Tweet Aggregator

Kévin Serrano, Gianni Scarnera, Clément Moutet, Timo Babst, Pierre Gouedard, Adrien Ghosn, Joris Beau, Mathieu Demarne, Cédric Bastin, Lewis Brown.

Supervisor: Mohammed El Seidy

Big Data

Spring 2014

Abstract

TweetAggregator project recently renamed to CrossTalk is the result of the Big Data class. This application offers the possibility to display for a specific area the correlation between Tweets according to two keywords.

Github repository: `http://www.github.com/TweetAggregator/TweetAggregator`

Keywords: Clustering, Big Data, Twitter, Aggregation

Contents

1 Introduction

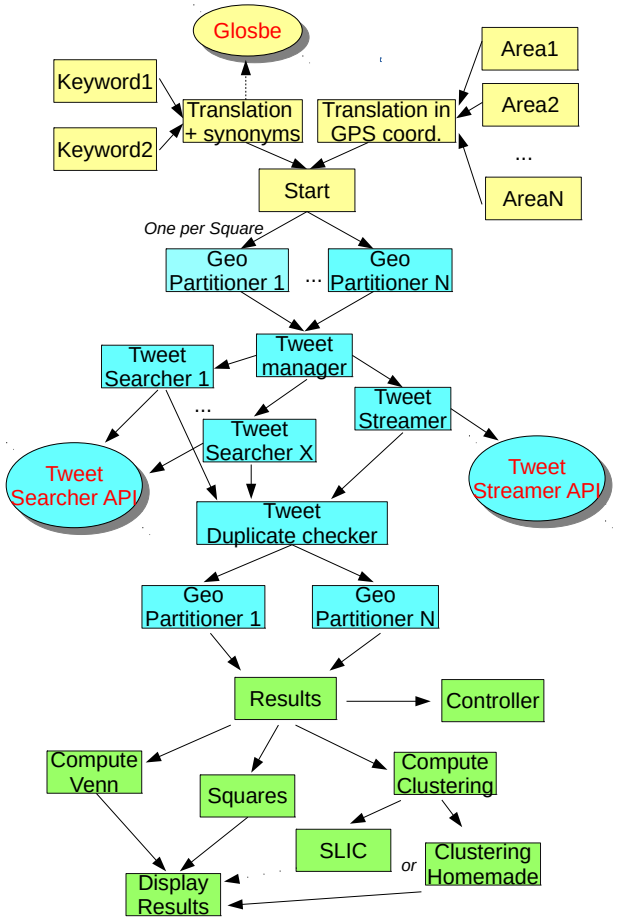
1.1 Description

The goal of the project was to create a minimalistic tool allowing users to select two topics, display the related Twitter activity on a map in real-time, and discover the trends linked to the given keywords. A web application based on the Play Framework [1] is the solution chosen since it fulfils the needs of this purpose (tweet collecting, result computation and display, ...). The implementation consists of Scala and JavaScript code and HTML for the UI. From the user's point of view, it is possible to input two keywords, translate them and find synonyms, define areas on the world map and then launch the search. Once the search has been launched, results can be dynamically displayed on the map for the selected areas. Displays can be either clusters or tweeting activity along with the corresponding Venn diagram.

1.2 Motivation

All public messages posted on Twitter are freely accessible through a set of APIs, allowing analysis of the way tweets propagate using some statistical tools.
Tweetping already provides a real-time feed display. All the new tweets are put onto a map, which allows interesting views. It is for example possible to see the sun rising above the US while the number of tweets grows from East to West, but it is impossible to read the messages, as there are way too many of them. One thing is lacking from this tool is user defined filtering of the tweets. Trends inside the Twitter platform already provide real-time feedback on various subjects, but not specifically displayed on a map. This could have interesting use cases, such as comparing what Swiss-German and Swiss-French people think about a popular vote, or analyzing the way people see Barack Obama in swing states. Such a thing is already partially implemented by Trendsmap.
But this tool could be made even more powerful by adding something currently lacking: population-based statistics. The idea here is to search for various trends and try to relate the populations of the users posting about them. Such analyses could for instance be displayed as a Venn diagram on the side of the map, allowing some interesting ways to observe correlation and intersection between trends. This tool could for example try to answer such questions as: do Swiss citizens speaking about the vote of February 9th also speak about Erasmus? Is it more the case in Zurich or in Lausanne?

1.3 Design overview



:Setup part: selection of keywords, translation & synonym (Globse API), selection of the Areas.

:Gathering part: GeoPartitioner sub-segments the squares, sends all requests to the manager, manager start a bounded number of Tweet Searcher and Tweet Streamer.

:Result and display part: Controller takes action onto the Tweet Manager, compute square transparencies, start the clustering algorithm and display cluster, compute Venn Intersection.

:External APIs. :Implemented code.

2 Getting the data

2.1 Tweet Manager

The Tweet Manager was designed to be the only interface with the sources of data. Starting a research requires sending a list of queries to the Manager, which will buffer them until all have been received.
The Manager will then split evenly the queries on multiple actors, both for the searchers and the streamers and select the proper update rate based on the number of keys available in the configuration.
Following some discussion and after some tests, we decided to avoid having dynamic reallocation of searchers, which means that once a research on a list of queries has started, the list of queries cannot be changed without stopping the whole search. This is a design choice to avoid heavy load on the Akka Scheduler and to avoid duplicated data as much as possible.
The manager is also in charge of coordinating the Searchers if the system is kicked by the Twitter API. Such a situation could arise if two searches were launched simultaneously with the same API keys or if we decide to have an aggressive search.
Moreover the Tweet Manager launches a Duplicate Checker which keeps track of the latest tweet IDs. Due to the way the Search API is designed, we cannot research on squares, but only on circles. We had to increase the size of the circles of research, leading to some overlapping we then need to deal with internally, in order to be sure to cover the whole map.

2.2 Tweet Searcher

The Twitter Search API is a great choice since it allows narrow filtering based on localisation (delimitation of squares of research), as well as logical operators inside the query string. The API keeps track of the previous requests sent in order to deliver only the newly indexed tweets while doing a callback. This is very suitable, since we don't need to keep track of tweet IDs to avoid duplicates.
The API also proposes a way to get more tweets by looking back into the past (one ping to the API returns at most 100 tweets, but it's possible to ask, in a second request, to get older data). Since we do not want to go too much in the past and since we assumed that we were already getting enough data without using this option, we discarded it.
The limitation of 450 tweets per 15 minutes is our biggest bottleneck in terms of data. Fortunately the Searchers were designed to be able to search with multiple keys and learn which

squares they should ask more often. They can also handle situations where the system is kicked by the API due to a heavy load, or in case of network failure.
Like the rest of our system, the Searcher is built on the Akka Actor System and therefore interacts with the other parts by message-passing. Depending on the properties of a machine or in case of multi-node deployment, the number of Searchers can be significantly increased using simple parameters stored in the configuration files. Hence, the data source can scale.

2.3 Tweet Streamer

Using the Streaming API, the Tweet Streamer is responsible to get brand new tweets, not yet indexed by the Search API. Due to the limit of 1% imposed by Twitter, the amount of tweets that will be found in both searches is small, but will still have to be considered. The amount of interesting data from the Streamer might be smaller, as we cannot ask the API to filter directly for both keywords and localization. We can also only have a few streams per connection. To compute interesting tweets, we launch one stream on the Tweet Streamer with all the location parameters. We do this because we only can launch few streams. Because the number of location parameters is limited to 20, we will also merge the location parameters that are close to each other, i.e. the squares. We will thus receive a lot of tweets in disorder.
This is typically a Big Data issue, we receive unparsed data from different sources that we need to put in the same format. In this case, we have one listener per word and per square, and we have a lot of data that is coming that will fit at least one square, but that will not contain a desired word in most of the cases.
From the received tweets, we look locally at their location parameter and the words they contain, search this for the correct listener, and send the tweet. We will receive a lot of data, and lots of tweets that are in the desired location but do not contain the keywords we are interested in. To be able to rapidly discard non relevant tweets, each time the Tweet Streamer receives a ping, he will discard all the non relevant tweets in his InputStream until he finds a correct one to return. A correct tweet is a tweet containing one of the desired keywords, because we already know that they will be in one of the desired squares, because we specified it in the parameters.

2.4 Data Simulator

In order to be sure our program would resist to a heavy load of data, a source of locally generated tweets can be used to test this kind of scaling. In addition to the need for a lot of data to be generated, we need the simulation to be as most complete as possible to take into account that some words are more popular than others, and that the emission of tweets with a specific set of keywords is not uniformly distributed.
The Tweet Tester will analyse the TweetQueries and their listeners and extract the list of desired words and their corresponding listeners. For each word, it will randomly assign a number N between 1 and the total number of queries, that will represent the importance of the word. Then, for each listener of this word, it will assign a random number L between 1 and N that will represent the interest of this zone for a given word, in order not to have a uniform distribution

of the tweets.
Once this is computed, the Tweet Tester will feed the listeners with tweets containing a random integer for a message. The Tweet Tester will automatically ping itself at a high rate (configurable in the config file), and for each ping ask N times for a random listener to send his L number of messages.

3 Map display

3.1 General introduction

The map library is used for three different sub-parts of our project. They represent an essential part of the front-end due to their unique ability to make sense of the large amount by abstracting away numerical data and making it implicit by showing it on a map with geometrical objects and their color intensity which can represent information as well. After some investigation in freely-available and open source libraries we decided to use the good-looking and lightweight PolyMaps tool which is entirely based on Javascript and vector graphics which made interfacing easy such as overlaying simple 2D objects. It was thus very easy to add SVG elements to the map, polygons such as rectangles are supported natively through the GeoJSON standard whereas circles need to be rendered individually. As most other open source mapping services PolyMaps uses the geographical information provided by OpenStreetMap in combination with some useful APIs from CloudMade.

3.2 Region selection

In the first view the user can move around the map and change the zoom level to show different parts of the world. While doing so he can also select geographic regions from which he wants to analyze the Twitter data. He can do so simply by dragging a new box on the screen. The coding effort behind this small action is quite big and can be decomposed in several steps. First of all the normal drag and drop action on the map is to move the navigate around. Whenever the user decided to draw a new region, which is done by clicking on a corresponding button, the focus is removed from the map such that the view is fixed. In the second step the movement of the mouse creates a rectangle on top of the map whose size is changed depending on the new position of the cursor. This rectangle is an SVG element which is not integrated with the map, only overlaid. When the user releases the mouse button the size of the rectangle longer changes, the boundaries of the given object are then translated to geographic data, meaning latitude and longitude, which is done using an API call to the PolyMaps library. Those coordinates in turn are used to create a GeoJSON polygon object which can directly be “stuck” to the map which means that when the user moves the map the created object stays on the corresponding geographical position. To distinguish between those different phases we used different colors to notify the user when the rectangle has been stuck to the map. The described action can be repeated to created several selected regions on the map.

When the user clicks on the button to save the selected regions, all geographical data is communicated to the server by saving them in a hidden form field in form as Json data which is transmitted via a POST request.

3.3 Tweet density display

On the server side the selected regions are subdivided in even smaller parts such that the analysis can be more fine-grained. For each subregion the number of tweets is collected and saved, this action is repeated for each one of the two keywords as well as for their intersection. Whenever the user wants to visualize this data he chooses the "mapresult" tab from the navigation in the UI, a view is presented where each subregion of the original regions has a color opacity relative to the number of tweets that occurred in that region. As described above each subregions is represented as a GeoJSON object an can thus be sticked to the map. Furthermore each keywords has a specific color assigned such that the user can quickly identify which part of the diagram on the map corresponds to which keyword. We used those colors uniformly between the venn diagram, region density and clustering views. We also included checkboxes which allow the user to choose which of the 3 sub-diagrams visualize on top of each other.

3.4 Clustering display

As the GeoJSON does not allow circle objects, meaning a disk with a radius given in some geographical related format, we had to use pure SVG circle objects and calculate their corresponding position depending on the current view on the map. This includes calculating their x, y pixel position on the screen as well as their radius which can change when the users zooms. The clustering algorithm, which runs on the server side, performs several levels of clustering which are passed as a JSON string to the interface. Through a UI slider the user can select the level of clustering he wants to visualize. Similar to the previous view the clustering also used 3 different colors to represent the clusters corresponding to each keyword as well as for their intersection to allow quick visualization of hot-spot areas for specific topics.

3.5 Conclusion

One important thing to note is that throughout these different views we always save the users select view-position which is done by recording the geographic center of the current view as well as the zoom level. This information allows us to present the same view to the user when he changes from one step to the other such that he does not need to navigate back to the same spot manually.

4 Intersection algorithm

4.1 Partitioning the map

Adrien and Lewis designed and implemented a scheme for partitioning queries to the Twitter API over the geographical area that the end-user selected. We could interact with the API by specifying individual squares in which to look for tweets. The initial design attempted to perform this task dynamically, using feedback from the tweets already received to increase the number of queries performed in certain areas. Thus, if there appeared to be many tweets in some place, more queries would be spawned to increase the granularity of the sampling in that area.

This scheme aimed to reduce the number of requests made to the Twitter API. However it created extra work for the server, which would need to deduplicate tweets and those tweets would take longer to collect. Instead, since the number of API requests didn't seem to be the limiting factor, we spawned requests at the maximum granularity right from the start.

Since we were using the Play framework, the implementation was performed in Akka. We created a GeoPartitioner actor which would perform the appropriate splitting for each user-selected square and start the queries. We initially had a separate Counter actor which would count tweets corresponding to a query. However Akka actors are actually very expensive and creating one counter for every query created a large amount of unnecessary overhead. In the final version these counters were moved into the GeoPartitioner itself.

Finally we had to decide on an interface that would be used to retrieve results from the GeoPartitioner. The results would be displayed using squares of varying opacity to distinguish between areas with different amounts of tweets, and it should be possible to get the counts of tweets in each subarea, possibly filtering out those not contained in a specified place. We simply created a message for each of these tasks, to which the GeoPartitioner responds with the desired data.

4.2 Interaction between the front and back end

Joris and Lewis designed and implemented the actions which are called by the front end once the user has selected the search parameters. These were implemented using Play `Action`s which are invoked upon HTTP requests to particular routes.

There are a handful of these actions, the most important of which are for starting the data collection and computing the display data. When the `Start` action is invoked the `GeoPartitioner` actors are instantiated and started with the desired parameters. In order to be able to display in-

tersections between the requested keywords, we spawned three partitioners for each user-selected area: one for each of the keywords, and another for their intersection. Twitter's search API allows keywords to be combined with logical conjunctions and disjunctions, which we used to create these queries.

When it comes to displaying the data, there are two requests the user can make: they can request either clustered data or squares with opacities. In both cases, the tweet counts for the Venn diagram are assembled and filtered to whatever square is currently displayed. The desired data (either clusters or opacities) are collected and transformed into the format expected by the front end, and everything is passed to a view which is shown to the end-user. There is also an action to refresh the Venn diagram to display the statistics for the area currently visible to the end-user.

In order to test these actions, it is helpful to be able remove the dependency on the Twitter API. By doing this, it is possible to write unit tests which only run the task controller code. This reduces the number of reasons for which the tests can fail, making them more reliable and useful. To accomplish this, we proceeded as is done in the Cake pattern. All of the actors that were used were put into an abstract trait, `actors`. There was then a concrete implementation of this trait which used the real actors, contacting the Twitter API and so on, and a test implementation which mocked those actors instead.

5 Clustering algorithm

5.1 Agglomerative clustering

In order to be able to extract as much information as we could from the gathered data, we decided to implement a clustering algorithm. Lewis and Adrien consulted some literature on the subject, and decided to devise a hierarchical clustering algorithm in order to have different levels of clustering to display on the map according to the zoom.
The challenging part was to find a good metric for clustering since, at that point, we had very little statistics about the data that we were able to gather. We tried to find some general relation that would adapt to any reasonable set of results. This algorithm is called "HomeCooked" clustering algorithm.
The algorithm proceeds as follows: The GeoSquares (which are the smallest geographical unit of tweet collection) are abstracted away and replaced by "leafClusters", that is, a simple square on an n x m grid. A cluster is a rectangle formed by contiguous leafClusters which are ordered according to the position of their center on the grid. They do not interleave.
At the very first level, we have the raw data (unclustered results, the basic GeoSquares). The higher we get, the more clustered the data becomes.
Each level results from applying the clustering algorithm to the previous level.
At each step, for each cluster, we look at the list of clusters that are after the current one in the ordered list, and try to find the one such that the area formed by aggregating them (that is, if we denote by i and j the clusters to be merged, the rectangle with bottom-left corner coordinates $(min_{k\in\{i,j\}}x_k, \min_{k\in\{i,j\}} y_k)$ and top-right corner coordinates $(max_{k\in\{i,j\}}x_k, \max_{k\in\{i,j\}} y_k)$) is $\leq \alpha \times b \times$ *(area of the whole region)* and the tweet density is maximal and superior to $(1-b) \times \frac{\textit{(total number of tweets)}}{\textit{(total area)}}$, where b corresponds to the level at which we are clustering and α is an application specific corrector constant.
The intuitive understanding behind this algorithm is that, at each level, we consider only regions of reasonable area in order to simulate proximity between areas. The higher the level, the greater distance we're allowed to aggregate. This generates a hierarchy since near clusters are clustered first and then, in the following steps we are able to cluster further ones. Another way to see that is the following: at low levels we allow only numerous small clusters, while in higher levels we have less clusters, but they are bigger. The second condition is there to ensure that we cluster elements only when their joint density is significant, with respect to the level at which we are. For low levels, we require that the region have only a small percentage of the overall density. The higher we get in the clustering level, the bigger this percentage has to be. As a result we get

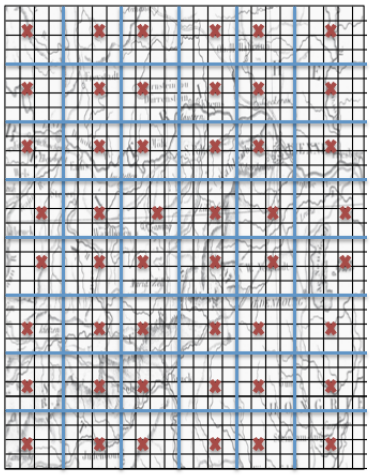
a nice hierarchy of clusters, where each cluster is formed only when the area considered seems interesting in terms of density of tweets.
Once the interface with the application was ready, we were able to test our algorithm and were satisfied with the results.
Pierre verified the theory behind our HomeCooked algorithm and proposed another one, based on a K-means algorithm called SLIC (see next section).
Our HomeCooked algorithm was finally kept and integrated into the application. At some point, we discovered that the implementation was a bottleneck for the application when we increased the number of initial GeoSquares. Luckily, Adrien was able to refactor it so that it is now more efficient. He also rewrote it in a Map-Reduce form so that, if ever needed, the addition of a Spark or Hadoop implementation could be easily done.
Finally, for the display, we decided to approximate a cluster by its center and a disk whose diameter is computed so that the area of the disk corresponds to the area of the original rectangle corresponding to the cluster. The opacity of the disk is computed according to the density of the cluster. We also added a slider on the View so that we can select the level of clustering to display.

5.2 SLIC algorithm

We implemented this algorithm, but did not integrate it to the final version of the project since the result was not really satisfactory. Adding it to the display shouldn't be hard though, since all the needed interfaces already exist.
The SLIC (Simple Linear Iterative Clustering) algorithm [6], published in June 2010, introduces a novel algorithm that clusters "pixels" in the combined n-dimensional density and image plane space to efficiently generate compact, nearly uniform superpixels. Due to its simplicity and its efficiency the algorithm quickly became very popular and often use in object class recognition, medical image segmentation and many other domains.
The SLIC algoithm is basically an adaptation of K-means for superpixel generation, with two important distinctions:

- The number of distance calculations in the optimization is dramatically reduced by limiting the search space to a region proportional to the superpixel size. This reduces the complexity to be linear in the number of pixels N and independent of the number of superpixels k.
- A weighted distance measure combines density and spatial proximity while simultaneously providing control over the size and compactness of the superpixels.

More information on the SLIC algorithm can be found in [6] (original paper) and [7] (State of the art of SuperPixel techniques).
The algorithm takes as input the segmented map image and, in order to produce roughly equally sized homogeneous regions, distributes K cluster centers $C_k, k \in \{1, \ldots, K\}$ uniformly over I. Let $S = \sqrt{K}$, It is done by over-sampling I, creating $S \times S$ "superpixels", placing C_k at the center of superpixel (figure 2) and computing the average density of tweets in the superpixel.



We then run a local K-means using the following distance:
$D_s(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ where x_i and y_i refer to the spatial location of pixel i and j

$D_t(i,j) = |(d_i - d_j)|$ where d_i refer to the density of tweets in pixel i

$D(i,j) = \sqrt{D_t^2 + (\frac{D_s}{S})^2 \times \gamma}$

The free parameters of the algorithm are K, γ and *Low threshold*. The parameter K impacts the granularity of clustering, it should be ≥ 10, γ is a parameter that allows a trade off between homogeneity in space and homogeneity in density of clusters of tweets. Finally *Low threshold* sets the number of iterations that will be necessary for the algorithm to converge.
A more detailed and complete adaptation of SLIC to our project can be found in here.

6 Venn diagram

6.1 Overview

The goal is to create a visualization of the result of the tweet gathering according to the keywords. Since we want to see the intersections between different keywords, it was decided that Venn diagrams would be ideal for this goal.
A Venn diagram or set diagram is a diagram that shows all possible logical relations between a finite collection of sets. The challenges implementing such diagrams is, first, to manage the placement of the circles in relation to each others. Secondly, we need to scale it for possibly up to ten keywords, which also brings optimization problems such as to where place so many circles and intersections automatically.

6.2 D3.js and placement optimization

D3.js is the tool that was used to create the venn diagram, as it allows to do that easily. The main problem was to make the positioning of the circles to be dynamic, and scale with more than two circles (even if it was not used in the end). To do this, the free library of Ben Frederickson was used, which implements the automatic placement of the circles using different optimization techniques, given sets of size of circles, size of the intersections.

6.3 Integration to play

Now that the Venn diagrams were working by themselves, they needed to be integrated with Play and the rest of the project. Play allows us to simplify the flow of data into the API library calls a lot, since it is possible to forward the inputs directly into the function call.

7 Keyword translation and synonym

7.1 Keyword entering and keyword translation

7.1.1 Translation

Goal

The goal was to be able to detect a higher number of relevant tweets by automatically looking for synonyms and translations of the keywords entered by the user.

Finding an API

Timo and Joris looked for an open and free API that enables us to accomplish this task. After trying out some of the alternatives that finally did not meet our requirements, we found Glosbe. Timo contacted the developers of Glosbe in order to ensure that we will not be blocked by making too many requests on the API. In the meantime, Pierre looked for another synonym API because we were not entirely satisfied by the results, since sometimes Glosbe returns non-relevant results. Pierre found http://words.bighugelabs.com which returns a lot of synonyms for the English language. We finally decided to keep only Glosbe, as Pierre's library returned even more non-relevant results in the end, even though they were better classified and selectable.

Using Glosbe

Timo and Joris tried a number of different ways in order to make the HTTP-request for Glosbe and processing the response in the JSON-format. First, we tried to use the built-in library of the play-framework. But there was no proper documentation about it, and it's behavior was somehow unpredictable. Then we tried to use a library called Dispatch, which he had to import into our sbt-project. Unfortunately, there was no documentation about it either, but we tried anyway. We ended up having to execute our code within a for-loop in order to get the HTTP-response, which resulted in hacked code, that did not completely work either. So we decided to use the HTTP-library from Apache, which is more a JAVA than a Scala library, but it works.

Integrating it into the project

Joris reformatted the code and made it compatible with the rest of the project.

7.1.2 Keyword entering interface

Goal

Timo implemented an interface that enables the user to enter his keywords in English, to select the destination languages, to translate his keywords into the selected languages, and to remove inaccurate translations.

Details

Timo started from an existing form-template of the Play framework, and adapted it to get fields enabling the user to enter a keyword, and also to remove them. There were some issues with the display of the forms and the adaptation of the JavaScript of the template to our purposes. Then, he needed to define a second page displaying the translation results, and 2 different data structures needed to be designed for the input-form and the translation-result-form. Finally, the existing Translator had to be integrated into the display and call it in an appropriate way for the user experience.

Integration into the project

The overall headers and footers of the project had a bad impact on the JavaScript code of the interface. Therefore, it took quite some time to get the interface working within the project. Moreover, the CSS of the project needed to be reconfigured to display the buttons and the forms in an elegant manner: what the template provided was not satisfactory.

8 Scalability

8.1 Scaling the computation

The Tweet Tester which returns a very large amount of tweets (about a million in a minute = around half a Gigabyte of data) was a good way to test the geographic partitioning, the display and the clustering at large scale.
Following some tests, we had to review our implementation of the clustering algorithm. The way the product of squares was handled has been changed to be faster, and the computation for the three keywords tracks (keyword 1, keyword 2 plus their intersection) needed to be parallelized. Fortunately asynchronous results are natively supported by the Play Framework.
Even with a billion tweets the display is still responsive, as well as the clustering, which takes about a minute to complete in such extreme circumstances.

8.2 Scaling the data source

Akka is Actor-based and not thread-based, i.e. there is not shared resources, only message-passing. This allow large and scalable systems which is one of the main reason why the Play Framework is so successful. Streamers and searchers could be split on different nodes (hence running on different JVMs) to use various API keys and simulate a faster connection to Twitter. However since we work with recent data, we need to wait large time frames to let people post their tweets. The more we wait, the better our statistics will be.
Hence scaling the searchers wasn't required to test our system, since a small amount of data still representative to do our statistics. Using the current repartition systems, we got up to 200'000 Tweets in about twelves hours. Since we are working on live data, twelve hours is a reasonable time frame to let people speak about the subjects of concern. The amount of data received could be multiplied by a large factor depending on time used to gather the data. In two and a half days our system would be able to do statistics on about a million tweets.

9 Member report

Timeline and Plans available here

9.1 Mathieu Demarne

I proposed the idea and was the team leader of the project. I did the setup on Git, making the skeleton of the Play Framework. I also organized the various subparts and scheduled our weekly meeting.
During the first phase, I tested the various Twitter API to find the more suited ones for our needs, implemented the Tweet Manager, the Tweet Searcher and the Duplicate Checker, while Clement was doing the Tweet Streamer. I ensured that our data source was stable and working for runs over a large period of time.
During the second phase, I was responsible for the general integration of the various parts withing the Play Framework. I also helped Adrien debugging the Clustering Algorithm and did a couple of tests to check that our system was scaling. I integrated the maps developed by Cedric, Kevin and Adrien to the design generated by Pierre and Gianni to have a coherent platform. I also helped Timo debugging a bit of Javascript.
Being responsible for the quality checked, I supervised and checked most of the parts of the project and coordinated the team.
On the last week I generated examples for the in-class demonstration.

9.2 Adrien Ghosn

During the first two weeks, I helped setting up the google group, the descriptive wiki on the github project, and helped establishing the different parts needed to be implemented. Then, I spent some time finding different alternatives for the map. After that I worked with Lewis on the GeoPartitionner responsible for splitting the research on different geoSquares. We actually designed two different versions before to chose the one that needed to be implemented. I was then attributed the implementation of the clustering part of the project. With Lewis, we devised the metric formula for our "Home cooked" algorithm, based on a classic hierarchical clustering algorithm. Then, I was responsible for its implementation that Mathieu later integrated to the project. After that, Pierre took the time to check our algorithm, verified that it respected the theory and proposed the SLIC algorithm. Therefore, I implemented the SLIC clustering algorithm too as a backup for our "home cooked" algorithm, and to see if K-mean algorithm

was better for us. Then we tested both implementations with real data once the interface with the application was ready and discovered that our algorithm yields more intuitive results. At that point a problem arise, once we increased the number of original squares in the application, the clustering algorithm became a bottleneck. As a conscequence I corrected the implementation in order to achieve better performance. I also rewrote it in a map-reduce way so that an eventual integration with Spark or Hadoop would be easy. Then, since the display team was busy finishing the views, I had to implement the view in order to display circles for the clusters (for the different levels of cluster). Thanks to Cedric, I was able to catch up quickly with the API and rapidly implement the display. He then helped me for the dynamic part and added a slide bar to select the level of clustering when we realized that the relation between the zoom level and the level of clustering was hard to find. Finally, on the very last week, I helped generating examples for the in-class presentation.

9.3 Lewis Brown

During the first phase, I worked on the design and implementation of the map partitioning with Adrien. This part was done using Akka, as described in section 4.1, so I spent some time getting familiar with the Play Framework and Akka beforehand. I then designed the custom clustering algorithm described in section 5.1 with Adrien, and contributed to the start of its implementation. During the final phase, I implemented the task controller which ties together the user interface and the Twitter API queries with Joris, as described in section 4.2. This required careful consideration of the possible errors that could arise during the task, and some restructuring of the back end to ensure that the many different actors in the system would synchronize properly.

9.4 Kévin Serrano

During the first phase, I was working on the map display with Cédric, as we tried to figure out what was the best way to accomplish our purpose. Then I implemented part of the code and managed to draw circles on an other map using Geojson, but after spending so much time to try to figure out how to draw these circles on our map with Geojson, it was decided to use only rectangles. The circle were added in the final phase with only svg components. I also implemented the part for storing data we needed for phase 2 such as coordinates. When the map was operational, it was ready for the integration with play (phase 2).
Phase 2 : during the second phase, I successfully integrate the map with the play framework, managed to get the coordinates of the regions from Scala in order to let the server process the data. This was done using a form and Json format. I also tried to display regions on the map, and with Cédric we finally draw the regions correctly, so with the play integration it was then easy to pass the result to the drawing functions to get the result.
Finally I implemented the part where the Venn diagram is calculated based on the view of the map, that means the border of the map are relevant to compute the Venn diagram.

9.5 Gianni Scarnera

During the first phase, I worked on finding ways to visualize the data, in which I considered multiple types of graphs, such as venn diagrams and chords diagrams. Once the venn diagram was the solution retained, I worked on implementing it with d3.js and solved the problem of optimizing the placement of the circles, finding in the process a very useful library (venn.js). I also worked on some of the structure of play (designing and creating first versions of views, routes, controllers) and drawed the initial concept design for the website as a workflow.
During the second phase, I finalized and integrated the venn diagram into the project using Play and worked on designing and implementing the website with Pierre, as well as some other tasks such as drawing a workflow for the whole process of the tweet aggregator.

9.6 Pierre Gouedard

During the first part of the project I looked for an API in order to improve and enlarge the proposition of synonym. I found Big Huge Thesaurus [8], a very simple API for retrieving the synonyms for any word in english. After testing and combining this API with the one found by Timo and Joris, we decided to not add the API.
In the second part of the project I made research in order to find a suit clustering algorithm, I proposed the SLIC clustering algorithm, a K-mean based algorithm as explain in 5.2. Finally I was actively involve in the design of the web-site with Gianni.

9.7 Clément Moutet

In the first weeks, I worked on getting tweets from the stream API of twitter. The first two weeks, I learned how to use the OAuth signpost and Apache HttpComponents API in order to authentificate to Twitter and implemented them in order to retrieve tweets from twitter and make it work with the TweetManager. After this, the projet setted-up and changed the input paramters, so I had to take one week to change the Streamer to take into account this new architecture. I was then confronted to the problem of the limitation of the search parameters. Indeed, according to the Twitter Streaming API specification I couldn't search for words and for geolocalisations in the same time, so I took one week to make a new search architecture, I now search with the geolocalisation parameters and filter locally the keywords. During hollidays, I had to adapt my output to the new input/output parameters specifications, meaning that I had to give each tweet to the correct listener. The last three weeks, I implemented the Tweet Tester and did not modified it like the Streamer because the project specifications didn't change.

9.8 Timo Babst

In the first week, I looked for a suitable Translation API. After trying out some others, I chose Glosbe and contacted the creators to make sure that we would not get blocked for sending too many translation requests. After that, in the following weeks, I implemented the Translator

with Joris. We faced a couple of issues since the Glosbe API is not documented. Moreover, we tried to formulate the GET request for the API using Play's built-in functions, which was very tedious. In the end, we decided to use Apache's framework for the request. During the last weeks, I implemented the Keyword Selection page, which prompts the user to enter his keywords, choose his translation languages and remove inaccurate translations if need be.

9.9 Joris Beau

During the first phase, I equally worked with Timo on the Translation part (section 7.1.1) of the project using Globse API [4]. This choice has been made after some searches for other solutions such that Google API, Yandex translate API and even parsing Google translate web application results. Then we faced troubles with the lack of documentation even with Globse API than Scala libraries (like Dispatch). We finally solved the issue by mixing Play Framework's functions with Apache's framework.
During the second phase, I worked with Lewis on the gathering part that consists in establishing the link between Twitter search API and GUI (section 4.2). We had to obtain data from the interface, launch all queries requested, collect the results and compute outputs for the interface. This work implied to deal with different structure of the code, correct some parts and bind them together.
I also took part in code debugging, report writing and proofreading.

9.10 Cédric Bastin

During the two phases of the project I was working on the front end for the map visualization; finding out how the PolyMaps library works as well as implementing our corresponding javascript code. In the beginning I worked with Kevin in parallel to find the features we needed and then we merged our work together. My main personal issues with this part was to figure out how to use the PolyMaps library and to debug javascript in general, which, due to it's dynamic design, can only be done by re-running the project or smaller tests. As described earlier, the visualization of circles on the map was also quite tricky, luckily Adrien took some time to implement this feature such that the nice clustering visualization was made possible! In the later phases I also worked on the templates which ensure the interoperability of the front end with the server-side Scala code.

10 Conclusion

After the completion of each subpart as well as the successful integration of those, our final project met all of our initial goals and requirements. It allows a search among large amounts of tweets and has a good scalability over all its parts. There are some restrictions in the Twitter APIs we used, otherwise we could easily imagine increasing the number of keywords for even more interesting results. It is also imaginable that CrossTalk could be applied to other domains using different data sources to fully exploit its potential.

Acknowledgment

We would like to thank our supervisor Mohammed El-Seidy and Prof. C. Koch for stimulating discussions on this class project and for helpful comments on how to conduct good team work.

Bibliography

[1] Play Framework 2.2.2, `http://www.playframework.com`
Last visited: 20th May, 2014

[2] Akka tookit, `http://akka.io/`
Last visited: 20th May, 2014

[3] PolyMaps, `http://polymaps.org/`
Last visited: 20th May, 2014

[4] Globse API, `http://glosbe.com/a-api`
Last visited: 20th May, 2014

[5] GeoJSON, `http://geojson.org/`
Last visited: 20th May 2014

[6] **SLIC Superpixels**, *Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Su sstrunk*

[7] **SLIC Superpixels Compared to State-of-the-Art Superpixel Methods**, *Radhakrishna Achanta, Member, IEEE, Appu Shaji, Kevin Smith, M̂ember, IEEE, Aurelien Lucchi, Pascal Fua, Fellow, IEEE, and Sabine Su sstrunk, Senior Member, IEEE*

[8] Big Huge Thesaurus, `http://words.bighugelabs.com/api.php/`
Last visited: 20th May, 2014