Final: The Algos of Wall Street

Live monitoring of stocks prices and executing trades on price targets

https://github.com/Flowersnow/CS311-Project.

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

Rust is a systems programming language that offers thread safety, speedy performance, and prevents segfaults. This report outlines the usage of Rust to implement a stock trading algorithm. In this paper, we will provide both a low-risk/high-risk approach, and discuss the design implementations of a prototype trading application on Rust.

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 have been exploring the language to create an application that executes trades based off of the commonly used *k_means* trading pattern.

2 THE SIGNIFICANCE OF RUST IN OUR PROJECT

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.

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 de-referencing raw pointers and mutating statics [2]. 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].

Another bonus for Rust is Cargo, inbuilt dependency and build management. Cargo removes the manual labor of obtaining the library and sticking it somewhere in your project. It even allows you to update all the packages you use in a project with a single command.

These qualities in a programming language are desirable to implement algorithmic trading for several reasons:

- Speed and memory efficiency: Language should be efficient in processing high volume of data. Trading algorithms have many moving pieces.
- Low latency execution: Need real time access to the API.
- Cargo inbuilt dependency and build management: Easy to use high-performing libraries make coding less verbose.
- Good design and data modeling: Trading algorithms are comprised of many complicated graphs and structures.

3 OVERVIEW

Based on some of the above research, we have found that Rust is a great candidate to develop a stock trading application. Our group created software that combines the initial version of our core algorithm, an API we had selected to analyze live stock prices, and a handler that updates prices using live stock data and outputs buy and sell signals to the console. This program is important as it will benefit trading firms by strengthening their decision to either buy or sell a particular stock. This report will extend our findings from our background report and planned proof of concept by concluding our research, and demonstrating how we reached our project goals. We have outlined our core goals and progress in the text and illustrated code pieces below.

4 CORE GOALS FOR FINAL PROJECT

The following are the goals we set and achieved for our final project documentation. This goal was satisfied with the *k_means* algorithm discussed further below.

- Live stock information can be pulled using an API.
- Can determine whether to buy or sell a given stock given a set of 90 price points. This is implemented via the algorithm, *k_means* function, coded in Rust.

Sections 5 to 8 document the project and include how we implemented these core goals. They describe how to integrate the API and use the *k_means* clustering algorithm

5 API CALL AND HANDLING THE RESPONSE

Our program requires real-time stock market data, therefore the application must continually make API calls to retrieve such information. We will be using the Rust crate, reqwest, which is a convenient HTTP client for Rust. More info regarding reqwest can be found at https://github.com/seanmonstar/reqwest. The program will make synchronous calls via reqwest::Client, as the application requires handling the response synchronously and there are relatively few HTTP requests. The Stock Time Series Data API from Alpha Vantage (https://www.alphavantage.co/documentation/) offers real time stock series data derived from the current trading day. We used the intraday time series option with one minute intervals to gather the last 90 data points of a certain equity, in this example we will be using Google. The output of the described API is shown below. *Note: a majority of the stocks data has been removed to save space.*

```
{
    "Meta Data": {
        "1. Information": "Intraday (1min) open,
        high, low, close prices and volume",
        "2. Symbol": "GOOGL",
        "3. Last Refreshed": "2018-11-21 16:00:00",
        "4. Interval": "1min",
        "5. Output Size": "Compact",
        "6. Time Zone": "US/Eastern"
    },
    "Time Series (1min)": {
        "2018-11-21 16:00:00": {
            "1. open": "1045.6400",
            "2. high": "1045.6400",
            "3. low": "1043.2000",
            "4. close": "1043.3000",
            "5. volume": "32766"
        },
        "2018-11-21 15:59:00": {
            "1. open": "1045.9725",
            "2. high": "1046.3300",
            "3. low": "1045.2800",
            "4. close": "1045.4000",
            "5. volume": "16491"
        },
        "2018-11-21 15:58:00": {
            "1. open": "1046.0601",
            "2. high": "1046.2600",
            "3. low": "1045.8400",
            "4. close": "1045.9100",
            "5. volume": "8915"
        }
}
```

Each object in the Time Series represents Google's stock price in USD at that time instance. We will only be considering the open price of the stock. For demonstration purposes we are using 90 minute and 30 minute averages instead of days.

The following code will be used to make a fetch request to the API and handle the response appropriately by correctly parsing the data.

```
    fn fetch() ->
    std::result::Result<Vec<f32>, reqwest::Error> {
    let client = Client::new();

// Parse the response into a serde_json::value
    let json = |mut res : Response | {
        res.json::<Value>()
    };
    let request1 =
        client
        // API below
            .get("https://www.alphavantage.co/query?
            function=TIME_SERIES_INTRADAY
            &symbol=GOOGL&interval=1min&
            apikey=IVO96IWUXGF22KP9")
            .send()
            .and_then(json);
         request1.map(|res1|{
            let obj = res1.as_object().unwrap();
            let meta = obj["Time Series (1min)"]
            .as_object().unwrap();
            // a vector containing the latest
            // 90 stock prices
            let mut prices: Vec<f32> = Vec::new();
            let start = meta.len() -90;
            let mut index = 0;
            // get the latest 90 stock prices
            for x in meta {
                if index >= start {
              let cur_stock = x.1.as_object().unwrap();
                let open = cur_stock["1. open"]
                .as_str().unwrap();
                let x = f32::from_str(open).unwrap();
                prices.push(x);
                index = index + 1;
                }
                index = index + 1;
            }
            return prices;
        })
        .map_err(|err| {
            return err;
        })
}
```

The function fetch is used to make a web request to the API and return a Result<Vec<f32>, reqwest::Error>. A vector of the latest 90 stock prices will be returned upon success of the fetch function, otherwise an Error will be returned.

6 THE RUNNING FUNCTION

Each day the function *main* updates the array prices to include only the past 90 days (removes the oldest price and adds in the current days open price) using helper functions such as sleep which lets the thread sleep for a day before updating the price again calling *get_most_recent_price* and then updating the price vector to reflect this new price. Then the *k_means* algorithm is called and prints out "Buy!" or "Don't buy" depending on the result from the algorithm before the thread sleep again. *main* contains three steps which are explained in depth below: delay, update prices, and determine buy. These steps will run continuously until the user tells the program to stop. To loop continuously, we will use the loop keyword which tells Rust to execute a block of code continuously forever or until you explicitly tell it to stop. This function includes calls to *get_most_recent_price* and *k_means* functions.
Note: The k-means algorithm uses the past 30 and 90 days to determine whether or not the user should buy the stock. We will model 1 day as 1 minute in our demo to provide a more interactive demo.

6.1 Delay

We need to retrieve data exactly once every day to update the past 90 days of the prices array. To do this we will use Rust's std::thread::sleep function to wait one minute before proceeding to the next steps. This function takes in a Duration struct and puts the current thread to sleep for the specified amount of time. A Duration type represents a span of time. Each Duration is composed of a whole number of seconds and a fractional part represented in nanoseconds. Recall that we represent 1 day as 1 minute so we will be delaying for 60 seconds. Here is the code we will use to execute this task:

```
use std::{thread, time};
let duration = time::Duration::from_secs(60);
thread::sleep(duration);
```

6.2 Update Prices

The next step is making the call to *get_most_recent_price* returns the current price of the stock in question. In our code we will put the result in the variable *recent_price* as below:

```
let recent_price = get_most_recent_price();
```

Now we will remove the first element of *prices* and add *recent_price* to the end of *prices* so our list holds the data from the past 90 days(minutes in our program). Here is how the code will look in Rust:

```
fn update_price_history(prices: &mut Vec<f32>,
                        recent_price: f32){
    prices.remove(0);
    prices.push(recent_price);
}
```

6.3 Determine Buy

The final step in this code is to make the call to *k_means* and print "Buy!" or "Don't buy" depending on the result from the algorithm. *k_means* returns a bool. It will return true if the stock should be bought according to the algorithm and false if not. Here is how this will look:

```
if k_means(&mut prices, recent_price) {
    println!("Buy!");
 } else {
    print!("Don't buy!");
 }
```

7 THE ALGORITHM

The subsections below describe our selected algorithm, the mean reversion models, reasoning behind the decision, and working code snippet of the function *k_means* which executes the algorithm on a hard coded array of 90 elements.

7.1 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 [3].

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.

7.2 Selection Reasoning

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.

7.3 The k_means Function

The function is divided into an update and execute portion. The update relies on the *getRecent* function to get the most recent stock price of the user's choosing. Once it receives the most recent pricing, it updates the price history vector, and the *k_means* function is called.

Within our code below, our algorithm first needs a vector array to function, thus is fed a mock array with floats pushed in to initialize. To illustrate our algorithm working, we will print out our results in the following subsection below.

```
fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 2 {
      println!("Please provide the equity of your choosing");
         return;
    }
    let query = &args[1];
    let x = fetch(query);
    match x {
        Ok(res) => {
        let mut tmp_prices = res;
    loop{
        let duration = time::Duration::from_secs(10);
        thread::sleep(duration);
      let recent_price = get_most_recent_price(query);
        tmp_prices.remove(0);
        tmp_prices.push(recent_price);
        if k_means(&mut tmp_prices) {
        println!("Buy!");
        } else {
        print!("Don't buy!");
            }
        }
    },
   Err(e) => println!("error parsing header: {:?}", e),
    }
}
```

The main, as described above, the mock vector has been set, a call to *k_means* is made, and *prices* are updated.

```
fn k_means(prices: &mut Vec<f32>) -> bool{
    let mut thirty_sum: f32 = 0.0;
    let mut x = 89;
    while x >= 60{
        thirty_sum += prices[x];
        x = x - 1;
    }

    let mut nintey_sum: f32 = 0.0;
    let mut k = 0;
    while k < 90 {
        nintey_sum += prices[k];
        k = k + 1;
    }
    println!("thirty sum: {}", thirty_sum);
    println!("nintey sum: {}", nintey_sum);

    let thirty_day_average: f32 = thirty_sum / 30.0;
    let nintey_day_average: f32 = nintey_sum / 90.0;
   println!("thirty day average: {}", thirty_day_average);
   println!("nintey day average: {}", nintey_day_average);

    if thirty_day_average <= nintey_day_average{
        return true;
    } else {
        return false;
    }
}
```

For a maximum of once per minute, the stock price is updated to the *prices* vector, and *k_means* function takes it in as an input. Because the function is mainly focused on the difference between the 90 day and the 60 day, the 30 day period, two while loops are used to calculate the sum of *prices*. The 30 day average and the 90 day average is calculated and if the 90 day average is larger or equal to the 30 day average, the boolean returns true. This will trigger a purchase call, otherwise *k_means* function will return false and a don't purchase call will be triggered.

```
fn update_price_history
(prices: &mut Vec<f32>, recent_price: f32){
    prices.remove(0);
    prices.push(recent_price);
}
```

The *update_price_history* function, as described above, updates the prices vector with the latest stock price. It does this by removing the oldest item and push the newest one.

7.4 Results

The following are the results that is being printed within the Rust playground to test whether if the function is running correctly. When the *k_means* function receives a new pricing information and is updated. The function takes in the last 30 days and sums up, as you can see at the bottom of the results. A 90 day sum is also calculated for comparison to decide whether or not to signal the trade. 30 and 90 day averages are then calculated and in this example, it is signaling false, don't buy signal, as the 90 day average is smaller than the 30 day average by exactly 30.

```
thirty sum: 2400
nintey sum: 4500
thirty day average: 80
nintey day average: 50
false
```

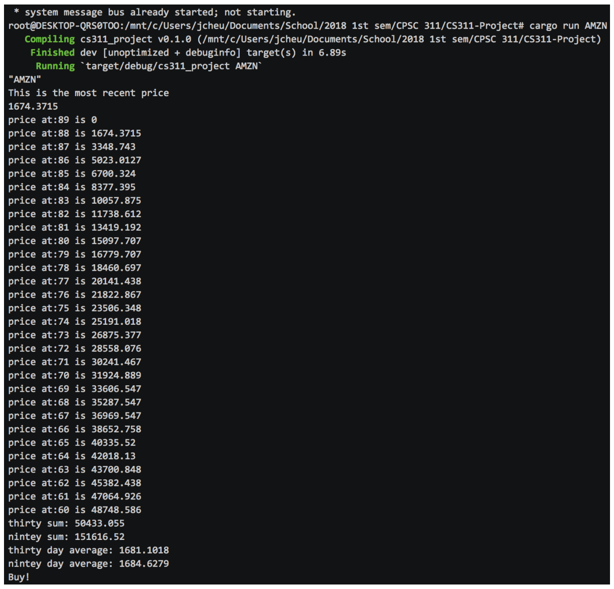
8 DEMO

Here we provide a link to our demo to see the algorithm in action. The k-means algorithm uses the past 30 and 90 days to determine whether or not the user should buy the stock. In the demo we have modeled 1 day as 1 minute to provide a more interactive demo. Below we have attached several screen shots of the demo showing. We see in the demo screen shots, a user inputs "AMZN", which is the ticker symbol for Amazon's stock price. The API call is made, bringing in the prices. 30 day and 90 day sum and the average are calculated and a "Buy!" signal is triggered as the recent price does favor the user to make the purchase. Refer to our source code at our Github repo:

https://github.com/Flowersnow/CS311-Project.

Link to demo:

https://asciinema.org/a/tL9bJOQDuexR2ToCrATZnwz0Z.



```
 * system message bus already started; not starting.
root@DESKTOP-QRS0TOO:/mnt/c/Users/jcheu/Documents/School/2018 1st sem/CPSC 311/CS311-Project# cargo run AMZN
   Compiling cs311_project v0.1.0 (/mnt/c/Users/jcheu/Documents/School/2018 1st sem/CPSC 311/CS311-Project)
    Finished dev [unoptimized + debuginfo] target(s) in 6.89s
     Running `target/debug/cs311_project AMZN`
"AMZN"
This is the most recent price
1674.3715
price at:89 is 0
price at:88 is 1674.3715
price at:87 is 3348.743
price at:86 is 5023.0127
price at:85 is 6700.324
price at:84 is 8377.395
price at:83 is 10057.875
price at:82 is 11738.612
price at:81 is 13419.192
price at:80 is 15097.707
price at:79 is 16779.707
price at:78 is 18460.697
price at:77 is 20141.438
price at:76 is 21822.867
price at:75 is 23506.348
price at:74 is 25191.018
price at:73 is 26875.377
price at:72 is 28558.076
price at:71 is 30241.467
price at:70 is 31924.889
price at:69 is 33606.547
price at:68 is 35287.547
price at:67 is 36969.547
price at:66 is 38652.758
price at:65 is 40335.52
price at:64 is 42018.13
price at:63 is 43700.848
price at:62 is 45382.438
price at:61 is 47064.926
price at:60 is 48748.586
thirty sum: 50433.055
nintey sum: 151616.52
thirty day average: 1681.1018
nintey day average: 1684.6279
Buy!
```

9 APIS AND LIBRARIES

Here is the URL to the API platform we will use to retrieve the data: https://www.alphavantage.co. This API is reliable and gives real time stock quotes and trading analyses with no request limits. Reqwest is an easy and powerful HTTP client in Rust. We will be using this library to make web requests to the API. More info can be found at https://github.com/seanmonstar/reqwest. 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.

10 FULL PROJECT GOALS

We outlined and successfully completed our main 100 percent goal which was originally set. Below is a review of possible full project goals. We plan to continue to work on this project using these goals which stretch beyond the scope of this project.

- Live buy and sell signals.
- Automatic stock investing. The program can automatically invest in stocks deemed worthy.
- Multiple stock review. Return from an input of a portfolio of multiple stocks.
- Easy to use UI such as simple console command.
- Threshold limitation. A user may set how much money will be spent.
- Stock Analysis that surveys multiple algorithms and only buys a stock if a majority agrees the stock should be bought.
- Modular or Genetic algorithm selection. To have the user choose which algorithm they would like to use.
- Facebook integration to share and compete with friends how much wealth you have accumulated.
- Chat bot integration that recommends you stocks that are not already in your portfolio to invest in based on their performance on the algorithm of the user's choosing.
- Stock shorting option, to take advantage well performing stocks that have intraday negatives.
- External category portfolio expansion. To diversify the user's portfolio, suggest stocks that are not familiar to user that are high performing to add to their portfolio.
- Cash out information. Have an option for users to cash out, with the user's local tax information to calculate when is the best time and amount to cash out.
- Region specific language support.

11 DISCUSSIONS

As shown above, Rust provides programmers with versatility that are not offered by many other programming languages. During our project cycle, we have encountered real world implications that benefit end-users. Bugs have been found immediately by having static type checking within the language and having access to crates makes it easier for end-users. Our group found Rust's zero cost abstraction to be especially useful as we were able to build within a safe environment, concentrating on our core functions without the need to be distracted by memory leaks or data races.

By understanding these concepts, we are ambitious to make our full project goals a reality. We have thousands of crates, such as text_io, at our disposal to accept user input, and generate a proper automated response. Creating a live input and output of information for multiple stock profiles and accepting user limitations of how many funds are to be used are achievable goals. Having multiple complex algorithms, such as genetic algorithm, would require extra computation to simulate trades, but isn't out of the scope of Rust's capability.

We leave the project by noting that with further studies, difficulty may arise with our cash out information goal and language support goal. As Rust is a relatively young language, there is no official support for non-Latin based languages, such as Korean. Also, local tax laws vary in cosmic levels between countries and regions within those countries, making it difficult to implement.

12 INSTALLING RUST

To install Rust refer to Rust's official installation documentations, https://www.rust-lang.org/en-US/install.html. *Note: Our source code currently has only been tested on Windows, other platforms may behave unexpectedly.*

REFERENCES

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

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

[3] The Ocean ZH The Ocean team, The Ocean JP. 2018. Algorithmic Trading 101 Lesson 2: Data, Strategy Design, and Mean Reversion. (2018).