Plan/Proof-of-Concept: 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

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. By elaborating our plan/proof-of-concept, 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 OVERVIEW

Based on our background report, we have found that Rust is a great candidate to develop a stock trading application on. Our group brought together the initial version of our algorithm, API we had selected, and have a skeleton model that will be prepared for our final project. This report will extend our findings from our background report by concluding our research, and plan out how we will reach our project goals. To showcase our progress, we have outlined our progress in the illustrated code pieces below.

2 MINIMUM CORE GOALS FOR PLAN/PROOF-OF-CONCEPT

With allowing pseudo code in mind, the following are the goals we set and achieved for our Plan/Proof-of-Concept.

- 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.
- Determine low-risk and high-risk plans for the final project and current completion and tasks to complete.

3 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 will be using 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 will be 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.

4 THE RUNNING FUNCTION

Each day the function *run* updates the array prices to include only the past 90 days (removes the oldest price and adds in the current days open price) and then prints out "Buy!" or "Don't buy" depending on the result from the algorithm. *run* 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 *getRecent* 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.

4.1 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);
```

4.2 Update prices

The next step is making the call to *getRecent* 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 = getRecent();
```

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);
}
```

4.3 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 kMeansAlg(prices) {
    println!("Buy!");
 } else {
    print!("Don't buy!");
```

```
}
```

5 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.

5.1 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 [1].

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.

5.2 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.

5.3 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 mut prices: Vec<f32> = Vec::new();
    for i in 0..90{
        prices.push(5.5 + i as f32);
    }

    let _ret: bool = k_means(&mut prices);
    println!("{}", _ret);

    let recent: f32 = 1000.1;
    update_price_history(&mut prices, recent);
    k_means(&mut prices);
}
```

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{
        println!("price at:{} is {}", x, thirty_sum);
        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.

The 30 day average and the 90 day averages are 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.

5.4 Results

The following are the results that are printed in our fully integrated and completed code. 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
```

6 PLANS FOR THE FINAL PROJECT

Currently we have a functioning API which pulls live stock data and a *k_means* function which given an array of prices can output a buy or sell signal. What needs to be done to meet our 100 percent goal is integrate the API with a loop which then calls the *k_means* function. The work-flow is as follows: inside of a while loop, start with a delay of 60 seconds (representing a day in the demo), after the delay get the most recent new stock price, call *update_price_history* with the new price and the array of prices passed by reference, then call the *k_means* algorithm which will output the buy signal. Once this integration is complete the project is complete.

6.1 Low-Risk Approach

Our core goals for a low-risk approach are as follows:

- Create a functioning k-means algorithm, which takes a list of stock prices and will determine whether or not to send a buy or sell signal.
- Instead of fetching real time stock market data using the API, create a static (hard-coded) list of stock prices that will be used as the input of the algorithm.

6.2 High-Risk Approach

Our core goals for a high-risk approach are as follows:

- 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 intra-day 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.

7 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.

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