CS261 Final Report

Team 29

Sylvester Cardorelle,**Tudor Cismarescu**, **Ollie Madelin**,
Josh Footman, **Joseph Parkins**, **Anil Kiral**

March 2017

Contents

1 Installation Guide/ System Requirements

1.1 Setup

1.1.1 Node

1.1.2 SQL

1.1.3 Angular

1.1.4 Npm Installation

1.2 Running

2 User Guide

2.1 Analysis tab

2.2 Live Stream tab

2.3 Historical Data tab

3 FAQs

4 Research

After the analysis and design of the project, our software development team had developed a stronger idea of the technologies that would be used to design the system. This section further expands on the decisions and research that was made after the first deliverable that allowed us to finalise our choice of technologies.

4.1 Front-end

We decided as a whole group quite early that the whole application would be web-based rather than a desktop app. This would increase the accessibility of the application being cross-browser compatible and mobile friendly. If the client integrated a sufficient security system, the system could be accessed from anywhere with a secure internet connection.
The **AngularJS** framework was used to build the frontend as planned. We considered other JavaScript frameworks, such as **React**, however we felt that AngularJS was the better choice being a full-fledged MVC (Model-View-Controller) framework. As our system is a single page application (SPA) that will update dynamically, AngularJS became a natural choice for our developers.
After this decision was made between the Project Manager and Developers, our front-end developers took the initiative to intensively learn AngularJS using a multitude online resources. For example, Codecademy a programming teaching platform was used to learn AngularJS basics, whilst AngularJS Community pages were used to further expand their knowledge.

4.1.1 JavaScript Graph Library

Deciding on a suitable JavaScript Graph Library was crucial for this project as data visualisation was one of our major non-functional requirements. After extensive research we decided to use the **PlotlyJS** library for the following reasons:

- It generates responsive graphs that are able to scale dynamically on different platforms e.g. desktops, tablets
- PlotlyJS is known for use in financial analytics and supports a variety of graphs e.g. candlestick, boxplots
- Uses WebGL technology for graph rendering with a speed that surpasses many competitors
- Provides the option to export the rendered graphs as PNG files for further inspection if needed
- Allows the zooming into of graphs, allowing the user to further inspect specific areas of the graph

4.2 Back-end

4.2.1 Server-Side

Node.js was used as our interface between the frontend and the Java backend using the Express framework. This decision was made by the Back-end developers built

on their exisiting skills with JavaScript. Node is extremely fast, compiling JavaScript code directly into machine code using Google's V8 engine, this means when facing large volumes of data the system will stay responsive due to the way Node handles concurrency. The modules and templates written in node can be easily reused throughout the serverside, thus reducing the size of our application and chance of bugs occuring.
Other popular server-side-scripts were considered such as Django and Ruby on Rails but given the unfamiliarity and time constraints, we discarded them because they did not fully utilise the current skills of the developers.

4.2.2 Database

Deciding whether to store stock data into a MySQL database became a frequently debated topic throughout the development of the application. If the system integrated a database ; storing the input stream into the database for a reasonable time period, it would allow us to select relevant data at will and improve the chance of the system finding potential perpetrators of any malicious anomalies.
Despite this, if the system relied on the database too heavily, a performance bottleneck would occur. Thus, making our system slower and less responsive. Our developer's countermeasure to this was to cache the data within Java's abstract data structures and storing a data for a certain amount of time then storing historical data within the database that would not be useful in further algorithmic calculations.
Once the Back-end developers had come to a decision that was then agreed between the whole group, both solutions to our data storage problem had been implemented with the MySQL database acting as a failsafe.

5 Design

5.1 User Interface

- color scheme - font size - abstraction - simplicity we wanted to develop something that did not require a manual essentialy, the apps fundamental operation can be grasped by the client with the assumption that they are computer literate.

5.2 Pseudocode

This section explores the type of algorithms that we used to analyse the input stream of data for anomalies.

5.2.1 Pump & Dump

Arguably, Pump & Dump was the most challenging type of price manipulation that we attempted to detect. After researching the characteristics of a Pump & Dump, our team concluded the following about Pump & Dump:

- A large batch of cancellation orders will occur before the dumping of stocks by a person or group (**Undetectable with current data set**)
- A steady increase of stock value from the average, when the pumping state begins (**Detectable**)
- A sudden drop in stock value when the dumping of stocks occur (**Detectable**)

Figure 1: Pump & Dump Example

By identifying the traits of a Pump & Dump, we were able to build the above model in fig:1. We found it more intuitive to design an algorithm that would first detect a dumping state, then look back on historical data to detect whether a pumping state had occured before the drop in stock value. Below in fig:2 is the pseudocode that would later be transformed into Java code.

```
PumpAndDump{

        if( PriceAverageList.Size() < 60) {
             return null ; // As there is not enough data to run analysis on
        }
        else {
             dumpingState = false ; // Set flag
             //Check for a dumping state (large decrease in price)
             //Look at latest and previous price averages
             //Look at percentage decrease, if there is a large enough decrease then a dump has occured

              difference = PriceAverageList.Get(LatestPrice) -  PriceAverageList.Get(PreviousPrice) ;
              percentage_decrease = (difference/PriceAverageList.Get(PreviousPrice)) *100 ;

              if (percentage_decrease > -30){
                  dumpingState = true;
              }

              if (dumpingState) {
                  // If the dumping state is true, we then lookback on the 50 prices,
                  // before the drop to look for a pumping state
                  Pumping = 0 ;
                  for the last previous 50 stocks {
                    if(PriceAverageList.get(currentIndex) > PriceAverageList.get(currentIndex-1)){
                      Pumping++ ;
                    }
                  }

                  if (pumping >= 30) { // Then a Pump & Dump has occured
                    // Build an Anomaly object containing relevant details about the Anomaly

                    return Anomaly ;
                  }
              }

        }

}
```

Figure 2: Pump & Dump Pseudocode

5.2.2 Volume Spike

To detect volume spikes, the algorithm keeps track of the amount of stocks bought within a certain time interval and calculates a volume moving average (VMA) that will be used as a comparison for the consequent interval. Therefore, if the total stock volume in a period greatly surpasses that of the previous VMA by a user-controlled threshold. The program will flag a volume spike, as seen below in fig:4. The threshold level would then adjust it's sensitivity to spikes depending on the user's feedback.

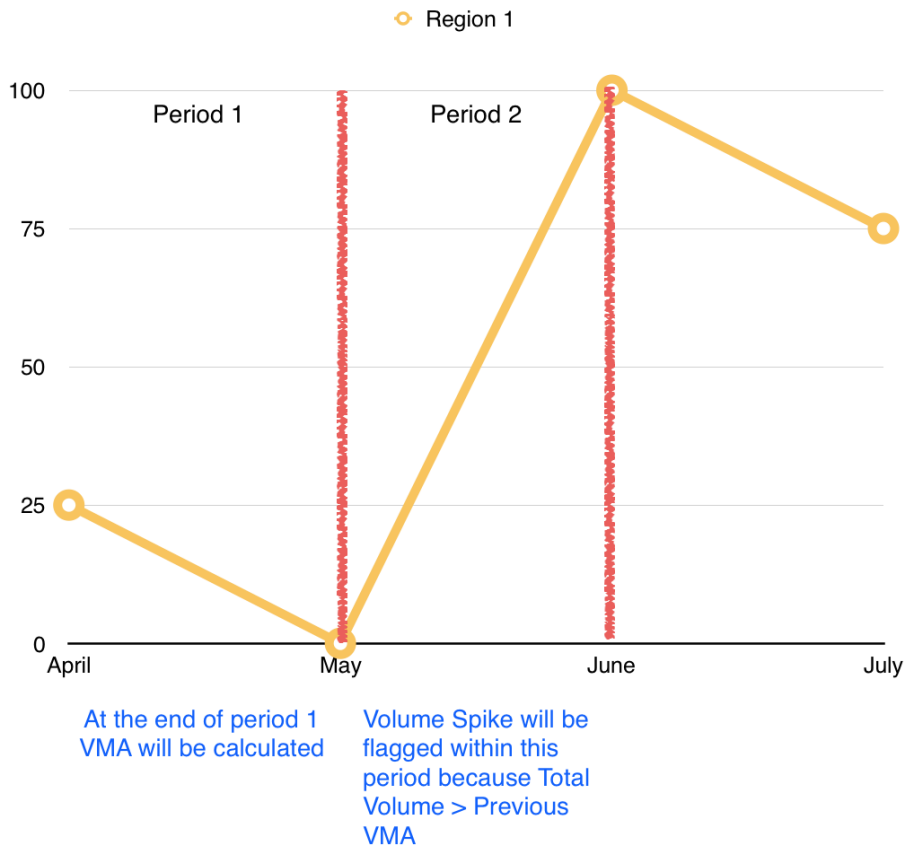

Figure 3: Volume Spike Example

```
Volume Spike {

    if (flag) { //if a volume spike has already occurred within the current time period
      return null;
    }
    else if ( currentPeriodVolume > limit) { //if amount of stocks sold has passed the dynamic limit
      //volume spike has occured
      flag = true;
      return anomaly; //send of the anomaly object containing all relevant information about the volume spike

    }
    else {
      flag = true;
      return null;
    }
}
```

Figure 4: Volume Spike Pseudocode

5.2.3 Fat Finger Errors

When detecting fat finger errors we decided to separate the error into two types:

- Fat Finger **Price Error**, which will detect fat finger errors related to the price of a stock transaction
- Fat Finger **Volume Error**, which will detect fat finger errors related to the size of a stock transaction

This is because it is possible for the Price or Size of a transaction to be extremely large compared to the average due to a typing fat finger error. The algorithm we developed to detect fat finger errors, uses orders of magnitude to determine whether the price or volume is extremely larger or smaller than the average (in factors of ten).

```
FatFingerPriceError {
        if( ((stock.getPrice() > lastVma*10^n ) || ( stock.getPrice() < lastVma* 10^-n )) && (lastVma != 0) ) {
            // if the volume of a stock is extremely large or small then ff has occured

            //then there has been a price ff error
            //calculate severity of the ff error

            severity = (stock.getPrice() * 100) / lastVma;

            //send anomaly
            return anomaly;
        } else {
            return null;
        }
    }
}
```

Figure 5: Fat Finger Price Pseudocode

```
FatFingerVolumeError {
        if( ((stock.getSize() > lastVma*10^n ) || ( stock.getSize() < lastVma* 10^-n )) && (lastVma != 0) ) {
            // if the volume of a stock is extremely large or small then ff has occured

            //then there has been a price ff error
            //calculate severity of the ff error

            severity = (stock.getSize() * 100) / lastVma;

            //send anomaly
            return anomaly;
        } else {
            return null;
        }
    }
}
```

Figure 6: Fat Finger Volume Pseudocode