

SENTIMENT ANALYSIS IN SOCIAL NETWORKS USING NAÏVE BAYES ALGORITHM

By

MOHAMED YAHYA MAHMOUD

**Dissertation submitted in partial fulfillment of
The requirement for the
Bachelor of Technology (Hons)
(Information and communication Technology)**

January 2011

**Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Rizduan**

CERTIFICATION OF APPROVAL

Sentiment analysis in social networks

By

Mohamed Yahya Mahmoud Ibrahim

A project dissertation submitted to the

Information and Communication Technology Programme

Universiti Teknologi PETRONAS

in Partial Fulfillment of the Requirements

for the Degree

Bachelor of Technology (Hons)

(INFORMATION AND COMMUNICATION TECHNOLOGY)

Approved by.



Dr Alan Oxley
Professor
Computer & Information Sciences Department
Universiti Teknologi PETRONAS

Prof. Dr. Alan Oxley

Main Supervisor

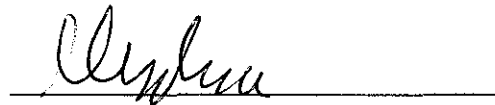
UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

APRIL 2011

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.



Mohamed Yahya Mahmoud Ibrahim

ACKNOWLEDGEMENTS

To UTP.

ABSTRACT

This report is concerned with the opinion mining and sentiment analysis in social networks especially Twitter, it aims to give a brief insight into the ongoing research in sentiment analysis algorithms and techniques, and graphical representation of the statistical results by applying the sentiment analysis on social networks. The objectives of this project is to perform a detailed research on the latest techniques in the process, and to enhance the current approaches of sentiment analysis by building a tool with an ability to provide statistical information, graphically represented –to an acceptable degree of accuracy- to show the collective consciousness of Internet users. The implementation of this project will most probably use third party tools available on the web to reduce time needed and for rapid prototyping in the initial stages of implementation.

Contents

CERTIFICATION OF ORIGINALITY 3

ACKNOWLEDGEMENTS 4

ABSTRACT 5

1. Chapter 1..... 8

 1.1. Introduction 8

 1.1.1. Background of study 8

 1.1.2. Problem statement 8

 1.2. Objectives and Scope of Study..... 9

2. Chapter 2..... 10

 2.1. Literature review and theory 10

 2.2. Sentiment analysis 10

 2.2.1. Naïve Bayes Classifier..... 11

3. Chapter 3..... 13

 3.1. Methodology..... 13

 3.2. Gantt chart 14

 3.3. Third Party APIs and Tools 15

 3.3.1. Twitter..... 15

 3.3.2. Google App Engine (GAE)..... 21

 3.3.3. Google Chart API 24

 3.3.4. Google Visualization API 25

4. Results and discussion 27

 4.1. Activity diagram 27

 4.2. Use Case Diagram 28

 4.3. Implementation 29

 4.3.1. Fetching Tweets (search) 29

 4.3.2. Wordlist handler 33

5. Conclusion 41

 5.1. Future development 41

Reference 42

Table of Figures:

Figure 1 Prototype Methodology.....	13
Figure 2: Twitter.com.....	16
Figure 3 : Twitter.com Traffic according to Quantcast	17
Figure 4: Twitter US Demographics	18
Figure 5: OAUTH Authentication Flow.....	20
Figure 6: GAE App Launcher	23
Figure 7: Static Pie Chart.....	24
Figure 8: Dynamic Pie Chart.....	26
Figure 9 Activity diagram	27

1. Chapter 1

1.1. Introduction

1.1.1. Background of study

Sentiment Analysis is a very new field, the first published academic book on the matter was in 2008 ^[1] by Bo Pang and Lillian Lee, the researcher at Yahoo co-wrote “Opinion Mining and Sentiment Analysis”, that’s why the materials available on this subject are very few, so far there hasn’t been a complete efficient enough system to solve the problem, but it is looking very promising as more and more researchers and companies take on the task of building a better, faster, more efficient algorithm to tackle the problem.

1.1.2. Problem statement

The social networks have been a great interactive tool to share everything from opinions, updates, photos, videos, links and much more with friends, family and colleagues, this is a very fast growing mountain of data that can be used to get a collective consciousness of the users online.

Currently the sentiment analysis programs are expensive and not available for everyone. This limits the benefits of the huge databases in these websites, All these databases (Twitter, Facebook, etc) have a search function but it doesn’t provide statistical information.

The main obstacle is the inability to understand if the statements made are positive or negative, most of these statements convey emotions not facts which is harder to detect by machines, also the statements are in slang most of the time so normal simple algorithms seem useless decrypting the human emotions in them.

1.2. Objectives and Scope of Study

The objective of this project is to conduct a research on Sentiment analysis and hopefully build a prototype of a web application to “opinion mine” some of the most famous social networks, especially Twitter, and to utilize the huge databases of opinions in these websites to define –to an acceptable degree of accuracy- a collective global opinion about any given phrase.

A well defined scope of work is essential to ensure the feasibility of the work undertaken. The scope of work for this project is as follows:

- Search should yield all needed raw data from twitter, the result of the requested data should be in a common format (XML) for example to ease exporting/processing the data.
- Search should also include a time intervals set by the user.
- Sentiment analysis algorithm should be interactive (gives the user the ability to edit the results) and the algorithm should learn from these modifications to enhance accuracy in the future.
- All the processed info should be represented in interactive graphs to help utilize it even more depending on the user’s preferences.
- Final Results can be exported into a couple different formats (XML, JSON).

2. Chapter 2

2.1. Literature review and theory

In order to facilitate understanding of the research material; the books, documentations, journals and articles examined in the literature review have been divided into those concerning sentiment analysis algorithms, and those concerning interactions with third party APIs and tools. The articles and reports mentioned here are the main ones.

2.2. Sentiment analysis

Sentiment analysis is the task of identifying positive and negative opinions, emotions, and evaluations. Most work on sentiment analysis has been done at the document level, for example distinguishing positive from negative reviews. However, tasks such as multi-perspective question answering and summarization, opinion-oriented information extraction, and mining product reviews require sentence-level or even phrase-level sentiment analysis. For example, if a question answering system is to successfully answer questions about people's opinions, it must be able to pinpoint expressions of positive and negative sentiments.

A typical approach to sentiment analysis is to start with a wordlist of positive and negative words and phrases. In these wordlist, entries are tagged with their a priori *prior polarity*: out of context, does the word seem to evoke something positive or something negative. For example, *beautiful* has a positive prior polarity, and *horrible* has a negative prior polarity. However, the *contextual polarity* of the phrase in which a word appears may be different from the word's prior polarity. Consider the underlined polarity words in the sentence below:

John smith, president of the university Environment Trust, sums up well the general thrust of the reaction of environmental movements: "There is no reason at all to believe that the polluters are going to become reasonable."

Of these words "well," "reason," and "reasonable" have positive prior polarity, but they are not all being used to express positive sentiments. The word "reason" is negated, making the contextual polarity negative. The phrase "no reason at all to believe" changes the polarity of the proposition that follows; because "reasonable" falls within this proposition, its contextual

polarity becomes negative. Similarly for “polluters”: in the context of the article, it simply refers to people who pollute. Only “well” has the same prior and contextual polarity.

Many things must be considered in phrase-level sentiment analysis. Negation may be local (e.g., **not good**), or involve longer-distance dependencies such as the negation of the proposition (e.g., **does not look very good**) or the negation of the subject (e.g., **no one thinks that it's good**). In addition, certain phrases that contain negation words intensify rather than change polarity (e.g., **not only good but amazing**).

This project should be able to automatically distinguish prior and contextual polarity. Beginning with a large stable of clues marked with prior polarity, we'll identify the contextual polarity of the phrases that contain instances of those clues in the list. We'll use a two-step process that employs machine learning and a variety of features. The first step classifies each phrase containing a clue as neutral or polar. The second step takes all phrases marked in step one as polar and disambiguates their contextual polarity (*positive, negative, both, or neutral*). With this approach, the system is able to automatically identify the contextual polarity for a large subset of sentiment expressions, achieving results that are significantly better than baseline.

2.2.1. Naïve Bayes Classifier

Bayesian classifiers are based around the Bayes rule, a way of looking at conditional probabilities that allows you to flip the condition around in a convenient way. A conditional probability is a probability that event X will occur, given the evidence Y. That is normally written $P(X | Y)$. The Bayes rule allows us to determine this probability when all we have is the probability of the opposite event occurring $P(Y|X)$ which means the probability of Y occurring given X has occurred, and of the two components individually $P(X)$ and $P(Y)$:

$$P(X|Y) = \frac{P(X)P(Y|X)}{P(Y)}$$

This restatement can be very helpful when we're trying to estimate the probability of something based on examples of it occurring.

In this case, we're trying to estimate the probability that a document is positive or negative, given its contents. We can restate that so that is in terms of the probability of that document occurring if it has been predetermined to be positive or negative. This is convenient, because we have examples of positive and negative opinions from our data set above.

The thing that makes this a "naive" Bayesian process is that we make a big assumption about how we can calculate at the probability of the document occurring: that it is equal to the product of the probabilities of each word within it occurring. This implies that there is no link between one word and another word. This *independence assumption* is clearly not true: there are lots of words which occur together more frequently than either do individually, or with other words, but this convenient fiction massively simplifies things for us, and makes it straightforward to build a classifier.

We can estimate the probability of a word occurring given a positive or negative sentiment by looking through a series of examples of positive and negative sentiments and counting how often it occurs in each class. This is what makes this *supervised learning* - the requirement for pre-classified examples to train on.

So, our initial formula looks like this.

$$P(\textit{Sentiment} | \textit{Sentence}) = \frac{P(\textit{Sentiment})P(\textit{sentence}|\textit{Sentiment})}{P(\textit{sentence})}$$

Where:

"Sentiment" is the polarity of the whole document.

"Sentence" is the polarity of a sentence in the document.

3. Chapter 3

3.1. Methodology

The methodology of choice for the implementation of this project is prototyping methodology. There was a fair amount of trial-and-error implementations until the optimum solution was reached. That is why prototyping methodology is perfect for this type of project since it is very flexible and allows for re-iteration and changes in the implementation of the project. The figure below shows the System Development Life Cycle (SDLC) using prototyping methodology.

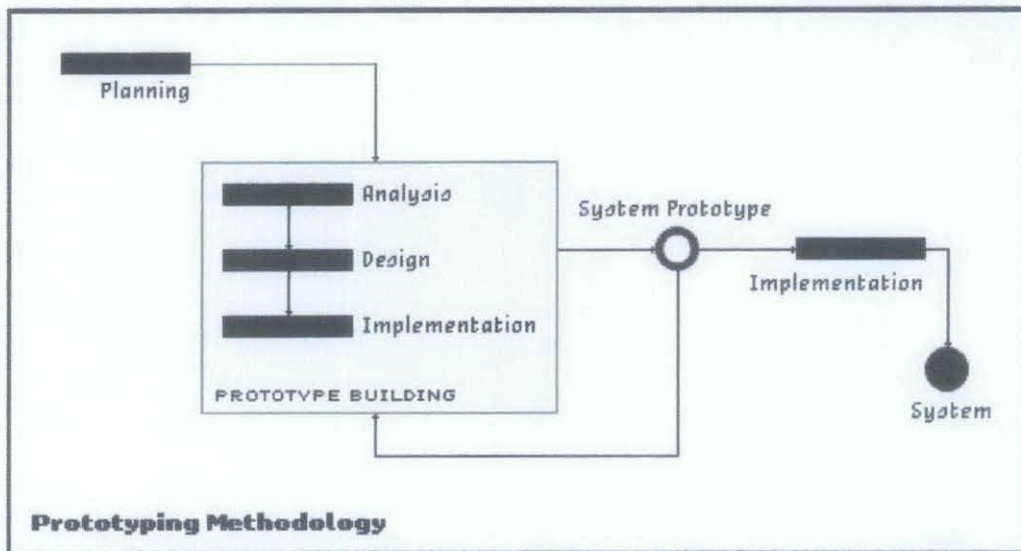
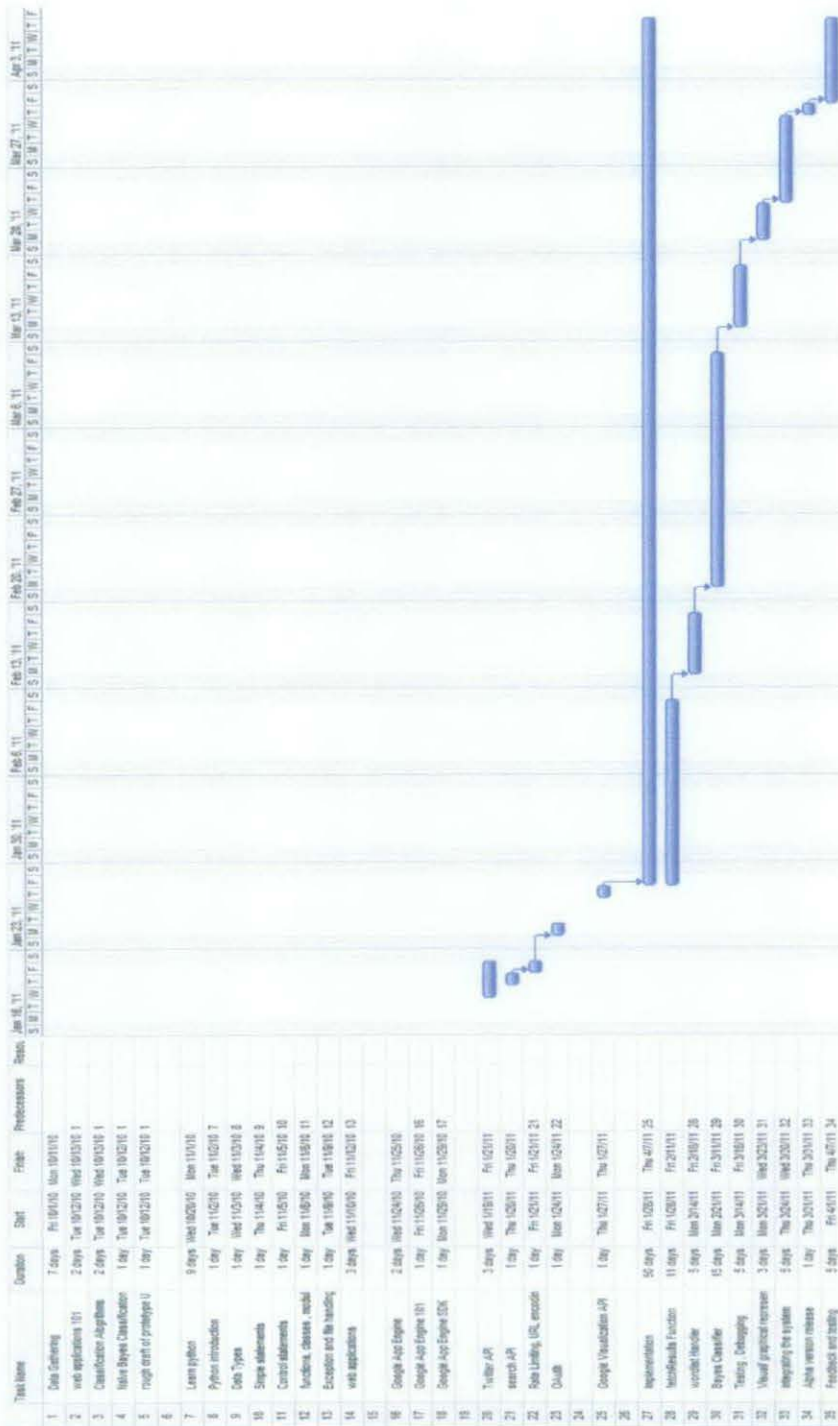


Figure 1 Methodology

For the Software Development LifeCycle of the sentiment analysis application, Rapid Application Development using phased development methodology was practiced to deliver the system. As the relation between the timeline and the technical expertise for the development is enhanced, the versions of the application is being developed and upgraded. This is to ensure the delivery of the application with the minimum requirements. Not all functions were developed in the first version, and so additional requirements may arise throughout the SDLC of sentiment analysis.

3.2. Gantt chart



3.3. Third Party APIs and Tools

The main tools required for the implementation of this project are the following

1. Twitter
2. Google App Engine SDK
3. Google Chart API
4. Google Visualization API

3.3.1. Twitter

3.3.1.1. *What is twitter?*

Twitter is a social network and a microblogging service that allows its users to send and receive other users' messages which are called tweets; tweet is a text post up to 140 characters. These tweets are displayed on the user's profile page and it's publicly visible by default, with the option of restricting access to one's page to their friend list. Users have the ability to subscribe to other users pages to get updates from this user, this is called following, subscribers are known as followers.

All users are allowed to update and receive tweets via twitter website, third party applications or devices such as smartphones, or by using SMS but It's only available in certain countries so far. Twitter is free for all users but accessing using SMS isn't since it's decided by the mobile service provider. Twitter was created by Jack Dorsey, an American software architect and an entrepreneur in 2006, and it became one of the most used social networks on the planet with 100 million users worldwide so far.

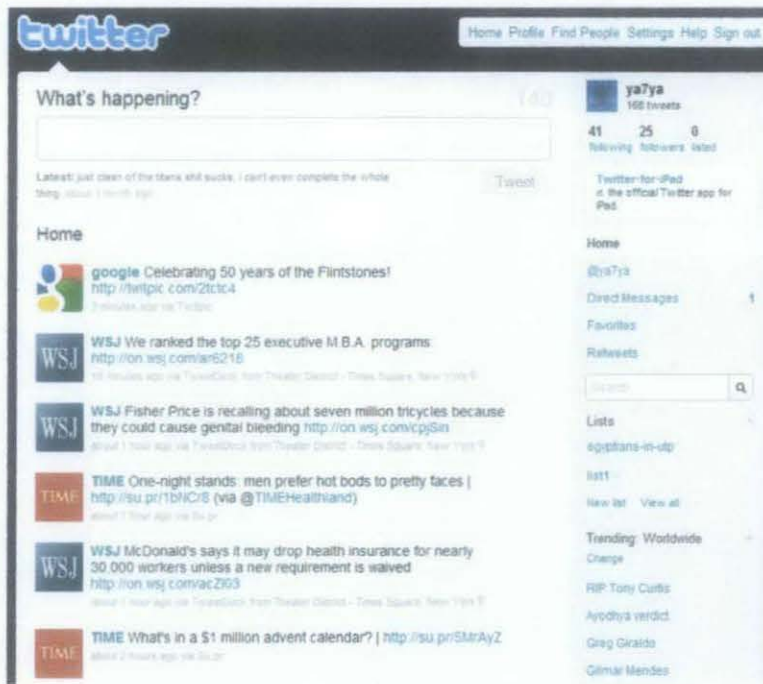


Figure 2: Twitter.com

3.3.1.2. Growth

Twitter grew from 400,000 tweets posted per quarter in 2007 to 55 million tweets per day in April 2010, it has 100 million registered users with a growth rate of new sign ups of 300,000 per day, the site itself is visited by 180 million unique visitors a month (worldwide) and 75% of the traffic comes from third party applications that are built based on twitter like tweetdeck.

The usage of twitter spikes during major events like sports events, in 2010, twitter set the record for the most amount of tweets during 2010 FIFA World Cup when fans tweeted 2,940 tweets per second in 30 seconds after Japan scored against Cameroon, however the record was broken during the NBA Finals of 2010.

According to Quantcast twitter.com is ranked 14th most popular site in US and during last year so far, Twitter recorded the highest amount of traffic which was 54.5M visitors from US in August 20th 2010.

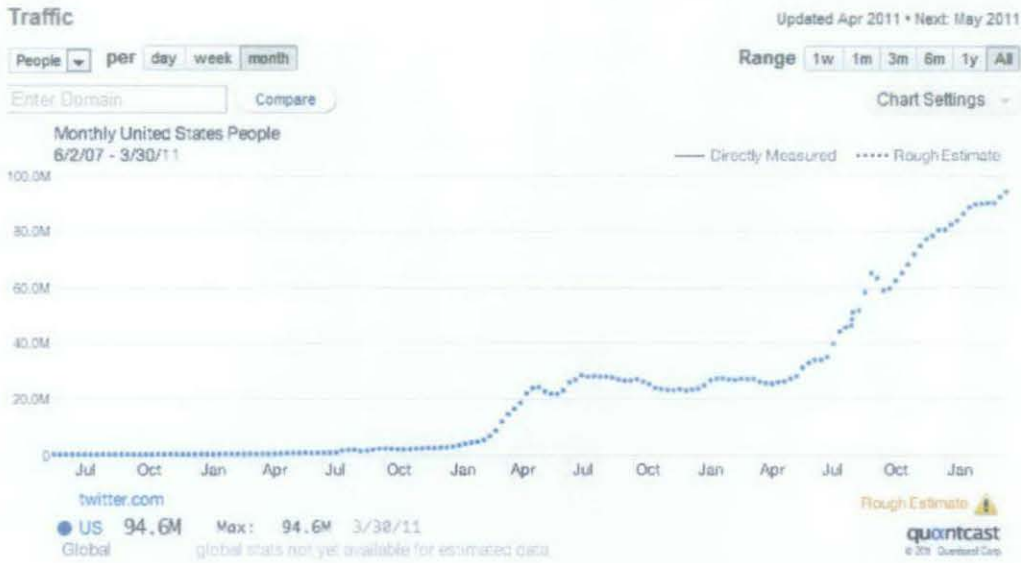


Figure 3 : Twitter.com Traffic according to Quantcast Stat search engine

3.3.1.3. Demographics

In United States, Twitter is used by young adults between 18 and 34 years old, they make around 45% of the total users. This is because this type of social network picked up more in the business and news settings hence it appeals more to an older crowd. Teenagers (13- 17 years old) make only 14% of twitter users.

The site also attracts higher than average numbers of households with more than \$100K income which makes around 30% of the site users and 28% of the users make around \$60-\$100K per year.

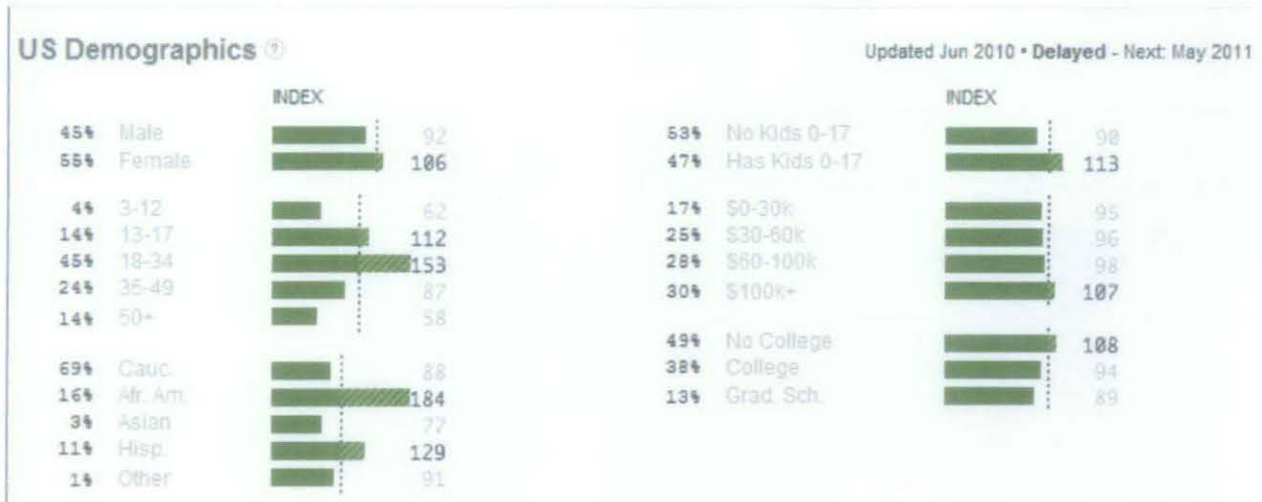


Figure 4: Twitter US Demographics

3.3.1.4. Twitter API

Twitter web interface was built using Ruby on Rails framework, the website maintains an application programming interface (API) to allow developers to integrate with twitter and build applications that uses twitter services and data.

The twitter API consist of 3 different parts, two REST (Representational State Transfer) APIs, they basically run the data transfer from and to the website, there are 2 of them because of earlier implementation of twitter was done by 2 companies, Twitter is planning to unify them soon according to twitter, the third API is a streaming API that support long lived connections to different architecture.

The REST APIs allow developers to access the core twitter data which include but not limited to status data and user info, the Search API allows developers to user twitter search to develop trending software like this project for example, the Streaming API allows near real-time high-volume access to tweets.

3.3.1.5. Rate limiting

The twitter APIs handle around 3 Billion API calls per day, and in order to keep the services offered up to acceptable performance and speed, the Twitter API has a Rate limiter which allows clients to make a limited number of calls in a given hour, this has different effects on the 3 APIs.

The default rate limit for calls to the REST APIs depends on the authorization method being used, if the calls are made anonymously based on the IP of the host , it's permitted to make 150 requests per hour. If the calls are made by OAuth client (authorized client) , twitter permits 350 calls per hour.

The rate limiting for REST APIs is usually applied to methods that request info with the HTTP GET command. Generally API methods that use HTTP POST to submit data to Twitter are not rate limited, however some methods are being rate limited now.

With REST APIs It is possible to whitelist both user accounts and IP addresses. Each whitelisted entity, whether an account or IP address, is allowed 20,000 requests per hour. It's recommended to consider whitelisting for this project to be able to handle more requests from the users.

The search API doesn't have the same Rate Limiting as the REST APIs, it has different limits which isn't made public by Twitter to stop abuse, it's known that it's higher than the normal REST API limits. This shouldn't be a problem since so far search has been sufficient enough to handle thousands of third party applications without major complaints from the developers.

3.3.1.6. Authentication the Requests

In order to use twitter API , the client application (this project) needs to register with Twitter, the client application will be provided a consumer key and secret, This key and secret scheme is similar to the public and private keys used in protocols such as SSH. This key and secret will be used –along with OAuth– to sign every request made to the API. This process is needed to identify the web application so it won't be black listed as spam.

OAuth is an Open Authentication standard used to allows the user to share private stored on one site with another site without having to hand out your username and password. This is achieved usually by retrieving a request token to access the data, Twitter requests user authorization by sending user to Twitter login page if needed, then finally OAuth exchanges request token for an access token.

The diagram below shows the usual flow of process during the OAuth Authentication process

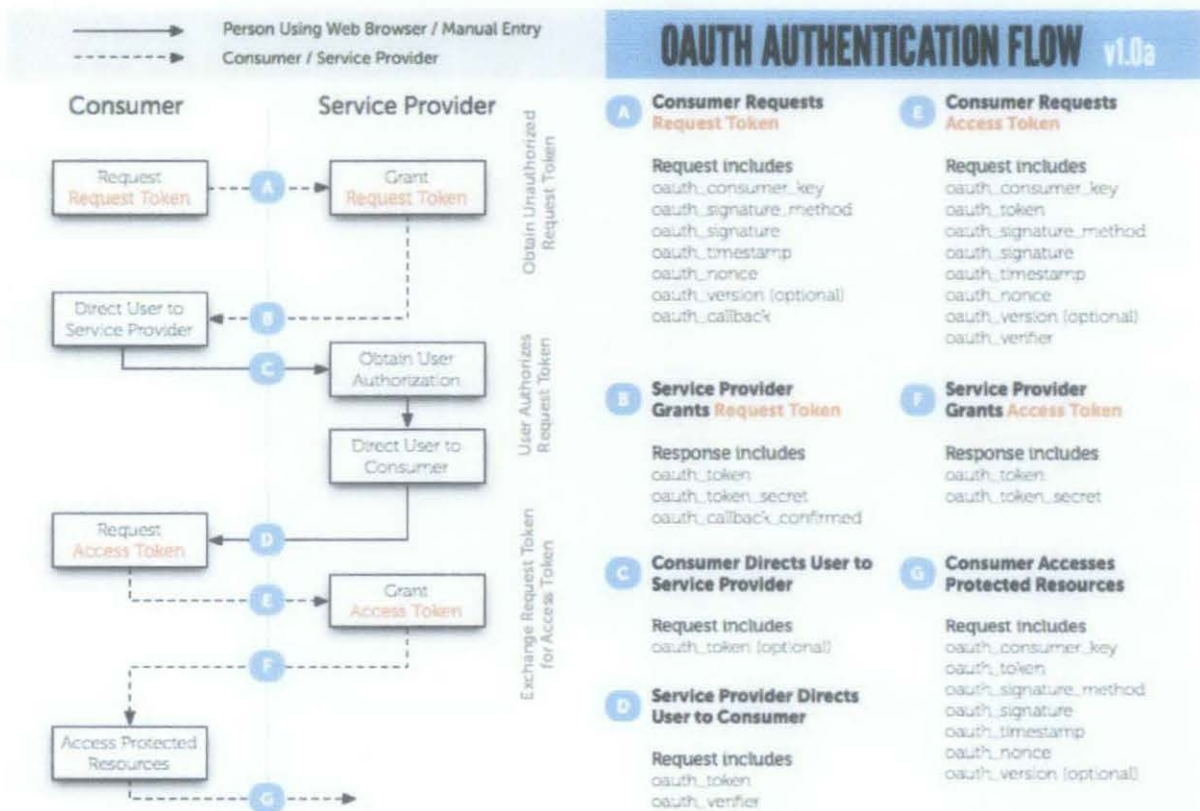


Figure 5: OAUTH Authentication Flow

3.3.2. Google App Engine (GAE)

GAE stands for Google App Engine which is a platform that power Google web applications. The web apps could be coded in 2 choices: Java and Python. The web app however needs to be designed to run on GAE which has some restrictions and standard when it comes to coding and implementation.

Like any platform, GAE has good and bad point, the following are some of both to clarify google App engine was chosen to build on.

The good

1. It's free

- The GAE allows users to create and publish un-restricted Web Apps for no cost or contract in the beginning as long as the software needs less than 1GB of storage and less than 5 Million page views a month. If the user need more space, GAE allows the user to set a daily budget for every resource according to the user's needs.

2. Easier to get started with

- The GAE comes with complete SDK, whether it's Python or Java, it also self manages the data storage, virtual machines, bringing up new instances of the app, and all the maintenance is taken care of by Google.

3. Automatic scalability

- Automatic scaling is built in with app engine, all the designer needs is to code the web app and google will take care of the resources depending on the amount of users using the web app.

4. Performance and security

- Google Keeps all its infrastructure up to date when it comes to performance and security which guarantees the web app stability and accessibility.

The bad

1. The system has to be coded using the App Engine SDK. The App Engine system is proprietary, so after you've written your system to use the Google APIs, it won't run anywhere else unless the code get the modification needed.
2. Google gets full access to the code and all the performance details, it's possible (but highly unlikely) for Google to copy the code or use it and there won't be a way to realize that.

Conclusion

Google App engine is great in minimizing starting costs for new developers and it's a great platform to run a web app on to test the reaction of the users and to see how popular the application gets without spending money on initial testing.

However, if the application picks momentum, the costs of changing to code to run anywhere else could vary since the code will need a lot of modification, but for now I think the ability to run the web app for free as long as it has less than 5 million users is priceless.

3.3.2.1. The Development Environment

I chose Python to develop this app over Java since it takes less time to develop in python in terms of number of lines of code. The Python SDK allows designers to build, locally test and upload the GAE web app, it includes the web server application needed for local testing and simulating App Engine environment, local version of the datastore (GAE database), Google Accounts and the ability to fetch URLs and send emails from the designers computer using the App Engine API.

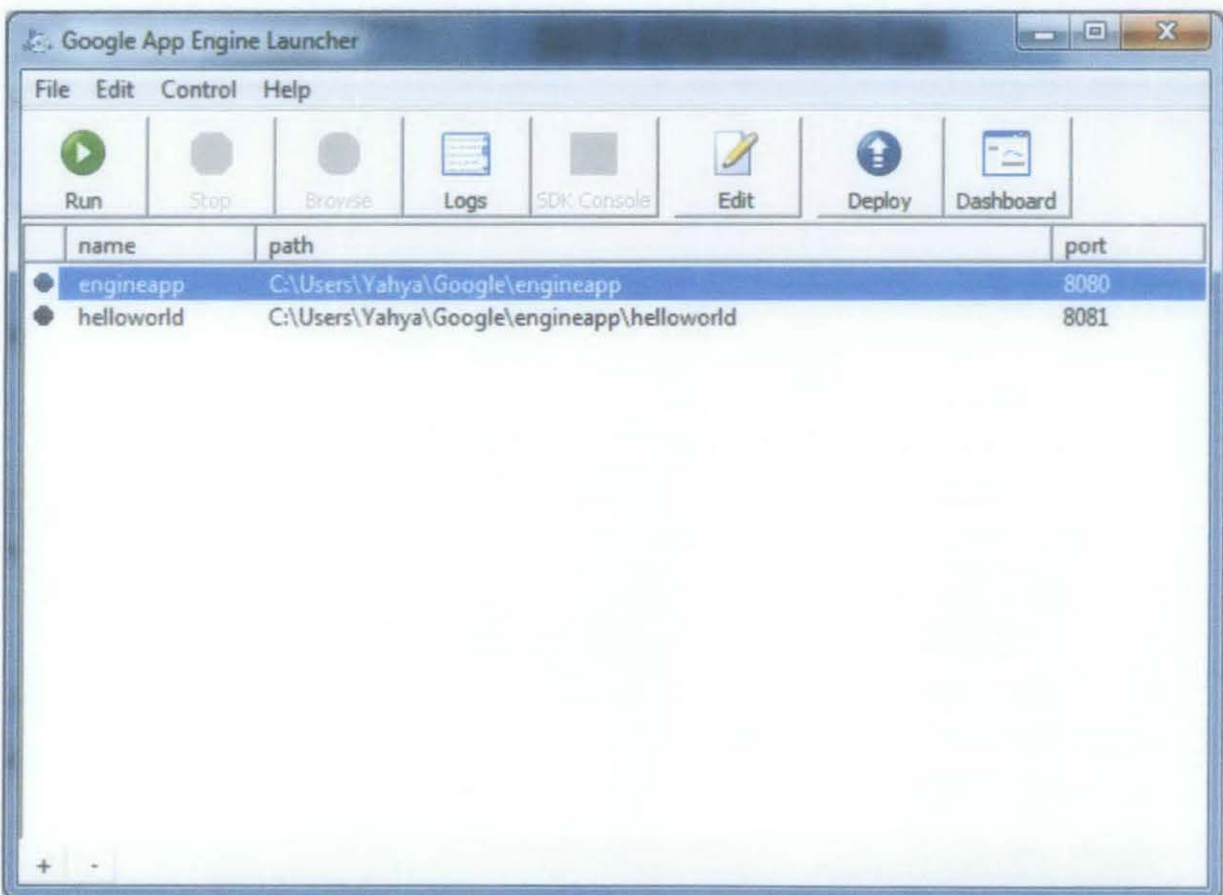


Figure 6: GAE App Launcher

The Python SDK requires Python 2.5 or higher to run but it's not compatible with Python 3.0 so far, it's also cross platform so it can run on either Windows or Mac environment, GAE also supports most of the original Python web frameworks, it also compatible with Django

framework, it's recommended to use Google's own web framework WebApp which comes included in the SDK of course.

3.3.3. Google Chart API

The Google Chart API is an extremely simple tool that lets people easily create a chart from some data and embed it in a web page. Google creates a PNG image of a chart from data and formatting parameters in an HTTP request. Many types of charts are supported, and by making the request into an image tag, people can simply include the chart in a web page. Originally it was an internal tool to support rapid embedding of charts within Google's own applications (like Google Finance for example). Google figured it would be a useful tool to open up to web developers. It officially launched on December 6, 2007.

The graphs can be made using a wizard or by hand to create the URL. For example in order to generate a 3D pie chart that shows a 60% positive and 40% negative section the URL will be

```
http://chart.apis.google.com/chart?
cht=p3&
chs=250x100&
chd=t:60,40&
chl=Postive|Negative
```



Figure 7: Static Pie Chart

The components

```
http://chart.apis.google.com/chart?
```

This is the base URL for all chart requests. (However,

```
cht=p3
```

The chart type: here, a 3D pie chart.

```
chs=250x100
```


The chart size (*width x height*), in pixels. Maximum chart size for all charts except maps is 300,000 pixels total, and maximum width or length is 1,000 pixels.

```
chd=t:60,40
```

The chart data. This data is in simple text format.

```
chl=Positive|Negative
```

The slice labels

All the graphs generated are static so in order to make interactive graphs, Google Visualization API is needed.

3.3.4. Google Visualization API

Google Visualization API is used to embed an interactive chart, graph, or other graphic onto your web page. Visualizations are interactive and also expose events, such as user mouse clicks, that enable the designer to write code to create great effects on your page, for example, to combine a map and a table that stay in sync when clicked. You can add a visualization either by using some simple JavaScript and HTML.

The following is a HTML and javascript example of an interactive pie chart.

```
<html>
<head>
<!--Load the AJAX API-->
<script type="text/javascript" src="http://www.google.com/jsapi"></script>
<script type="text/javascript">

// Load the Visualization API and the piechart package.
google.load('visualization', '1', {'packages':['corechart']});

// Set a callback to run when the Google Visualization API is loaded.
google.setOnLoadCallback(drawChart);

// Callback that creates and populates a data table,
// instantiates the pie chart, passes in the data and
// draws it.
function drawChart() {

// Create our data table.
var data = new google.visualization.DataTable();
data.addColumn('string', 'Task');
data.addColumn('number', 'Hours per Day');
data.addRows([
  ['Work', 11],
```

```

    ['Eat', 2],
    ['Commute', 2],
    ['Watch TV', 2],
    ['Sleep', 7]
  });

  // Instantiate and draw our chart, passing in some options.
  var chart = new
google.visualization.PieChart(document.getElementById('chart_div'));
  chart.draw(data, {width: 400, height: 240, is3D: true, title: 'My Daily
Activities'});
}
</script>
</head>

<body>
  <!--Div that will hold the pie chart-->
  <div id="chart_div"></div>
</body>
</html>

```

The results of that code

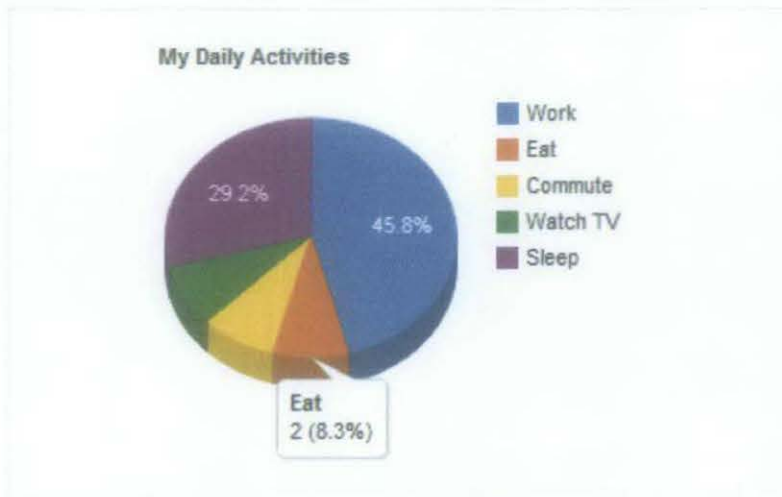


Figure 8: Dynamic Pie Chart

The difference between this chart and the one before it is this one is interactive, when the mouse is rolled over a slice of the chart, a comment box pops up with the size and percentage of the slice.

4. Results and discussion

4.1. Activity diagram

This web application has a very simple activity diagram which allows user to search, edit and view results.

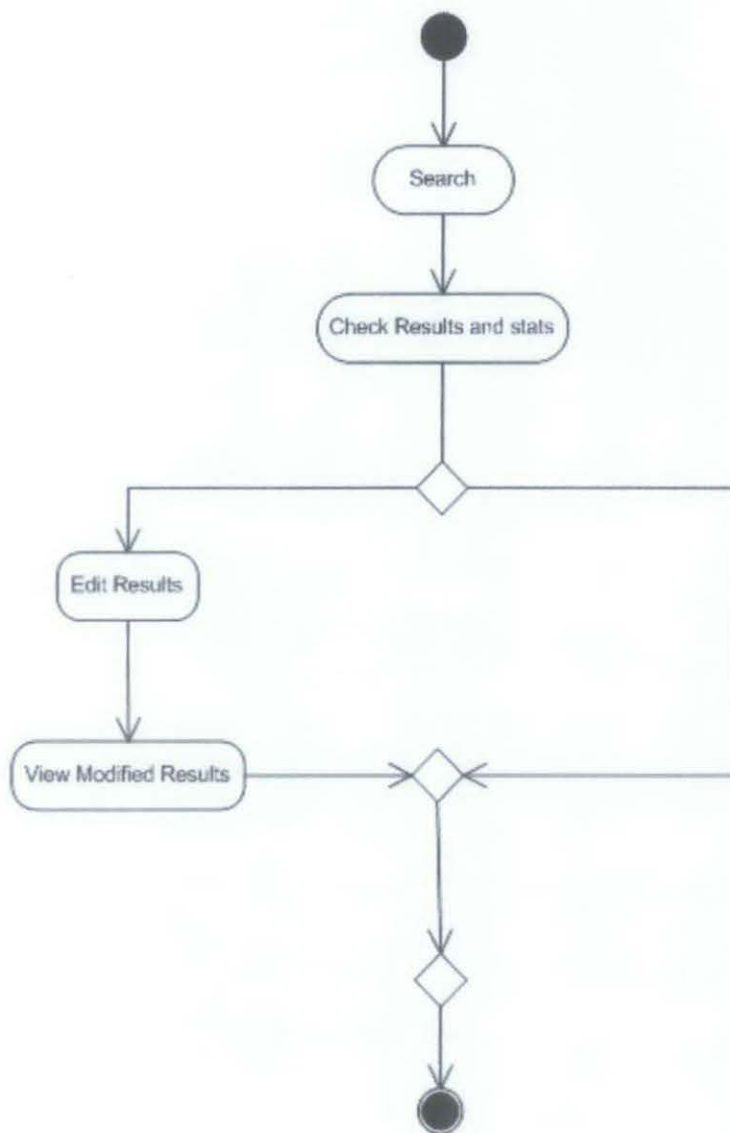
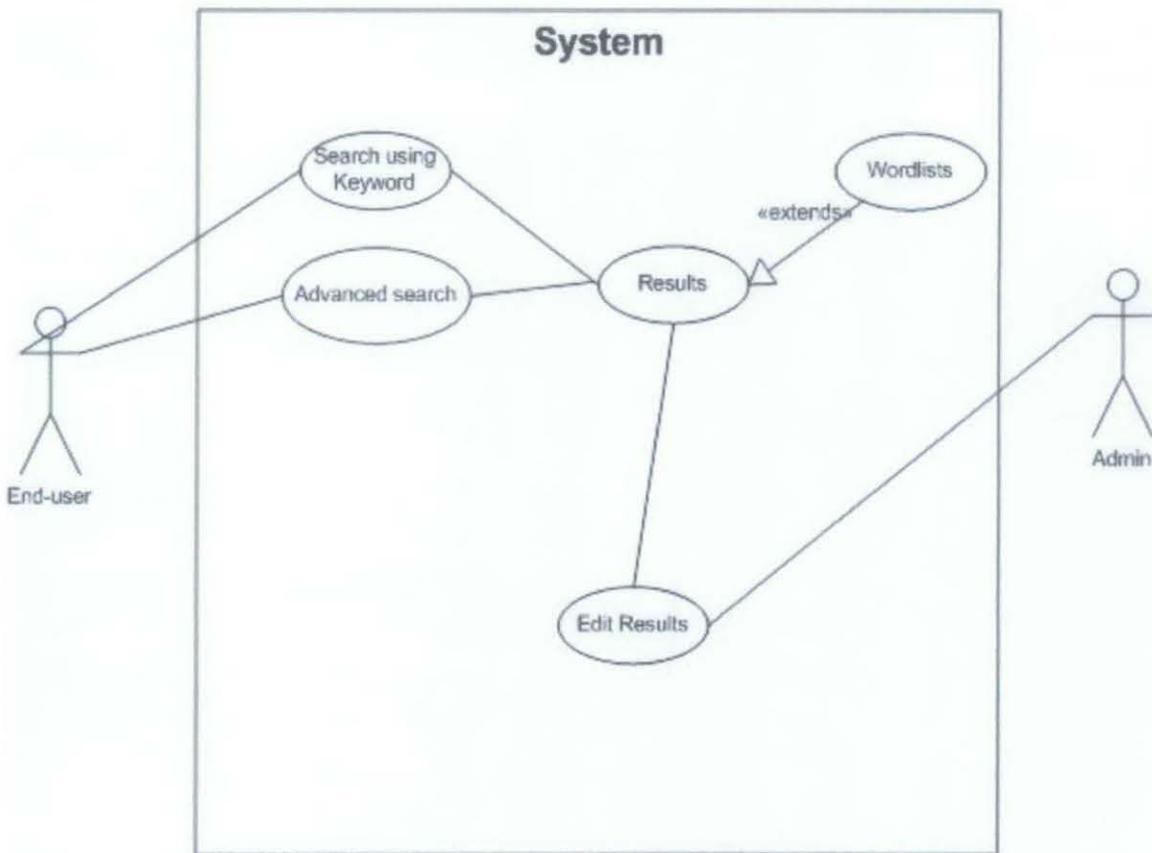


Figure 9 Activity diagram

4.2. Use Case Diagram



In this use case diagram, there are 2 actors:

1. Admin: the person in charge of checking result accuracy, wordlist aggregation and overall webapp maintenance.
2. End-user: normal user who can perform searches using a keyword or within a period of time, end user can correct results too.

4.3. Implementation

4.3.1. Fetching Tweets (search)

The first part of the process is getting all the tweets containing the keyword within a certain period of time, could be pre-determined or determined by default. The Twitter search API can return results in both atom and JSON format, both formats are lightweight data-interchange formats to exchange data between different processes or scripts, between Javascript and python for example.

Atom : XML based language used for web feeds like RSS, it is mainly used in the blogging communities to share latest headlines and by news websites to send data to web readers.

JSON : a data format, stands for JavaScript Object Notation which is a lightweight text based open standard used to exchange information between machines in a Human readable text.

I chose to use JSON because it's available in native python data, which means it won't be hard to implement by basically importing simplejson library and calling `simplejson.loads(json_string)`, also it won't make a difference if the final system runs on Google App Engine or other web application platform since simplejson is python library, not third party.

4.3.1.1. Twitter Search API

URL (basic syntax) : `http://search.twitter.com/search.json?q=keyword`

Twitter provides a wide range of parameters to customize the query required, the only required parameter is the keyword , the rest is optional depending on the needs of the developer. The following is a list of some available parameters:

- q (required) : Search query
- lang : restricts tweets to the given language given by an ISO 639-1 code. (English= en)
- rpp: number of tweets to return per page up to 100.
- Page: number of pages (starting at 1) to return, up to 1500 depending on rpp

- Until: return tweets generated before a given date. Date format YYYY-MM-DD
- Geocode: return tweets by users located within a given radius (geographically) given latitude/longitude. Example: <http://search.twitter.com/search.json?geocode=37.781157,-122.398720,1mi>
- Result_type: specifies what type of results to receive it could be:
 - o mixed: Include both popular and real time results in the response.
 - o recent: return only the most recent results in the response
 - o popular: return only the most popular results in the response.

4.3.1.2. URL encoding

Query strings should be URL encoded which means it should adhere to URI standards (Uniform Resource Identifier) for example if the query is “android 2.3” it should be encoded to “android%202.3” (notice the space character is replaced by %20).

I have coded an on-screen form for the end-user to enter the parameter values, upon completion of the form, the encoded URL is generated.

This process can be coded in either Javascript or Python, in Javascript this can be done using the built in function `encodeURIComponent(query_string)`, the following is an example I wrote for a script that translates the keyword into URL encoding

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>Encoder</title>

<script type="text/javascript" charset="utf-8">

function encoder(){
```

```
    var keyW =
encodeURIComponent (document.getElementById('keyToencode').value);

    document.getElementsByTagName('h2')[0].innerHTML = "Encoded:" + " "+
keyW;

    document.getElementsByTagName('h2')[1].innerHTML = "Decoded:" + " "+
decodeURIComponent (keyW);

}
</script>
</head>

<body>

<form action="" method="post" name="form1" target="_self"><label>Encode: <br
/>

<input name="keyword" type="text" id="keyToencode" value="android 2.3
(gingerbread)" size="50" maxlength="140" />

</label>

<input name="GO" type="button" value="submit" onclick="encoder()" />

</form>

<h2></h2>

<h2></h2>

</body>

</html>
```

4.3.1.3. Restrictions and Challenges

Twitter search API has some annoying restrictions, the most crucial one is it limits the number of results to 100 for every request, it's understandable why this is done since a search function without a limiter can be easily abused and dramatically reduce the efficiency of the system. This is a major challenge for developers that use this API, since it's very important to get all the raw results in order to generate informative statistics.

The other crucial problem with the system is it can only make 150 requests per hour, each request is a 100 tweets per result, this is a huge problem when dealing with very popular keywords like "iPhone" or "Google", but it's no big deal for less popular keywords like "Jeremy Clarkson" for example.

Running the fetch code recursively would be a nightmare if the keyword is popular, this will cause the system to dramatically slow down plus the hourly limit of 150 requests will be finished within 5 to 10 minutes.

For the sake of building and testing a prototype of an alpha version, I limited the requests per query (keyword) to 3 requests which return 300 recent tweets, I think this would be enough to test the function and provide enough result to test the analyzer.

4.3.2. Wordlist handler

The basic idea behind the classification method used is to make 2 wordlist, one that includes "POSITIVE" words like "happy, fun, great, love, ,etc" and a "NEGATIVE" wordlist that includes negative words like "sad, horrible, terrible, ,etc", and the application counts how many positive words vs. how many negative words and this will classify the sentence (or tweet) to either positive or negative.

Although the system sounds very simple, but theoretically it works very well given a good wordlist on both sides, but the more importantly, the web application will function much better if I append a learning mechanism to allow the end-user to correct the result, for example if a sentence is a positive one but it was added as a negative, the user can change that, and the system will take it into consideration next time.

I initially thought the wordlist should be saved in an external file on the server then it can be loaded when needed, but after some online research, it turns out it's a better practice if the words were saved in an internal data structure because:

1. It's much faster to crawl compared to an external file.
2. It's dynamic (easier to modify , delete, or add new entries)

Obviously a stack or a queue is out of the question since this will take an immense amount of time to search, in Python both stacks and queues are called Lists with multiple different implementations. Array (also known as tuple in Python) is okay but it's still not optimum, this brings us to Sets which is defined as unordered collection with no duplicates, this is very useful since it gives the code the ability to find duplications very easily, it also allows the code to find words on both negative and positive sets, these words will be used to make a third set which is called the neutral since the system can't count the word as positive or negative.

4.3.2.1. Class BayesData

Wordlists in this case are treated like objects in Python, the following is the definition of the object wordlist.

```
class BayesData(dict):

    def __init__(self, name='', wordlist=None):

        self.name = name

        self.training = []

        self.wordlist = wordlist

        self.tokenCount = 0

        self.trainCount = 0

    def trainedOn(self, item):

        return item in self.training

    def __repr__(self):

        return '<BayesDict: %s, %s tokens>' % (self.name, self.tokenCount)
```

The previous code defines a data type called BayesData, this type has 5 different attributes which are:

1. Name: name of the wordlist.
2. Training: it's the action adding words to the wordlist or removing incorrect words.
3. Wordlist: return wordlist name. (default is None).
4. TokenCount: how many tokens (words) in the list (default = 0)
5. trainCount : how much training did the wordlist have, this calculates new words added plus words that was removed, which is also considered training since it leads to better results.

the `trainedOn` returns all the words in the wordlist, it's kind like `get/set` methods in Java.

`__repr__` basically returns the name of the wordlist and how many words does it include.

this class sole job is to define the datatype `BayesData`, however all operations (`add word`, `add wordlist`, `delete`, `train`, `save`, ..., etc) is implemented in a different class, this class is called `Bayes()`

First thing to do is to initiate the class and define the datatype as `BayesData`

```
def __init__(self, tokenizer=None, dataClass=None):

    self.dataClass = BayesData

    self._tokenizer = Tokenizer()
```

After initiation comes the operations available like adding or removing wordlists:

```
def newWordlist(self, wordlistName):

    # create new wordlist

    self.dirty = True

    return self.wordlists.setdefault(wordlistName,
self.dataClass(wordlistName))
```

```
def removeWordlist(self, wordlistName):

    del(self.wordlists[wordlistName])

    self.dirty = True
```

```
def renameWordlist(self, wordlistName, newName):

    self.wordlists[newName] = self.wordlists[wordlistName]

    self.wordlists[newName].name = newName

    self.removeWordlist(wordlistName)
```

```
self.dirty = True
```

The dirty statement is a way to check if there has been any changes to self (this class) it's not essential but it's a good practice to keep track of various errors.

The following methods are to retrieve (GET) data or from a certain wordlist:

```
def wordlistData(self, wordlistName):
    return self.wordlists[wordlistName].items()
```

The following code implements the function of Training (adding words to the wordlist), and also unTraining which is deleting words from the wordlists but it still counts as training,

```
def train(self, wordlist, item, uid=None):
    tokens = self.getTokens(item)
    wordlist = self.wordlists.setdefault(wordlist,
self.dataClass(wordlist))
    self._train(wordlist, tokens)
    self.corpus.trainCount += 1
    wordlist.trainCount += 1
    if uid:
        wordlist.training.append(uid)
    self.dirty = True

def untrain(self, wordlist, item, uid=None):
    tokens = self.getTokens(item)
    wordlist = self.wordlists.get(wordlist, None)
```

```

if not wordlist:
    return

self._untrain(wordlist, tokens)

self.corpus.trainCount += 1

wordlist.trainCount += 1

if uid:
    wordlist.training.remove(uid)

self.dirty = True
    
```

The reason trainCount increases even when we delete words from the list is that it's still training when we remove wrong words from the list which enhances the overall experience and accuracy.

4.3.2.2. The Tokenizer()

This class is a very simple one that takes a string or multiple words and return them as tokens and lower case to make sure the comparison is correct. It uses Regular expressions library (also known as Regex), it's available through the "re" module.

```

import re

class Tokenizer:

    WORD_RE = re.compile('\w+', re.U)

    def __init__(self, lower=False):
        self.lower = lower

    def tokenize(self, obj):
        for match in self.WORD_RE.finditer(obj):
            if self.lower:
                yield match.group().lower()
    
```

```

else:
    yield match.group()

```

Compile() : expressions are compiled into pattern objects.

\w+ : since \ is a reserved character, the first one is to cancel that. \w+ matches any alphanumeric character , the range is [a-zA-Z0-9_]

Tokenize(self, obj) : checks all tokens and make sure everything is in lowercase.

4.3.2.3. *getProbs() and Guess()*

Guess() function is responsible for determining if any of the tokens (single words) are equal to any of the words in the wordlists and determining the probability using the getProbs() function

```

def getProbs(self, wordlist, words):
    """ extracts the probabilities of tokens in a message
    """
    probs = [(word, wordlist[word]) for word in words if word in
wordlist]
    probs.sort(lambda x,y: cmp(y[1],x[1]))
    return probs[:2048]

def guess(self, msg):
    tokens = Set(self.getTokens(msg))
    wordlists = self.wordlistProbs()

    res = {}
    for pname, pprobs in wordlists.items():
        p = self.getProbs(pprobs, tokens)
        if len(p) != 0:
            res[pname]=self.combiner(p, pname)
    res = res.items()
    res.sort(lambda x,y: cmp(y[1], x[1]))
    return res

```

4.3.2.4. *Robinson spam algorithm*

The following algorithm is a spam detection open source algorithm used to filter the tweets, it's needed to make sure the tweet sample doesn't include spam like links to articles and such.

```

def robinson(self, probs, ignore):
    """ computes the probability of a message being spam (Robinson's
    method)
        P = 1 - prod(1-p)^(1/n)
        Q = 1 - prod(p)^(1/n)
        S = (1 + (P-Q)/(P+Q)) / 2
    """

    nth = 1./len(probs)
    P = 1.0 - reduce(operator.mul, map(lambda p: 1.0-p[1], probs), 1.0)
    ** nth
    Q = 1.0 - reduce(operator.mul, map(lambda p: p[1], probs)) ** nth
    S = (P - Q) / (P + Q)
    return (1 + S) / 2

def robinsonFisher(self, probs, ignore):
    """ computes the probability of a message being spam (Robinson-Fisher
    method)
        H = C-1(-2. ln(prod(p)), 2*n )
        S = C-1(-2. ln(prod(1-p)), 2*n )
        I = (1 + H - S) / 2
    """
    n = len(probs)
    try: H = chi2P(-2.0 * math.log(reduce(operator.mul, map(lambda p:
    p[1], probs), 1.0)), 2*n)
    except OverflowError: H = 0.0
    try: S = chi2P(-2.0 * math.log(reduce(operator.mul, map(lambda p:
    1.0-p[1], probs), 1.0)), 2*n)
    except OverflowError: S = 0.0
    return (1 + H - S) / 2

def chi2P(chi, df):
    """ return P(chisq >= chi, with df degree of freedom)

    df must be even
    """
    assert df & 1 == 0
    m = chi / 2.0
    sum = term = math.exp(-m)
    for i in range(1, df/2):
        term *= m/i
        sum += term
    return min(sum, 1.0)

```

4.3.2.5. Challenges and restrictions

The problem with this technique is on the long run, the wordlists will keep growing which means the time needed to crawl the wordlists will increase which slows down the whole system, and because of that the wordlists need to be modified and up to date, this can be done by checking for duplications, making sure words doesn't exists on 2 wordlists in the same time.

5. Conclusion

This research was intended to discover the potential of using probability theory to find out the sentiment of a sentence to turn web chatter (tweets in case of this research) into usable statistical info that can be used to help make decisions.

This research proved that it's possible with further research and development, this technique could be enhanced in terms of speed and accuracy, it can also enhance the world of marketing by giving real-time genuine surveys for product research and as a feedback method for manufacturers.

5.1. Future development

This project can be modified to integrate it with multiple social networks like Facebook and yelp, this will increase the web chatter and the overall number of users.

The ability to share results on multiple networks will be available in the next implementation to give the app more exposure.

Reference

1. Bo Pang; Lillian Lee, *Opinion Mining and Sentiment Analysis*, Volume: 2, 2008.
2. Harry Zhang, *The Optimality of Naive Bayes*. FLAIRS 2004 conference. Page 1
3. Tetsuya Nasukawa; jeonghee Yi, Sentiment analysis: capturing favorability using natural language processing, IBM Research, Almaden Research Center, San Jose, CA
4. I. Rish, An empirical study of the Naïve Bayes Classifier, T.J Watson Research Center, NY
5. Twitter API Wiki , http://dev.twitter.com/pages/every_developer
6. Twitter User Statistics , http://www.huffingtonpost.com/2010/04/14/twitter-user-statistics-r_n_537992.html
7. Twitter Statistics , Quantcast.com statistics engine <http://www.quantcast.com/Twitter.com>
8. Twitter Rate Limiting , <http://dev.twitter.com/pages/rate-limiting>
9. OAuth Overview, <http://hueniverse.com/2007/10/beginners-guide-to-oauth-part-i-overview/>
10. Google App Engine Python Overview,
<http://code.google.com/appengine/docs/whatisgoogleappengine.html>
11. Google chart Tools, <http://code.google.com/apis/chart/>
12. <http://christophe.delord.free.fr/en/index.html>
13. <http://www.divmod.org/trac/wiki/DivmodReverend>
14. Alec Go, Richa Bhayani , Lei Huang , Stanford University
<http://www.stanford.edu/~alecmgo/papers/TwitterDistantSupervision09.pdf> , Twitter Sentiment Classification using Distant Supervision.
15. Robinson Spam filtering : <http://www.linuxjournal.com/article/6467>