Twitter Analyzer

*Cab 432 Cloud Computing*

Daniel Maxwell n9670262

**Introduction**

Twitter Analyzer is an application created to analyze twitter tweets against each other to find the most similar tweets. A user will be prompted to enter different tweets and will receive the latest tweet, this tweet is then compared to a list of up to one hundred tweets using the same search word and any previous search word the user has inputted. Using the natural language processing library the tweet is analyzed against this list to find the most similar tweet to itself and similarity is displayed. As the number of tweets being compared to each other grows, the number of comparisons grows exponentially thus using more and more cpu. To solve this issue, Amazon Web Service(AWS) is used to setup a scalable cloud to increase when the usage is increased.

**API used**

Twitter API - <https://developer.twitter.com/en/docs/api-reference-index>

twitter API is primarily used to receive a large input of data, using a keyword the API will fetch the latest 100 tweets with that word, an issue occurred because of the API limit of 450 calls per 15 mins, this however is rectified by creating multiple accounts and thus increasing the amount of calls that can be made

Natural - <https://github.com/NaturalNode/natural>

Natural is a library that gives access to language processing tools, Natural is used to compare all tweets to each other to discover the most similar tweet. To do this the Levenshtein Distance algorithm is used to count the amount of changes needed to make the strings an exact match.

**Technical description**

Twitter Analyzer was developed using node.js with the express generator to create the backbone of the application. After a basic server was created it was then edited in javascript using a text editor called ATOM. Jade/pug was used as the template engine that created the html and css was developed alongside bootstrap and then sent off to the client. The only thing on the client side was a small javascript script embedded in the html to update the page after a period time had collapsed. After the application was developed, it was then transferred using github to a computer running ubuntu which was hosted from AWS. When the server was running correctly on that computer the image was copied and a auto scaling group was created.

**Server Architecture**

The server architecture for Twitter Analyzer will be broken up into three parts, these are: Twitter data gathering, data processing and comparing and checking. Firstly gathering the enough data from twitter became an issue with a limit of 450 requests per 15 mins, this was overcome by adding more accounts and checking them if the account above it is used up, a good demonstration is shown in fig 2.0. After all the data has been successfully received the next stage of the process is to compare the data to one another, this is done in a separate function with 3 for loops. Essentially the script will choose the first tweet and compare it to every other tweet that was received, for one search term a maximum of one hundred tweets can be found. This leads to a lot of comparing where the CPU is needed as shown in fig 2.1. Lastly the processed information is send off and passed into the html where it is then displayed to the user, periodic checks are then used to ensure that the information is up to date.

**Difficulties**

Developing Twitter Analyzer had some difficulties to overcome, firstly there was the difficulty of receiving enough data because of the account limit of twitter. This issue was overcome by creating multiple accounts as described above. The next issue that arose was generating enough CPU usage for use over multiple instances, this was resolved by changing to the Levenshtein Distance algorithm which was also much more accurate to use as well. The last of the major issues was the implementation of a graphical interface of the data, d3.js was considered but unfortunately this issue was never resolved.

**Scaling and performance**

Twitter Analyzer’s architecture shows exponential growth when more search words are added, for better performance automatic scaling is used. To scale this application when needed using a E2 instance from AWS the image of this server was created. After an image is created a scaling group is created and told when the CPU is over 40%, add another instance with the server’s image, This is shown in fig 3.0. Another rule was created that stated when the CPU dropped below 30%, remove a instance as shown in fig 3.1. Finally the scaling group is applied to a load balancing to check whether a instance needs to be created or removed. Fig 3.2 to 3.4 show the server increasing its performance and assigning new instance to take on the extra load.

**Testing and limitations**

|  |  |  |
| --- | --- | --- |
| Action | Expected/Actual | Appendix Num |
| submit with no text | E: entry as just #  A: entry as just # | 4.1 |
| Submit with random letters | E: error stating no hashtag  A: error stating no hashtag | 4.2 |
| Press the back button | E:nothing happens same page  A:previous page last hashtags | 4.3 |
| Very large list of hashtags fast | E: infinite loading  A: infinite loading | 4.4 |

**Possible extensions**

This application has multiple ways to be improved, some of the possibilities are: make the data more visual by using an API like google charts or use a library like d3.js, using another string distance algorithm such as Jaro–Winkler distance algorithm and comparing them to each other. With the applications current architecture these improvements could easily be implemented but would require more time to be developed properly.

**References**

Natural - <https://github.com/NaturalNode/natural>

Twitter - <https://developer.twitter.com/en/docs>

Levenshteindistance - <https://en.wikipedia.org/wiki/Levenshtein_distance>

**Appendix**

2.0 – Showing the architecture of how to overcome the twitter bottleneck

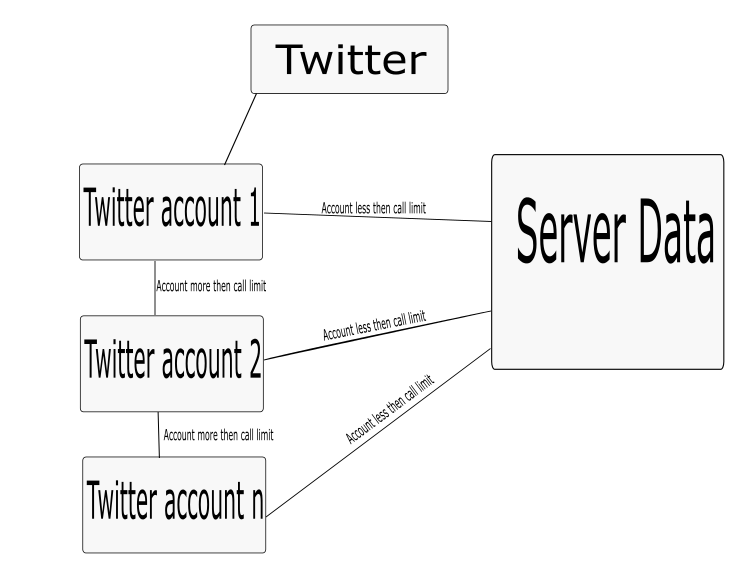


Fig 2.1- showing how many comparisons each term will have to process

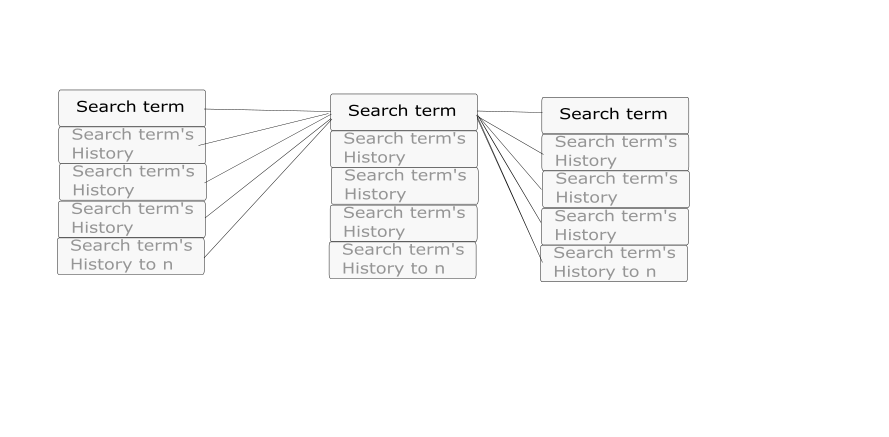


Fig 3.0- create an alarm to trigger when cpu is high then 40%

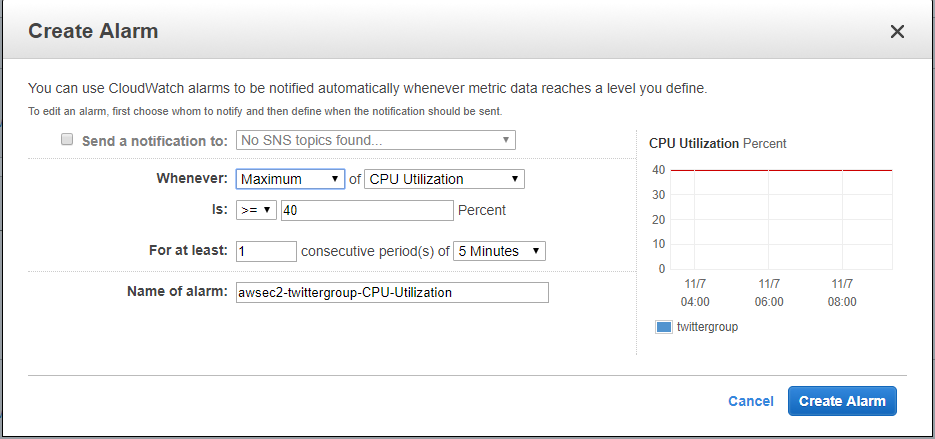


Fig 3.1- create an alarm to trigger when cpu is lower then 30%

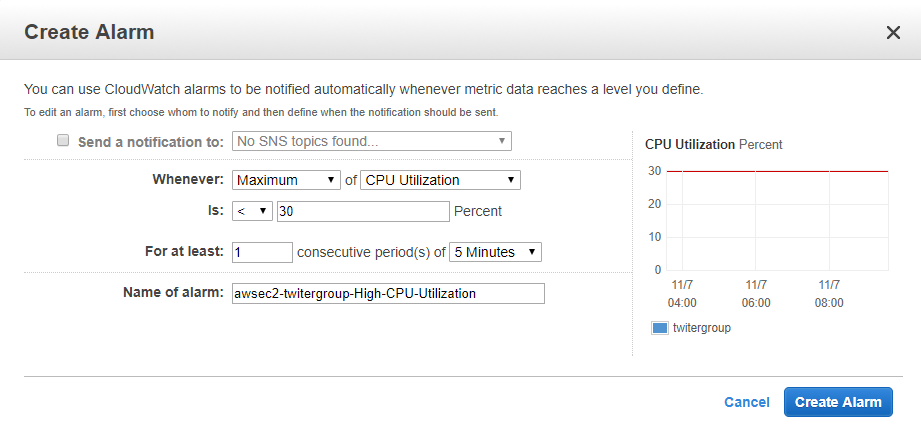


Fig 3.2 – adding instances for increased load

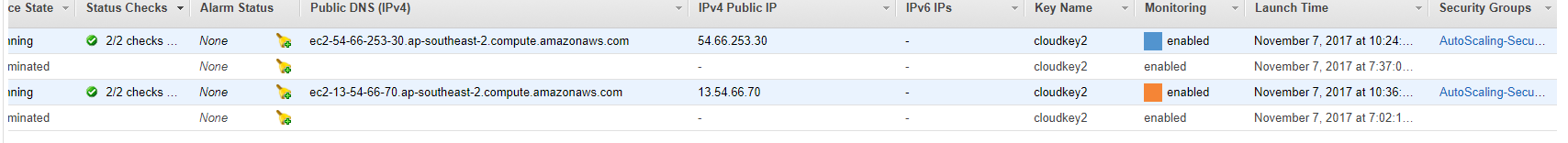


Fig 3.3 – decreasing instances for decreased load

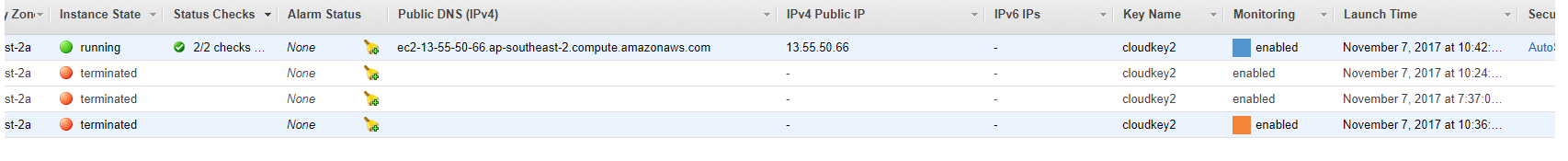


Fig 3.4 – showing when the load was increase 2 more instances was created

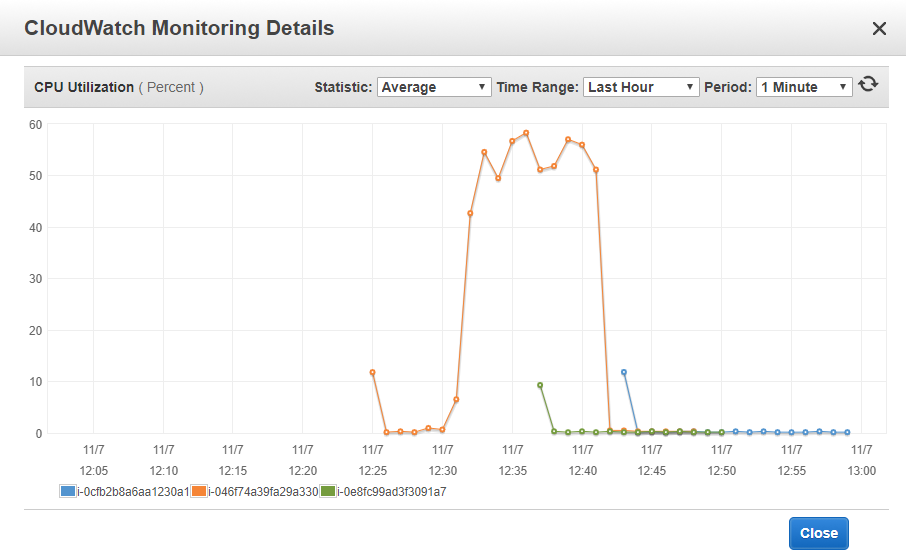


Fig 4.1 - Testing

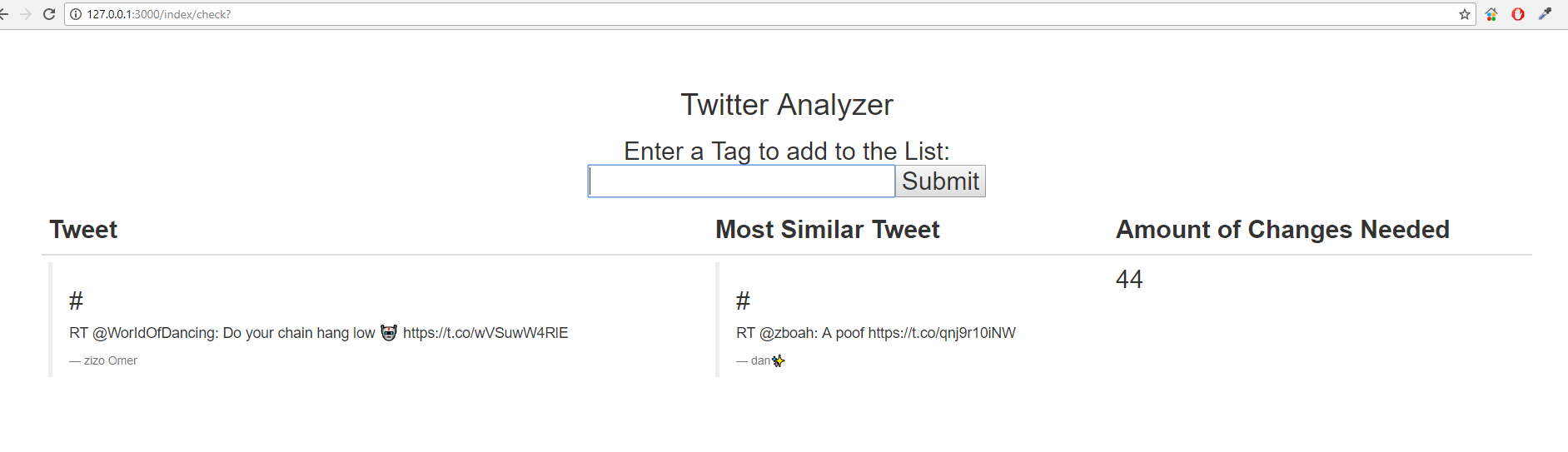


Fig 4.2- Testing

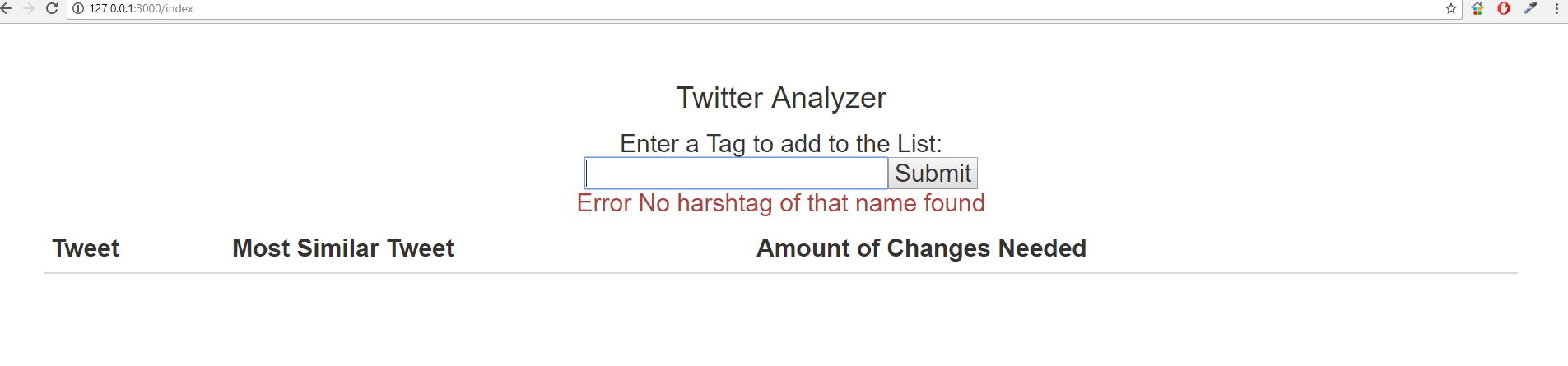


Fig 4.3- Testing

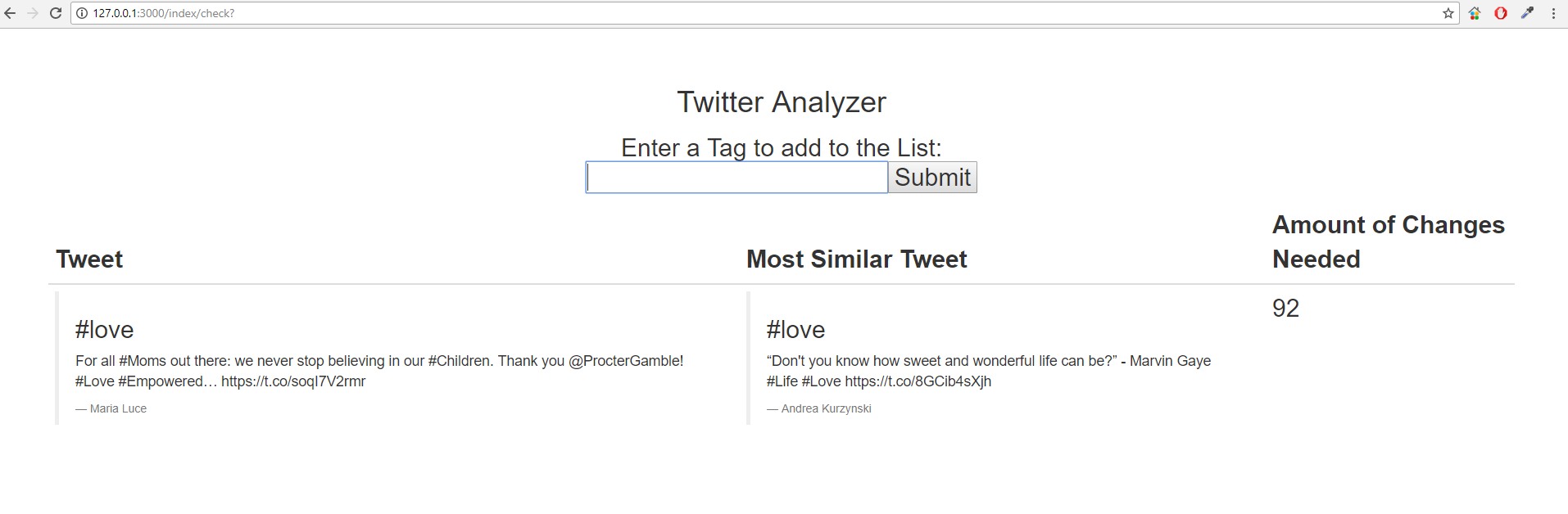


Fig 4.4- Testing

