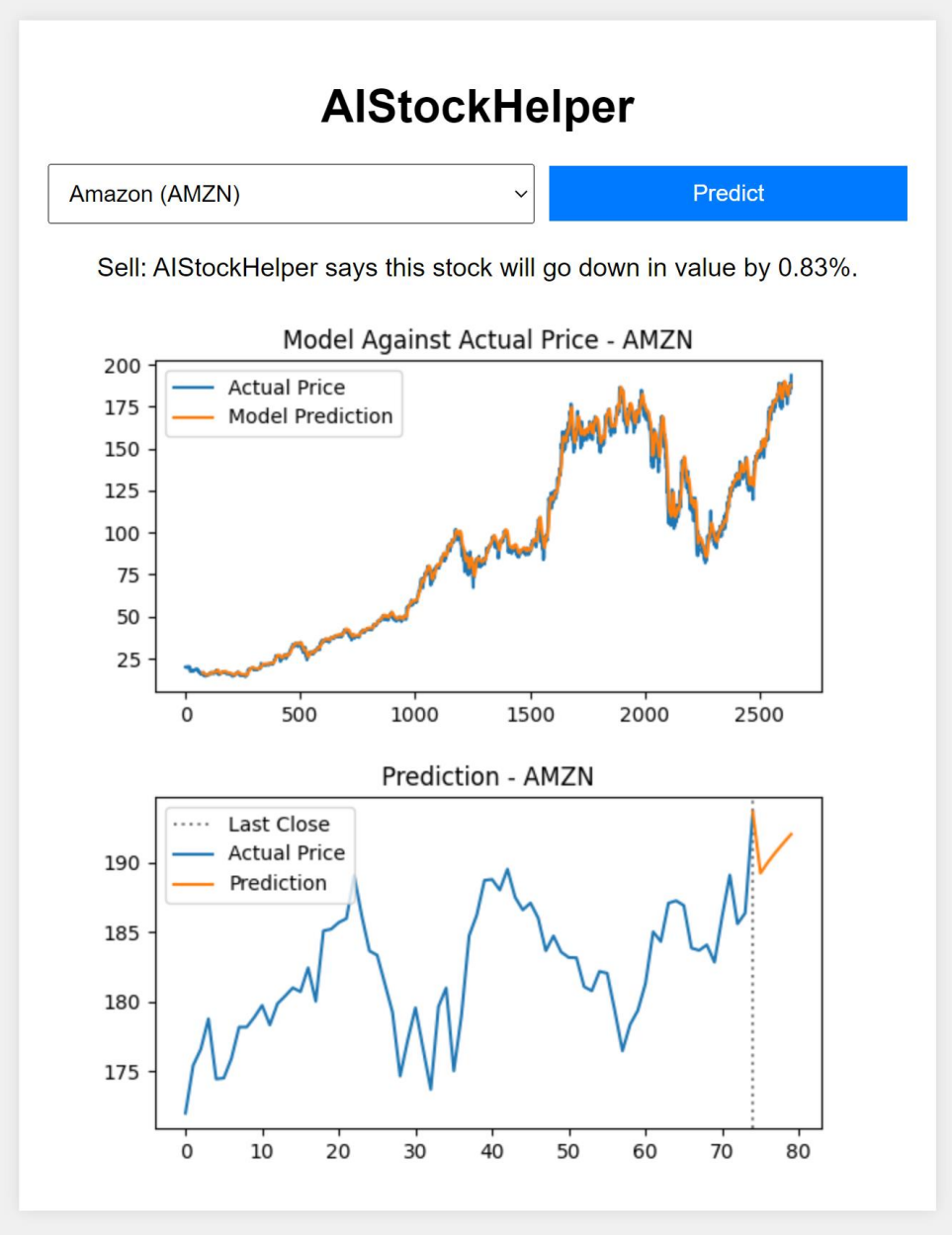

Figure 13: A screenshot of the web app predicting ticker – 'AMZN,' 6/27/2024

When I developed the web app, I strayed from the initial design to add graphs to give the user additional context. There are two sections of the application – the form and the result. The form has the title, ticker, and button, while the result gives a text recommendation, Model Overlay, and Future Prediction images.

User Guide

1. Access AIStockHelper via the Web App Link
2. Input a stock ticker and select ‘Predict.’
    a. Please note that a time delay of 5 seconds is typical. However, if the chosen ticker has not been run since the last close, the time delay increases to 20 seconds while the models are retrained. Retraining only happens once per ticker per 24 hours, with the cutoff time at market close – 4 pm EST.
3. The program outputs a Buy/Sell recommendation, an explanation of its recommendation, and a newly generated Model Overlay and Model Prediction visualizations.