Background Report: The Algos of Wall Street

Live monitoring of stocks prices and executing trades on price targets

Amelia Milne
University of British Columbia
amelia.milne@gmail.com

Seungyup (Steven) Lee
University of British Columbia
syl24@live.com

Karan Grover
University of British Columbia
karan.kg.grover@gmail.com

Pahul Panesar
University of British Columbia
pahulpn@gmail.com

Josh Cheung
University of British Columbia
se77enn@gmail.com

ABSTRACT

This paper proposes the usage of Rust to implement a stock trading algorithm.

KEYWORDS

Rust, Stock, Algorithmic trading, Generic, Arrowhead, Liquidity, Detrending

1 INTRODUCTION

In 2015, Mozilla Research released the first stable version of a new programming language named Rust. Many features make this new language attractive for the scientific community. It is open source and it guarantees memory safety while offering zero-cost abstraction. We choose this topic because we were interested in how we would be able to use a functional language like Scheme, Rust or Haskell to create an algorithm that buys and sells stocks based on a researched algorithm. We choose Rust as our language of choice and will be exploring the language to create an application that will execute trades based off of a trading pattern we find through research. We have identified some potential algorithms below and further discuss why we choose Rust.

2 ALGORITHMS

2.1 Genetic Algorithm

The following is based on the paper *Genetic Algorithm as a Tool for Stock Market Modelling* by Urszula Markowska-Kaczmar, Halina Kwasnicka, and Marcin Szczepkowski [8].

To summarize the Genetic (one) algorithm, it uses a set of agents (simulated stock trader), to generate similar behaviors as of the real stock market. It collects information within a virtual setting emulating the real-world stock market.

The set of agent simulate investors working sessions (K); a predetermined number of times the simulation runs. This is compared to the performance of the real market, and agents are evaluated for the performance. The same is performed over multiple times, with successful candidates being noted via amount of cash, register of desired investments, positive patterns, and negative patterns. The agents are then subjected to genetic operations such as mutations and crossover. The operation continues until a stop condition is met.

Mutation within the algorithm means that a number of agents within the set will be replaced with randomly generated ones. Crossover within the algorithm means that a new set of agents will be created by breaking existing sets into two and taking pieces of them at random. This will ensure that there is a good pool of diverse agent sets.

The sets of agents, referred to as individuals are evaluated by the fitness function R; by taking the difference of correlation and difference between stock quotations generated by a model to the real-world quotations in (K) sessions. Basically meaning that prediction within the model is compared to the real-world performance, trying to predict the stock performance.

The Virtual Market, the system that contains agents, keeps track of transactions, clear transactions, update stock prices, and provides agents with price information. Min/Max prices are updated as well. It is vital that the VM is updated in tandem with the real-world market to track agent performance.

Within the VM, agent bids are collected and sorted. The VM proceeds to search for stocks that have the same bids as the agents within the system. It then matches sales and purchase bids, then it updates the current stock price, creating a correlation between predictions made and real-world stock performance.

At the end of K sessions, individuals/sets of agents are evaluated, and generates a prediction model for the next steps.

It is important to note that with the Genetic (one) algorithm model, the less available stocks the VM tracks for simulation, the better the average of prediction. This is because the lesser the amount of stocks to replay quotations the more likely it can predict accurately.

This model of algorithm would be useful to track and predict a few selected amount of stocks with good accuracy. However, it would fail to correctly predict accurately a diverse portfolio.

2.2 Particle Swarm Optimization

The following research is based on the paper *Novel Binary Particle Swarm Optimization* by Mojtaba Ahmadieh Khanesar, Hassan Tavakoli, Mohammad Teshnehlab and Mahdi Aliyari Shoorehdeli [6].

The particle swarm optimization was originally introduced by Eberhart and Kennedy (Ebarhart, Kennedy, 1995; Kennedy, Eberhart, 1995; Ebarhart, Kennedy, 2001). It is a population based search algorithm based on the simulation of behaviors of groups. Each individual within a swarm is represented by a vector in a multidimensional search space. That vector also has a vector inside which determines the next movements called velocity vector. The PSO

will then process how to update each velocity vector based on the behavior shown so far and global swarm position.

The PSO iterates over a fixed number of times or until a minimum number of error cases has been surpassed.

Binary PSO, each particle within the swarm is represented via a binary value. Thus, each particle's value can be mutated over time based on swarm behaviors.

The algorithm assumes that our search space is on a dimension, and a search can happen within the swarm, for a particle, which is positioned in that dimension as a vector $X = (x1, x2, ...xd)$. The velocity (the predicted next step) is also a vector within that $X, V = (v1, v2...vd)$. For that particle, it has two other vectors that represent best visited and best explored. $P - best(p1, p2...pd)$ and $Pg - visited(pv1, pv2...pvd)$. All are relative traits to stock traders. So the position of particles and its velocity can be explained in this equation:

$$V_i(t+1) = wV_i(t) + c_1\varphi_1(P_{ibest} - x_i) + c_2\varphi_2(P_{gbest} - x_i)$$

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

$C1$ and $C2$ are positive constants, and $x_i(t+1) = x_i(t) + v_i(t+1)$ are random variables with uniformity between 0 and 1. P-best and P-visited are updated every iteration when individual particle has found a better position or when the whole swarm did.

The best feature about the PSO is the social interaction of particles, particles can learn from each other and based on that the whole swarm can become better at what the swarm intends to do. This can be directly applied to the stock market. The algorithm can be updated with stock market movement of individual buy and sell bids. Once there is a movement, the swarm will intuitively start to bid on similar stocks or the same stock. Thus, algorithm will predict the swarm movement.

2.3 Bacterial Foraging Optimization

The following research is based on the paper *Efficient prediction of stock market indices using adaptive bacterial foraging optimization (ABFO) and BFO based techniques* by Ritanjali Majhi , G. Panda, Babita Majhi , G. Sahoo [7].

Bacterial foraging optimization or BFO was purposed by Passino (2005), inspired by the pattern of bacterial foraging behavior. The system consists of four sequential mechanism, named chemotaxis, swarming, reproduction and elimination-dispersal.

2.3.1 Chemotaxis. In BFO, a unit walk with random direction represents a tumble, and unit walk in same direction indicates a run. This all means that chemotaxis dictate location of a bacteria, with in consideration of loss/gain; reproduction, elimination, and dispersal. It is represented in this equation

$$\theta^i(j+1, k, l) = \theta^i(j, k, l) + C(i)\phi(j)$$

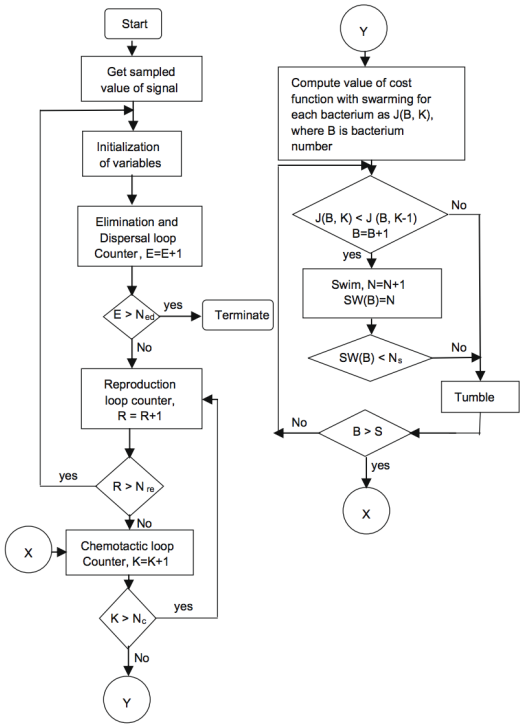
where $\theta(j, k, l)$ denotes the location of i bacterium, at j chemotatic, k reproductive, l elimination, and dispersal step. $C(i)$ is the length of unit walk, and $\phi(j)$ is the direction in j steps. This is an mirror image of agents from genetic algorithm. Bacteria representing particles, thus each movement can be referring to stock trades, cash, and desire to invest.

2.3.2 Swarming. Modelled after a bacteria's ability cluster or disperse when needed with other bacteria, much like discussed in article on top about particle swarm optimization, BFO also has the same function. The mathematical equation is the same as the article above. Same reasoning can be applied to the swarming as PSO's reasoning in terms of stock trades.

2.3.3 Reproduction. After the N chemotactic steps, a stop condition, a step called reproduction occurs. This is the same step in genetic algorithm's crossover stage. It halves the bacteria and creates a new one with a another half from a another bacteria.

2.3.4 Elimination and dispersal. Some bacteria can become trapped at a certain position. These bacteria are to die off as bacteria needs others to survive, thus eliminated. In terms of stock trade, the algorithm would not keep track of localized stock performance, that may have resulted in no trade volume.

The dispersion operation is triggered after a certain number of reproduction has completed. A bacterium is chosen by a preset probability, and moved to a another position. This helps prevent sudden elimination but could create optimization disturbances. Here is a graphic representation of the algorithm:



2.3.5 Application of BFO. An assumption is made about the stock market before applying BFO. The stock market is to be an adaptive linear combiner with parallel inputs. It boils down to a simulation of the real stock market, with filter inserted between the inputs, to distill an input pattern. A population of a simulation to chosen at random and each are to walk, swarm, reproduce, and die based on inputs. The BFO is a happy mix of genetic algorithm and PSO, as it takes both into consideration and creates a hybrid formula.

2.4 K-Means Clustering Algorithm

The following research is based on the paper *Algorithm AS 136: A K-Means Clustering Algorithm* J. A. Hartigan and M. A. Wong [5].

The aim of the K-means algorithm is to divide M points in N dimensions into K clusters so that the within-cluster sum of squares is minimized. It is not practical to require that the solution has minimal sum of squares against all partitions, except when M, N are small and K = 2. We seek instead "local" optima, solutions such that no movement of a point from one cluster to another will reduce the within-cluster sum of square.

2.5 Mean Reversion Models

The following research is based on the paper *Algorithmic Trading 101 Lesson 2: Data, Strategy Design, and Mean Reversion* by The Ocean Trading [9].

This algorithm is based off of the idea that there is a core stability to the current price of a stock and that the stock price fluctuates randomly in the short term around this core trend. A buy signal is indicated when the 30 day moving average dips below the 90 day moving average. A sell signal is triggered by the opposite effect.

An alternative way to represent this is to say that a stock is purchased when the current price is below what the core stock price is. The core stock price is determined by using moving averages to see what price the stock price has been fluctuating between.

3 ALGORITHM SELECTED

We have selected the Mean Reversions Model as our algorithm to implement. This algorithm is core to the idea that the stock price fluctuates around a core trend and allows for users to benefit from this fluctuation in the short term. This algorithm is still core to the daily purchases and sells of trades around the world.

4 RUST AND THE PROGRAM

4.1 Intro to Rust

Rust is a statically and strongly typed systems programming language, designed for low-level applications such as the design of operating and embedded systems. As Rust is statically typed, all its types are known at compile time versus dynamically typed languages, which do not enforce type checking at compilation (E.g. JavaScript, Python). One of the advantages of static typing is that the compiler is aware of all the types beforehand and can perform certain optimizations which leads to faster generation of machine code. One of the trade-offs is that it makes it more tedious and verbose for end users to write code in statically typed languages, whereas in dynamically typed languages it is less verbose and easier to understand. Rust's compiler does not optimize code unless explicitly specified.

4.2 What makes Rust unique

The biggest difference between Rust and other low-level languages is default memory safety; end users are still able to manually manage memory, while guaranteeing the absence of illegal memory access or memory related errors (E.g. dangling pointers, memory leaks). This is true for Safe Rust (default), which will never cause undefined behaviour, however end users are still capable of explicitly writing Unsafe Rust which is exactly like Safe Rust except for the ability to do "unsafe" work such as dereferencing raw pointers and mutating statics [4]. Like other low-level languages (C++), Rust provides zero-core abstraction, which allows for abstraction without overhead. The foundation of abstraction in Rust are traits, which are a collection of functions defined for an unknown type: Self [1].

4.3 Design of Rust

Rust was initially designed to solve two prominent problems in low-level languages:

- How do you do safe systems programming?
- How do you make concurrency painless? [2]

In order to solve both problems, Rust designed one mechanism that provides memory safety (no segmentation faults) without the need of a garbage collector and safe concurrent programming (concurrency without data races) through ownership and borrowing [1]. The benefits of avoiding garbage collection are deterministic resource cleanup and lower overhead for memory management [3].

One big idea that came out of Rust is their unique ownership system and the notion of "zero-cost abstractions." Ownership enables Rust to make memory safety guarantees without the need for a garbage collector. The Rust ownership system abides by this set of rules that the compiler checks at compile time:

- Each value in Rust has a variable that's called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value is dropped.

These rules are illustrated in following block of code:

Assigning a value to another variable moves its ownership and when a variable that includes data on the heap goes out of scope, the value will be cleaned up by drop unless the data has been moved to be owned by another variable. Using ownership in functions can become tedious as it would need to follow this pattern of taking and returning ownership with every function. This leads us to Rust's concept of references and borrowing.

In harmony with languages like C and C++, Rust uses an ampersand with a variable to create a reference that refers to the value of the variable but does not own it. Because it does not own it, the value it points to will not be dropped when the reference goes out of scope. Rust also allows for mutable references but with a couple restrictions:

- Only one mutable reference to a particular piece of data is allowed in a particular scope.
- A mutable reference cannot exist while there is have an immutable one.

The compiler also guarantees that references will never be dangling references: for a reference to some data, the compiler will ensure that the data will not go out of scope before the reference to the data does.

These restrictions on leveraging ownership and type checking allows Rust to handle many concurrency errors in compile-time rather than in run time. This is extremely beneficial as it prevents undefined behavior and lets code be free of subtle bugs while also letting it be easy to re-factor without introducing new bugs.

Raw pointers are allowed to ignore many of the rules that references have to follow. As talked about before, they are allowed to do "unsafe" work. They compile to standard C pointers without additional tagging or run time checks.

All of these compile time optimization encompass Rust's unique ability of having "zero-cost abstractions." All the analyses are performed at compile time, hence incurring "zero-cost" at run time.

5 OUR 100 PERCENT GOALS

For this goal we will have an algorithm that is coded into a program that can pull live stock price information and determine whether or not to send a buy or sell signal.

Steps:

- Complete the 90 percent goals. Refer to our project proposal
- Call an API to bring in live stock data into the program instead of hard coded data

This could look something like: Have an API updating a data set which a program runs on. When more data is passed into this dataset the core algorithm is run again and determines if there should be a trade based on certain criteria. If so a buy signal is sent out.

6 IMPLEMENTATION DETAILS

Below is pseudo-code on an example implementation of the algorithm:

```
double[] prices = new double[90];
double thirtyDayAvg = -1;
double ninetyDayAvg = -1;

// called when a new stock price
// is available, max every minute
public Signal addNewPrice (double price) {
    prices.addToEnd(price);
    thirtyDayAvg = calculateThirtyDayAvg(prices);
    ninetyDayAvg = calculateNinetyDayAvg(prices);
    if (thirtyDayAvg > ninetyDayAvg)
        return sellSignal;
    else
        return buySignal;
}
```

7 INDUSTRIAL BENEFIT TO THIS PROGRAM

Rust is fundamentally a modern multipurpose systems language that features memory safety, fast speed, concurrency support, and a built-in management system. The industrial benefits all fit into multiple categories. Currently many existing models of trading application use C++ as their language of choice, but an application that has the same output in Rust will gain all the benefits, being safer and faster.

Taking online stock trading applications and banks as example, they all benefit from the outlined features of Rust. Memory safety is crucial, as much as programmers try to keep memory management a priority, it may leak. For online applications and banks, these leaks can contain sensitive information, unconfirmed trades, or even system crashes, which big institutions do not ever want.

Rust is better than C++ in terms of speed, as outlined in many online articles, it has a better handle on concurrency. This benefits stock trading exponentially as seconds can mean the loss or gain of large sums of money. Concurrency will benefit trading as bids can be submitted multiple times, spammed by many, all across the world. The system must handle large amount of requests and Rust can benefit the institutions.

Last, Rust has a great assortment of libraries that Cargo manages. If a bank or online trading application company would want to support a new feature of Rust, it wouldn't take too long for a backend team to implement a new assortment of functions, saving time and resources for the companies.

8 CORE ALGORITHM PROGRAM RUST CODE

The following is a block of code in Rust showing how to calculate the mean of the last 30 elements and the mean of the last 90 elements in an inputted array of integers. This is a helpful calculation for the mean reversion model. Every time a new stock price comes in (checking each minute) we calculate both these means and if meanLast30 < meanLast90 we buy the stock.

```
let sumLast30 = 0;
let sumLast90 = 0;

for i in range(70, 100) {
    sumLast30 += input[i];
}
for j in range(10, 100) {
    sumLast90 += input[j];
}
let meanLast30 = sumLast30/30;
let meanLast90 = sumLast90/90;
```

9 APIS AND LIBRARIES

Here is the URL to the API platform we will use to retrieve the data: https://iextrading.com/apps/stocks/. This API is reliable and gives real time stock quotes and trading analyses with no request limits. The Rust Standard Library can be found here: https://doc.rust-lang.org/std/index.html. It is divided into a number of focused modules. Their documentation includes an overview of the module along with examples. Implicit methods on primitive types are also documented here. Also, although primitives are implemented by the compiler, the standard library includes and implements methods directly on the primitive types.

REFERENCES

[1] Rust blog. 2018. Abstraction without overhead: traits in Rust. https://blog.rust-lang.org/2015/05/11/traits.html

[2] Rust blog. 2018. Fearless Concurrency with Rust. https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

[3] Official Rust documentation. 2018. Rust FAQs. https://www.rust-lang.org/en-US/faq.html

[4] Official Rust documentation. 2018. What Rust Unsafe Does. https://doc.rust-lang.org/nomicon/what-unsafe-does.html

[5] J. A. HARTIGAN and M. A. WONG. 1979. A K-Means Clustering Algorithm. Blackwell Publishing for the Royal Statistical Society. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979). http://www.labri.fr/perso/bpinaud/userfiles/downloads/hartigan_1979_kmeans.pdf

[6] Mohammad Teshnehlab Mojtaba Ahmadieh Khanesar, Hassan Tavakoli and Mahdi Aliyari Shoorehdeli. 2009. Novel Binary Particle Swarm Optimization. *Particle Swarm Optimization* 6251, 1 (2009), 1–12.

[7] Babita Majhi G. Sahoo Ritanjali Majhi, G. Panda. 2009. Efficient prediction of stock market indices using adaptive bacterial foraging optimization (ABFO) and BFO based techniques. *Expert Systems With Applications* 36, 6 (2009), 10097–10104.
[8] Urszula Markowska-KaczmarHalina KwasnickaMarcin Szczepkowski. 2008. Genetic Algorithm as a Tool for Stock Market Modelling. *Artificial Intelligence and Soft Computing - ICAISC* 5097, 44 (2008), 450–459.
[9] The Ocean ZH The Ocean team, The Ocean JP. 2018. *Algorithmic Trading 101 Lesson 2: Data, Strategy Design, and Mean Reversion. (2018).*