Table of Contents

Abstract **2**

Background **2**

Objectives **3**

Database Management **3**

Database Structure 3

The Django ORM 5

Performance 6

Monte Carlo Simulations **9**

Conclusions **12**

References **13**

Abstract

The purpose of this paper is to detail the planning, construction, management and deployment of a SQL database in a web application and the usage of APIs to create a useful website. To do this a MySQL database was deployed alongside a Django web server allowing queries to be written in a compact and object-oriented way. The database was developed by eliciting requirements and decomposing use-cases into objects, then creating an entity-relationship (ER) Diagram. The table definitions for the application's database were then derived from the ER diagram. Performance of queries on these tables is then evaluated using a tool included in the Django framework to demonstrate the optimizations being performed by the ORM. Finally, this report gives an overview of the statistical analysis being performed on historical stock data retrieved from the Yahoo Finance API [1].

GitHub repository: https://github.com/graemsheppard/MonteCarlo

Background

The foundation of this project is built off the mathematics of statistics and the role it plays when it comes to risk analysis of future stock prices. In order to provide the end user with a means of forecasting, the web application utilizes the Monte Carlo method in order to simulate the price volatility of future prices. This numerical algorithm can be used to solve a large number of problems by implementing repeated random sampling in order to generate results. The way in which these simulations are generated for the end user will be outlined in a later section. When it comes to other work related, there are a number of software tools geared towards enterprise environments, such as Oracle's Crystal Ball [2] and PALISADE's @RISK [3], that also has the ability to utilize the Monte Carlo method. Because of being aimed at this sort of audience, the prices associated with using the software can be fairly steep for a regular consumer.

The development stack that was chosen for this web application includes these main tools:

- Python: chosen for rapid development.
- Django: chosen for the MVC paradigm.
- MySQL: chosen for security and reliability.
- Bootstrap: chosen for responsive user interface, as well as fast layout implementation.
- jQuery: chosen for dynamic front-end content and AJAX calls.

With this stack, the MVC (Model-View-Controller) architecture was chosen due to its high modularity and scalability.

Objectives

This MonteCarlo web app was created with two objectives in mind. The first being that this should be an app usable by the average person. There are other stock market tools that can do very detailed and complex analysis on stocks that assist already knowledgeable people with their day trading. This app is meant to be used by people who do not do extensive trading and are not familiar with the deep intricacies of the stock market. To do this, only the essential and easy to understand information is used and displayed on the site to make the app intuitive to use. The intention is not to provide the end user with financial advice, but to provide estimations given by the model governing the Monte Carlo simulations.

The second objective is that the site should be free to use. The other high end trading tools can be very costly. The target audience of an app like this, who are likely interested in long term saving, would not be willing to spend a large sum of money on a trading app. Therefore the site should be able to run at a minimal cost in order to provide a free service to the users.

Database Management

Database Structure

The database used for this project consists of 6 MySQL tables. These tables are: comment, trader, stock, message, trader_favourite, trader_message. The tables contain the following data:

trader

id	user_id

comment

id	text	posted_on	about_id	posted_by_id

stock

id	name	ticker	popularity

message

id	text	date	sender_id

trader_favorites

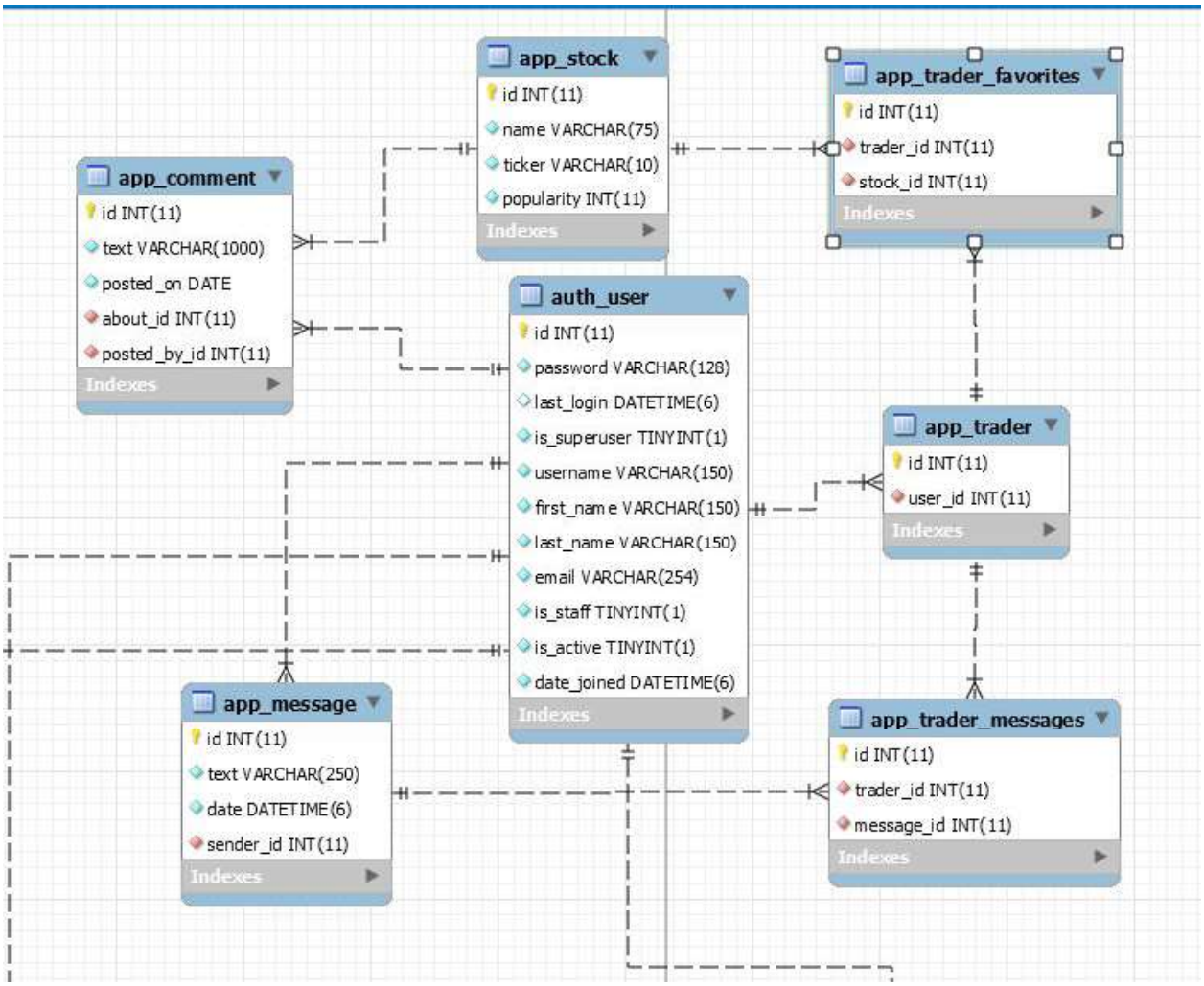
id	trader_id	stock_id

trader_messages

id	trader_id	message_id

The trader table contains information about each user who uses the site. The stock table contains data about every stock that is visited on the website. The comment table contains all comments posted under each stock page while the message table contains the messages sent between users. Trader_favorites and trader_message are many to many relationships that contain each user's favourite stocks and each user's messages respectively.

In order to create various views and complex queries, the tables use foriegn keys to execute joins such as displaying the comments that a user has posted. The following ER diagram shows how the tables in the database are connected.



In the diagram above, the auth_user table is created by Django to keep track of all users who use the site. This table can provide useful information such as login information, check if the user is logged in among other things. The relationships between the manually created tables are also shown.

The Django ORM

Django's ORM (Object-Relational Mapper) makes querying the database much easier and it eliminates the need to write raw SQL queries in the code. Another benefit is that the syntax is the same for all supported databases such as MySQL, PostgreSQL, and SQLite. The way this works is by defining the model for a table in the same way as a class. When the model is defined, characteristics are set for each attribute to specify the datatype, whether it is a primary key or foreign key. The ORM also allows the programmer to define relationships such as one-to-one and many-to-many on an attribute. An example of a Django model used in the code can be seen below:

```
class Comment (models.Model):
      text = models.CharField(max_length=1000)
      posted_on = models.DateField(auto_now_add=True)
      posted_by = ForeignKey(User, on_delete=models.CASCADE)
      about = ForeignKey(Stock, on_delete=models.CASCADE)
```

Once the model has been created the programmer can execute the command 'makemigrations' which looks for changes in the models and then 'migrate' to translate the models that have changed into SQL create or alter statements. A powerful feature is that if a relationship is specified, for example a *user* having a favourite_stocks field and a model for *stock* has been defined, then Django will automatically generate the create statement to generate the table that relates each *user* to a favourite *stock* called *user_favourite_stocks*.

In many instances, a database might already be in place and the programmer only wants to use the models for the ease of use and object-oriented aspects. In this case the programmer can connect the program to the database and write the models to describe the existing tables, or they can run the command 'inspectdb' which will output the code for the models needed.

In addition to making the creation and structuring of the database easier, the main advantage of Django's ORM is its querying capabilities. Queries are invoked as methods of the model for the table that is being queried, this makes for a much more readable codebase and allows chaining of methods. Another benefit is that joins can be done implicitly. Using the favourite_stocks example mentioned previously, to find a user's favourite stocks the programmer could simply write:

```
User.objects.get(username="test").favourite_stocks
```

Which is equivalent to the following SQL query:

```
SELECT name, ticker
FROM user AS U
JOIN user_favourite_stocks AS F
      ON U.id = F.uid
```

```
JOIN stocks AS S
      ON F.sid = S.id
WHERE U.username = “test”
```

This query will return an array of all of the favourite stocks of the user named “test.” There are many more useful methods that can be used such as filter, order_by, and aggregate functions to shape the data however it is required. Because of the capabilities of the ORM it ended up being a great tool for all of the database operations required for the Monte Carlo stock simulation website.

Performance

Throughout the development of the MonteCarlo web application, it was imperative to the team to maintain stable performance of the application. This meant that minimization of database hits and efficient queries were of paramount concern to the team. Using an abstraction such as the ORM perhaps made the development process in general more efficient, but we still had to ensure the quality of the queries used to render the various user views. We had considered two options and both were weighted against each other. These include:

1. Writing the SQL queries ourselves and using the ORM simply as an interface to the backend.
2. Learn the best practices of the Django Queries API and rely on native tools to monitor performance.

The former made sense due to the fact that all the members in the team were already well versed with writing raw SQL queries and most of the views had already been implemented during phases 1 and 2 of development. In hindsight, this option may have been useful, but we had decided to forego it due to several reasons, one of which included the programming of additional modules to provide interfaces with the local database. This was a significant challenge as we were developing the application in our local machines respectively. An obvious consequence of this fact was that there were various host operating systems used by the individual developers. In sum, there was a challenge in providing a common environment unless a server was set up - which comes with its own costs. Therefore, the latter choice seemed more suited even though we did lose a layer of control by letting django handle the queries. Upon later inspection, we did find that this choice was accurate.

The Django Debug Toolbar:

The factors that we had identified early on that would contribute towards a reduction in performance were mitigated to a large extent by the use of this tool. It is simply a sidebar that appears globally throughout the site when the debug flag is set to true. This ensures that it is only visible to the developers who are actively working on the site, and not the end user. This toolkit extended our ability to maintain the required threshold of performance in the following ways:

1. It logged the time required to fetch various parts of the website individually on to a client machine.

2. It logged the list of all database queries that are required to render a specific page of the website.
3. It monitored the said list sequentially, logging the specific characteristics of the queries such as the actual sql command associated with it as well as the time it took for the database to return its results.

This allowed us to assess our own performance quite accurately, and draw certain conclusions that we shall further elaborate on.

Observations:

a. Which page took the longest to render?
b. What are the specific characteristics of this page that caused it to render slowly?
c. Were we efficient in our use of the Django ORM?
d. What are the challenges faced in mitigating performance loss?

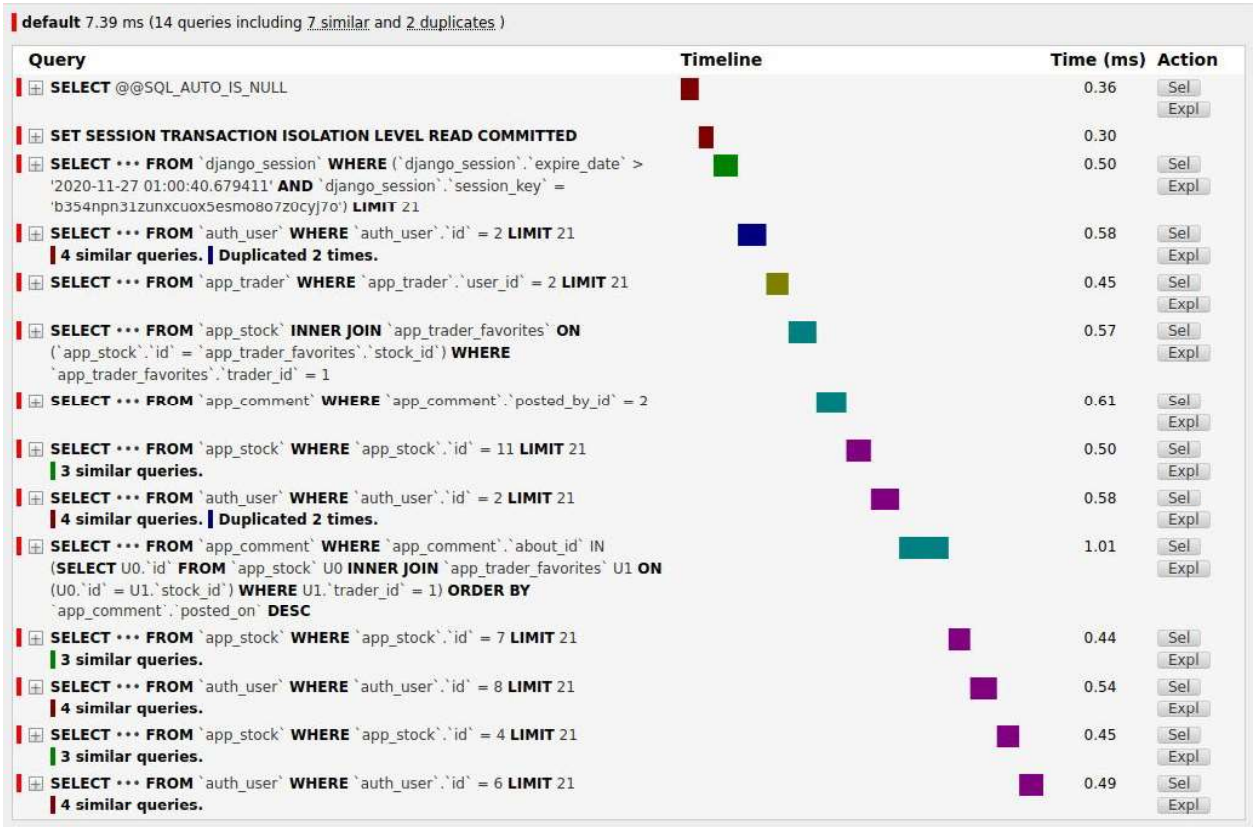
It came as no surprise that the page that took longest to render was the one that fetched data from the Yahoo API. This is the only part of the backend that is not handled entirely locally and such a result is to be expected. An evidence of this is shown below.

Resource	Value
User CPU time	216.300 msec
System CPU time	40.129 msec
Total CPU time	256.429 msec
Elapsed time	3021.781 msec
Context switches	30 voluntary, 15 involuntary

Perhaps a more interesting result is that among all the other pages, the ones that are handled locally, it is the Favorites page that took the longest time to render with an average delay of 384.484 msec. But this is not all, since we can further investigate as to why this is the case. The following is the list of all queries that are required to render the favorites page.

Upon first glance, it is shown that there exist 7 similar and 2 duplicate queries. The team was quite baffled at this result. To display the list of the user's favorite stocks, the ORM made separate calls to retrieve the corresponding stock objects. We had previously expected this to be done in one single unified query as we had not used constructs like loops in order to retrieve this data as can be seen in the following code snippet.

```
80  def favourites (request):
81      """
82      """
83
84      if request.user.is_authenticated:
85          comments = Comment.objects.filter(posted_by = request.user)
86          favInfo = request.user.trader.favorites.order_by('id')
87          discussions = Comment.objects.filter(about__in=favInfo).order_by('-posted_on')
88          print(discussions)
89          return render(request, 'app/favourites.html', {'content': favInfo, 'comments': comments, 'discussions': discussions})
90      return render(request, 'app/favourites.html')
```



Our first conclusion was that these queries are handled internally as separate and are later appended together with the SQL Union command. Perhaps this is how queries of such nature are handled by SQL itself. We were unable to verify this as it is too closely related to the inner workings of the specific SQL implementations, of which we were all running different ones. This was far from the case, however.

What we did not realize was that we were in fact creating new queries by using the template to render the name of the stocks that the user has commented on. And that these queries were contained within the loop that was generating the frontend content which displays the user's recent comments. This is illustrated by the following code snippet:

```
    <article class="media content-section">
        <div class="media-body">
            <div class="article-metadata">
                <h3 class="d-inline-block"><a class="article-title" href="#">{{ data.about.name }}</a></h3>
                <br>
                <small>{{ data.posted_by }}</small>
                <small class="text-muted">posted {{ data.posted_on }}</small>
            </div>
```

/faze/MonteCarlo/src/django/app/templates/app/favourites.html

Proposed Improvements:

Upon identifying the culprit, we could easily fix this by passing the individual stock names as a separate JSON to the frontend. And this is the fix that was chosen.

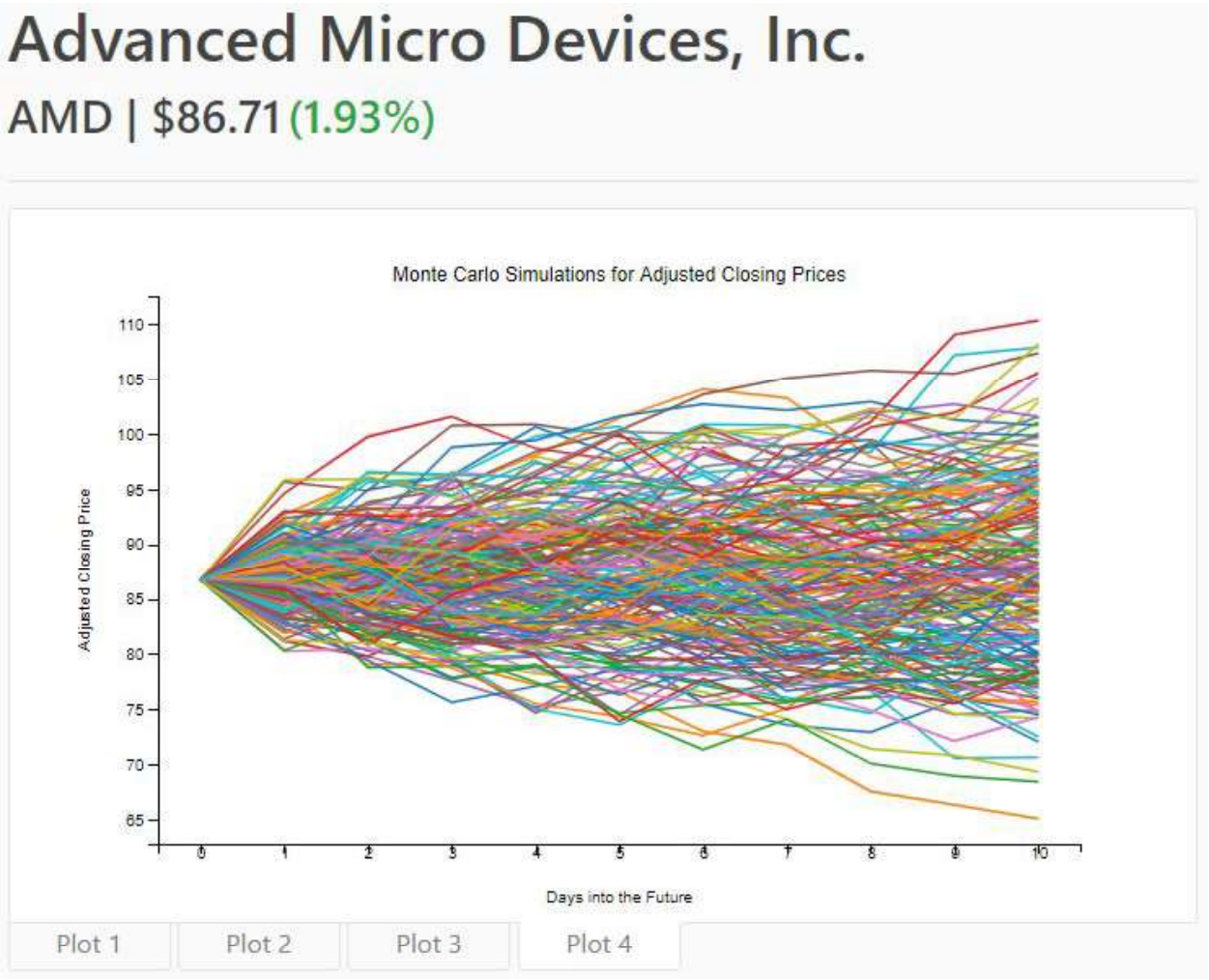
Monte Carlo Simulations

The main feature of the web application is to provide a tool for conducting risk analysis when it comes to predicting a given stock's adjusted closing price at some point in the future. These predictions are made by implementing the Monte Carlo numerical method for simulating a future price with an associated level of uncertainty. The way in which this feature is implemented is through a number of python libraries used to help with the statistical analysis necessary to generate results: numpy, pandas, pandas-datareader, scipy, matplotlib, and mpld3.

When the user submits the form on the simulation page of the web application, the user's input parameters are sent to the server and used to instantiate a Monte object which contains the attributes and methods necessary to do the simulation and plotting. The dates chosen by the user are used to pull the historical data of the adjusted closing price of a given stock. These parameters are used to interact with the Yahoo Finance API and stored in a DataFrame data structure supplied by the pandas library. This data structure is a labelled 2D array. With the help of numpy and scipy, the logarithmic returns, mean, variance, standard deviation, and stochastic drift values are calculated from the historical data. With these values, the simulation calculation follows this formula:

NextPrice = CurrentPrice * e^(Drift + Sigma*RandomVariable)

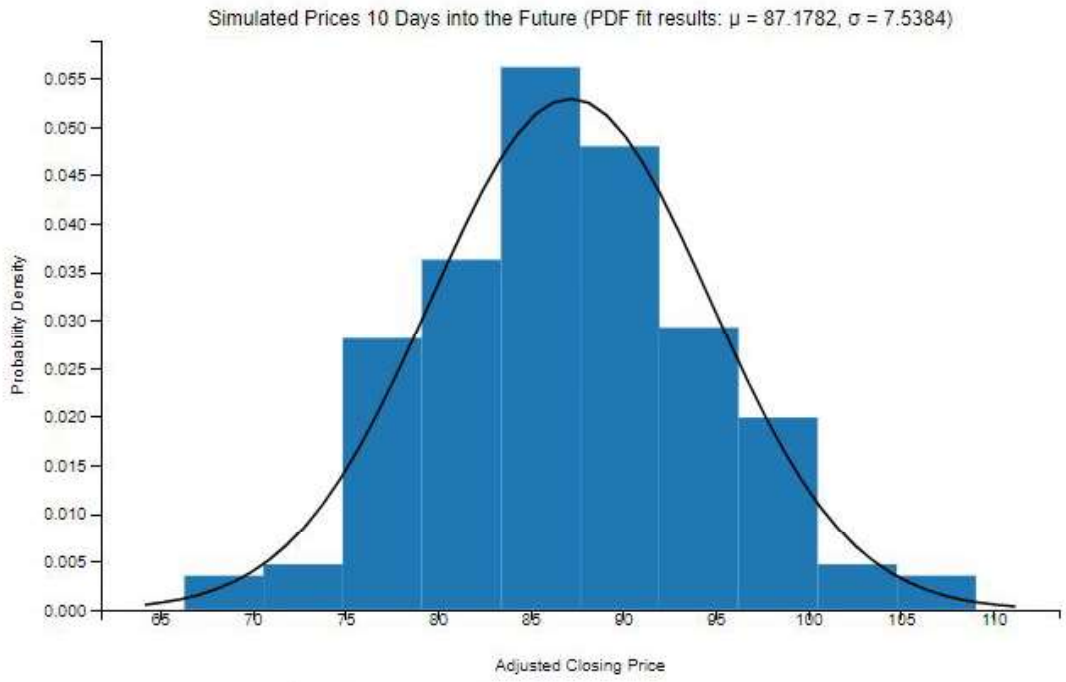
Where the current price is the most recent data point in the historical data. With the stochastic drift and standard deviation values already calculated, the random variable is pulled from a normal probability distribution. These simulated prices are stored in their own array as this equation generates all the outputs at each interval (time_step = 1 day) given by the user. As an example, if the user asks to generate 200 simulations and wants a forecast for 10 days into the future, at the first time step interval 200 prices are generated, stored, and indexed for 1 day into the future. Using those newly simulated prices, the next time step interval continues on with the simulation and future price calculations. At each of the 10 days there are 200 simulated prices used to create the plot shown below using the matplotlib and mpld3 libraries.



At day 10, a histogram of the prices is plotted and a normal distribution is fit to the data. This is shown in the plot below with the associated mean and standard deviation to showcase the uncertainty in the simulated price prediction governed by the model. The simulated price can be compared to the current price shown underneath the stock page title.

Advanced Micro Devices, Inc.

AMD | $86.71 (1.93%)



Plot 1 | Plot 2 | Plot 3 | Plot 4

Conclusions

The aim of this website is to provide a free and easy to use tool for stock risk analysis. The use of a monte carlo simulation displayed in four different graphs gives a very visual indication of the possible trends a stock can take. The simulation does not require any math or difficult inputs for the user to use. They can simply enter in the number of simulations they want and the data is gathered for them. This design makes it so that anybody who is not experienced in trading can still use this platform as a means of estimation. The web application itself does not provide any financial advice.

Improvement that could be made regarding the simulation itself could be to have a different model governing the simulated prices. As well, having a different probability distribution to fit the mean price forecast might provide a more thorough analysis. In terms of the data source for the historical stock prices, by fetching from more than just Yahoo Finance, a more accurate simulation can be done. If the method of prediction were to be attempted again, machine learning tools such as deep neural networks might be more suited for this type of problem. On the other hand improvements that could be made to the database and application as a whole were recognized and thoroughly documented within the performance section of the report.

The challenges faced by the team throughout the development process were also outlined along the pages of this report, making it a comprehensive overview of the work that was done over the semester. Undoubtedly, this was an invaluable learning experience in the avenue of data management systems as well in the art of working together as a team.

References

[1] "Yahoo Finance" *Yahoo.* [Online]. Available: https://ca.finance.yahoo.com/. [Accessed: 26-Nov-2020].

[2] "Oracle Crystal Ball" *Oracle.* [Online]. Available: https://www.oracle.com/ca-en/applications/crystalball/. [Accessed: 14-Oct-2020].

[3] "@RISK" *PALISADE.* [Online]. Available: https://www.palisade.com/risk/. [Accessed: 14-Oct-2020].

[4] "How to Embed Interactive Python Visualizations on Your Website with Python and Matplotlib" *Nick McCullum*. [Online]. Available: https://www.freecodecamp.org/news/how-to-embed-interactive-python-visualizations-on-your-website-with-python-and-matplotlib/. [Accessed: 26-Nov-2020].

[4] "US Stock Market Stock Forecast," *Walletinvestor.com*. [Online]. Available: https://walletinvestor.com/stock-forecast. [Accessed: 14-Oct-2020].

[5] "Singleton in C#," *Refactoring Guru*. [Online]. Available: https://refactoring.guru/design-patterns/singleton/csharp/example. [Accessed: 14-Oct-2020].