Map Reduce for Algorithmic Trading

Akshaan Kakar, Alice Berard
Dept. of Computer Science
Columbia University
Email: ak3808@columbia.edu, alice.berard@columbia.edu

***Abstract*—Algorithmic Trading is an extremely competitive sector of financial markets. Developing trading algorithms involves the pivotal step of backtesting, where the performance of the algorithm is validated against large amounts of historical securities pricing data. These time series require large amounts of computational resources for storage, processing and visualization. In this paper, we describe the implementation of an algorithmic trading backtest engine that uses "Big Data" tools as a backbone to backtests for trading algorithms in an efficient and scalable fashion. We also show how this system can be extended to support live trading as well as a plethora of other features.**

***Keywords*-Algorithms, Finance, Trading, Big Data, Hadoop, Spark, Backtesting**

I. Introduction

With the computerization of order flows in financial markets, traders were afforded the ability to have computers place buy and sell orders on securities according to predefined strategies. The advent of such Algorithmic Trading strategies eventually developed into an extremely competitive sector of financial markets. The development of trading algorithms is now a formalized process that involves intricate research and mathematical models. Although a significant portion of this development effort is invested in the underlying mathematical theory, backtesting of algorithms is the most pivotal step in the process. Backtesting is the phase in which a trading algorithm is validated against large amounts of historical pricing data. Since trading algorithms may place orders extremely frequently and involve multiple securities in a single portfolio, large amounts of data must be stored, read and processed in order to run these tests. Moreover, the result must be visualized so that developers can gauge the performance of their algorithms quickly and conveniently.

There is an abundance of well developed and well maintained tools which have been built expressly to grapple with the large amounts of data that have now become commonplace. These tools leverage novel algorithmic and system level techniques to to deal with the space and time bottlenecks that come with large datasets. Since the problem of backtesting algorithmic strategies is inherently a 'Big Data' shaped problem, we decided to use a combination of such tools in order to to build an efficient and flexible system for the task.

II. Related Works

We modeled our implementation of the backtesting engine with some other tools and works in mind. One of the prominent platforms that we sought to emulate is Quantopian [2]. The Quantopian platform allows in-browser implementation of trading algorithms in Python. It also allows the use of internal APIs as well as most external python modules for statistics, numerical algorithms etc. Although we have not extended our system to allow in-browser coding, we have also used python as the language of choice for trading algorithm implementation. Python is readable, flexible and popular, making it a good choice for algorithmic trading; a field where not everyone is well versed with computer programming. We also sought to emulate the charting and metric calculations performed by the Quantopian platform. The benchmark as well as the returns time series are plotted to provide a lucid understanding of the algorithm performance, without confusing the user with unnecessary detail. Also, important metrics such as the Sharpe Ratio, Maximum drawdown etc. are provided to the user for further insight.

III. System Overview

The main objectives in our implementation were to keep the platform efficient and easily extensible. In order to achieve these goals, we partitioned the system into independently functioning subsystems which we then integrated to realize the final backtesting engine. For the purpose of demonstration, we utilized the free historical pricing data from QuantQuote[3]. The dataset as well as the subsystems are described in detail in the following subsections.

A. QuantQuote dataset

We decided to use the QuantQuote free historical stock price data for the purpose of demonstration. This data consists of daily stock tick data for the 500 symbols that are listed on the Standard and Poor's 500 Index from 1998 to present. Although we used the dataset throughout the design and testing process, we structured the backtesting engine such that it is dataset agnostic. More specifically, we designed our subsystems so that they do not impose any tight constraints on input data formatting. Further descriptions of input data specifications are provided in the following subsection.

B. Hadoop Data Warehouse

Apache Hadoop is a distributed framework that is designed for the storage and processing of large datasets[1]. Hadoop is especially well suited to read-only, batch accessing of large amounts of data. Our system only requires the reading of financial time series data, without frequent writes or editing, making Hadoop a perfect choice for the data warehouse subsystem. We stored the finance symbol pricing data in the Hadoop Distributed File System (HDFS), which can be configured to run on any number of machines without a single point of failure. We structured the file system so that the time series data for each symbol were stored in a separate file. This simple, flat structure makes accessing the data extremely simple, since for a particular symbol, only a single file must be accessed. Each file was named after the symbol, to facilitate programatic access using symbol names directly, without the need for any lookup or translation. In the case that the data repository needs to be updated with newer pricing data, only a single file need be written to. This simple organization does not compromise on efficiency since Hadoop is designed to work well with a relatively small number of very large files, as opposed to small fragments of a larger dataset. A directory listing showing the file structure is shown in Figure 1.

Within each file, the time series were stored in a comma separated value (CSV) format, with time stamps and price values as the fields in each row. The CSV format is simple and widely used, making our system largely data agnostic. Any data in CSV format with the appropriate fields can be used with our data warehouse system since our engine does not impose and other restrictions on data formatting.

C. Spark Algorithm Processing Engine

Apache Spark is a tool that was developed for processing general large-scale data efficiently[4]. Spark come with support for multiple languages and platforms and an also integrate seamlessly with the Hadoop Distributed File System. We chose Spark to be the workhorse for the main trading algorithm processing. In order to leverage the features offered by Spark, we used pySpark, which is the Spark API in the Python scripting language. Our algorithm processing engine is responsible for detecting all the trading symbols that are involved in a user-defined trading strategy, and to retrieve the corresponding data from the Hadoop data store. After retrieving the required time series, the engine then runs the rules specified in the strategy against these series and computes the basic returns for the overall portfolio. Once the returns have been computed for the entire data, further portfolio performance metrics are computed. These include the mean return, standard deviation, maximum drawdown and Sharpe Ratio. The final returns time series and the metrics are then written to temporary output files on disk for use further downstream in the processing pipeline.

D. Javascript Server Back End

We sought to build a simple web app which could concisely present the results of algorithm backtesting to the user. In order to do this, we implemented a very basic server using the node.js JavaScript runtime.

E. Visualization

In order to visualize the performance of user defined trading algorithms viz. the benchmark, we built a JavaScript front end for our web application. In order to produce detailed and informative plots, we utilized the Highcharts and Highstock charting libraries.
The schematic in Figure 2. shows an outline of the system design.

IV. Algorithms

A. Map Reduce

The algorithmic paradigm we leveraged was the map reduce paradigm. We use a map reduce job to retrieve the required time slices of price time series from the Hadoop data warehouse. The data is stored in a distributed fashion across multiple machines hosting the Hadoop warehouse. The Map-Reduce Job simply accesses a file with a particular name from the Hadoop Distributed File System. In later versions, we might require only particular slices or windows of the temporal data and this can easily be achieved by augmenting the current Map Reduce job logic.

B. Mapping a function across a Spark RDD

We used the functional paradigm of mapping a function over an iterable object. Spark provides this functionality as a lazily evaluated map function that can be used with Redundant Distributed Datasets (RDDs). This map function was used for purposes such as normalizing the stock prices, processing the calculated returns for each strategy etc.

C. For-each sequential processing of an RDD

Finally, we used the foreach sequential processing of the stock price RDD in order to apply the actual trading strategy rules to the prices. This sequential processing is similar to a reduce operation, in that it must be the final action performed on an RDD and is sequential. Unlike the reduce operation, the foreach function does not return a single value but instead performs some action on each element.

V. Software Package Description

The software package that we have open sourced is written in Python. It contains scripts corresponding to each of the modules shown in Figure 2. The scripts are named in a self explanatory fashion. Our web application has an interface for charting the returns and the benchmark. A

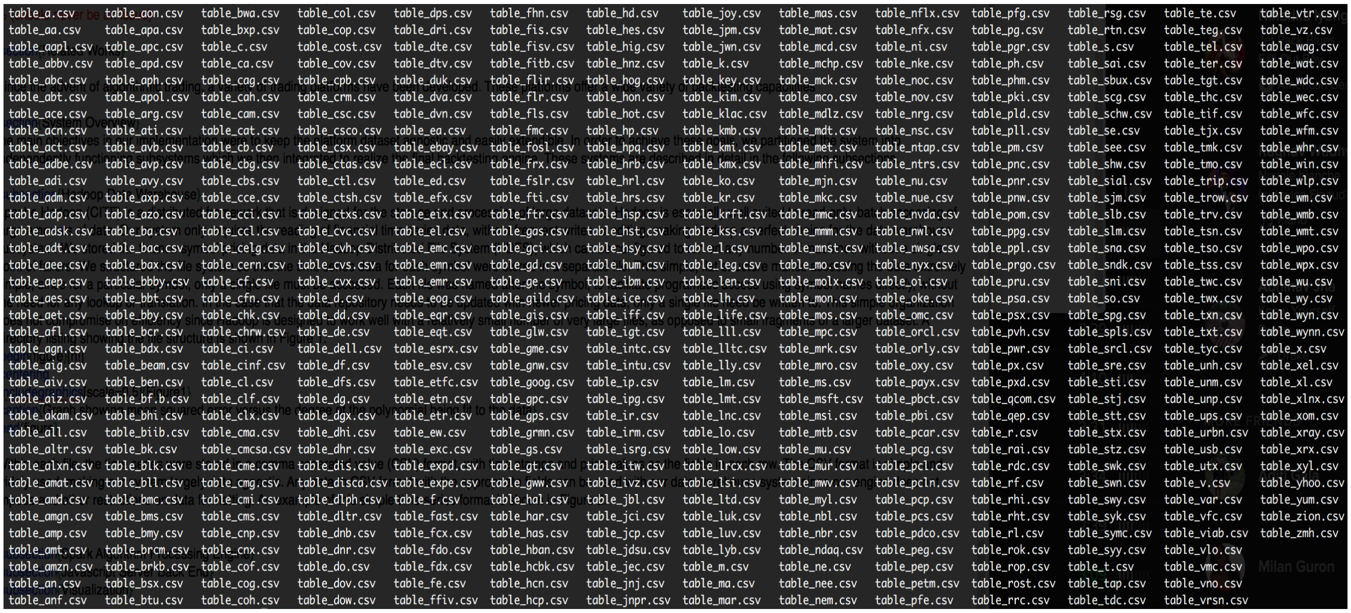

Figure 1. HDFS directory listing showing the flat directory structure for the symbol price data from the QuantQuote dataset.

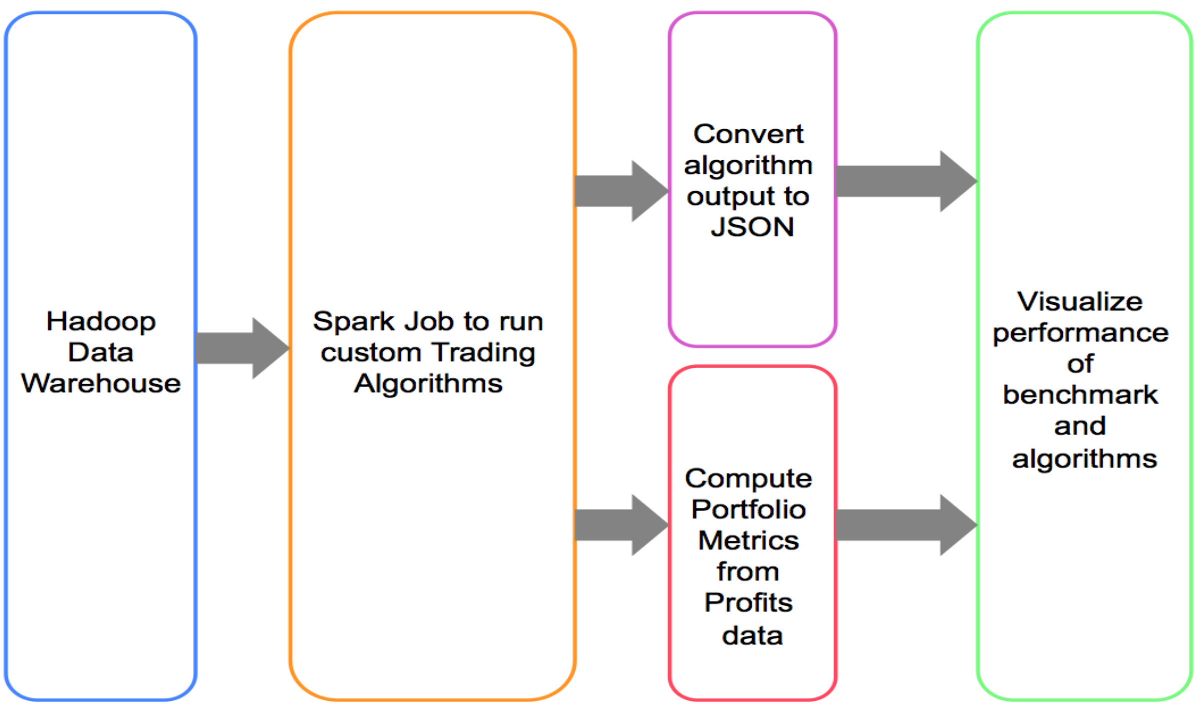

Figure 2. Schematic depicting the overall system organization

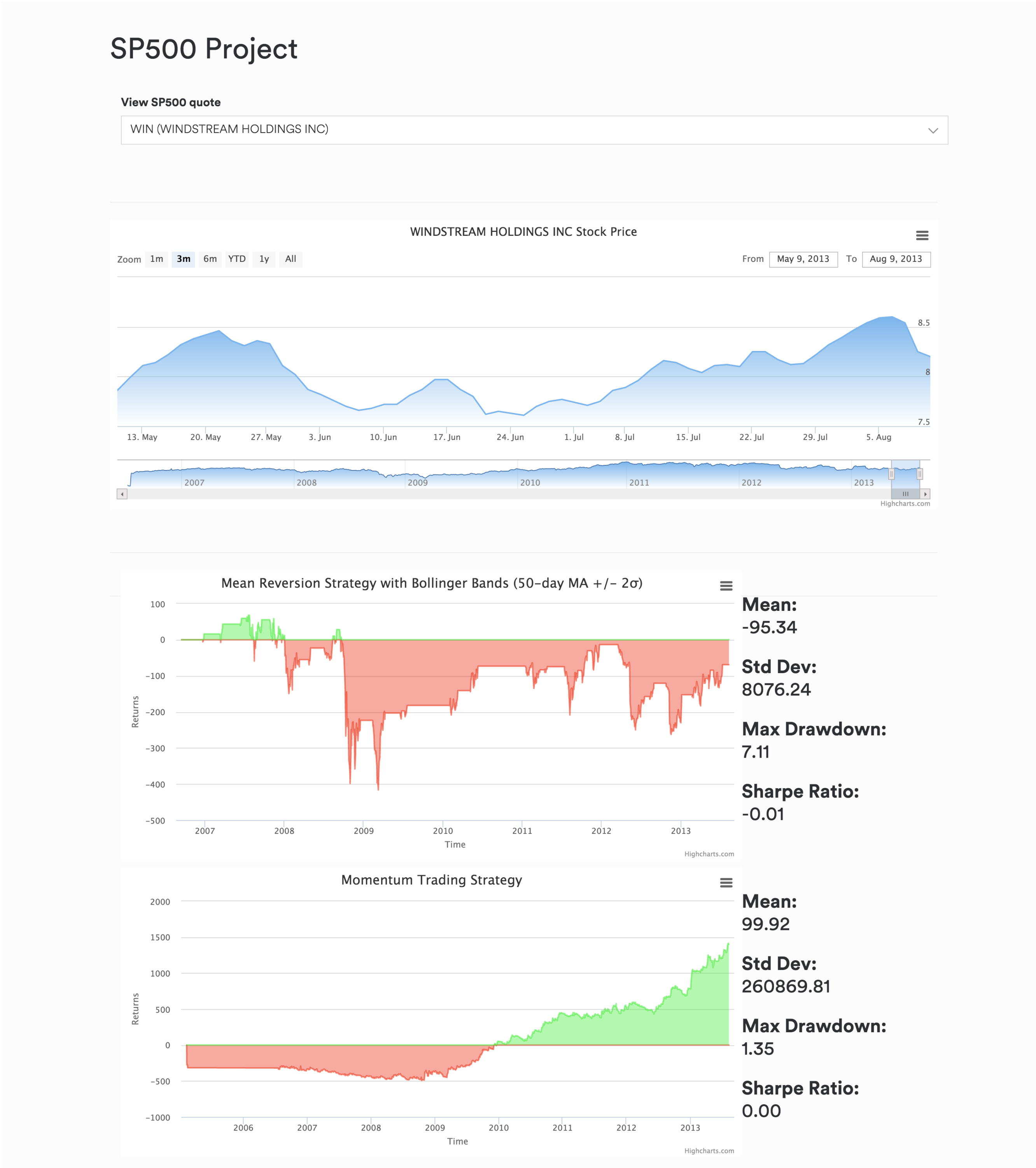

Figure 3. Image of the entire User Interface

screenshot of the overall UI for our app is shown in Figure 3.

The dropdown box at the top of the page enables the user to select a single stock symbol that is to be traded using the user defined strategies. The first chart at the top is the graph depicting the stock price time series. Below this first chart are the charts for all the user defined strategies loaded onto the engine. We used two specific strategies for testing our engine. These strategies are shown and described in more detail in the next section.

VI. Experiment Results

We a used two distinct trading strategies to test our trading engine. These are described below:

A. *Mean Reversion with Bollinger Bands*

The mean reversion strategy assumes that a stock price can be modeled as a mean reverting stochastic process. Every day, a moving average and standard deviation is calculated over the prices from the last five days. The bands two standard deviations from the mean are used to generate buy and sell signals depending on whether the stock prices crosses these thresholds. This strategy worked well on our engine and was easy to implement within our framework. A screenshot showing the returns for this algorithm are shown in Figure 4.

B. *Momentum Trading*

The momentum trading strategy attempts to ride trends in the stock price. If a decreasing trend is noticed, the stock is deemed cold and sold. In the opposite case, the stock is deemed hot and is bought. This strategy shows staggeringly large returns on most stock symbols. This strategy was also extremely easy to implement in our framework and worked in near real time. Figure 5. shows the return chart for this strategy.

VII. Conclusion

We were able to implement a complete trading platform based on Apache Hadoop and Apache Spark. We were able to test popular real world trading algorithms and achieve visual as well as numerical insight into their performance on any of 500 stock symbols. We plan to extend the features of our system in the future so as to build a more flexible and universal securities trading engine. Some ideas we have are as follows

A. *Multiple security types*

There are many different sorts of financial instruments that can be traded algorithmically, such as options, bonds etc and we wish to enable trading of these on our engine.

B. *Live testing*

We would like to enable the testing of algorithms in live, real-time stock tick data

C. *Custom visualizations*

We would like to allow uses to plot custom indicators and series on the charts and also to extend charting to portfolios with more than a single stock.

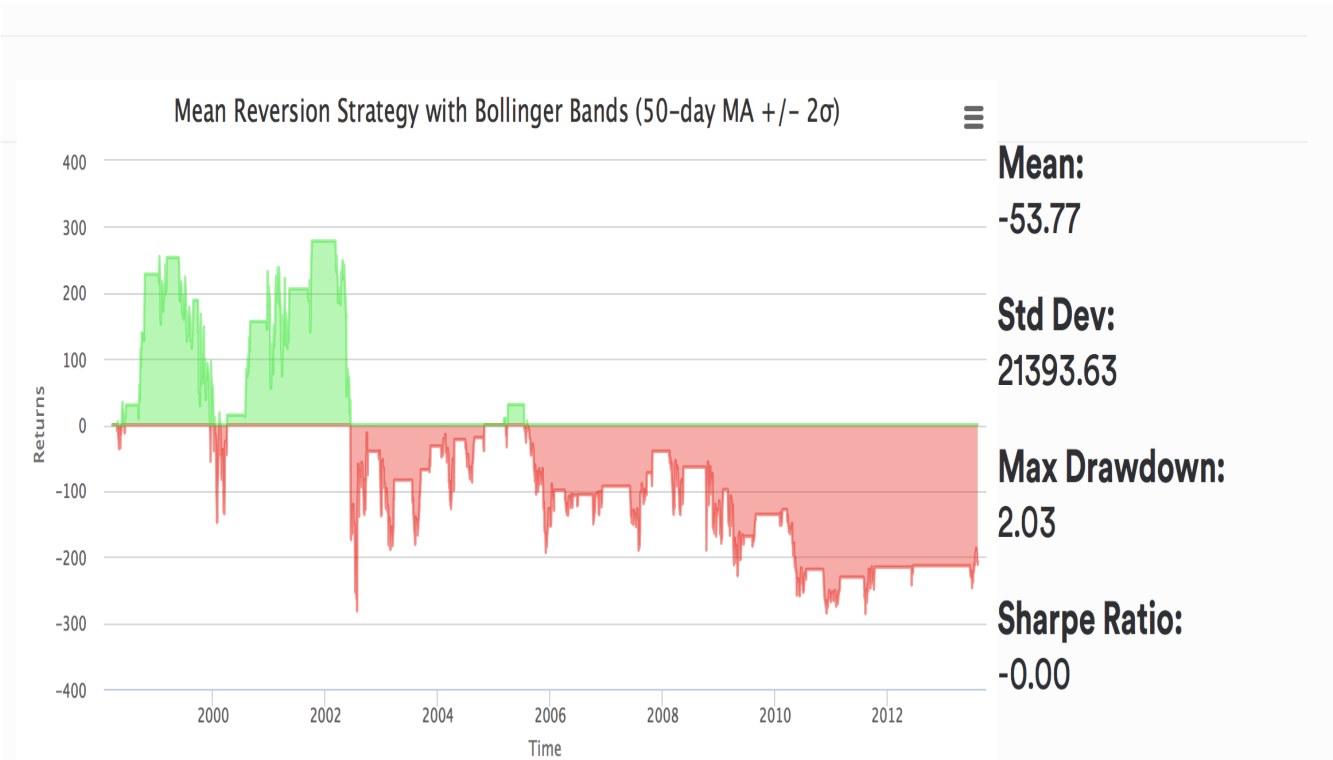

Figure 4. Returns from the Mean Reversion Trading Algorithm

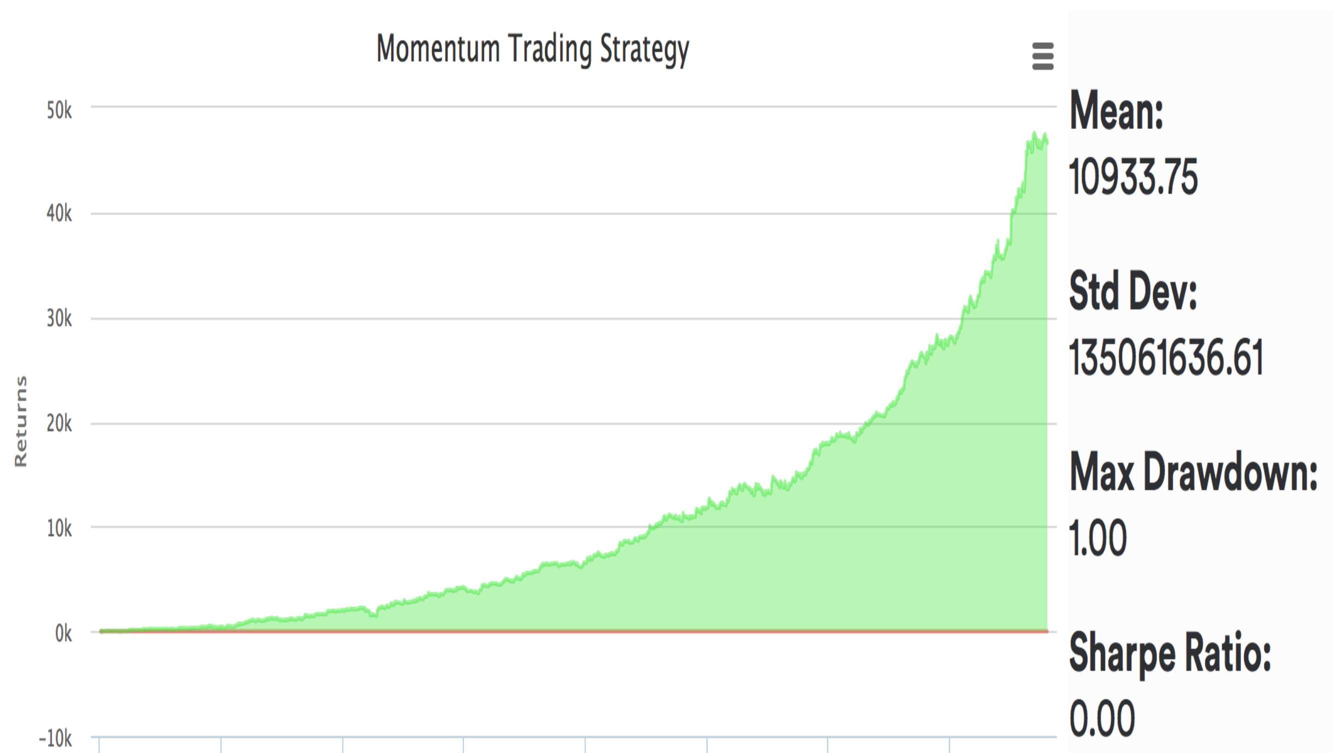

Figure 5. Returns from the Momentum Trading Algorithm

Appendix

References

[1] Apache Hadoop (http://hadoop.apache.org)

[2] Quantopian (https://www.quantopian.com)

[3] QuantQuote (https://quantquote.com)

[4] Apache Spark (http://spark.apache.org)