Genetic Programming:

Predicting Stock Closing Values a Day in Advance

**Overview**

For my final project I implemented a genetic programming algorithm in LISP to predict stock prices. More specifically, the objective of the program is to generate a function capable of predicting the closing price of a given stock for the day after the day up to which it has current data for. For my implementation chose to use the price of General Electric (GE) for three reasons. Historical data was easily obtained back to 1962, it is traded on the New York Stock Exchange, and they work primarily in industrial goods which tends to reflect the health of the market as a whole without being overly volatile.

**Problem and Approach**

Before I go on to describe the problem and my approach in more detail, I want to mention that my principle guide for this project was the book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* by John R. Koza. The book contains a detailed description of genetic programming techniques, and an appendix with a set of portable functions in LISP that can be used for any basic genetic programming system.

I represented individuals in the population with a struct that contains fields for the function the individual represents (the function to predict the stock value) and all of the fitness statistics for the individual. Individual functions are created by taking elements from a terminal set and a function set (provided by default in my implementation, but they can also be supplied by a user if desired) and consing them into a list which can then be evaluated with the LISP EVAL function. The initial population is essentially random. However, based on advice from Koza's book I implemented a generation method called ramped-half-and-half. Using this method, the functions for the initial population is created in a depth-limited fashion with half of the trees being "full" trees and half being "grown." Full trees only have terminals at the leaf nodes while grown trees have a random chance of choosing an element from the combined set of terminals and functions at any node. This produces trees with interesting shapes and sizes in the initial set.

Because stock prices are theorized to be essentially random, it's very difficult to know what variables (terminals) will have explanatory power on the closing price for the next day. The terminals I chose for my system were: the high, low, open, close, adjusted close, and trading volume from the "current" day, as well as the a 7-day average of the closing price, a 30-day average of the closing price, and the 3-day slope of the closing price. These were fairly easy to obtain and calculate and I thought that they were the most likely to have some explanatory power on the closing price. For the function set I first used just the basic arithmetic operators (+, -, \*, /) and then experimentally expanded the set to see what would happen as I became more familiar with the program.

Once the initial random population is created the fitness of each individual is assessed and the population is sorted based on fitness. Overall fitness for an individual is based on a number of fitness cases which are, in this case, separate days of trading. So, using ten fitness cases would use trading data from ten different days to test the function and calculate a fitness based on how the algorithm performs on all ten days. For the stock prediction problem I decided that the best measure of standardized fitness (which is a fitness measure set so that a smaller value is always better) is the average percent error over all fitness cases. This means that an individual with a good standardized fitness will (theoretically) have a good return on investment. The population actually sorted according to a normalized fitness value which ranges between one and zero and sums to one for all of the individuals in the population. This is because it is easier to select an individual probabilistically based on the normalized fitness. The normalized fitness is calculated from the standardized fitness as follows:

The function that evaluates individuals' fitness also calculates a statistic called hits. The number of hits for an individual is the number of fitness cases for which its standardized fitness is below a given threshold. For this problem, I specified a hit as a standardized fitness less than or equal to one percent.

Once the population is sorted, individuals are selected for breeding based on fitness. The methods of reproduction in my system are: fitness-proportionate, crossover, and mutation. The fitness-proportionate individuals are simply passed into the next generation. Individuals selected for crossover have a node randomly selected, and then the subtree starting at that node is swapped between two selected individuals (with reselection allowed). Individuals selected for crossover are also broken down internally to those with crossover performed at internal points and those where crossover can be performed at any point. The fraction allowed to crossover at any point is kept small in order to reduce the amount of crossover operations that involve simply swapping leaves. After fitness-proportionate and crossover selections are made the remaining individuals are created via mutation. Mutation involves replacing a randomly selected node with a subtree generated using the grow method. The fraction of mutants is usually small because the crossover operation performs a process similar to mutation on its own. However, I found that increasing the fraction of individuals created through mutation sometimes led to better results.

Once the new population has been bred it is sorted and the process starts over again. This continues until either the maximum number of generations has been reached or the termination criteria are reached. I set the termination criteria (in addition to the maximum number of generations) to be fairly strict (the best of the generation individual having a hit on every fitness case) because I was more interested in how close an individual could get to the goal than in reducing run times.

Beyond the kernel, I had to write a set of functions that were specific to the problem in order to give the kernel the information it needed to run. In this file I imported the data for the fitness cases from a text file, using a BASH script to parse the space-delimited data and add parentheses at the beginning and end of each line so that each file could be read as a LISP S-expression using the READ function. The data is then added into an array which holds the fitness cases. This is also where I wrote the function to calculate the fitness of each individual. As a final touch to the project I also added the capability to produce a running log file or a verbose mode that will print out a report on each generation as the program is running.

**Challenges**

The most difficult parts of creating the genetic programming system, for me, were understanding the code in the LISP kernel from Koza's book and creating the protected functions for the function set. Because the program randomly generates functions that may evaluate to virtually any value, the function set can't use (only) the built-in LISP operators. Special "protected" functions have to be written in order to filter input and handle errors like division by zero and floating-point-overflows and underflows. The most difficult part of the entire process was trying to get the EXPT function to work. And even though I did have some successful runs with it in the function set, I never managed to get it to work consistently.

**Testing and Results**

In order to test my algorithm I created a testing function, TEST-GP-SYSTEM, which allows the user to specify almost every aspect of the system (details given in the readme file). The test file begins by opening a file containing the entire range of available dates (from 1/2/1962-5/8/2013) into a hash table so that the user can call the tester function using a date string rather than having to know the line number of the file (I’m actually very proud of both this and the fitness cases array mentioned earlier, as they let me combine my BASH scripting skills with LISP in a way that actually helped increase the usability of the program).

The best test case to show a single run is with 30 fitness cases (around a month and a half of sequential days), 200 generations, and a population size of 1000. A simple run of the algorithm on any valid date in the range with 30 fitness cases will produce a function with an average percent error (standardized fitness) of roughly 1.3%. This means that the function produced was had an average prediction value within 1.3% of the closing price for the next day for 30 straight days.[[1]](#footnote-1)

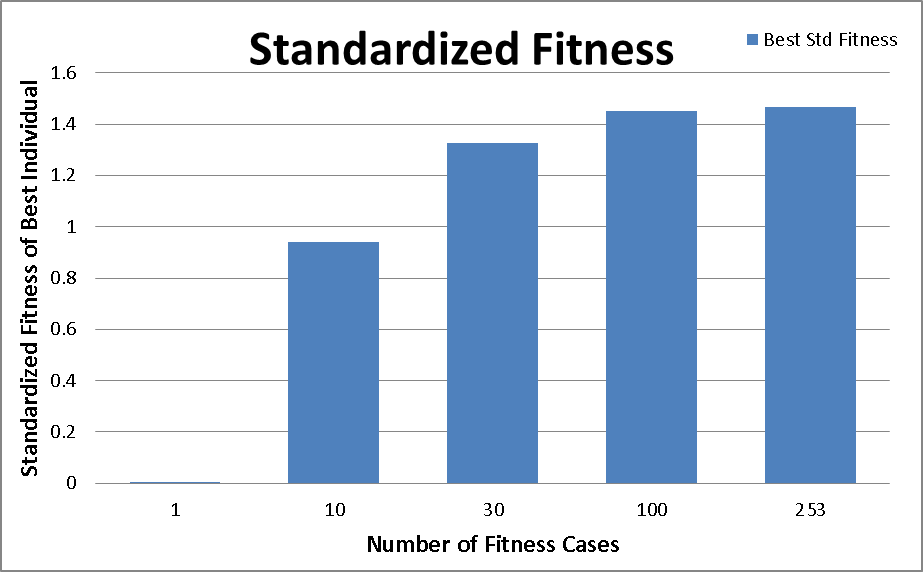
Figure 1 below shows how the average fitness of subsequent generations evolves over run. As can be seen from the figure, the trend follows a quickly decaying exponential which means that the value converges very quickly.

Figure 1: The Average Standardized fitness for 30 fitness cases over 200 generations.[[2]](#footnote-2)

For the run shown above, the best program in the run was found on the 10th generation and had an average standardized fitness of 1.33% . The function produced was:

(+ \*STOCK-CLOSE\* (RSIN (\* \*STOCK-HIGH\* (+ \*STOCK-CLOSE\* \*STOCK-3-SLOPE\*))))

This is far from an intuitive relationship, and it is unlikely that a human could have easily come up with this. And while this is very interesting as a preliminary result, I ran many other tests to explore how the input parameters affect the run of the system. I don’t have time to analyze each test in detail but my results are presented with brief explanations below.



**Effect of the Number of Fitness Cases**

The nest parameter I investigated was the number of fitness cases. As can be seen in Figure 2, the effects of diminishing returns appear very quickly.

**Effect of the Fraction of Mutants**

Upon seeing how quickly the population converged on a solution (as seen in Figure 1) I also became interested in how the fraction of mutants allowed in the population effects the results. So I repeated the test from Figure 1, changing only the fraction of individuals selected based on fitness-proportionate selection. In the original test, this was 0.1, I also tried with values of 0.05 and 0.01, which correspond to a mutant fraction of 0.05 and 0.09, respectively. Unfortunately, the introduction of more mutants makes it very difficult to show any kind of evolution in the populations over time as can be seen in Figure 3. I put in an exponential fit trend-line in orange to show that there still some evolution, but it’s a pretty poor fit.

Figure 2: Effect of the number of fitness cases on best individual.

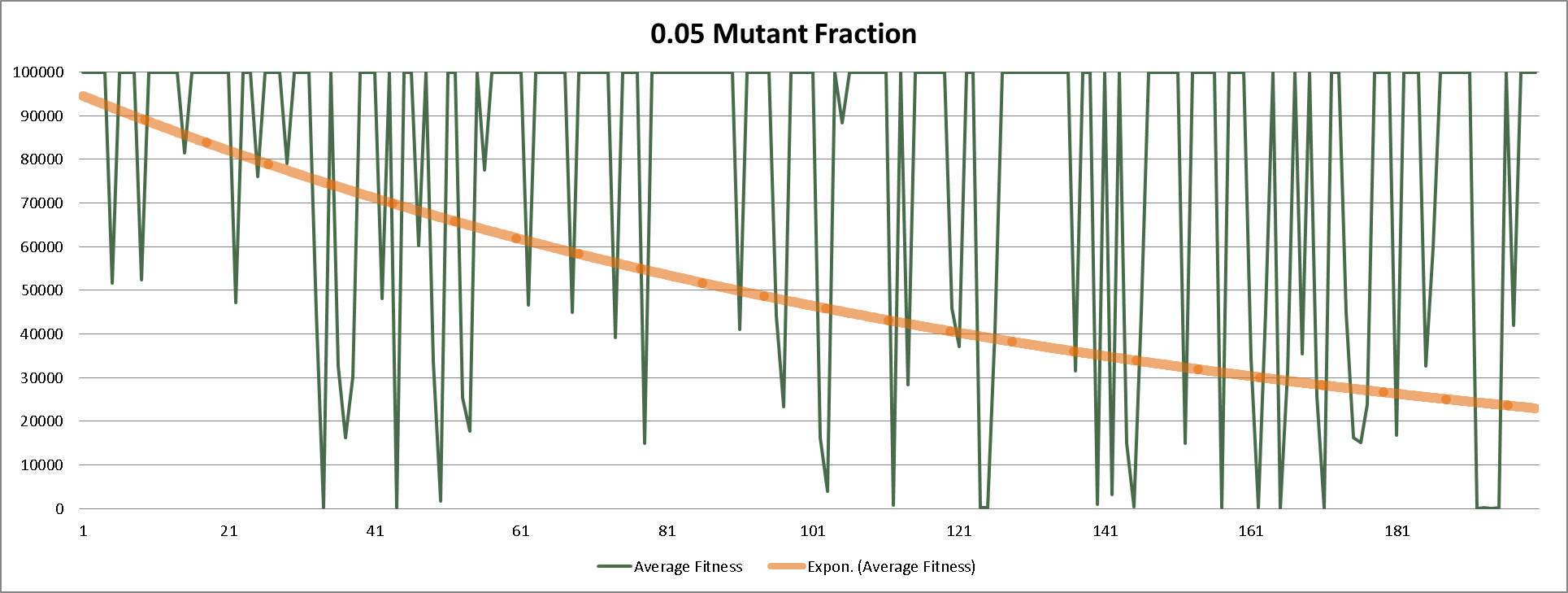
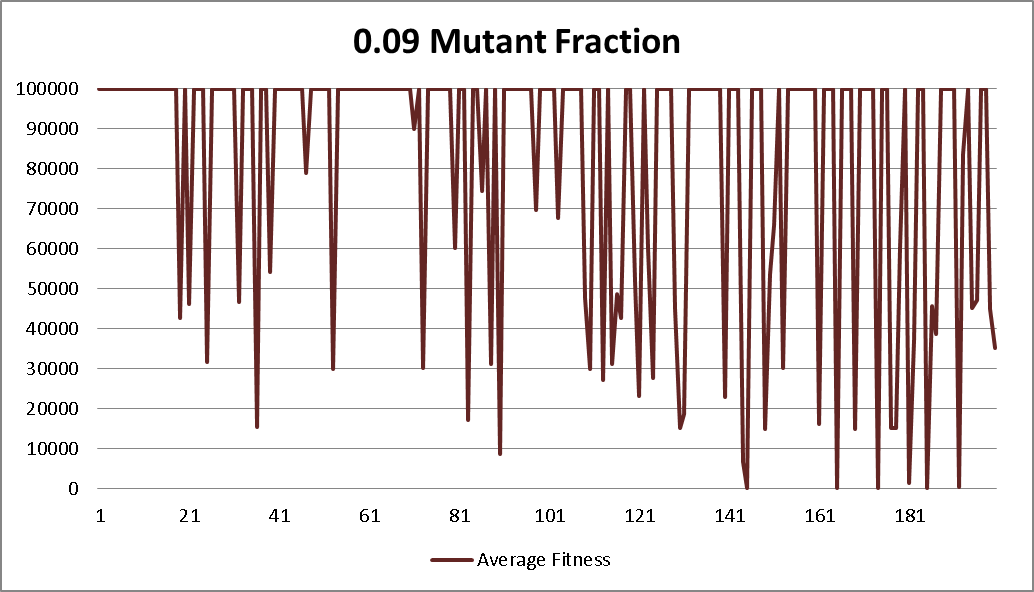


Figure 3: Evolution of average fitness in a population of 1000 over 200 generations with a 0.05 chance of mutant production.

However, even though the evolution over the run of the program is difficult to show, the results show noticeable improvement. The run in on the right was one of the most successful I have seen to date. The table below shows these results.

|  |  |
| --- | --- |
| **Mutant Fraction** | **Fitness of Best Individual** |
| 0.1 | 1.3267 |
| 0.05 | 1.1347 |
| 0.09 | 1.0613 |

**Effect of the composition of the Function Set**

I also wrote a script to loop through the default function set (+ - \* % srt rlog rsin rsinh rtan rexp), adding a new function on every run. This also showed a distinct point of diminishing returns.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  | | --- | --- | | **Run** | **Best Std Fitness** | | 1 | 1.749535 | | 2 | 1.549782 | | 3 | 1.5172652 | | 4 | 1.3191286 | | 5 | 1.3925176 | | 6 | 1.2791356 | | 7 | 1.109904 | | 8 | 1.1366882 | | 9 | 1.1899785 | | 10 | 1.3162029 | |

|  |  |
| --- | --- |
| **Run** | **Best Std Fitness** |
| 1 | 1.86172 |
| 2 | 1.501653 |
| 3 | 1.342195 |
| 4 | 1.167182 |
| 5 | 1.161134 |
| 6 | 1.17631 |
| 7 | 1.158321 |
| 8 | 1.127607 |
| 9 | 1.154732 |

**Effect of the composition of the Terminal Set**

I also wrote a similar script for the terminal set and again observed a similar result.

I believe that both of these results point to a sufficiency argument for the respective set, or that some of the elements are unnecessary. However, I don’t believe that the testing I have done has been exhaustive enough to make an informed conclusion. I also found that the magnitude of the standardized fitness was highly correlated with the base price of the stock. Results for more recent periods where the value of the stock was lower gave more accurate results (even though I used an arbitrary date for testing in this write-up). This may mean that my program has a future in penny stocks.

**Other Testing**

In addition to testing to work on the stock prediction problem, I also wrote a script to perform a stress test on the algorithm to see if I could break it. The stress test ran the system 100 times with a population of 1000 and 200 generations and the system prevailed!

**Conclusions**

Overall, I had a great time writing the program and learning about genetic programming, the system performs well for a large host of (practical) test cases and produces desirable (if not profitable) results.

I think the most important addition that I would like to add that I didn’t have time to finish was adding the exponential function to the function set. Even though it often produced results that caused the fitness measures to explode I think that there was a potential for it to produce some good results. If I had more time I would work on refining the set of protected functions and creating more automated testing functions to perform regression analysis on the outputs so that, maybe I could eventually modify the program to create some kind of trading strategy.

**Sources**

Artificial Intelligence, A Modern Approach. Chapter 4.1

Koza, John. Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge MA. 1998.

Black, Fisher and Scholes, Myron. The Pricing of Options and Corporate Liabilities. *The Journal of Political Economy*, Vol 81 Issue 3 (May-Jun 1973), 637-654

Marczyk, Adam. Genetic Algorithms and Evolutionary Computation. The Talk Origins Archive 2004 Available Online: <http://www.talkorigins.org/faqs/genalg/genalg.html>

Grace, Bruce, K. Black-Scholes Option Pricing Via Genetic Algorithms. *Applied Economics Letters*, 7:2, 129-132

G., Anuradha, Dimple, Borha. Optimizing and Analyzing Returns in Commodity Trading Using Genetic Algorithm, Simulated Annealing, and a Novel Algorithm (GaSa). *International Journal of Emerging Technology and Advanced Engineering*, 2:12 December 2012.

Appendix

**Sample Output**

This is the output of the run used for the 0.09 mutant fraction data:

CL-USER(3): (test-gp-system "1/11/1999" 30 :max-gens 200 :pop-size 1000

:log? t :log-file "test-mutant-09.log")

Parameters used for this run.

=================================================

Number of fitness cases: 30

Maximum number of Generations: 200

Size of Population: 1000

Maximum depth of new individuals: 6

Maximum depth of new subtrees for mutants: 6

Maximum depth of individuals after crossover: 30

Fitness-proportionate reproduction fraction: 0.01

Crossover at any point fraction: 0.2

Crossover at function points fraction: 0.7

Selection method: FITNESS-PROPORTIONATE

Generation method: RAMPED

Randomizer seed: 1.0

Best of run individual program:

Generation: 199

Hits: 16

Std Fit Vector: #(1.004509 0.21916962 1.1156006 2.9924774 0.24276733 0.29356384

0.98267365 0.47953796 1.7712784 0.5027695 1.4998779 1.0102692

3.5054932 0.62890625 0.0015945435 1.7795029 1.1417389 0.50626373

1.1151428 0.9693451 1.6594162 1.3800278 0.7935562 0.20339966

1.1345291 0.11859131 2.7520828 0.3695984 1.5282364 0.2523346)

Std Fit (sum): 31.954254

Std Fit Avg %: 1.0612613

Function Nodes: 185

Function Depth: 25

Function: (+ (RSIN (+ (\* (+ (RSIN (+ (\* (- (+ (SRT (+ (RSIN (RSIN \*STOCK-CLOSE\*)) \*STOCK-CLOSE\*)) \*STOCK-30-DAY\*)

(- (+ (+ (RSIN \*STOCK-CLOSE\*) \*STOCK-CLOSE\*) \*STOCK-CLOSE\*)

(+ (RSIN (RSIN (+ (RSIN (+ \*STOCK-7-DAY\*

(+ (+ \*STOCK-CLOSE\*

(RSIN (% (+ (RSIN \*STOCK-7-DAY\*) \*STOCK-3-SLOPE\*) (RSIN (- \*STOCK-OPEN\* \*STOCK-ADJ-CLOSE\*)))))

(+ (+ (+ (RSIN (% (- \*STOCK-OPEN\* \*STOCK-HIGH\*) (RLOG (RSIN \*STOCK-CLOSE\*))))

(% (RSIN \*STOCK-ADJ-CLOSE\*) \*STOCK-LOW\*))

\*STOCK-CLOSE\*)

\*STOCK-3-SLOPE\*))))

(RLOG \*STOCK-OPEN\*))))

(RLOG \*STOCK-OPEN\*))))

(- (+ (+ (SRT (% \*STOCK-CLOSE\* (RSIN (+ (SRT \*STOCK-30-DAY\*) (RLOG (- (SRT \*STOCK-VOLUME\*) \*STOCK-LOW\*))))))

(- (RLOG (RLOG (+ \*STOCK-7-DAY\* \*STOCK-CLOSE\*))) \*STOCK-30-DAY\*))

(+ (+ (\* (+ (RSIN (+ (SRT \*STOCK-7-DAY\*) \*STOCK-HIGH\*))

(\* \*STOCK-ADJ-CLOSE\* (\* \*STOCK-LOW\* (+ (RSIN (RSIN (% \*STOCK-OPEN\* (RSIN \*STOCK-3-SLOPE\*)))) \*STOCK-CLOSE\*))))

(% (+ (RSIN (% (RSIN \*STOCK-LOW\*) (+ (RLOG \*STOCK-30-DAY\*) \*STOCK-ADJ-CLOSE\*))) \*STOCK-CLOSE\*)

(+ (RSIN (SRT (SRT \*STOCK-3-SLOPE\*))) (SRT \*STOCK-7-DAY\*))))

\*STOCK-VOLUME\*)

\*STOCK-3-SLOPE\*))

\*STOCK-3-SLOPE\*))

\*STOCK-CLOSE\*))

(+ \*STOCK-CLOSE\*

(RLOG (RLOG (\* \*STOCK-30-DAY\*

(+ \*STOCK-3-SLOPE\*

(RSIN (+ (+ (+ (RSIN (- \*STOCK-3-SLOPE\* (\* \*STOCK-CLOSE\* (RLOG (RSIN (- \*STOCK-CLOSE\* \*STOCK-30-DAY\*))))))

(- (% (RSIN (RLOG \*STOCK-VOLUME\*)) \*STOCK-3-SLOPE\*)

(+ (RSIN (RSIN (+ (+ \*STOCK-3-SLOPE\* \*STOCK-CLOSE\*) (RLOG \*STOCK-OPEN\*)))) (RLOG \*STOCK-OPEN\*))))

\*STOCK-3-SLOPE\*)

\*STOCK-30-DAY\*))))))))

(RSIN \*STOCK-30-DAY\*))

(+ \*STOCK-3-SLOPE\* (+ (SRT (% \*STOCK-CLOSE\* (RSIN (+ \*STOCK-HIGH\* \*STOCK-HIGH\*)))) \*STOCK-CLOSE\*))))

\*STOCK-CLOSE\*)

1. For all of the tests in this write up I arbitrarily chose a starting date of 1/11/1999. [↑](#footnote-ref-1)
2. Please note that the first four values have been removed from this graph because they were so large that they overshadowed the rest of the data. [↑](#footnote-ref-2)