Virginia Polytechnic Institute and State University

CS 4624

Multimedia/Hypertext/Information Access

# Analyzing Microblog Feeds to Trade Stocks

Project Documentation

*Authors:*
Joseph Watts
Nick Anderson
Connor Asbill
Joseph Mehr

*Supervisor:*
Dr. Edward Fox

*Client:*
Saurabh Chakravarty

May 10, 2017

Blacksburg, VA 24061

# Contents

**List of Figures** **3**

**List of Tables** **4**

**1 Executive Summary** **6**

**2 Introduction** **8**

**3 User's Manual** **9**
- 3.1 Usage Environment . . . . . 9
- 3.2 Use Cases . . . . . 9
  - 3.2.1 Populating Databases . . . . . 9
  - 3.2.2 Stock Opinion Aggregation . . . . . 10
  - 3.2.3 Backtesting Trading Strategies . . . . . 10

**4 Developer's Manual** **11**
- 4.1 Preparing a Development Environment . . . . . 11
- 4.2 Project Structure . . . . . 11
- 4.3 Defining New Trading Strategies . . . . . 12
- 4.4 Defining Sources of Data . . . . . 12

**5 Lessons Learned** **13**
- 5.1 Problems Encountered . . . . . 13
- 5.2 Future Work . . . . . 13
  - 5.2.1 Improving Trading Simulation Software . . . . . 13
  - 5.2.2 Advanced Trading Strategies . . . . . 14

**6 Acknowledgements** **15**

**References** **16**

**Appendices** **18**

**A Requirements** **18**
- A.1 Who Will Be Served . . . . . 18
- A.2 Scenarios Served . . . . . 18

A.3 Data To Be Processed . . . . . . . . . . . . . . . . . . . . . . . . 19
A.4 Existing Code Base . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**B Design** **21**
B.1 System Architecture . . . . . . . . . . . . . . . . . . . . . . . . . 21
B.1.1 Pricing Data . . . . . . . . . . . . . . . . . . . . . . . . . . 21
B.1.2 Opinion Aggregation . . . . . . . . . . . . . . . . . . . . . . 21
B.1.3 Trading Simulation . . . . . . . . . . . . . . . . . . . . . . . 22
B.2 Technology Used . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
B.2.1 Pricing Data . . . . . . . . . . . . . . . . . . . . . . . . . . 23
B.2.2 Opinion Aggregation . . . . . . . . . . . . . . . . . . . . . . 23
B.2.3 Trading Simulation . . . . . . . . . . . . . . . . . . . . . . . 23
B.3 Schema/Data Structures . . . . . . . . . . . . . . . . . . . . . . . 24
B.3.1 Pricing Data . . . . . . . . . . . . . . . . . . . . . . . . . . 24
B.3.2 Opinion Aggregation . . . . . . . . . . . . . . . . . . . . . . 24
B.3.3 Trading Simulation . . . . . . . . . . . . . . . . . . . . . . . 26
B.4 Data Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
B.4.1 Pricing Data . . . . . . . . . . . . . . . . . . . . . . . . . . 27
B.4.2 Opinion Aggregation . . . . . . . . . . . . . . . . . . . . . . 28
B.4.3 Trading Simulation . . . . . . . . . . . . . . . . . . . . . . . 28
B.5 Group Roles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
B.6 Timeline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**C Implementation** **31**
C.1 Pricing Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
C.1.1 Setting Up Database Schema . . . . . . . . . . . . . . . . . 31
C.1.2 Database Interactions . . . . . . . . . . . . . . . . . . . . . 31
C.1.3 External Sources . . . . . . . . . . . . . . . . . . . . . . . . 32
C.2 Opinion Aggregation . . . . . . . . . . . . . . . . . . . . . . . . . 33
C.2.1 Setting Up Database Schema . . . . . . . . . . . . . . . . . 34
C.2.2 Database Interactions . . . . . . . . . . . . . . . . . . . . . 34
C.2.3 Sentiment Analysis . . . . . . . . . . . . . . . . . . . . . . . 34
C.2.4 Aggregation Model . . . . . . . . . . . . . . . . . . . . . . . 35
C.3 Trading Simulation . . . . . . . . . . . . . . . . . . . . . . . . . . 35
C.3.1 Virtual Portfolio . . . . . . . . . . . . . . . . . . . . . . . . 36
C.3.2 Trading Context . . . . . . . . . . . . . . . . . . . . . . . . 36
C.3.3 Trading Strategies . . . . . . . . . . . . . . . . . . . . . . . 37

**D Prototyping** **39**
D.1 Program Entry Point . . . . . . . . . . . . . . . . . . . . . . . . . 39
D.1.1 Specifying the Simulation Time Range . . . . . . . . . 39
D.1.2 Customizing the Trading Events . . . . . . . . . . . . . 40
D.1.3 Customizing the Trading Strategies . . . . . . . . . . . 40
D.2 Running a Simulation . . . . . . . . . . . . . . . . . . . . . . . . . 40
D.3 Interpreting the Output . . . . . . . . . . . . . . . . . . . . . . . . 41

**E Refinement** **42**
E.1 Stock Split Consideration . . . . . . . . . . . . . . . . . . . . . . 42
E.2 Moving Average Strategy . . . . . . . . . . . . . . . . . . . . . . . 42
E.3 Moving Average with Sentiment Strategy . . . . . . . . . . . . 43
E.4 Selection by Sentiment Strategy . . . . . . . . . . . . . . . . . . 43

**F Testing** **45**
F.1 Trading Simulation Software . . . . . . . . . . . . . . . . . . . . 45
F.2 Trading Strategy Performance . . . . . . . . . . . . . . . . . . . 45
F.2.1 Results for 2014 . . . . . . . . . . . . . . . . . . . . . . . 46
F.2.2 Results for 2015 . . . . . . . . . . . . . . . . . . . . . . . 48
F.3 Additional Refinements . . . . . . . . . . . . . . . . . . . . . . . . 49

# List of Figures

1 Example microblog post from StockTwits . . . . . . . . . . . . 20
2 The stock price data structure as a Scala case class (left) mapped onto its corresponding database schema (right). . . . 25
3 The `MicroblogPost` and `MicroblogAuthor` data structures (left) mapped onto the corresponding database schema (right). 26
4 The virtual portfolio data structure. . . . . . . . . . . . . . . . . 27
5 The `StockPriceDataSource` trait. . . . . . . . . . . . . . . . . . 32
6 The `EndOfDayStockQuote` data structure. . . . . . . . . . . . 33
7 The `MicroblogDataSource` data structure. . . . . . . . . . . . 34
8 The `TradingStrategy` trait. . . . . . . . . . . . . . . . . . . . . 37
9 The `TradingEvent` trait and the `TradingEventEmitter` trait. 38
10 A `TradingEvent` signifying a new microblog post was made. . 38
11 Comparison of the Baseline Strategy and S&P 500 Index Portfolio Values for the year of 2014. . . . . . . . . . . . . . . . . . 41
12 A graph of the portfolio values for the year of 2014 for several trading strategies. . . . . . . . . . . . . . . . . . . . . . . . . . 46
13 A graph of the portfolio values for the year of 2015 for several trading strategies. . . . . . . . . . . . . . . . . . . . . . . . . . 48

## List of Tables

1 Baseline Stocks . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

## 1 Executive Summary

The goal of this project is to leverage microblogging data about the stock market to predict price trends and execute trades based on these predictions. Predicting the price trends of stocks with microblogging data involves a complex opinion aggregation model. For this, we built upon previous research, specifically a paper called "CrowdIQ" submitted by a team consisting of some Virginia Tech faculty [2]. This paper details a complicated method of aggregating an accurate opinion by modeling judge reliability and interdependence. Once the overall sentiment of the judges was deduced, we built trading strategies that take this information into account to execute trades.

The first step of the project was a sentiment analysis of posts on a microblogging site named StockTwits [22]. These messages can contain a label indicating a bullish or bearish sentiment, which will help indicate a specific position to take on a given stock. However, most users choose not to use these labels on their StockTwits [2]. A classification of these unlabeled tweets is required to autonomously utilize StockTwits to drive the proposed trading strategies.

With a working sentiment analysis model, we implemented the opinion aggregation model described by *CrowdIQ* [2]. This can gauge an accurate market sentiment for a particular stock based on the collection of sentiments that are received from users on StockTwits.

The next step was the creation of a trading simulation platform, including a complete virtual portfolio management system and an API for retrieving historical and current stock data. These tools allow us to run quick and repeatable tests of our trading strategies on historical data. We can easily compare the performance of strategies by running them with the same historical data.

After we had a viable testing environment setup, we implemented trading strategies. This required research and analysis of other attempts at similar uses of microblogging data on predicting stock returns. The testing environment was focused on a set of stocks that is consistent with those used in *CrowdIQ*. The implementation of the *CrowdIQ* strategy served as a baseline against which we compared our results.

Development of new trading strategies is an open-ended task that involved a process of trial and error. It is possible for a strategy to find success in 2014, but not perform quite as well in other years, because market climates can be fickle. To assess the dependence of the market climate on our strategy's success, we also tested against data for the year of 2015 and compared the performance.

The final deliverable is a viable trading simulation environment coupled with various trading strategies and an analysis of their performance in the years of 2014 and 2015. The analysis of each strategy's performance indicated that our sentiment-based strategies perform better than the index in bullish markets like that of 2014, but, when they encounter a bear market, they typically make poor trading decisions which result in a loss of value.

## 2 Introduction

There have been numerous instances in the past where tweets have been used to create short term trading positions on specific stocks. On September 23, 2016, a tweet reported speculation on a possible takeover of Twitter by Google. The tweet was sent at around 9 AM, and, by 11 AM, the Twitter stock had gained about 20% on its previous day closing price [20]. We want to build a bot that can analyze a given set of people who tweet about stock trading. By analyzing their historical predictions and matching them with current data, we would filter the junk analysts and only follow the ones who have good accuracy in their predictions. For those tweeters, we would follow the tweets in real-time and employ sophisticated machine learning algorithms in text classification to analyze stock sentiments. Based on this analysis, we will develop, test, and compare multiple trading strategies. We would want to test the strategies with historical data from multiple intervals to determine how the strategies perform in different market climates. With analysis of these tests, we will determine particular strengths and weaknesses of particular strategies and which strategies perform the best.

In the *User's Manual*, we describe how to use our software deliverables to execute tests of your own. Next, in the *Developer's Manual*, we describe our development environment and how to get started with defining new trading strategies and data sources in our simulation software. Finally, in *Lessons Learned*, we reflect on our experiences in the development of this project and propose future work which can expand upon our work.

# 3 User's Manual

In this section, we describe the usage of our trading simulation software, including setting up databases, using the opinion aggregation implementation, and executing backtests for trading strategies.

## 3.1 Usage Environment

Our software does not come with a graphical user interface or even a single command line entrypoint. Our code is written with Scala [14] and uses sbt [12] to manage project structure and dependencies. Additionally, our system depends upon an HBase [5] instance or cluster. We have been provided a Hadoop cluster that runs HBase by Virginia Tech, so this manual will assume that you have an HBase instance set up and that you have sbt installed. Please consult an external reference such as *Deploying HBase on a Cluster* by Cloudera [1] if you need any assistance setting this up. If you are affiliated with Virginia Tech and are interested in the DLRL Hadoop cluster that we used, please contact Prof. Edward Fox for more information.

## 3.2 Use Cases

### 3.2.1 Populating Databases

Before you can run any simulations, you must populate HBase with stock prices for the date range of interest. Additionally, if you want to use our sentiment-based strategies, you must populate HBase with microblog posts. To populate HBase with stock prices, you can use our `YahooGoogleFinanceDBPopulate` utility, which will query Yahoo Finance and Google Finance for daily stock price summaries on a given date range and populate the database with the results. There are two variables, `startDate` and `endDate`, which define the date range of daily stock prices to query. You should avoid setting a range that spans more than one year due to the limitations of the respective APIs. You can configure the symbols to query from Yahoo Finance and Google Finance by modifying the two lists named `yahooSymbols` and `googleSymbols`. The `dataSource` variable specifies that the `YahooFinance` table should be used. Therefore, you should create the HBase table named `stockprices_yahoo` with the `price` column family (by executing `create 'stockprices_yahoo', 'price'` in the HBase shell). For more information, see the Design report in Appendix B. To run the utility, use the command `sbt ‘‘PricingData/run-main cs4624.prices.test.YahooGoogleFinanceDBPopulate’’`.

### 3.2.2 Stock Opinion Aggregation

We implemented the opinion aggregation model from a paper called *CrowdIQ* [2]. A library to use our implementation is provided with the Opinion Aggregation subproject. The `AggregatedOpinions` class provides an interface that will compute the aggregated sentiment towards a certain stock. This class requires that you have a source of stock price data. Additionally, you must specify a window of time which determines the amount of time after a post that we will confirm the accuracy of the post and adjust the weight for the post's author. To use an instance of this class, you pass the microblog posts in the interval (i.e., between market open events) to the `on` method. Using the `sentimentForStock` method, you can get the aggregated sentiment (bullish, bearish, or inconclusive) towards a stock symbol. After you've gotten the sentiments for that date, you use the `reset` method to start a new interval of posts and repeat the process.

### 3.2.3 Backtesting Trading Strategies

Our software provides the ability to test trading strategies by replaying historical data (a "backtest"). Trading strategies are defined as extensions of the `TradingStrategy` trait. All data that a trading strategy can use to make decisions is formed into a class that extends the `TradingEvent` trait. In a backtest, these events are collected for a particular time interval and passed in-order to each trading strategy by an instance of the `TradingContext` class. The `TradingContext` class takes the sources of these trading events, the set of trading strategies, and the time interval in its constructor. The `run` method, which will execute the backtest, takes a callback which allows you to see the change in each strategy's portfolio after an event. Our tests are written in the `TradingSimulation.scala` file. To execute this main class, run `sbt ‘‘TradingSimulation/run-maincs4624.trading.TradingSimulation’’`.

# 4 Developer's Manual

In this section, we describe our development environment, the structure of the project's code, and how to define new trading strategies and sources of data in our platform.

## 4.1 Preparing a Development Environment

To develop on our system, you will need to have sbt[12] installed and an HBase[5] instance ready to use. Virginia Tech's DLRL cluster is running the Cloudera 5.6.0 distribution which contains HBase version `1.0.0-cdh5.6.0`. To ensure that there are no incompatibilities, it is best to run this version of HBase. Because our project is using sbt, you should easily be able to import this code into your favorite Scala IDE, or you can simply edit the code in your favorite text editor and compile using the command line interface.

## 4.2 Project Structure

Our root sbt project is broken down into three subprojects: Pricing Data, Opinion Aggregation, and Trading Simulation. The Pricing Data subproject (located within the `PricingData/` folder) contains all of our code related to accumulating information about stock prices and populating it within our database. This includes APIs which query Yahoo and Google Finance, as well as data structures which represent a stock price tick and a daily stock price summary. The Opinion Aggregation subproject (located within the `OpinionAggregation/` folder) contains all of our code related to microblog posts, sentiment analysis, and the CrowdIQ model. This includes data structures to represent microblog posts, an API to read in StockTwits posts from the CSV format provided to us by our client, and an interface which facilitates the use of the CrowdIQ model in forming a crowd-sourced sentiment towards a stock. The Trading Simulation subproject (located within the `TradingSimulation/` folder) contains all of our code related to defining and testing trading strategies. This includes all of the strategies that we have developed and tested (Baseline, Moving Average, Selection by Sentiment, etc) and a program which executes a simulation with all of our strategies, outputting a CSV file with the results. Our project structure is outlined in extensive detail in the Design, Implementation, and Prototyping reports found in Appendixes B, C, and D, respectively.

## 4.3 Defining New Trading Strategies

One critical design feature of our simulation software was that it had to allow developers to easily define new trading strategies. This can be done by extending the `TradingStrategy` trait. This trait defines a `currentPortfolio` method which should return the current portfolio. Typically, your trading strategy will define a variable in the constructor to represent the portfolio, and you should override this method to return that portfolio. This is done so that other objects will not be able to mutate the portfolio instance within your trading strategy. The `TradingStrategy` trait also defines an `on` method which you should override to react to the delivery of a trading event. When your strategy receives an event, you can check the type of the event using pattern matching and perform different logic when a particular type of a event is sent. For example, if you wanted to buy/sell a stock when the market opens, you would use pattern matching to check for an event of the type `MarketOpen` to be sent, then perform this logic.

## 4.4 Defining Sources of Data

The ability for developers to easily extend our simulation software by adding new sources of information which strategies can consider in their decision-making was another crucial design choice. This was the main motivation for the `TradingEvent` trait. By creating case classes which extend this trait, you can make it easy for strategies to use Scala's pattern matching feature to react to your event type. To add a source of data, simply create a case class which extends `TradingEvent` and encapsulate your data within this case class. Then define a corresponding class, which will be used to query for these events during a backtest, that extends the `TradingEventEmitter` trait. This trait contains one method called `eventsForInterval`, which returns an iterator of the events (that is in-order by time) in a given time interval. An instance of the class extending `TradingEventEmitter` will need to be passed to your `TradingContext`'s constructor to generate the events during the backtest.

# 5 Lessons Learned

In this section, we reflect on our progress, documenting the problems that we encountered, and outline some possible directions of future work.

## 5.1 Problems Encountered

The main problem that we encountered was related to our project plan. We underestimated the amount of work that was required to produce trading simulation software that was viable for running our tests. Due to the limited time we had to work on this project, we could not build a comprehensive trading simulation with exceptional real-world accuracy from scratch. Our simulation software lacks realistic models which consider the trading volumes of each stock when placing orders. In our simulation, your orders are always fulfilled, but this is not always the case in real-life. The development of our simplified simulation and other parts of the project – including the CrowdIQ model, structures and schemas for stock prices and microblog posts, APIs to access stock price information, and more – put us behind schedule in the research and development of novel trading strategies. We would have liked to spend time on machine learning-based strategies and day trading strategies, but our time was too constrained for this to be possible. This project was a learning experience for our group because we had essentially no prior knowledge about the stock market. The effort behind this project has taught us some valuable lessons about developing and committing to a project timeline.

## 5.2 Future Work

There are many different directions in which future work could take this project. This section will detail a few of the directions that we have considered.

### 5.2.1 Improving Trading Simulation Software

As discussed in the *Problems Encountered* section, our trading simulation software has an oversimplified ordering model that isn't always consistent with stock trading in the real world. One way to expand upon our work is to improve our simulation using a realistic ordering model that considers whether your stock orders would be fulfilled. Quantopian's slippage models are an example of an implementation of a realistic order fulfilling model [15]. This would require a modification of our pricing data schema to include the trading volume for each price tick. It would also be beneficial to modify the schema to include the bid/ask price instead of just one trade price. This would allow the simulation to consider the bid/ask spread. Additionally, our simulation software lacks a comprehensive graphical or command-line interface. One way to develop a UI around this software is to turn it into a web server which you can access from your browser to start simulations with adjustable parameters (the trading strategies, trading event emitters, and time intervals) and easily export/view the results of past simulations. This would streamline the development and analysis of trading strategies using our system. Another addition to our trading simulation software would be the integration with platforms that execute real trades (like the Bloomberg Terminal [19] and Robinhood [16]), allowing your strategies, as defined in this software, to drive real trades.

### 5.2.2 Advanced Trading Strategies

In the *Problems Encountered* section, we mentioned that, due to time constraints, we were unable to explore some more advanced trading techniques. To expand upon our work, a future project could research machine learning-based strategies that learn relationships from various sources of information, such as stock price trends and the sentiment of microblog posts, to make decisions on whether to buy/sell a stock. Additionally, it would be interesting to experiment with day trading. This would require a high-resolution source of stock quotes. It may be beneficial to address the potential inaccuracies of the simulation before adding day trading support, because the error introduced by not considering the bid-ask spread or not implementing a realistic order fulfillment model may compound as you start to trade at higher frequencies.

## 6 Acknowledgements

We thank our colleagues from Virginia Tech who provided insight, expertise, and resources that greatly assisted the research, especially Professor Edward Fox. We thank Saurabh Chakravarty for serving as our client and providing us with valuable feedback and guidance.

# References

[1] Cloudera. *Deploying HBase on a Cluster*. 2017. URL: https://www.cloudera.com/documentation/enterprise/5-8-x/topics/cdh_ig_hbase_cluster_deploy.html (visited on 05/10/2017).

[2] Qianzhou Du et al. "CrowdIQ : A New Opinion Aggregation Model". In: *Proceedings of the 50th Hawaii International Conference on System Sciences* (2017), pp. 1737–1744.

[3] Internet Engineering Task Force. *RFC 4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files*. 2017. URL: https://tools.ietf.org/html/rfc4180 (visited on 04/11/2017).

[4] The Apache Software Foundation. *Apache Hadoop*. 2017. URL: http://hadoop.apache.org/ (visited on 04/11/2017).

[5] The Apache Software Foundation. *Apache HBase*. 2017. URL: https://hbase.apache.org/ (visited on 04/11/2017).

[6] The Apache Software Foundation. *Apache Spark*. 2017. URL: https://spark.apache.org/ (visited on 04/11/2017).

[7] The Apache Software Foundation. *Apache Spark MLlib*. 2017. URL: http://spark.apache.org/mllib/ (visited on 04/11/2017).

[8] Git. *Git Version Control*. 2017. URL: https://git-scm.com/ (visited on 04/14/2017).

[9] Google. *Google Finance*. 2017. URL: https://www.google.com/finance (visited on 04/11/2017).

[10] GitHub Inc. *GitHub*. 2017. URL: https://github.com/ (visited on 04/14/2017).

[11] Lightbend Inc. *Play Framework - Build Modern & Scalable Web Apps with Java and Scala*. 2017. URL: https://www.playframework.com/ (visited on 04/11/2017).

[12] Lightbend Inc. *sbt - The Interactive Build Tool*. 2017. URL: http://www.scala-sbt.org/ (visited on 04/11/2017).

[13] Lightbend Inc. *The Play WS API*. 2017. URL: https://www.playframework.com/documentation/2.4.x/ScalaWS (visited on 04/11/2017).

[14] Lightbend Inc. *The Scala Programming Language*. 2017. URL: `http://scala-lang.org/` (visited on 04/11/2017).

[15] Quantopian Inc. *Quantopian Help - Slippage Models*. 2017. URL: `https://www.quantopian.com/help#ide-slippage` (visited on 04/14/2017).

[16] Robinhood Markets Inc. *Robinhood - Free Stock Trading*. 2017. URL: `https://robinhood.com/` (visited on 04/14/2017).

[17] Ecma International. *JSON*. 2017. URL: `http://www.json.org/` (visited on 04/11/2017).

[18] Investopedia. *What is a stock split? Why do stocks split?* 2017. URL: `http://www.investopedia.com/ask/answers/113.asp` (visited on 04/14/2017).

[19] Bloomberg L.P. *Bloomberg Terminal — Bloomberg Professional Services*. 2017. URL: `https://www.bloomberg.com/professional/solution/bloomberg-terminal/` (visited on 04/14/2017).

[20] Paul R. La Monica. *Twitter skyrockets on more takeover rumors.* Sept. 23, 2016. URL: `http://money.cnn.com/2016/09/23/investing/twitter-takeover-rumors-google-salesforce/` (visited on 05/01/2017).

[21] Jesse Solomon. *Apple stock now costs $94. Fans love it.* 2017. URL: `http://money.cnn.com/2014/06/09/investing/apple-stock-split-reactions/` (visited on 04/14/2017).

[22] StockTwits. *StockTwits, A Communication Platform for the Investing Community*. 2017. URL: `https://stocktwits.com` (visited on 04/11/2017).

[23] Steven Skiena Wenbin Zhang. "Trading Strategies to Exploit Blog and News Sentiment". In: *Proceedings of the Fourth International AAAI Conference on Weblogs and Social Media* (2017), pp. 375–378.

[24] Yahoo. *Yahoo Finance*. 2017. URL: `https://finance.yahoo.com/` (visited on 04/11/2017).

# Appendices

## A Requirements

### A.1 Who Will Be Served

The main drive for this completion of this project is Saurabh Chakravarty, our client. By iterating over his feedback, we are treating him as the eventual end user. Eric Williamson is also working with Saurabh as an additional client for the project and a potential end user. Theoretically there is room for expansion when it comes to future users. If we are successful in “predicting” the stock market, there is potential for our work to be used for monetary gain, but no plan is in place at the current time.

Automated stock trading is a rapidly developing research area. Our paper will help to further expand this research. Its availability on VTechWorks will allow future researchers to expand upon our work. As evidenced by our group’s use of earlier work done at Virginia Tech through CrowdIQ, we can provide a roadmap to jump start others’ forays into utilizing microblogging data as predictive input.

Formally there are no current plans for extended support. As opposed to other potential projects in this class ours is more conceptual/research based. Hopefully, the ability of others to reference our research will result in its future use.

### A.2 Scenarios Served

We are tasked with building an extensible trading simulation software that can be used to define and test any arbitrary trading strategy. This trading software should be able to operate on arbitrary market-related events, such as posts on microblogging websites, stock price changes, and market open and close events.

The simulation software must allow the querying of the most recent stock price for a symbol at a given time. In addition, there must be a comprehensive virtual portfolio implementation that allows strategies to easily execute transactions that buy or sell shares of a stock. The virtual portfolio must take transaction fees into account with every transaction.

We also must implement the opinion aggregation model from *CrowdIQ*.

| Symbol | Name |
|---|---|
| `$APPL` | Apple Inc. |
| `$FB` | Facebook |
| `$GILD` | Gilead Sciences Inc. |
| `$KNDI` | Kandi Technologies Group Inc. |
| `$MNKD` | MannKind Corporation |
| `$NQ` | NQ Mobile Inc. |
| `$PLUG` | Plug Power Inc. |
| `$QQQ` | PowerShares QQQ Trust, Series 1 (ETF) |
| `$SPY` | SPDR S&P 500 ETF Trust |
| `$TSLA` | Tesla Inc. |
| `$VRNG` | Vringo Inc. |

Table 1: Baseline Stocks

This complex aggregation model takes a judge's historical accuracy and the interdependence of each post into account to accumulate an accurate opinion. This will be used to implement strategies that operate on the perceived market sentiment toward a stock.

Our focus is to test and compare the performance of trading strategies for 11 stocks in the year of 2014. These 11 stocks are shown in Table 1. This set of historical data is used to compare our results to a baseline. Focusing our testing to the same set of stocks allows for a more accurate comparisons of performance. Otherwise, the performance of the chosen stocks for a strategy would influence the potential returns.

Our reach goal is to test our strategy on other time intervals in addition to the year of 2014. This will allow us to ensure that our trading method is not created just to specifically fit the data from 2014. This additional testing comes at a much lower cost than going straight to real-time testing as it will be quicker, and much of the implementation overlaps from the baseline set.

A prospective future use case of this system is for real-time testing. Real-time access to stock prices and microblogging data often comes at quite a high cost. Therefore, we are not expecting to perform any real-time testing.

## A.3 Data To Be Processed

The main dataset consists of microblog posts from the StockTwits website. Each post will contain a stock symbol at the front as a way to determine

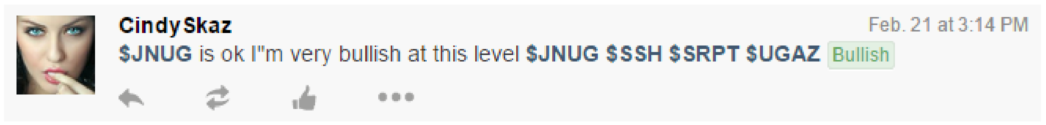

CindySkaz Feb. 21 at 3:14 PM
$JNUG is ok I"m very bullish at this level $JNUG $SSH $SRPT $UGAZ Bullish

Figure 1: Example microblog post from StockTwits

what stock the user is talking about (see Figure 1). Some posts will also contain a tag referring to the tweet as either bullish or bearish. We will use sentiment analysis to detect the sentiment of untagged posts.

The initial data set will focus on 11 stocks, shown in Table 1, for the year 2014. However, we hope to expand this time frame to include the year of 2015 as well. This will require us to obtain a set of microblog posts for 2015 from StockTwits.

We will use stock price data to verify past predictions by judges. The verification of past predictions will be used to assign a weight to each judge. A judge's weight corresponds to the historical performance of that judge in predicting stock price trends. Additionally, we will use the order in which posts arrive to determine the interdependence of a judge's opinions on other judges. This will be used to prevent judges who are subject to groupthink from affecting the overall market sentiment. All of this judge performance analysis will be implemented from *CrowdIQ*.

By analyzing the portfolio value over time of different strategies, we will determine which strategies perform most optimally. We can further analyze specific large changes in portfolio values to determine why a strategy might have gained or lost money. The prices of the 11 stocks during the change and the microblog posts during the change will be referenced in this analysis.

## A.4 Existing Code Base

Saurabh and Eric have provided us with some code that will help implement the sentiment analysis. This code is in the Scala general-purpose programming language [14]. Due to this code already being created we have decided to do the rest of the program in Scala as well and will be storing in a public GitHub repository.

We will also make use of previous implementations of chosen machine learning algorithms (specifically those provided in the Spark.ML library). This will allow us to save time by not creating anything that already exists.

# B Design

In this section, we discuss the planned design of our system. We start by outlining a very high-level architecture of interconnected components that make up our system in the *System Architecture* subsection. Next, we discuss the external tools that will be used to facilitate the development and execution of each component in the *Technology Used* section. In the *Schema/Data Structures* section, the details on how each component represents information is clarified. Finally, in the *Data Flow* section, we elaborate on the specifics of how the separate components communicate with each other.

## B.1 System Architecture

The architecture of this project is separated into three interconnected components. One component retrieves and stores information on stock prices from various sources so that it can be easily queried on-demand by other components. Another component implements the opinion aggregation model from a related work. The remaining component is responsible for testing a trading strategy and analyzing the results.

### B.1.1 Pricing Data

The Pricing Data component contains utilities that can be used to query stock prices from a given data source. All stock prices from each source are preloaded into separate tables in an HBase [5] database. For our tests, we download data from our sources on the set of stocks that we are considering (see Table 1) for the entire year of 2014. Once we have downloaded the data, we can populate a table on the cluster provided to us by Virginia Tech with the price information so that it is in a standardized format. The main purpose of this component is to provide facilities for other components to access stock price information.

### B.1.2 Opinion Aggregation

The Opinion Aggregation component is an implementation of the model described by Qianzhou Du, Hong Hong, G. Alan Wang, Pingyuan Wang, and Weiguo Fan in *CrowdIQ: A New Opinion Aggregation Model* [2]. It provides utilities that aggregate an opinion towards a stock symbol based on the microblog posts in a given time interval as well as the historical performance and consistency of each microblog author. This component is also responsible for storing and querying microblog posts from our database and provides utilities for other components to do so as well. Finally, it also provides an implementation of a sentiment analysis utility that categorizes microblog posts as “bullish” or “bearish”.

### B.1.3 Trading Simulation

The Trading Simulation component brings the other two components together to provide a full simulation of a trading environment. This involves implementing a virtual stock portfolio and the ability to execute transactions to buy and sell stocks. Additionally, it involves the ability to define a strategy that responds to specific events (for example, a microblog post, a stock price change, and/or market open or close events). This component provides utilities that allow you to define a strategy like an event listener which manipulates a virtual portfolio. Within this component, we will also implement the strategies that we are evaluating.

## B.2 Technology Used

In this section, we discuss the external tools or dependencies of each component. There are a few tools that all components depend upon. The entire project will be implemented in the Scala programming language [14]. Scala was chosen because it integrates well with Apache Spark [6], which the project also depends upon. Spark allows for large-scale data processing with high performance by distributing work across a cluster of computers. We are using the Simple Build Tool (SBT) to build and execute our Scala code [12]. SBT allows us to organize the code for each component in separate projects and introduce build dependencies easily between the projects. We use Apache HBase [5] for our database needs, which is a distributed database that runs on Apache Hadoop [4]. Apache Hadoop is a framework that facilitates the development of distributed applications. The Hadoop/HBase cluster in use was provided to us by our colleagues at Virginia Tech and allows us to store a massive amount of information (since it runs on a cluster and is optimized to store big data). We are using versions 2.10.6 and 1.5.0 of Scala and Spark (respectively) so that we can match the versions available on the cluster.

### B.2.1 Pricing Data

The Pricing Data component contains code to pull stock prices from various data sources and store them in HBase. Apart from the Apache HBase libraries, we also depend on the Play framework's [11] WS library, which provides a Scala API to perform web requests [13]. This component depends upon two different sources of data to retrieve stock prices. The Play WS library is used to query the Yahoo Finance and Google Finance APIs.

We will pull stock prices from Yahoo Finance [24] and Google Finance [9]. Yahoo Finance has historical daily stock prices available for all but one of the stocks that we are using for testing (see Table 1). The one stock that we couldn't retrieve from Yahoo, `$VRNG`, is available from Google Finance. We have developed small utilities that request daily stock prices from the web APIs given a set of stock symbols and a date range. When the API responds with the list of prices in a JSON [17] or CSV [3] file format, we parse the response, massage the data into our own data structures, and write this information to the corresponding HBase table.

### B.2.2 Opinion Aggregation

The Opinion Aggregation component is responsible for performing sentiment analysis on microblog posts and aggregating these sentiments according to a complex model from a related work. This component makes use of the HBase libraries to query the microblog posts from our database. The sentiment analysis portion depends upon the machine learning library provided by Spark called "Spark.MLLib" [7]. This library provides implementations of commonly used feature extraction, regression, and classification algorithms that are useful when implementing sentiment analysis. This component also depends on a source of microblog posts. Our client, Saurabh Chakravarty, has provided us with a CSV file containing about 1.5 million posts from the site StockTwits [22] for the year of 2014. The HBase libraries are used to build an interface to store and retrieve these microblog posts from our database.

### B.2.3 Trading Simulation

The Trading Simulation component has no external dependencies outside of those common to the entire project.

## B.3 Schema/Data Structures

This section describes the data structures and schemas used to represent and organize information in this system.

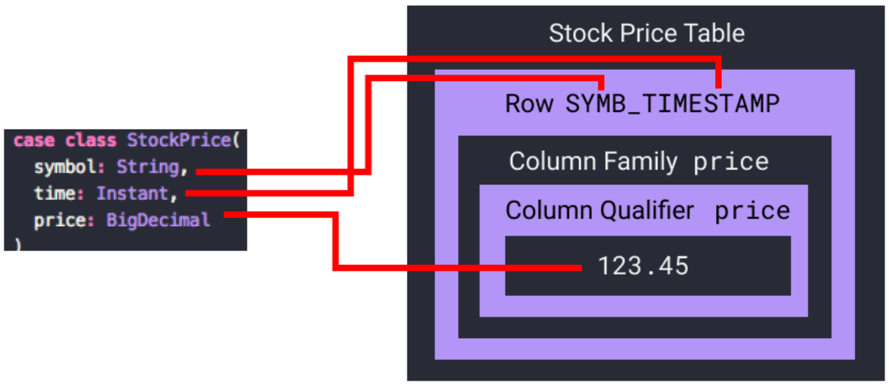
Understanding the database schemas requires some background information about how HBase works [5]. An HBase table stores records in the following manner. Each record is uniquely identified by its row key. Each row contains a set of columns which are identified by the column family and column qualifier. Our schemas are simple enough that it is not very important to know what distinguishes between the column family and column qualifier, but be aware that the combination of the column family and column qualifier uniquely identifies a column in a row.

### B.3.1 Pricing Data

The Pricing Data component defines a simple structure for a stock price within this system and a matching schema within our HBase database. Figure B.3.1 shows how the structure in Scala is stored in the database. Data from various sources must be processed into this structure before it can be stored in our database. We pull stock prices from three different sources which all use their own structures to represent the information. Google Finance's API returns information in a CSV format [3]. Every row (line) in the CSV contains columns for the date, open price, high price, low price, close price, and trading volume. The CSV will contain a line for each market day in the date range that is specified. Yahoo Finance's API returns information in a JSON format [17]. The information of interest in the JSON response is an array within the nested object structure. This array is found by following these keys in the nested object structure: "query", "results", and, finally, "quote". This array is filled with objects containing keys for the stock symbol, date, open price, high price, low price, close price, volume, and adjusted close price. These daily price summaries don't give us granular information about the price at a particular time during the day, but by associating a predefined time with the open and close of the market, we can generate a `StockPrice` object for the open and close of each market day.

### B.3.2 Opinion Aggregation

The Opinion Aggregation component is responsible for storing and retrieving microblog posts from our database, performing sentiment analysis on those

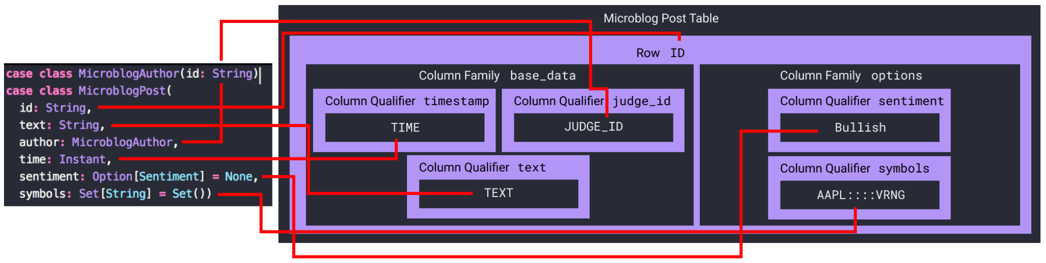


```
case class StockPrice(
  symbol: String,
  time: Instant,
  price: BigDecimal
)
```

Figure 2: The stock price data structure as a Scala case class (left) mapped onto its corresponding database schema (right).

posts, and aggregating sentiments towards a particular stock symbol. A structure is defined to represent a microblog post, and a matching schema defines how it is stored in our database. Additionally, we represent the author of a microblog post as a structure of its own, even though it only contains one field at this point. Figure B.3.2 shows both the structures and schema and how to convert between the two. The conversion between the CSV of StockTwits data from 2014 provided by our client to the microblog post structure described above is done by parsing each row of the CSV and extracting the columns that correspond to the post ID, text content, symbols mentioned, tagged sentiment, time, and author ID. These columns translate directly into values stored in the `MicroblogPost` data structure.

Aggregating the opinions of microblog authors requires many data structures. In order to future-proof our implementation for the support of live data, we are modeling the opinion aggregation as an online algorithm so it processes input incrementally and doesn't need to have all posts available at the start of execution. This decision influences the intermediate values used in the calculation of the aggregate sentiment and the data structures required to represent them. The aggregate sentiment of a stock requires that we know the sum of the post author's weight multiplied by the order in which each post that mentions the stock with the same sentiment was posted. Since we only need the sum of these, we don't need to store every post in the time interval with its associated order. In order to know the order in which each post referencing a particular stock symbol and with a particular sentiment was created, we can store these values in a map with keys taking the form of a tuple containing the symbol and sentiment and values being an integer. Additionally, we need to store each author's weight. This can be represented by a map of microblog authors to an object that calculates the author's weight incrementally. An author's weight is calculated by dividing their average prediction score by the standard deviation of their prediction scores. The object that calculates the author's weight must store enough data to calculate the average and standard deviation of the set of prediction scores incrementally (i.e., adding one score at a time). The score of a prediction is only available after we know if the prediction was correct or not, which happens after a confirmation time window. Therefore, we also need to queue up posts until after the confirmation window in order to update each author's weight.

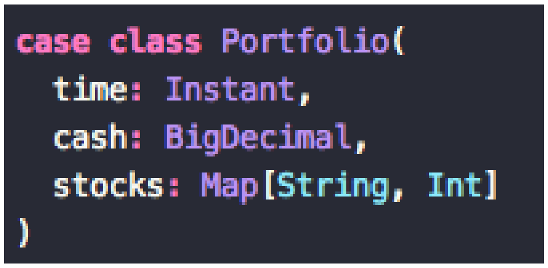


```
case class MicroblogAuthor(id: String)
case class MicroblogPost(
  id: String,
  text: String,
  author: MicroblogAuthor,
  time: Instant,
  sentiment: Option[Sentiment] = None,
  symbols: Set[String] = Set())
```

Figure 3: The `MicroblogPost` and `MicroblogAuthor` data structures (left) mapped onto the corresponding database schema (right).

### B.3.3 Trading Simulation

One of the main structures required in the Trading Simulation is the virtual portfolio. Storing a virtual portfolio involves storing the amount of each stock that is owned, as well as the amount of cash available. This structure is shown in Figure B.3.3. Because we've decided to model a trading strategy as a listener to trading events, we must define what a trading event is. A trading event is any action or occurrence that is recognized by a trading strategy. The structure of a trading event will be defined to contain a time (so that all events are guaranteed to be sorted by their time of occurrence). Other than that, any implementation of a trading event may contain any arbitrary data. For example, a microblog event would be an implementation of a trading event that contains a microblog post. Trading event emitters create iterators of trading events (that are ordered by time). In order to process each event in order from all the emitters efficiently, a min priority queue that is sorted by the event time will be used to store the next event from each emitter. When an event is pulled from the priority queue, it will be processed by the strategy and then the next event from that emitter is inserted into the queue. This process repeats until there are no more events available.



```
case class Portfolio(
  time: Instant,
  cash: BigDecimal,
  stocks: Map[String, Int]
)
```

Figure 4: The virtual portfolio data structure.

## B.4 Data Flow

This section gives a detailed view of how data flows between each component of the system.

### B.4.1 Pricing Data

Data within the pricing data component flows into our system through external sources of prices. As described in previous sections, these sources consist of the Yahoo and Google Finance APIs. Data from these sources is processed and stored into HBase tables. Then, the stock price information flows out of the pricing data component through an interface than pulls the data from the appropriate HBase table.

### B.4.2 Opinion Aggregation

External sources of microblog posts flow into the opinion aggregation component. As described in previous sections, the microblog data that we use is a set of posts from the StockTwits site in CSV format. These posts are processed and stored in an HBase table. An interface that pulls these posts from our database is provided so that other components can easily make use of this information. A utility is provided that analyzes the sentiment of a post. An additional utility calculates the aggregated sentiment towards a stock from a set of microblog posts. Both of these utilities receive microblog posts from another component (the Trading Simulation component) and return additional information (a sentiment or the current set of aggregated sentiments).

### B.4.3 Trading Simulation

As discussed earlier, in the trading simulation component, trading events flow into a trading strategy object. In response to these events, a trading strategy will manipulate virtual portfolio objects to execute transactions. When executing transactions, the portfolio will use an interface provided by the pricing data component to get the current price of a particular stock. Additionally, trading strategies may use interfaces provided by the opinion aggregation and pricing data components as factors in their decision making.

## B.5 Group Roles

Apart from Saurabh and Eric, we have a four person team to which we can delegate tasks. Joseph Watts functions as the Data Processing Specialist, which leads the effort on any interactions with the distributed data processing aspect of this project. This will require proficiency with Spark and HBase. Nick Anderson is our Portfolio Management and Machine Learning specialist. Nick will develop the portfolio management system that was detailed earlier and will assist Joseph Mehr in developing and researching machine learning algorithms. Joseph Mehr will specialize in machine learning and will specifically organize the effort behind implementing a Support Vector Machine (SVM). Connor Asbill is our Data Aggregation Researcher. His role involves researching new ways to best aggregate the data that we obtain (tweet sentiments, judge reliability scores, stock price changes, etc) to make informed trading decisions. On top of all of our roles, each of us is responsible for participating in the literature survey that is ongoing throughout this project.

## B.6 Timeline

A timeline of milestones over the course of our project can be seen below. Our group plans to meet once a week to delegate tasks and review work that has been done independently.

- Feb. 15 - Draft Trading Simulation & Pricing Data
  - Before we can evaluate any trading strategy, we need to develop a system that simulates the trading environment (by replaying historical events and allowing a strategy to buy/sell stocks). By February 15, we want to have an initial draft of the implementation of these pricing data and trading simulation components.
- Mar. 1 - Complete Trading Simulation & Pricing Data
  - By March 1, we want to have the pricing data and trading simulation components completed, so that we can start to implement and test trading strategies on top of this framework.
- Mar. 15 - Complete Opinion Aggregation & Baseline Strategy
  - By March 15, we aim to complete our implementation of the opinion aggregation model laid out by *CrowdIQ: A New Opinion Aggregation Model* [2].
  - Additionally, we want to have the trading strategy described by this paper implemented by then.
- Apr. 1 - Research and Start Implementing Trading Strategies
  - From March 15 to April 1, we'd like to have completed our research on trading strategies. By this time, our goal is to have identified three trading strategies and to start on their implementation.
- Apr. 15 - Complete Implementation and Analysis of Trading Strategies / Assemble Presentation

- By April 15, our goal is to have working implementations of the trading strategies and analyze their results for the year of 2014.
- Additionally, we want to assemble the presentation that we will be giving to the rest of our capstone class on our work.

- Apr. 25 - Complete Documentation of Work in Final Report
  - By April 25, we want to have completed our documentation for users and developers who are interested in our work.

## C Implementation

In this section, we outline the plan to implement each component of our system as it was designed in the previous section. Specifically, we are outlining the implementation plan for the trading simulation software required to test the strategies that are being researched and will be implemented in a later stage of this project. Additionally, the responsibilities for each member of our team will be made clear.

Our team will make use of the Git version control tool to help maintain our project [8]. We will use GitHub's free web service to host our source code publicly [10]. This repository is available at `github.com/saurabh/cs4624`.

### C.1 Pricing Data

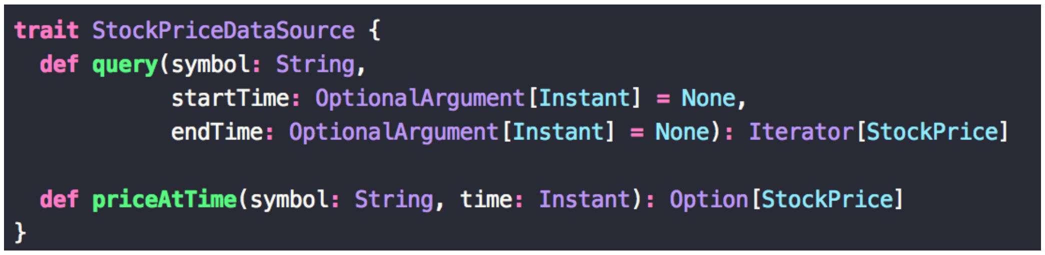
To recap the design section, the implementation of the pricing data component involves setting up the database schema, implementing an interface that stores and retrieves stock prices (in the data structure described) from our database, and, finally, implementing utilities that request and parse the stock price information from external sources. The effort behind the implementation of this component is led by Joseph Watts.

#### C.1.1 Setting Up Database Schema

In order to prepare HBase to store our data, we need to define the table that will store the stock prices from a given data source. With HBase, tables need to be predefined with a name and a list of column families before they can be used to store any information [5]. In this case, the name of our table will correspond to the stock prices that we are storing (for example, `dailystockprices`). Then, this table will contain one column family named `price`. To actually execute the command that defines this table in the database, we first start the HBase shell by running `hbase shell`. Then we run the command `create 'dailystockprices','prices'` in this shell.

#### C.1.2 Database Interactions

Assuming that a table to hold stock prices is predefined, we need interfaces to query stock prices from this table and write stock prices to this table. `StockPriceDataSource` (see figure C.1.2) is a Scala trait (which is similar to an interface in Java) that abstracts away the actual source of the stock prices. All objects that implement `StockPriceDataSource` must define two methods that allow you to request stock prices. The `query` method allows you to do a range query for stock prices of a particular symbol between two timestamps. The `priceAtTime` method allows you to get the most recent stock price of a particular symbol (that is, on or before a given timestamp). `StockPriceDataSource` affords this system the flexibility of moving away from HBase towards a different database solution. We define a `HBaseStockPriceDataSource` class which implements the contract laid out by `StockPriceDataSource`. The constructor for the `HBaseStockPriceDataSource` class takes the table name as a parameter. `HBaseStockPriceDataSource` provides functions that write stock prices back to the source (the database). These functions are used by the tools which query the external sources and populate HBase with their data.



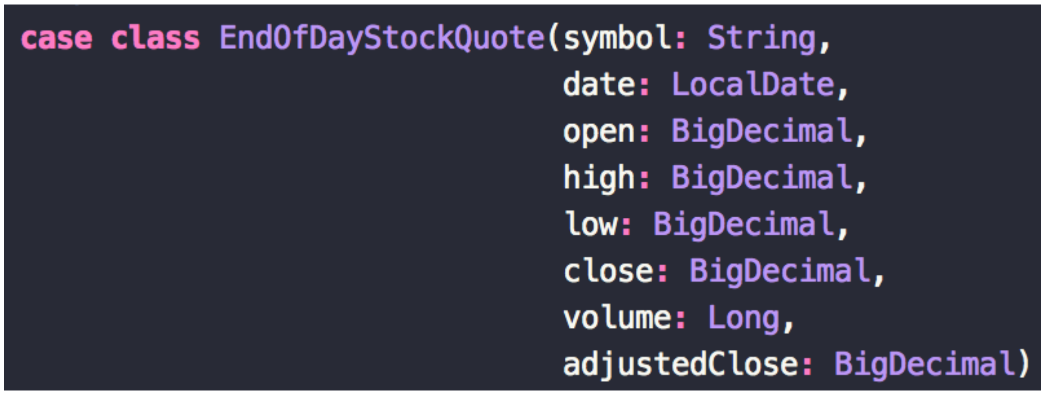
```
trait StockPriceDataSource {
  def query(symbol: String,
            startTime: OptionalArgument[Instant] = None,
            endTime: OptionalArgument[Instant] = None): Iterator[StockPrice]

  def priceAtTime(symbol: String, time: Instant): Option[StockPrice]
}
```

Figure 5: The `StockPriceDataSource` trait.

### C.1.3 External Sources

We use two different external sources to retrieve daily stock prices for the full set of stocks of interest. This set of securities consists of the 11 stocks in Table 1 and the S&P 500 index. The two data sources are Yahoo Finance and Google Finance. Since both of these sources give us access to daily summaries of the stock price, we will define an intermediate data structure that will store this daily summary and have functions to convert it into singular stock prices. This intermediate data structure can be seen in Figure C.1.3. The daily stock price summary contains the open, close, high, and low prices and the transaction volume for a particular security on a given day. For every daily summary, we will use the open and close prices to generate the open and close stock prices for that day. In the context of our simulation, we have set the time of open price to be at 7:59 am and the time of the close price to be at 4:59 pm. Once we have `EndOfDayStockQuote` objects retrieved from the external data source, we can insert the open and close prices into an HBase table using an instance of the `HBaseStockPriceDataSource`.

In order to retrieve these daily stock price summaries from the external sources, there must be some interface that queries the corresponding Web APIs. For this, we define a trait named `EODStockQuoteAPI` that has a method called `getQuotes`. Implementing classes of `EODStockQuoteAPI` (that is, `YahooFinanceAPI` and `GoogleFinanceAPI`) will implement the `getQuotes` method to construct a query to the Web API in the correct format and return a list of `EndOfDayStockQuotes`. Therefore, instances of the `YahooFinanceAPI` and `GoogleFinanceAPI` classes can retrieve the stock price information in the form of daily summaries, which we process into two stock prices at the open and close of each day, and use an instance of the `HBaseStockPriceDataSource` to populate our database.



```scala
case class EndOfDayStockQuote(symbol: String,
                              date: LocalDate,
                              open: BigDecimal,
                              high: BigDecimal,
                              low: BigDecimal,
                              close: BigDecimal,
                              volume: Long,
                              adjustedClose: BigDecimal)
```

Figure 6: The `EndOfDayStockQuote` data structure.

## C.2 Opinion Aggregation

The opinion aggregation component contains sentiment analysis code, facilities to aggregate an accurate crowd-sourced sentiment towards a stock from microblog posts based on the *CrowdIQ* model, and facilities to store and retrieve microblogs from our database.

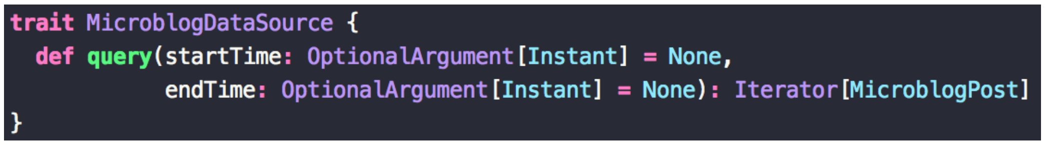
### C.2.1 Setting Up Database Schema

Similar to the pricing data component, we need to setup a table in HBase to store our microblog posts. Recall from Figure B.3.2 that a microblog post table has the column families `base_data` and `options`. From the HBase shell, we execute the command "`create 'stocktwits','base_data','prices'`" to create the table that will store all of our post data.

### C.2.2 Database Interactions

In the same pattern as the pricing data component, we define a trait named `MicroblogDataSource` (see Figure C.2.2). `MicroblogDataSource` contains a `query` method that allows you to perform a time range query on the set of posts. The `HBaseMicroblogDataSource` class, which implements the contract laid out by `MicroblogDataSource`, implements the `query` method so that it retrieves the posts from a table (specified in the class' constructor) in our HBase database. Additionally, the `HBaseMicroblogDataSource` contains a `write` function that will write a set of posts into the HBase table.

Our microblog post data from StockTwits that we received from our client is in the form of a CSV file. We implemented a `CsvMicroblogDataSource` class that reads the posts from our CSV file, so that we can write them to our database with an instance of the `HBaseMicroblogDataSource`.



```
trait MicroblogDataSource {
  def query(startTime: OptionalArgument[Instant] = None,
            endTime: OptionalArgument[Instant] = None): Iterator[MicroblogPost]
}
```

Figure 7: The `MicroblogDataSource` data structure.

### C.2.3 Sentiment Analysis

The implementation of our sentiment analysis model is simply a slightly modified version of the code that was provided to us by our client Saurabh Chakravarty and his colleague Eric Williamson. The provided sentiment analysis implementation used the Word2Vec feature extractor and the logistic regression classifier from Spark's MLlib [7]. We trained and tested our sentiment analysis model using the posts from StockTwits that already had an author-specified bullish/bearish sentiment. We sampled 50% of the pre-classified posts to serve as the training data and the remaining pre-classified posts were used as testing data.

### C.2.4 Aggregation Model

A class named `AggregatedOpinions` will be implemented to aggregate the sentiments towards each stock based on the method outlined in *CrowdIQ* [2]. An instance of `AggregatedOpinions` is passed `MicroblogPost` objects (that include a sentiment that has been analyzed by our sentiment analysis model) one-by-one through a method named `on` (like an event listener). The quantity of author individual weight times the degree of independence needs to be accumulated for each stock symbol and sentiment pair. Therefore, with each post, `AggregatedOpinions` will accumulate these values in a map that is keyed by a tuple of the stock symbol and sentiment. The degree of independence of a post is an exponential decay function that requires the order in which a post appears among others with the same sentiment and stock in a given time frame. This requires an additional map (also keyed by a tuple of stock symbol and sentiment) to store the order in which the next post appears. Individual weights for each author are also maintained in this object. These weights are calculated by determining the post's score, which measures the accuracy of the prediction made by the post. It does this by checking the price change of the stock in a time window after the post was made and checking if the trend is consistent with the prediction. Therefore, to adjust the author weights, this object needs to store each post (using a queue) until after the time window when it can verify the price change. The `AggregatedOpinions` class exposes a method named `sentimentForStock` which allows you to query for the aggregated sentiment of a stock in the current time window. It also exposes a `reset` method which resets the time window (clears the current aggregated sentiments).

## C.3 Trading Simulation

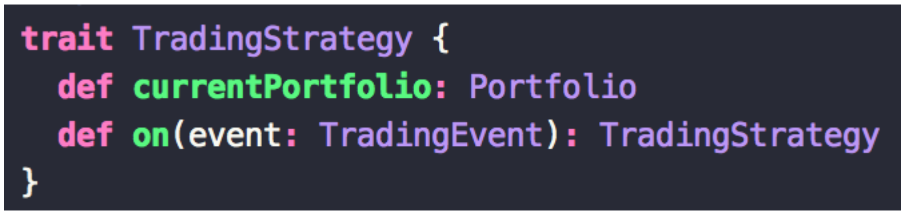
The implementation of the Trading Simulation component consists of the development of a virtual portfolio model, a trading context that collects events for a given time interval and passes them off to a trading strategy, and the trading strategies themselves. This is quite an open-ended task. Due to the complexity and importance of this component, which is required before we can implement and test any trading strategies that we encounter, all group members are responsible for contributing to this component.

### C.3.1 Virtual Portfolio

The virtual portfolio stores a cash value as well as the number of each stock that you own. It provides several functions that allow you to buy and sell shares of a stock. Additionally, it performs error checking so that you cannot perform invalid trades. Nick Anderson was responsible for implementing the first draft of the virtual portfolio. After the first draft of the virtual portfolio was completed, our team convened to discuss the final form of this API. This API consists of a class named `Portfolio` which stores the amount of cash available, and a map containing the stock symbols mapped to the amount of each stock that is currently held in the portfolio. This class is modeled as an immutable object. The `Portfolio` object requires a `StockPriceDataSource` instance to buy/sell stocks or calculate the total value of the portfolio. A `TransactionError` object is defined to represent an error performing a transaction. This error object contains the previous portfolio and an error message describing what went wrong. The `Portfolio` object contains several methods to perform transactions, all of which return either a `TransactionError` or the resulting `Portfolio` object. The methods that perform transactions include `withSharesAtTargetAmount`, `withSharesAtTargetValue`, `withSharesPurchasedAtValue`, `withSharesPurchased`, `withSharesSold`, and `withAllSharesSold`. The `withSharesAtTargetAmount` and `withSharesAtTargetValue` methods perform a transaction so that the number of shares or the total value of the shares for a particular stock is a particular value. The `withSharesPurchasedAtValue` and `withSharesPurchased` methods will try to purchase more shares of a stock using the number of shares or the amount that you can buy with a dollar value, respectively. The `withSharesSold` method will try to sell a number of shares of a stock. The `withAllSharesSold` will sell all shares in the portfolio.

### C.3.2 Trading Context

We are required to be able to easily define different trading strategies and execute them by passing old trading events. At the core of this functionality is a class named `TradingContext`, which contains a list of all the trading strategies to test and a list of the event emitters that will generate events to pass to the strategies. The `TradingContext` is responsible for obtaining the trading events from the emitters and passing them to the strategies in order. This class contains a `run` method that runs a simulation for a given time interval. This calls on the event emitters to generate the events for the provided time interval and pass them off to all of the strategies. Each trading strategy is implemented by extending a Scala trait named `TradingStrategy`. This involves implementing an `on` method that responds to an instance of `TradingEvent`. In this method, you can execute transactions on your portfolio or maintain information that influences the decision to perform a transaction (like analyzing the sentiment of a `MicroblogPost`). The `TradingEvent` trait defines each event as containing a time (so that they can be sorted). Extensions of `TradingEvent` can choose to provide more information. For example, we define the `MicroblogPostEvent` as containing a `MicroblogPost` object. The `TradingEventEmitter` trait allows you to define how events are generated by implementing an `eventsForInterval` method where you may return an interval of events for a given time interval. In the `TradingStrategy`'s `on` method, we can use Scala's pattern matching feature as a convenient way to check the type of event and extract data from it. Figures C.3.2, C.3.2, and C.3.2 show our `TradingStrategy` trait, `TradingEvent` trait, and `MicroblogPostEvent` class, respectively.

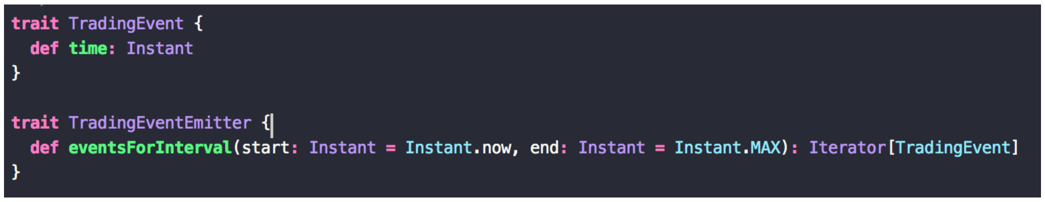


```
trait TradingStrategy {
  def currentPortfolio: Portfolio
  def on(event: TradingEvent): TradingStrategy
}
```

Figure 8: The `TradingStrategy` trait.

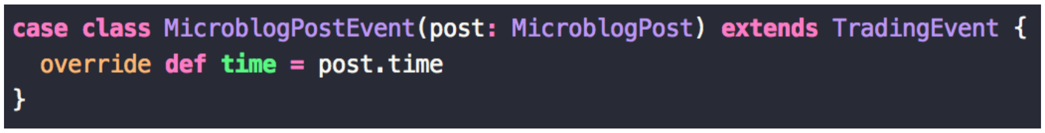
### C.3.3 Trading Strategies

Referring back to our timeline from the design report, we are planning to implement the trading strategy described by *CrowdIQ* before March 15 [2]. We can run simulations with this strategy and compare the results achieved by our implementation to see our simulation software in action and to make



```scala
trait TradingEvent {
  def time: Instant
}

trait TradingEventEmitter {
  def eventsForInterval(start: Instant = Instant.now, end: Instant = Instant.MAX): Iterator[TradingEvent]
}
```

Figure 9: The `TradingEvent` trait and the `TradingEventEmitter` trait.



```scala
case class MicroblogPostEvent(post: MicroblogPost) extends TradingEvent {
  override def time = post.time
}
```

Figure 10: A `TradingEvent` signifying a new microblog post was made.

sure that our implementation of *CrowdIQ* is working properly. More rigorous testing and analysis will be performed at later stages of the project. Additionally, we will implement a simple strategy that invests in the S&P 500 index fund to compare our results against the market performance. The three trading strategies that we'd like to start implementing on April 1 will be implemented after we have ensured that we have a working system on which to test our strategies. The intricate details of these strategies will be discussed in detail in the Refinement report.

# D Prototyping

At this stage, we have implemented the trading simulation software that has been previously described in the requirements, design, and implementation reports. This section details the usage of this software. We describe how to compile and run this software and how to interpret its output.

## D.1 Program Entry Point

We have defined the program's entry point (the main function) in an object called `TradingSimulation` within the Trading Simulation component. The trading strategies that will be tested, the trading events that will be emitted during the simulation, and the time interval that the simulation will test all need to be defined in the entry point of the program.

At this point, the baseline strategy and a buy-and-hold strategy are the only strategies that have been implemented. The Buy-and-Hold strategy is used to compare the baseline against the S&P 500 index by tracking the change in value of the index over time. The implementation of the Baseline strategy operates on one stock at a time, so you must define 11 instances of the Baseline strategy (one for each stock). The entry point receives a callback after each trading event, allowing it to track the value of each strategy's portfolio. On each market open event, we output a line to a CSV file with three columns. The first column is the time. The second column is the sum of all 11 Baseline strategy portfolio values. The third column is the S&P 500 Buy-and-Hold strategy portfolio value. A CSV file is a universally recognized file format that facilitates analysis on our data.

This project does not provide any interface that is especially usable, but we can describe a few things about the code in the program entry point that will aid in customizing the parameters of the simulation.

### D.1.1 Specifying the Simulation Time Range

Since we are backtesting on old data, we need to specify a time range over which to run the simulation. This date is specified in the `TradingSimulation` program in the `start` and `end` values that are passed in the constructor of the `TradingContext`. These values are constructed using the `OffsetDateTime` class from Java 8's time library. Therefore, it is easy to read and modify the date and time of the timestamp being passed to the `TradingContext`.

### D.1.2 Customizing the Trading Events

The trading event emitters are specified in the `eventSources` value in the `TradingSimulation` program. `eventSources` is a list of event emitters that you can easily modify to add or remove the current defined event emitters. By default, the `eventSources` value is constructed to contain the `MarketEventsEmitter`, which sends market open/close events, and the `MicroblogEventEmitter`, which sends an event for each microblog post.

### D.1.3 Customizing the Trading Strategies

The trading strategies in the simulation are more complicated to specify than one might expect. Because some strategies only operate on one stock and you need to define several instances of them to cover all 11 stocks, the strategies are specified in a map of the strategy name to a list of strategies. This allows us to easily sum the value of a list of strategies and output a CSV with a header containing the strategy names and rows containing the summed up portfolio values. For example, you can store the 11 instances of the baseline strategy with the key "Baseline", and the CSV file will contain a column that corresponds to the summed up portfolio values of these 11 strategies. This allows for a very declarative specification of trading strategies without requiring any modification of the code that writes the CSV.

## D.2 Running a Simulation

Before you can compile and run our code, you first obtain it from our GitHub repo by using the git command line utility: `git clone https://github.com/saurabh/cs4624` [10]. As we explained in the design report, we chose to use the sbt build tool for our project. The code can be compiled by executing `sbt compile` in the project directory. This will compile all three components. Running a main class in a particular component or sub-project can be done by executing a command with the following form: `sbt ''ProjectName/run-main package.ClassName''`. Therefore, to run the `TradingSimulation` program, we execute `sbt ''TradingSimulation/run-main cs4624.trading.TradingSimulation''`.

## D.3 Interpreting the Output

As stated previously, our program outputs data points for the portfolio values of each strategy at the beginning of every market day. Because this information is output in a CSV file, we can easily import it in spreadsheet software that is suited to perform analysis on the data. An easy visualization of this data can be made by creating a line graph with the dates on the horizontal axis and the portfolio value on the vertical axis. Each strategy would be a separate series on this graph. This allows you to compare the performance of the strategies throughout the year. See Figure D.3 for an example of this that compares the S&P 500 index fund with the Baseline strategy.

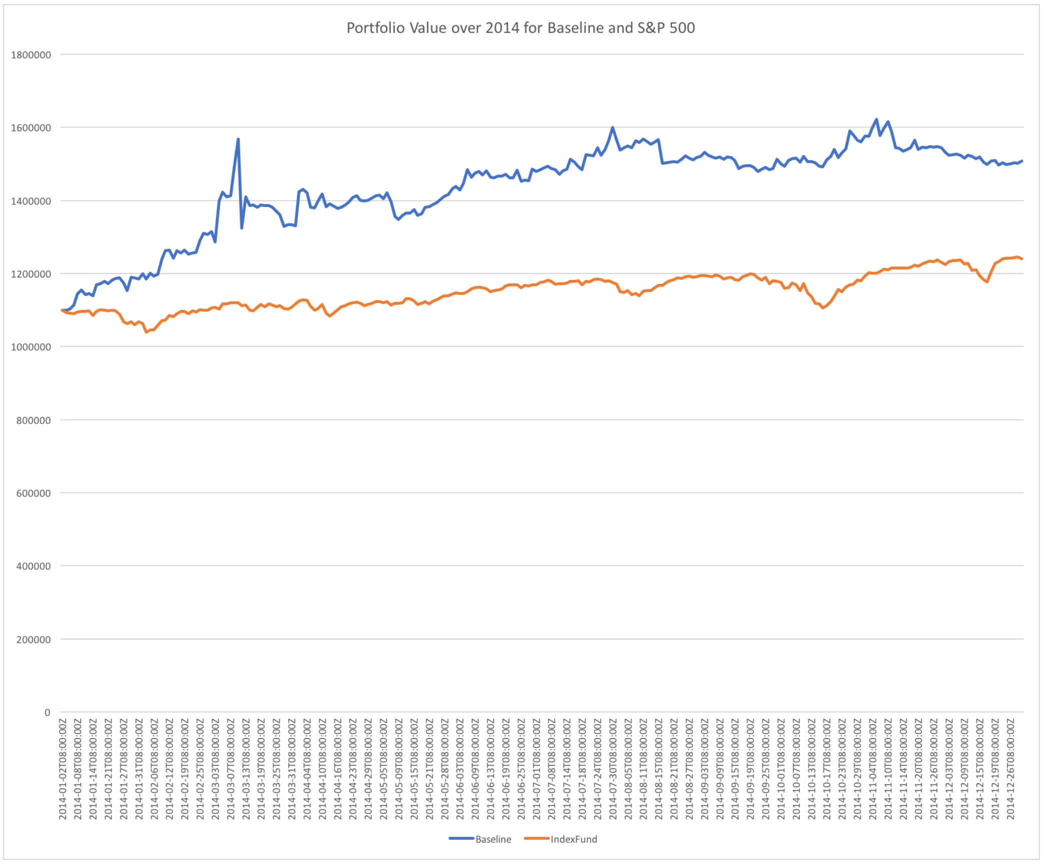


Figure 11: Comparison of the Baseline Strategy and S&P 500 Index Portfolio Values for the year of 2014.

# E Refinement

After implementing the prototype, which consists of the trading simulation software, the baseline strategy, and a simple buy-and-hold strategy, we reviewed our progress with our client, Saurabh. Together, we determined that our simulation is as designed and the baseline implementation is working as anticipated. In this section, we discuss the refinements that we make on our prototype and three trading strategies that we will implement, test, and compare against the baseline.

## E.1 Stock Split Consideration

We discovered that we neglected to consider stock-splits in the design of our trading simulation software. A stock split is when a security's shares split so that the number of shares is adjusted (and as a consequence, the value of each share changes) [18]. For example, in a 1-for-3 stock split, one share splits into three shares with values that sum to the same value of the existing share. There are also reverse stock splits (or "stock merges"). In our research, we discovered that the only stock split for the year of 2014 for our selection of 11 stocks was `$AAPL`'s 1-for-7 split on June 9, 2014 [21]. We were able to implement the adjustment for stock splits in our virtual portfolio by hardcoding the split ratio and adding a function `withSplitAdjustments` that will adjust the number of `$AAPL` stocks that are owned once June 9, 2014 is passed in the simulation. This served as a short term solution and allows us to prioritize our work on implementing additional trading strategies.

## E.2 Moving Average Strategy

Using moving averages of stock prices is a very common indicator used to make trading decisions. A moving average is calculated by averaging the stock price for the last $n$ days. We devised a "Moving Average" trading strategy that takes a long moving average and a short moving average of a stock's price, and compares them against the closing price of the day. If the closing price is greater than both the short and the long moving averages, then we buy the stock. If the closing price is less than both the short and the long moving averages, then we sell the stock. Otherwise, we hold. This strategy is based on the idea that a stock's price will continue on its current trend.

To implement this strategy, we used a queue of `StockPrice` objects to store the open price of the last $m$ days where $m$ is the number of stocks to consider in the long moving average. On every market open, we query for the current stock price and add it to the queue, dropping the front of the queue while the length is greater than $m$. With this queue, it is simple to compute a moving average.

Since we are doing relatively short-term trading, we decided on using 5 days for the short moving average and 10 days for the long moving average.

## E.3 Moving Average with Sentiment Strategy

The baseline strategy does not consider the trend in stock price in the decision-making process. With the "Moving Average with Sentiment" strategy, we attempt to consider the price trend in addition to the aggregated market sentiment. The strategy works by only trading when the baseline strategy and the "Moving Average" strategy would both make the same decision. That is, if the "Moving Average" strategy would expect the stock price to continue rising (if it would buy), but the baseline strategy thinks that the market is bearish, it will not make a trade. Similarly, if the baseline detects a bullish market, but the "Moving Average" strategy is expecting the price to drop, then it will not make a trade. It only trades when the expectations of both strategies are the same. The hypothesis with this strategy is that baseline can lose a significant amount of money by not considering the current price trend and only going off of others opinions, and by augmenting this strategy with the information about the price trend, the decision-making considers more factors.

## E.4 Selection by Sentiment Strategy

The "Selection by Sentiment" strategy differs from the baseline in the way that it allocates funds to bullish stocks. The baseline strategy allocates funds for each stock in a separate portfolio. This is inefficient when a stock is bearish because the funds are sitting in cash as opposed to being invested in a different stock that is bullish. This strategy is a more efficient allocation of funds because it keeps all the cash in the same portfolio. On each market open event, it evenly distributes the value of the portfolio among a maximum of $n$ stocks that it detects as bullish. By more efficiently allocating funds, we can significantly increase profit. This is because profits will compound across all stocks since they are all in the same portfolio and more money will typically be invested in each bullish stock since it rarely happens that all stocks are bullish.

We will experimentally test several values for $n$, the maximum number of stocks that you can be invested in, to determine which performs the best. Interestingly, a related work on sentiment-based stock trading strategies claimed that diversification (investing in more stocks) leads to diminished returns [23].

## F Testing

In this section, we will test the performance of our strategies by comparing their profit rate and portfolio values. We will also go over the testing that we perform on the trading simulation software to ensure that it operates as expected.

### F.1 Trading Simulation Software

The trading simulation software has a few unit tests that ensure that certain operations are working correctly. The virtual portfolio is tested with a unit test that performs operations on a portfolio instance and checks that the output of each operation is as expected. That is, it tests all of the transaction methods with valid and invalid inputs, making sure that the correct output is returned each time. We also perform unit tests on `TradingContext` to ensure that it delivers events to the strategies in order. This test is implemented by defining several mock trading event emitters that emit a mock trading event class that only contains a timestamp. The test also defines a trading strategy that accumulates all of the events that it receives and an assertion is performed at the end to make sure that all of the events occur in the expected order. We perform unit testing on the `MarketEventsEmitter` to make sure that the market open/close events are only sent for days when the market is open. It also ensures that a market open event is always followed by a market close event on the same day. This unit test is performed with a hardcoded time range (so that we can manually figure out a list of events that should be emitted and easily compare this with the result).

As a precaution, we also perform extensive logging during the execution of a simulation, so that it is possible to detect any issues by looking back at logs. This has allowed us to detect issues that are harder to write tests for, such as incorrect author weight calculation in the opinion aggregation implementation.

### F.2 Trading Strategy Performance

Trading strategies are tested by running a simulation with historical data and observing their performance. This performance can be analyzed to determine which strategies are performing the best, and which ones perform the worst.

### F.2.1 Results for 2014

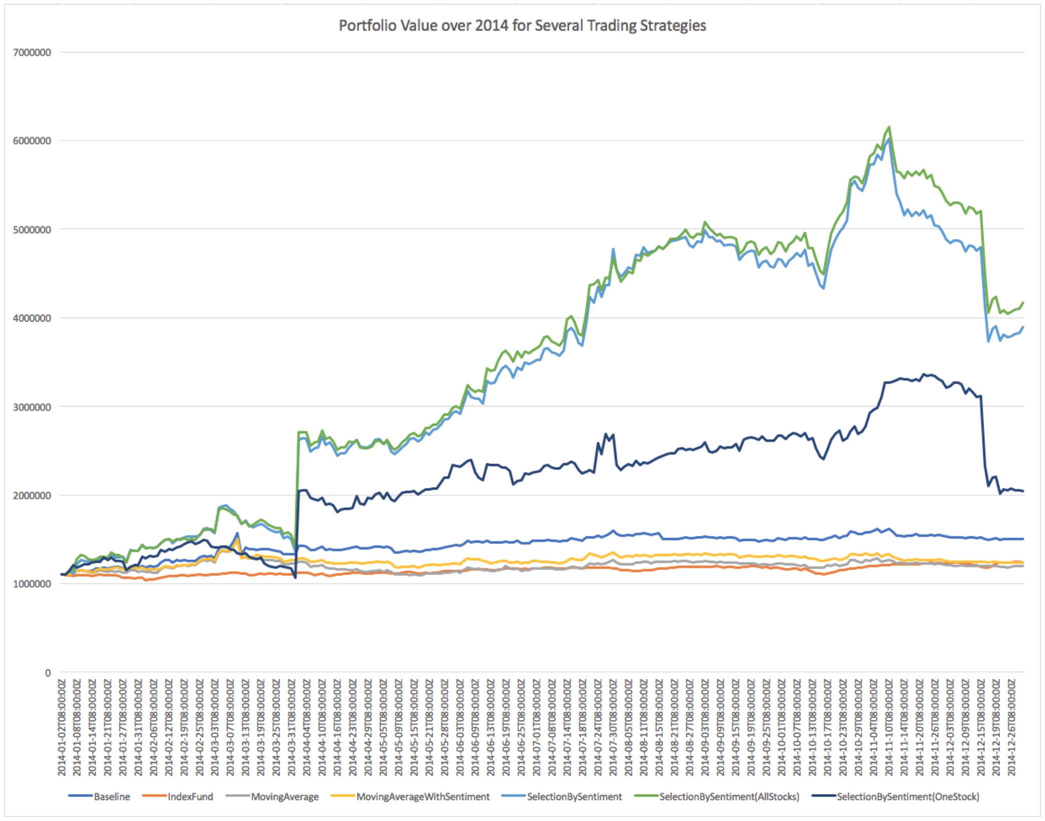


Figure 12: A graph of the portfolio values for the year of 2014 for several trading strategies.

For 2014, we tested 7 different strategies, including the baseline strategy and the buy-and-hold S&P 500 strategy. The "Moving Average" and "Moving Average with Sentiment" strategies detailed in the refinement report were also tested. Additionally, three variations of the "Selection by Sentiment" strategy were tested. The first is "SelectionBySentiment(OneStock)", which is the "Selection by Sentiment" strategy with $n = 1$. "SelectionBySentiment(AllStocks)" is the "Selection by Sentiment" strategy with $n = 11$. "SelectionBySentiment" is the "Selection by Sentiment" strategy with $n = 3$. A graph of each strategy's portfolio values over time for the year of 2014 is shown in Figure 12.

The Moving Average strategy only shows some marginal improvement over the S&P 500 index. By considering the sentiment with the moving average, we see some additional improvement over that, but unfortunately, the Moving Average with Sentiment strategy yields lower returns than our baseline implementation.

The Selection by Sentiment strategy yields the greatest returns of any strategy that we have tested by far. Interestingly (and maybe intuitively), greater values of $n$ yield a higher return, even though this disagrees with the assertion made by the related work [23].

We start each strategy off with a total of \$1.1 million. The "SelectionBy-Sentiment(AllStocks)" strategy's maximum portfolio value over the year was \$6.15 million (on Nov. 10), which is an incredible 459% profit. This strategy closed the year (on Dec. 31) with \$4.17 million (a 279% profit) after dropping quite a bit in a short time frame, but this number is still significantly higher than any other strategy.

There is a large jump in the graph for all Selection by Sentiment strategies from April 1 to April 2. After some analysis, we discovered that this due to an investment in the MNKD stock (which was \$4.02 on April 1 and closed at \$6.99 on April 2). This occurred because MNKD received a recommendation for FDA approval from an assembled Adcom committee after two previous denials from the FDA. Looking at the posts from our StockTwits data on March 31, we received an influx of speculative posts about FDA approval (like "@the_twit `$MNKD` , I fully expect to get at least partial approval, but more importantly-soon Europe will also approve. Insulin lab there+2" and "`$MNKD` in the end with approval I expect that it will shove it into the 7.87 range.-then fade-as people come to grips with the work ahead..").

There was also a large fall-off towards the end of the year on December 16. This was due to the purchase of `$VRNG` on December 12 in anticipation of the Federal Circuit's ruling on Vringo's request for en banc. This request was denied by the Federal Circuit on December 15, sending the stock into a free-fall. After looking at the StockTwits posts from December 11, some seemed to be hopeful that the en banc would be approved, while others seemed pessimistic. Here are some examples of these posts: "`$VRNG` Thus far charts looking nicely. We may take a nice rise.", "`$VRNG` I'm here. Good morning VRNGites! Hope Mr Buies is ready to deal with corupt system and we go to \$10!!!!", "`$VRNG` Chief Judge Prost has stood up for the little guy in the past... `http://stks.co/e1MT6`", and "`$VRNG` Trying to get licensing fees from ZTE is like stopping piracy. It's not going to happen, and def not for a significant amt of money." These posts give some insight into what's happening, but there doesn't seem to be a clear consensus. This indicates that experimentation with the cutoffs for bullish and bearish aggregated opinions could improve performance.

### F.2.2 Results for 2015

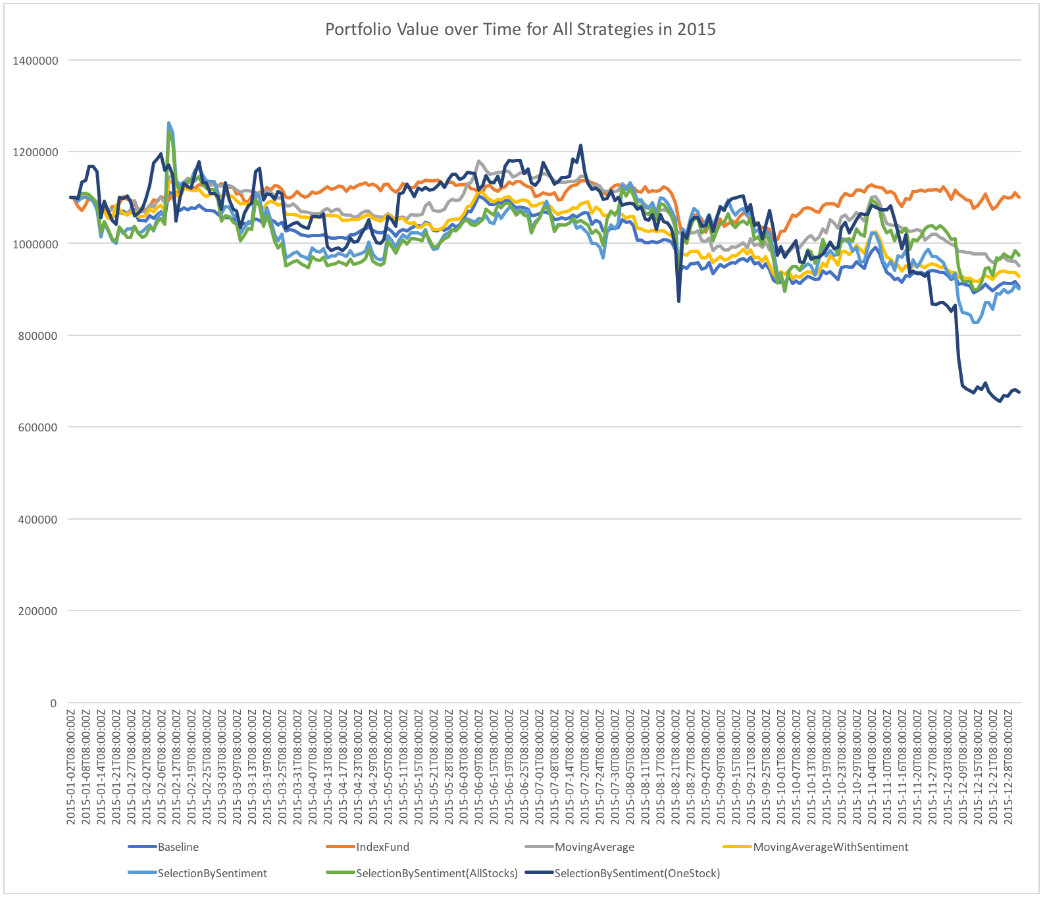


Figure 13: A graph of the portfolio values for the year of 2015 for several trading strategies.

For 2015, we tested the same 7 strategies as we did in 2014. A graph of each strategy's portfolio values over time for the year of 2015 is shown in Figure 13.

The results for all strategies are quite underwhelming in comparison to the 2014 results. The S&P 500 index fund had the best performance judging by the portfolio values at the end of the year, but it still did significantly worse than it did in 2014. The index fund ended the year roughly at the same value with which it started, achieving essentially a zero percent return. This means that the overall market performed very poorly. All other strategies performed worse than the index fund, which shows that our strategies do not perform well in a bearish market. The only non-sentiment-based strategy other than the index fund is the moving average strategy, and this strategy performed consistently better throughout the year than all of the sentiment-based strategies. We can conclude that our sentiment-based strategies with the current sentiment analysis and opinion aggregation models perform very poorly in bearish markets.

## F.3 Additional Refinements

In the future, we'd like to further refine our system by adding more comprehensive support for corporate actions like stock splits and dividends without having to hardcode them in our simulation. Our goal is to perform more testing, specifically with data from the year of 2015. This will help us determine whether our strategy is scalable or if it is market-dependent. We would also like to add support for volume information and more complex and realistic stock ordering mechanisms in our simulation software. Our current simulation makes the assumption that the volume of stock that we would like to buy is available for purchase and that the volume of stock that we would like to sell is always sold. This is not always true in the real world, and it requires a more complicated simulation that takes the trading volumes into account to model this phenomenon. A more realistic simulation might support realistic ordering mechanisms like market orders and limit orders and implement slippage models like Quantopian does [15]. On top of this, we'd like to experiment with machine learning based strategies that perform price prediction based on sentiment and past price trends. More long-term refinements might include full support for live testing with real-time data, and integration with real trading platforms like Robinhood [16] and the Bloomberg Terminal [19].