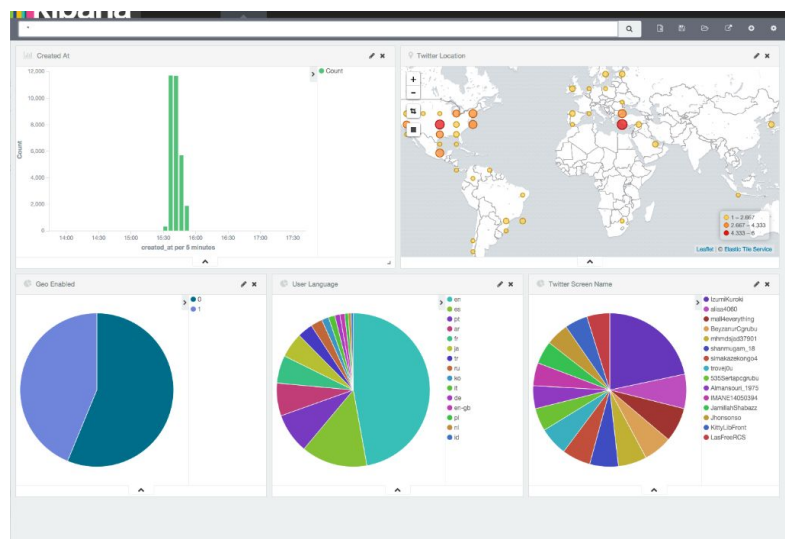


Twitter Realtime Sentiment Analysis

01.06.2018



Class:

Universitat Politècnica de Catalunya -
Cloud Computing and Big Data Analysis

Author:

Patrick Schneider

Table of Content

Twitter Realtime Sentiment Analysis	0
1. Introduction	2
2. Code Functionality and Scope	3
3. Architectural Design	6
3.1.1 BackEnd	6
3.1.2 FrontEnd and Authentication	10
3.2 Components	15
3.3 APIs	16
4. Main Problems and Challenges Faced	18
5. Twelve-factor methodology	24
6. Project Management Methodology	29
7. Service and resource components evaluation and description	31
8. Conclusions	34

1. Introduction

Twitter offers a fast feedback cycle for events happening in this world. In the recent years political analysis on Twitter helped campaigns to get quick insights into opinions of peoples in different regions. Such an insight can be not only valuable for political analysis, but also for product reviews, tracking of marketing campaigns or public relationship management. With this in mind I decided to create a generalized dashboard to analyze location based sentiments.

The goal of this project was to benefit from this fast feedback cycle and deliver a real time insight about tweet sentiments. A user can get information about a specific topic and see the current sentiment on a global scale. By using the browser based analysis platform Kibana, a in debt analysis is facilitated for a user to get e.g. details on the origin of a tweet and its related sentiment.

Possible use cases:

- Political elections
- Analysis of a election with the sentiment about topics and political parties, to adapt quickly or gain first insights for targeted campaign management
- Marketing Campaign Support
- Product release or other marketing campaigns, where it's important to know which audience gets reached
- Public relationship management at unexpected events
- (Bad) Events about the Company e.g. Facebook data scandal, to see where the people are talking about this topic to maybe address them in a specific way to restore the public opinion
- (General) Opinion Mining for research purpose
- Geographical twitter analysis about the interest for specific topics or the spreading(diffusion) of information
- Stock Market Analysis
- Following a Stock/Cryptocurrency to see current activities and be alarmed about quick changes. Most of the times news occure faster on twitter than in the main news

2. Code Functionality and Scope

Features vs unimplemented features

Implemented

- User can display tweets origin on a spatial map based on tweet coordinates.
- User can view tweet sentiment for a given topic
- Time series of tweet sentiment development
- Political affiliation overview
- Tag Cloud
- The relevant URLs found on tweets are scraped for information about the page's title and description. An URL is considered relevant after a given number of appearances in tweets.

Unimplemented

- User can view news sentiment for a given topic
- Sophisticated historical data analysis (machine learning for prediction and alarming)

Architecture differences: First Draft vs Now

S3 Bucket and kinesis firehose in the speedlayer was removed and Elasticsearch is used directly

The Sentiment API calls were outsourced in a lambda function and not in the listener directly.

Google Maps API call for coordinates lookup of addresses was initially not planned.

The application data flow was changed to the one more deeply described on section 3

Elastic beanstalk instance for the web application was taken out and only the initial static fronted via S3 is used.

Batch Layer for past data processing was taken out (DynamoDB + lambda function).

With the integrated FIB OAuth, the planned setup of a user database was taken out.

The requirements to this project were set on the initial planning phase. Based on the identified priorities for each requirement, the scope for the project was set. In the following table, the result of the requirement analysis can be found.

Priorities:

Rank 1 - are in the scope of the project due to necessity for the proof of concept

Rank 2 - are not important for a functional proof of concept, but are tried to be implemented in the scope

Rank 3 - are left out for future implementation

Field	Id	Priority (1-high, 2-medium, 3-low)	Description
01. Data Ingestion	1	1	Setup of AWS API Gateway
01. Data Ingestion	2	1	Twitter API integration in Listener
01. Data Ingestion	3	1	Google Maps API integration in Listener
01. Data Ingestion	4	3	Google News API integration in Listener
02. Processing	5	1	Tokenization of retrieved tweets/articles text via lambda function
02. Processing	6	1	AWS Comprehend integration for sentiment analysis via lambda function
02. Processing	7	2	Obtain and format information from the most relevant URLS
02. Processing	8	2	Indico API integration for sentiment analysis via lambda function
03. Speed Layer	9	1	Setup Kibana for visualization
03. Speed Layer	10	1	Setup Elasticsearch
04. Frontend	11	1	Setup static S3 Webfrontend

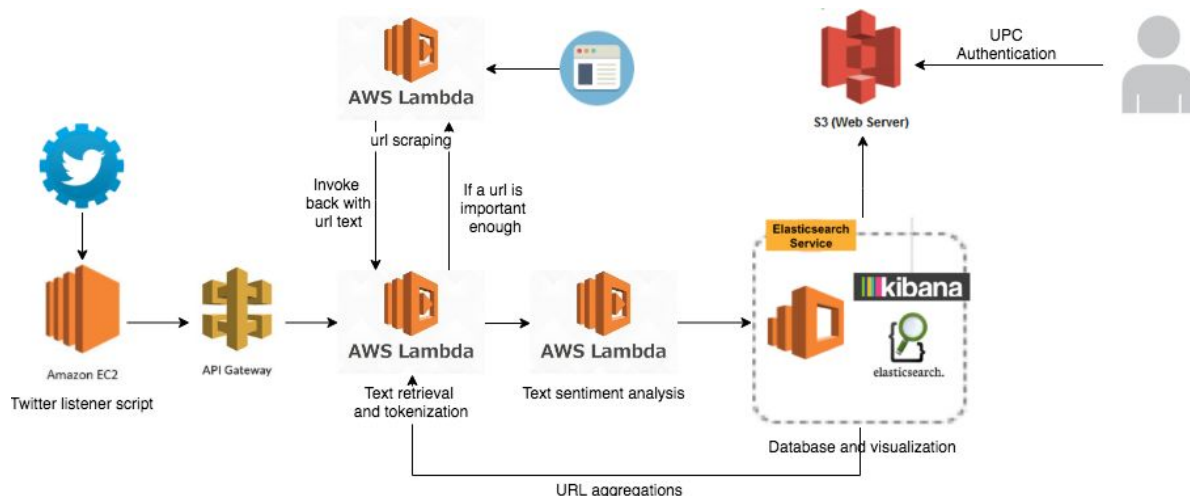
04. Frontend	12	1	Creation of Visualization plots via Kibana
04. Frontend	13	1	IFrame integration of Kibana Dashboard to static S3 frontend
04. Frontend	14	3	User configuration interface for Listener Topic selection, start/stop command
04. Frontend	15	2	Integrate user authentication via FIB oAuth
05. Batch Layer	16	3	Setup DynamoDB
05. Batch Layer	17	3	Integration of DynamoDB to Visualization framework
05. Batch Layer	18	3	Visualizations of sophisticated requests on the batch layer (DynamoDB)
06. General	19	1	Permission Management
06. General	20	2	Creation of AWS CloudFormation template of infrastructure
06. General	21	2	Configuration of AWS CloudWatch to specific needs
06. General	22	3	AWS CloudFront setup

Non-functional requirements contain mainly an intuitive frontend to illustrate dashboards to a user containing:

- Twitter analytics dashboard tab
 - Heatmap
 - Sentiment Count
 - Political affiliation overview
 - Time series of sentiment development
- News analytics dashboard tab (out of scope)
 - Sentiment Count

3. Architectural Design

As I started working on the project, I realized that the initially proposed architecture was overly complicated and incomplete, since I had misunderstood the functionality of some of the AWS services. The final architecture I implemented can be seen in the following figure and will be explained in more detail below.



3.1 Architecture Description

3.1.1 BackEnd

The first issue with the original architecture was discovered on the first sprint, when I realized that there was no way to set the twitter API as a trigger for the API gateway directly without an intermediate application that called the GW. The proposed solution was to have a twitter listener script running on an EC2 instance with load balancing, this way I could be continuously sending tweets to the gateway on an isolated and fail-safe manner.

Despite not being able to set it as originally wanted, using the gateway allowed to have more control and security, isolating the lambdas from the data retrieval module.

The next change in the architecture came when I realized that there was no need to use DynamoDB, since I would already be using the Elastic Stack in order to generate the Kibana plots, and thus it made more sense to use Elasticsearch as the main and only database.

I ended up using only a single index which I wanted to split in different entry types according to the source of the text analyzed, but this was not possible since the

version of Elasticsearch used by AWS (6.2.2) only allowed one type per index; this was not a major issue and the solution was to create an additional field with the text source. The final database entries had the following structure:

```
{
  "id" (string) - external ID of the text, did not affect the database's internal ID,
  "text_source" (string) - either "Twitter" or "URL" depending on how we got the
  text,
  "text_language" (string) - extracted by the AWS Comprehend API,
  "has_coord" (boolean) - Indicates whether we have the coordinates of the
  place where the text was posted, only true for certain tweets which contain the
  information,
  "c0" (float) - Longitude coordinate,
  "c1"(float) - Latitude coordinate,
  "location" (string) - Location in the format "latitude,longitude" (c1,c0)
  "date" (datetime) - Date of the tweet of scraping in the format YY-mm-dd
  hh:mm:ss,
  "text_raw" (string) - Original text of the tweet or webpage,
  "text_tokenized" (list of strings) - List of non-punctuation and non-stop words
  on the original text,
  "emotion" (string) - Detected sentiment associated with the text,
  "libertarian" (float) - Political lean of the text from 0 to 1,
  "green" (float) - Political lean of the text from 0 to 1,
  "liberal" (float) - Political lean of the text from 0 to 1,
  "conservative" (float) - Political lean of the text from 0 to 1,
  "urls" (list of strings) - List of URLs contained on the tweet or associated with
  an scraped web page
}
```

Some examples of the final entries made on the Elasticsearch database can be seen below. The first capture shows text from a tweet, with its associated analysis, url and coordinates, while the second one contains text scraped from a URL classified as relevant and has the date of the scraping associated with it.

#	Conservative	Q Q □ *	0.114
#	Green	Q Q □ *	0.605
#	Liberal	Q Q □ *	0.126
#	Libertarian	Q Q □ *	0.155
t	_id	Q Q □ *	gB_Jq2MBZWS70_2KkAt
t	_index	Q Q □ *	sentiment_analysis_index
#	_score	Q Q □ *	2
t	_type	Q Q □ *	doc_type
#	c0	Q Q □ *	-120.74
#	c1	Q Q □ *	47.751
⌚	date	Q Q □ *	May 29th 2018, 14:04:33.000
t	emotion	Q Q □ *	POSITIVE
📍	has_coord	Q Q □ *	true
t	id	Q Q □ *	1001434115030306816
📍	location	Q Q □ *	47.7510741,-120.7401385
t	text_language	Q Q □ *	en
t	text_raw	Q Q □ *	Just had a great experience at the San Rafael @tesla Service center (fixed glovebox latch). Thanks @elonmusk for a great car and excellent service staff.
t	text_source	Q Q □ *	Twitter
t	text_tokenized	Q Q □ *	great, experience, san, rafael, @tesla, service, center, fixed, glovebox, latch, thanks, @elonmusk, great, car, excellent, service, staff
t	urls	Q Q □ *	

#	Conservative	Q Q □ *	0.226
#	Green	Q Q □ *	0.153
#	Liberal	Q Q □ *	0.428
#	Libertarian	Q Q □ *	0.193
t	_id	Q Q □ *	KB-_q2MBZWS70_2mkAz
t	_index	Q Q □ *	sentiment_analysis_index
#	_score	Q Q □ *	2
t	_type	Q Q □ *	doc_type
⌚	date	Q Q □ *	May 29th 2018, 13:54:09.000
t	emotion	Q Q □ *	NEGATIVE
📍	has_coord	Q Q □ *	false
t	id	Q Q □ *	0000
t	text_language	Q Q □ *	en
t	text_raw	Q Q □ *	What It's Like When Elon Musk's Twitter Mob Comes After You: When female journalists like me dare to question the SpaceX and Tesla founder, there's a predictable result: We get called bitches, idiots, and worse.
t	text_source	Q Q □ *	URL
t	text_tokenized	Q Q □ *	like, elon, musk, twitter, mob, comes, female, journalists, like, dare, question, spacex, tesla, founder, predictable, result, get, called, bitches, idiots, worse
t	urls	Q Q □ *	http://thebea.st/2IVCd3q

I decided to redistribute the processing which was intended to do on three distinct lambdas. Since one must pay for the execution time consumed by the lambdas, they have a timeout that can be increased from the default of 3 seconds and up to 5 minutes, but it is recommended to keep it low so that possible issues on an specific call will not consume too many resources. In this project I had calls on the lambdas that could run overtime, so it was in the best interest to keep them separated so that

if a specific request to a webpage or API took too long, we simply lose that entry instead of consuming excessive resources or affecting the others. The lambdas used on the final architecture were:

- Text reception lambda: Triggered by the API gateway and invoked by the scraping lambda. It parses the following fields: text (string), created-at (datetime), id (string), text_source (string), has_coord (boolean), urls (list of strings) and in case of the has_coord variable being true c0 and c1 (floats). After parsing the values it tokenizes the text removing stop words and punctuation, then it adds the new list of strings to a json with the rest of the values and invokes the sentiment analysis lambda, passing the document through the call.

This lambda also verifies if the urls field received is empty, if not it makes a call to the elasticsearch database, with an aggregation to verify if any of the urls associated with the text has been introduced on the database more than a set number of times, after which the url is considered “important” and the web scraping lambda is invoked to scrape it.

The call to both of the lambdas is made using the InvocationType “Event” instead of “RequestResponse”. This way we cannot know if the invoked lambda was able to successfully complete its processing, but we will know if it received the call and will have the benefit of not increasing the original lambdas execution time, since it is an asynchronous call that has no further monitoring.

- Sentiment analysis lambda: It is invoked by the tweet reception lambda and parses the same fields with the addition of the tokenized list of strings. It first invokes the Comprehend API to extract the language of the raw text, if it is in Spanish or English it calls the API again to request the sentiment attached to it, otherwise it calls the Indico API to request the sentiment; it also calls the Indico API to get the probability distribution of the text’s political lean. Afterwards it pushes all the fields as a new entry to ElasticSearch.
- Web scraping lambda: It is invoked by the tweet reception lambda and parses a URL, which is checked against Elasticsearch to verify that it has not been scraped before. The scraping is done using BeautifulSoup, a Python library to simplify this task and that is used instead of scrapy since it fit better the needs of the current project.

The only fields scraped are the title and description of the page, joining to form a full string of text. This is done since many sites did not follow the same

patterns or used the same tags for their content, but most filled at least the basic metadata. The date and time associated with the scraped texts is taken from the system at the moment of scraping, instead of from the article's publication date, since the location of this information varies greatly from site to site and I could not manage to extract it consistently.

3.1.2 FrontEnd and Authentication

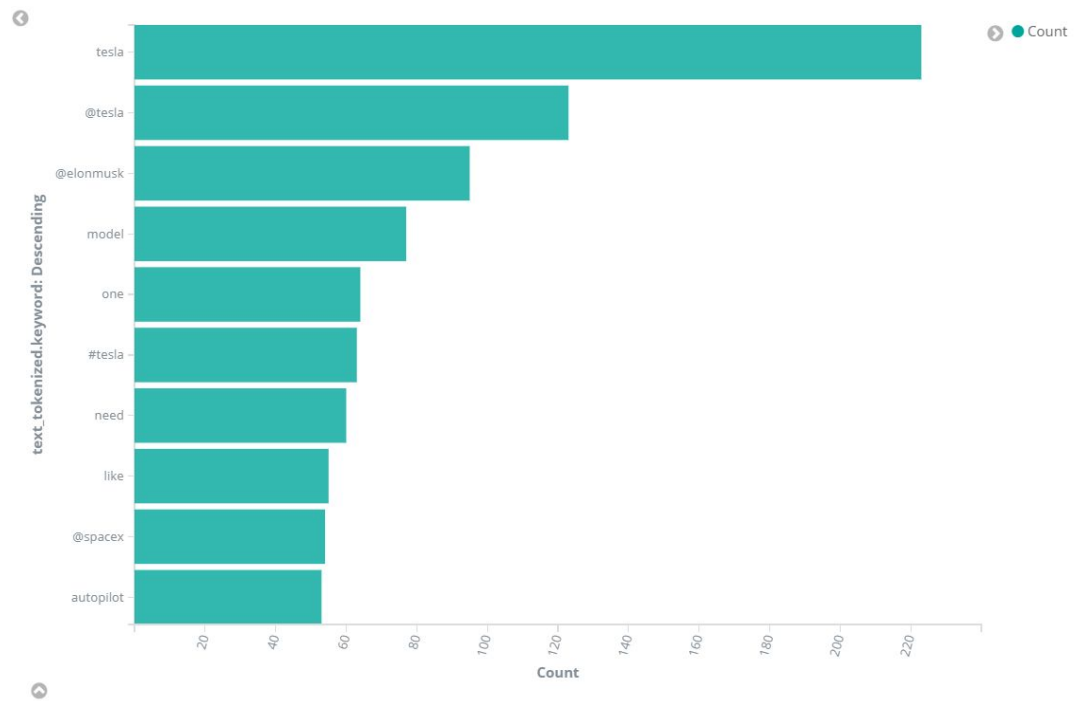
The static web application is designed to show the output of the tweets in a better way and is hosted on the S3 bucket.

It has used the following technologies.

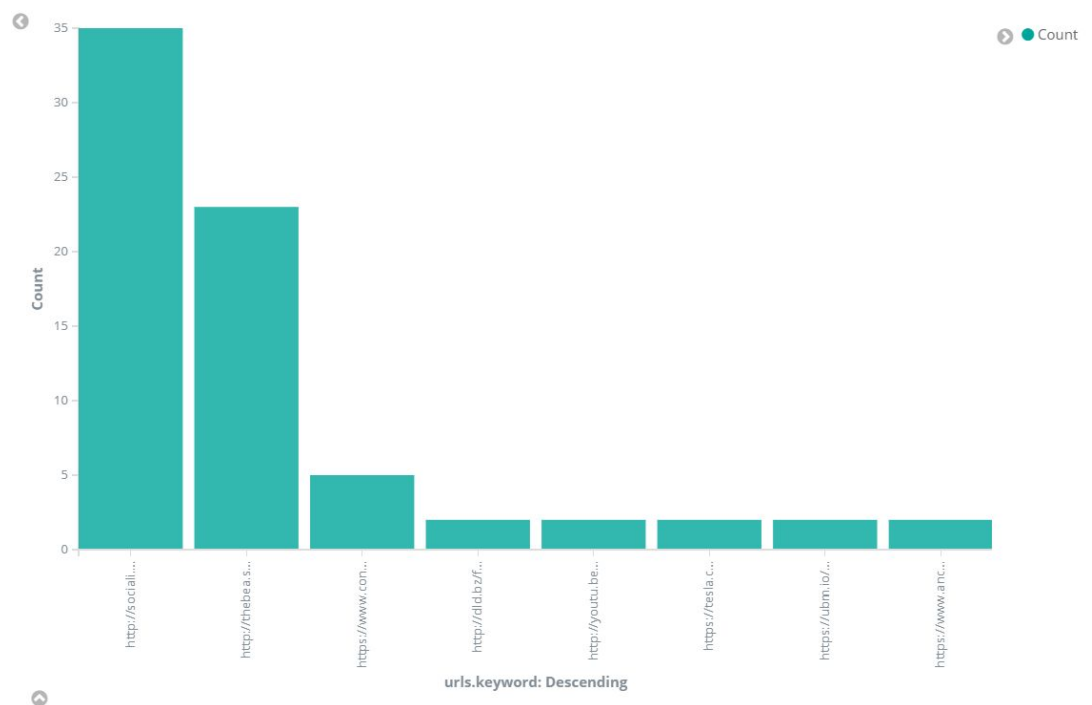
- Static Front End: The static front end serves the kibana charts in a well organized way, and provides the user to navigate between different charts simultaneously.
- Kibana Iframes: Kibana provides the powerful facility to read and visualize the data into graphs, charts, geomaps, line charts, bar charts etc

I have created the following type of visualizations available from the kibana.

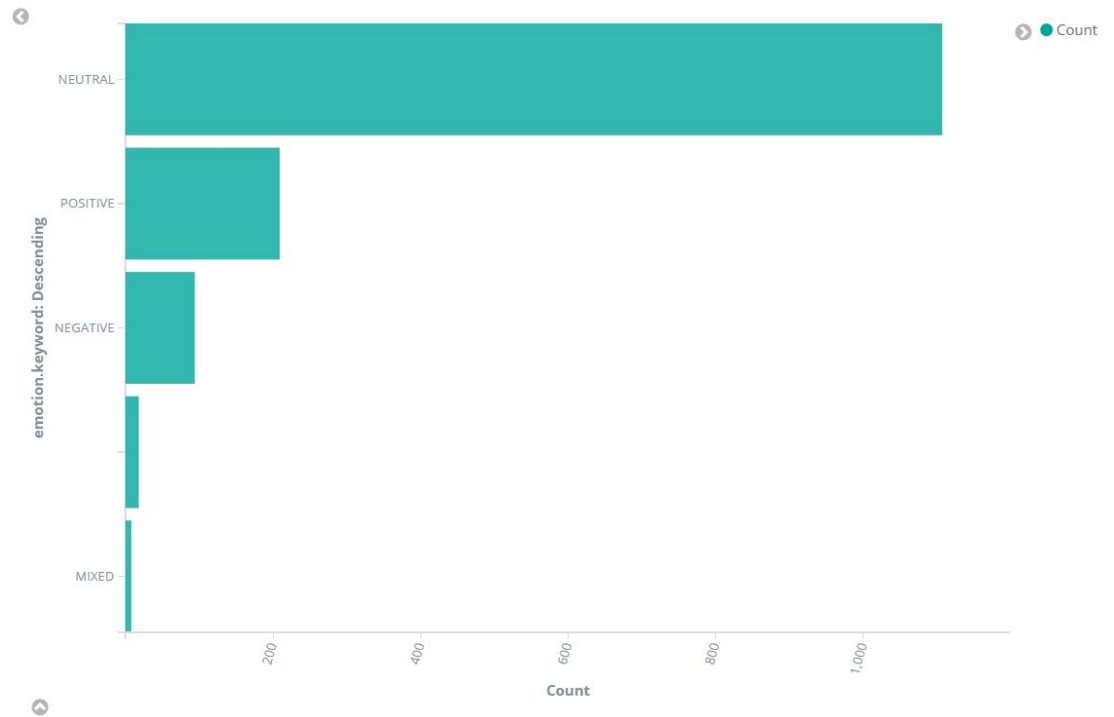
- Keywords: The keyword visualization shows the count of highest occurring keywords fetched and processed from tweets in descending order.



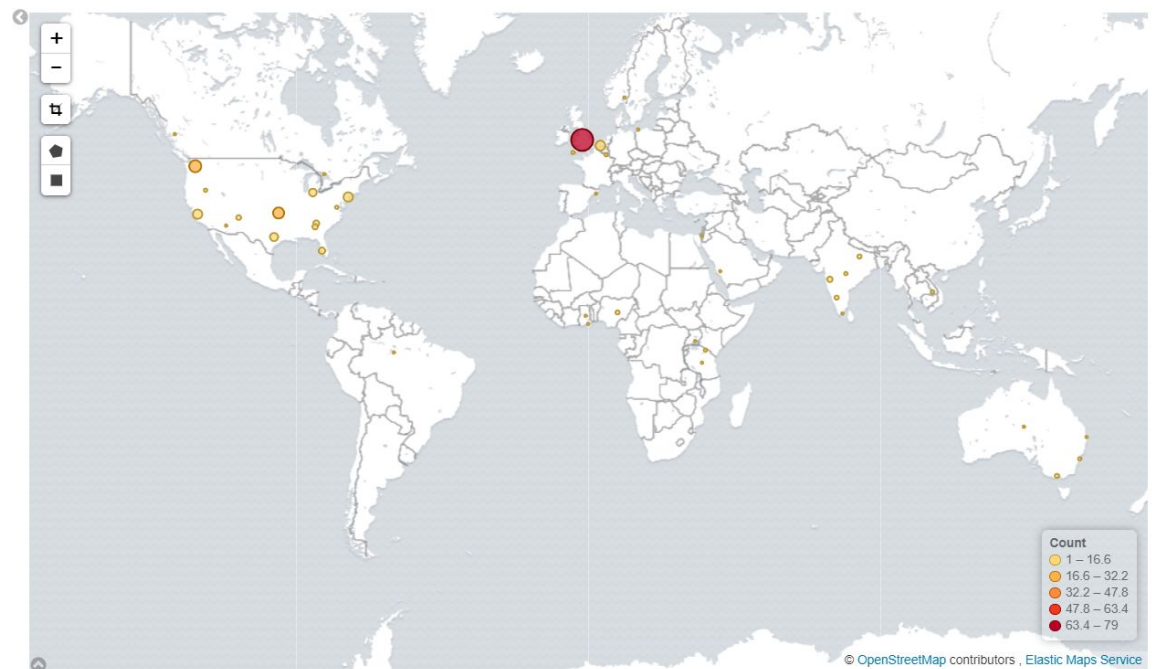
- Urls: The url chart shows the highest number of keywords fetched from the urls.



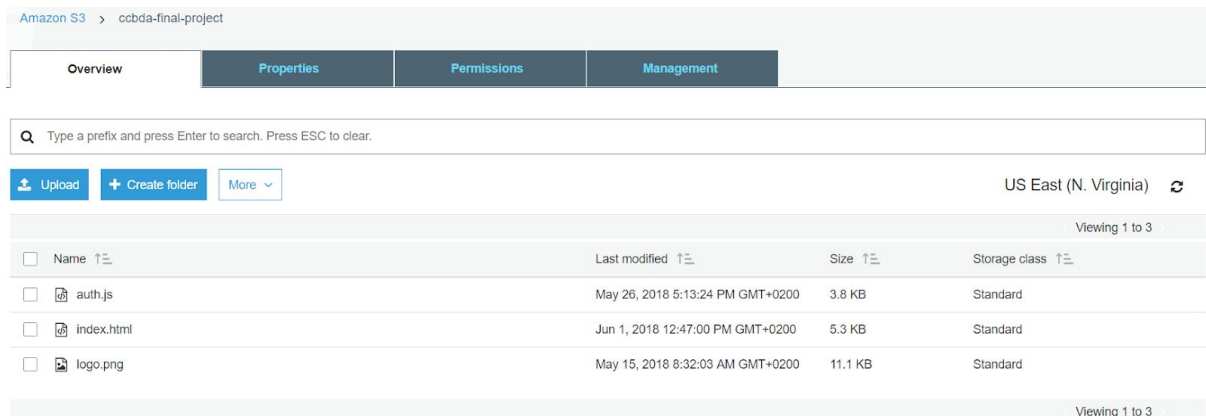
- Sentiments: The sentiment chart, display the total number of emotions with their category from the available tweets.



- Tweets Map: The tweets map shows the location of the tweets on a geo-coordinate map. The field location is indexed to show the tweets on the map, also not all the tweets have the location due to privacy issues. In the legend, it shows the count of tweets with the intensity of colors as the tweets count increases.



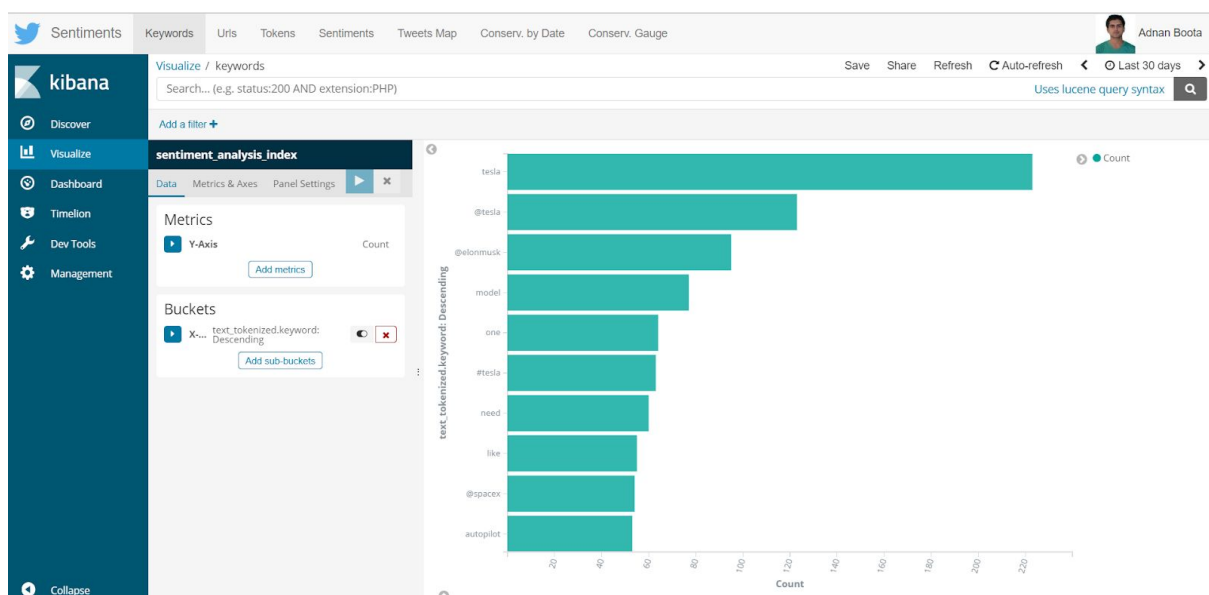
The web app is served on S3 bucket as a static site, the architecture of the static site inside the S3 bucket is like this:



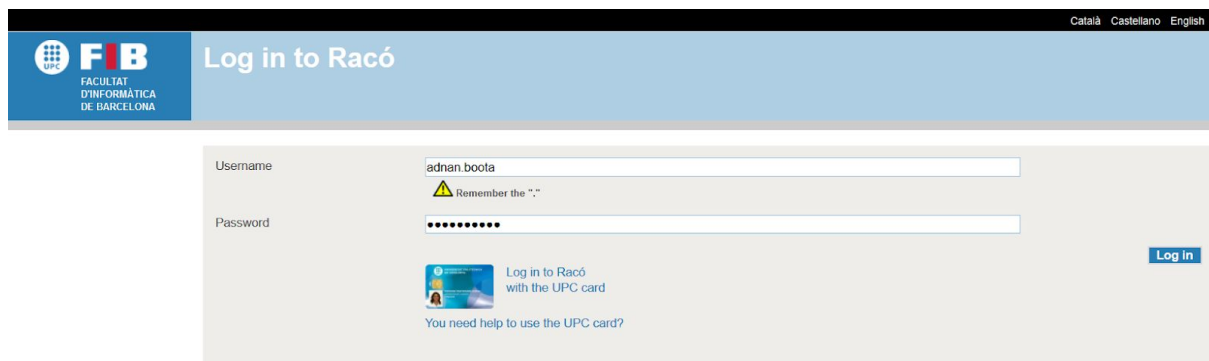
Name	Last modified	Size	Storage class
auth.js	May 26, 2018 5:13:24 PM GMT+0200	3.8 KB	Standard
index.html	Jun 1, 2018 12:47:00 PM GMT+0200	5.3 KB	Standard
logo.png	May 15, 2018 8:32:03 AM GMT+0200	11.1 KB	Standard

“ccbda-final-project” is the name of the bucket, which contains the contents of the static site.

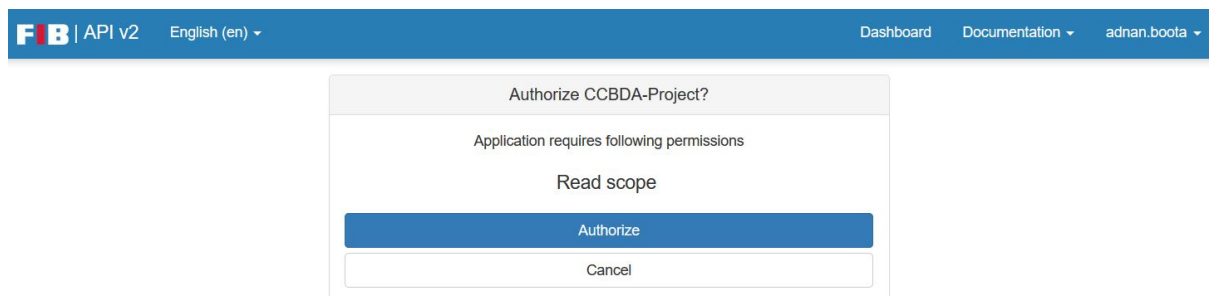
The index.html displays the static site with the kibana iframes which shows the different type of visualizations like keywords, tokens, urls, tweets map, etc. The skeleton of the website is developed using bootstrap, css and javascript. In bootstrap, tabs are used for the responsive and dynamic view like immediately changing from one chart to the other. The kibana iframes are then embedded into each tab based on the chart type.



The auth.js file acts as middleware between the web application and the user, the user needs to have an account from the FIB - UPC to login into our application and view the content. It contains the authentication code provided from the FIB OAuth API. The user needs to click on the “Login with FIB API” button and then it redirects to the FIB Login page, where the user has to login with his/her credentials.



After the successful login with correct credentials, the api will ask for the permission to share the information with the web application.



After you click “Authorize”, it will send back the information

- First Name
- Last Name
- Profile Picture

When we receive this information about the user, we just display it on the website, confirming that the user is logged in.

3.2 Components

Twitter Listener: The Tweet Listener was implemented in python 3 and runs on an amazon EC2 instance. Retrieved tweets get analyzed for included web links as well as location data gets processed for later use.

API Gateway: Is an AWS service designed to connect other services, such as lambdas, with public or private websites and applications. It provides consistent RESTful application programming interfaces (APIs) for mobile and web applications to access AWS services. It also acts as a security and isolation layer for the services that lay behind it.

ElasticStack: Open source suite for data visualization and analytics, comprised of ElasticSearch, LogStash and Kibana, a data visualization tool.

Elasticsearch: A RESTful, distributed search and analytics engine built on Apache Lucene, which can be used to store data and make interesting queries, aggregations and mathematical operations with it.

Kibana: An analytics and visualization platform designed to work with Elasticsearch.

AWS Lambda functions: An event-driven, serverless computing platform provided by AWS, whom automatically manages the underlying computing resources for the user.

S3 Bucket: Logical unit of storage in AWS object storage service. They are used to store objects, which consist of data and metadata, and also have resources for hosting dynamic websites.

3.3 APIs

Twitter API: Source interface to retrieve twitter details about users and their posts.

Google Maps API: With google maps api, user location data like "Santa Clara County, California, US" can be resolved in the latitude and longitude value. Most of the Tweets does not have location coordinates of location place data in form of longitude and latitude. With google maps api, the retrieved tweet coordinate info was increased by plus ~80% (total ~85%).

Sentiment Analysis APIs: At the moment of writing this report the AWS Comprehend API could only analyze texts in english and spanish, on the other hand the Indico API had support for 12 languages. The Nevertheless, the analysis made by Comprehend seemed better as a primary source of text sentiment, since the three category classification was clearer than the probability given by the Indico API, from which was not easy to set a range for "neutral" emotions.

Since both APIs were able to detect the text's primary language I decided to use the output of the Comprehend API for texts in english and spanish, and the one from the other API for the rest of supported languages. In both cases I drew the political analysis from the Indico API. The information from each one of the APIs used was:

- **AWS Comprehend:** The emotion associated with the text given as a string that could either be "POSITIVE", "NEGATIVE" or "NEUTRAL"
- **Indico:** The emotion associated with the text given as a float from 0 to 1, where 0.5 could be taken as the threshold between positive and negative. The other text analysis provided by the API was a political classification, through a function that returned a dictionary with 4 key-value pairs, which represent the likelihood that each of 4 political leanings are expressed in the analyzed text (libertarian, green, liberal and conservative)

4. Main Problems and Challenges Faced

Problem 1: For each tweet, Twitter includes two types of location data in the data that makes up the tweet - Exact location data for the tweet and general location data for the user. Most people have turned off sharing of geodata for individual tweets. This means that there is no information on where they tweeted that exact tweet from.

Solution 1: Google maps API. The location detection runs in 3 levels by descending precision of a tweet.

Level 1. Extracts the enabled coordinates on where the tweet originated. (~3% of tweets, very high precision)

Level 2. In case the coordinates are not enabled, the place of a tweet gets retrieved and the border_box coordinates are used. (~2% of tweets, high precision)

Level 3. In case the place information are not shared, the location address that users provide voluntarily in their profile description are used. The coordinates to this address are retrieved by looking it up at the google maps api. (~80% of tweets, precision depending on address resolution)

Problem 2: Google maps allows a day only X lookups and y in a month

Solution 2: Storing the retrieved location data in a database. Approximate location information like “California, USA” represent a decent amount of the lookups. By checking an internal database for those entries, a very high amount of requests can be reduced over time.

Problem 3: Retrieving URLs from tweets, several issues came up related to Twitter’s URL shortening and placement.

- 1) At the time of doing this project Tweets could be original, retweets or quotes, which may or may not contain “extended objects”. This variety in the types of tweets meant that it was needed to be careful about which urls should be retrieve and where from.
- 2) Twitter substitutes most links for shortened versions. Since these links were temporary and different for each person that published the same page originally, I needed to get the full URL instead. There was no interested on

URLs to other Twitter pages, since they were most commonly only references to the original tweet where a retweet came from and so it would be duplicating the information already obtained.

Solution 3: The decision fell on always check if a tweet was a retweet and in that case if it had extended entities or quotes, in order to retrieve all the links both from the tweet and from the original which it was potentially quoting. If the tweet was not a retweet or a quote it was only check if it had or not extended entities and get the URLs accordingly.

In order to get the real links on the tweet instead of Twitter's shortened version, it should look for the "expanded_url" in the appropriate position, which depended of the type of tweet. Afterwards the url URLs that started with "<https://twitter.com>" are filtered and added the rest as an array of strings on the message that would be send to the API GW.

Problem 4: Scraping web pages with unknown format. I noticed that the information useful to this project was placed on different places of the html according to the page.

Solution 4: Most sites did not follow the same patterns or used the same tags for their content, but almost all filled at least the basic metadata. For this reason decision fel on only scraping the fields "title" and "description" found on the html header, creating a joint string with the format "title: description" which was passed to the text reception lambda. If one of the two fields was empty (usually the description) it will have at least something to analyze in order to know what information the relevant URL contained.

Another option that was considered to solve this issue was to use [Diffbot](#), a set of APIs which allow to analyze a URL in order to know which type of content the page contains and then call a more specialized API to retrieve content from it using a merge of different scraping techniques with AI. This would have allowed to extract more information, but since the service is very expensive after trial and this solution worked well for this purpose the decision fell against it.

Problem 5: I wanted to only scrape URLs that were "important", a concept which was defined as "having appeared in more than a given number of tweets".

Solution 5: The aggregation capabilities of Elasticsearch in order to query for all URLs which had a count of at least the set threshold. If a URL was not on the returned list, it could not be scraped yet. A sample of the query used, which can be tested on Kibana would be:

```
GET /tweets-with-emotion-comprehend/_search{
  "query": {
    "match": {
      "text_source.keyword": "Twitter"
    }
  },
  "aggs": {
    "url_count": {
      "terms": {
        "field": "urls.keyword",
        "min_doc_count": 10 } } } }
```

Problem 6: I needed to verify when a URL had enough counts to be considered important and scraped, but the query with the aggregation had to be followed by a verification step in order to avoid scraping the same URL more than once as its count kept increasing above the threshold.

Solution 6: The first approach was to have a lambda triggered by a CloudWatch timer check every few minutes which URLs had gone above the threshold, verify if they were already scraped and if not, call the scraping lambda, nevertheless upon testing it was realized that the verification step would be extremely long to be done on a single lambda and decided to move it to the scraping lambda, where it only added some milliseconds per execution.

The decision fell to transfer the aggregation call to the text analysis lambda, since it would also only add a small latency, and it could check against the URL associated with the given tweet, instead of going through the whole database in a single instant.

Problem 7: Debugging lambdas and passing from test environment to deployment

Since only external libraries were used for many of the lambdas, I had to add them as zipped deployment packages. The problem was that when a package weighs more than 3MB, the inline code editing functionality of the lambda console is

disabled, so if an error was detected I needed to modify the code locally, zip it and upload it again

Solution 7:. This was time consuming and troublesome, so the decision fell on using the environment variables to define different things on the code during development, such as trying out different implementations or values.

For deployment I was able to use the final deployment package which passed all the tests and load it on the final lambda in a few minutes, just writing the environment variables to define the level of logging I wanted, the API call information, the threshold for scraping, etc.

Environment variables

You can define Environment Variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the need to change function code. [Learn more.](#)

debug	true
es_host	https://search-sentiment-visualizer-sh6s22lr6f24vs6ujdgzcaplx.eu-west-1.es.amaz
es_index	sentiment_analysis_index
es_type	doc_type
max_urls_response	1000
min_url_reps	3

Problem 8: Elasticsearch cluster health deterioration

Solution 8: After gathering over a thousand entries on the AWS Elasticsearch, I noticed that the cluster health had deteriorated, as can be seen in the capture below:

Domain	Elasticsearch version	Endpoint	Searchable documents	Cluster health ⓘ	Free storage space ⓘ	Minimum free storage space ⓘ	Configuration state
ccbda-imdb	6.2	Internet	0	Green	7.24 GB	7.24 GB	Active
sentiment-visualizer	6.2	Internet	1,571	Yellow	7.23 GB	7.23 GB	Active

The yellow health indicator meant that the cluster was running well and requests could be handled successfully, but since only one nodes replica rules where not satisfied and it at risk losing data in the event of a server malfunction. To fix the issue another node was added to the cluster and the health indicator went back to green.

Problem 9: Adding coordinates to Elasticsearch as a valid data type for plotting.

Solution 9: When entries were added to Elasticsearch from the lambda, it generally inferred the types correctly, except for the location and date fields, which needed to have very specific types to generate the Kibana plots needed. The best solution found was to first create the index, defining the two fields with their specific types as can be seen in the capture below, and then add the entries from the lambdas as before, since Elasticsearch had no problems with the new types of field added along with the ones originally defined.

```
PUT sentiment_analysis_index
{
  "mappings": {
    "doc_type": {
      "properties": {
        "location": {
          "type": "geo_point"
        },
        "date": {
          "type": "date"
        }
      }
    }
  }
}
```

Problem 10: Creating CloudFormation templates from scratch

Solution 10:

One of the challenges was; AWS has a design template system where you can create your CloudFormation architecture, but these templates are nothing more than visual guides with empty format.

Problem 11: AWS Design IDE does not perform as other external tools

Solution 11:

The AWS design console does not offer very promising SDE in terms of efficient code management or debugging or any kind of error handling tool (very basic level).

It took a bit time to learn a proper way of creation the CloudFormation templates. When it RollBacks the stack, there is an option to modify(update) the stack.

CloudFormation ▾ Stacks				
Create Stack ▾ Actions ▾ Design template				
Filter: Active ▾ By Stack Name				
	Stack Name	Created Time	Status	Description
<input type="checkbox"/>	EC2ValidatedStack	2018-05-29 14:09:44 UTC+0200	CREATE_COMPLETE	
<input type="checkbox"/>	EC2Stack7	2018-05-29 08:32:57 UTC+0200	CREATE_COMPLETE	
<input type="checkbox"/>	KibanaStack5	2018-05-28 16:06:44 UTC+0200	ROLLBACK_COMPLETE	ELK Stack - Elasticsearch, Logstash, Kibana 5
<input type="checkbox"/>	ECStack6	2018-05-28 14:40:40 UTC+0200	CREATE_COMPLETE	
<input type="checkbox"/>	ECStack5	2018-05-28 14:16:44 UTC+0200	CREATE_COMPLETE	
<input type="checkbox"/>	EC2Stack3	2018-05-28 00:46:49 UTC+0200	CREATE_COMPLETE	
<input type="checkbox"/>	lambda-s3-event	2018-05-27 19:36:02 UTC+0200	CREATE_COMPLETE	Upload an object to an S3 bucket, triggering a Lambda
Overview Outputs Resources Events Template Parameters Tags Stack Policy Change Sets Rollback Triggers				

Problem 12: FIB Auth Token Issue, the auth api was not sending back the access_token for the authentication, and it was not working. In fact in the documentation of the API, it says to have the token type as “Authorization Code”, but it was not able to get the access_token with this type.

Solution 12: First, tried different solutions, like debugging even changin the original example available from the documentation, but it didn’t work. After that, Submitted a ticket to the management of the api, they changed the option of access_token to implicit at the backend of the api, and then it was possible to get the access_token. After receiving the access_token, it was possible to get the first_name, last_name and picture of the logged in user.

5. Twelve-factor methodology

How the twelve-factor methodology was implemented in this project.

I. **Codebase**

Use Github version control system to deploy and manage the infrastructure.

II. **Dependencies**

Libraries are installed through “pip” dependency

Twitter.py

```
tweepy
json
argparse
```

TwitterListener.py

```
tweepy
from tweepy.streaming import StreamListener
json
sys
boto3
datetime
requests
```

Sentiment Analysis Lambda function

```
boto3
indicoid
```

Tokenizer Lambda function

```
boto3
nltk
requests
requests_aws4auth
```

III. **Config**

The environment variables were set in the AWS console directly.

IV. **Backing services**

Backing services were mainly in the form of lambda functions.

V. **Build, release, run**

Build, release and run the components on different aws account in an isolated manner. Amazon offers many options to test components like lambda function with an simulated input and resulting output.

VI. **Processes**

With this architecture and the used lambda functions do we have a stateless process, where only information are stored in elasticsearch. The complete process can be run, terminated and re-run at any time.

VII. **Port binding**

The port binding was mainly done via lambda functions that operate in an synchronous manner.

VIII. **Concurrency**

There are no concurrent processes in this scenario. Theoretically could the lambda functions operate in a concurrent manner, where they consume the tweets on a producer. But this was not seen as necessary in our speed layer.

IX. **Disposability**

Tracing the lambda function and speed via AWS X-Ray - on a future scope.

X. **Dev/prod parity**

By working with github and the AWS accounts, there was no problem of testing the atomic functions and push it into productive when it was done.

XI. **Logs**

The logs are basically managed with CloudWatch, that gives a sophisticated overview of everything that is happening in the architecture. Furthermore does the twitter listener give information about (un)successful operations in the console. This could be on a future step optimised to alarm on specific exceptions.

XII. **Admin processes**

Admin processes can be executed separately from the productive environment, due to the stateless architecture and the isolated services.

QA environment easily set up with cloud formation and the easy deployment options of aws. Only the elastic search database is stateful. One-off admin tasks can be tested outside of the productive environment on separate instances.

For this problem, the implementation of a basic CloudFormation architecture was done to deploy the infrastructure.

a. Replicating the architecture stacks with Templates

The requirements was to create an EC2 instance by the yaml or json template with the right parameters.

It is a formal way of creating CloudFormation templates for an architecture.

In order to create a complete architecture, it is possible to use only one stack (template) including every “resources” in use. Which means for each AWS components, you might add all the configuration parameters in one template and just launch it. It is also possible to create a stack set to define each service configuration in different stacks to visualize them in an easier way.

I was not able to complete the CloudFormation stacks for each service in use.

b. Samples for the stacks in templates

```
Resources:
  EC2Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: ami-db710fa3
      InstanceType: t2.micro
      UserData: !Base64 |
        #!/bin/bash
        apt-get update -qq
        apt-get install -y apt-transport-https ca-certificates
        apt-get install pip
        apt-get install requirements
        apt-get install tweepy
        apt-get install boto3
        apt-get install request

    Metadata:
      AWS::CloudFormation::Init:
        configSets:
          ascending:
            - "config1"
            - "config2"
          descending:
            - "config2"
            - "config1"
        config1:
          commands:
            test:
              command: "echo \"${CFNTEST}\" > TwitterListener.py"
              env:
                CFNTEST: "Setting up the Twitter Listener."
              cwd: "~"
          build:
            - echo "Validating resources..."
            - python TwitterListener.py
            - exit $?

```

This .yaml configuration file is to automate EC2 instance(s). I prefer .yaml format because of its simplicity and more readable syntax.

The important parameters here are ImageId and InstanceType to identify which of the options of the EC2 instance that should be deployed. There are many other options as well, like I explained in the previous paragraphs. Depends on the region you would like to set your EC2 instances, the ImageId will be based on that specific region. You can use multiple regions to create your instance and just have to add the "ImageId" of the desired location.

```

ElkLaunchConfig:
  Type: AWS::AutoScaling::LaunchConfiguration
  Properties:
    ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', ImageId]
    SecurityGroups:
      - !Ref ElkSecurityGroup
    InstanceType:
      !Ref ElkInstanceType
    BlockDeviceMappings:
      - Fn::If:
          - UseEBS
          - DeviceName: "/dev/sdk"
          Ebs:
              VolumeSize: !Ref EBSVolumeSize
              VolumeType: gp2
              Encrypted: true
          - !Ref AWS::NoValue
    IamInstanceProfile: !Ref InstanceProfile
    KeyName: !Ref KeyName
    UserData:
      Fn::Base64: !Sub
      - |
        #!/bin/bash -ev

        # Update repositories
        wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
        sudo apt-get install apt-transport-https
        echo "deb https://artifacts.elastic.co/packages/5.x/apt stable main" | sudo tee -a /etc/apt/sources.list.d/elastic-5.x.list
        echo "deb http://packages.elastic.co/curator/4/debian stable main" | sudo tee -a /etc/apt/sources.list.d/curator.list
        apt-get -y upgrade
        update-ca-certificates -f
        # Install prerequisites
        apt-get -y update && apt-get -y install language-pack-en ntp openjdk-8-jdk unzip libwww-perl libdatetime-perl
        # Install logstash, Elasticsearch, Kibana, etc...
        apt-get -y update && apt-get -y install elasticsearch kibana logstash elasticsearch-curator nodejs npm

```

Another example of the Kibana and ElasticSearch Deployment. This has not been completed but the idea here is to install the UserData: configuration !Ref to your instance and your instance profile you have assigned to EC2 stack. Updating your virtual machine environment with the required Kibana components and make it ready to use.

6. Project Management Methodology

Tools & Technologies used for project management:

- Trello & Kanban with backlog (Sprint management)

Sprint planning:

Sprint	Task
Sprint 1	Elasticsearch as an AWS service can be accessed from a lambda and new information can be stored to index
	Tweets captured and sent to API GW (local execution)
	Test Amazon Comprehend API on a lambda
	Set up project management tools
Sprint 2	Create static site with Kibana IFrame integration
	Information can be retrieved from given index on the lambda using basic queries
	Test the Indico API on a lambda
	Lambda captures text sent, tokenizes it (without stop words, punctuation, etc) and stores it on Elasticsearch's texts-index
	Adequate Kibana plots can be created from the texts-index
	Comprehend API results are stored on index with text, tokenized text and the simple emotion associated with it
	A second lambda can be called from the main one
	Fix issue with twitter's URL shortening and improve twitter's links retrieval
	Add the external emotion API to the emotion lambda call
	Establish the final lambda chain with the first one receiving and tokenizing text, and the second one making the sentiment analysis and writing to elasticsearch - with the basic comprehend API

Sprint 3	Improve elasticsearch queries to go beyond the basic filters
	Obtain the most relevant URLs to scrape them (avoid continuous re-scraping)
	Fix issues with truncated text on the tweet retriever
	Make filters work on the Kibana plots of the static page
	FIB User Auth Example - Setup
	User Authentication - Tweets App
	Improve the tweets coordinates' extraction
	Initial Cloud formation test
Sprint 4	Cloud execution of the twitter capture script
	Cloud formation -> Deployment environment
	Add final visualization to the static site
	Merge the external API call with the current sentiment analysis lambda
	Obtain and format information from the most relevant URLs -> Scraper
	Integrate scraper lambda with the rest of the architecture
	Final testing and troubleshooting
	Start working on the report

7. Service and resource components evaluation and description

Sentiment analysis

The evaluation considered a self developed machine learning classifier using the python library sklearn and Sentiment Analysis APIs Indico, Amazon Comprehend

and Google NLP API. The development and training of a classifier was excluded early for the proof of concept, due to the high effort of developing it. Amazon Comprehend, Google NLP and Indico are easy to integrate and the offered amount of requests for sentiment analysis under the free lizenz is enough for a proof of concept. In the end my decision fell on AWS Comprehend, due to the closeness to the AWS architecture, as well as Indico, as a more sophisticated analysis tool.

	AWS Comprehend API	Indico API	Google Natural Language API	Own Development
Cost	low	low	low	no cost
Easy to implement and maintain	very easy	easy	easy	complex
Performance	0	0	0	0
Quality of sentiment	normal	very detailed	normal	topic trained classifier
Internet Resources and help	many	many	many	-

On a future scope, it can be evaluated whether an own developed sentiment classifier results in a higher precision for specific topics. Furthermore, in an application environment with a high amount of processed tweets or news articles.

Visualization

The evaluation of the data visualization contained Kibana and a self developed approach with d3.js and the google maps heatmap api. In establishing a proof of concept, decision fell to use the Kibana due to the simple integration into the elasticsearch environment. A self development approach would have allowed to build more complex solutions, but on the other hand more time need to develop.

	Kibana	Tableau	d3.js and other
Cost	low	low	low
Frontend integration	good	only with viewer licence	very good
Implementation	very easy	easy	hard
Performance	-	-	-
Design	good	very good	very good
Internet Resources and help	many	many	many
plot variety	normal	complex	no limits

Datastore - Elastic Stack

On the first draft I already stated that the intention was to use at least two components of the Elastic Stack for the project, Elasticsearch and Kibana, nevertheless when I started to implement it I realized that there were several feasible options to do it:

- Running ELK on an EC2 instance: Since the stack is open source we could download it and install it on an EC2 instance, for this option I would have to ensure load balancing but this could be handled by amazon as well. This option provides the most flexibility since I could install the preferred version, but it is also the one that requires the most management.
- Using ELK as a service on the cloud offered by elastic.co: The official cloud offering for the stack. It was probably the easier to use since everything was very well documented and I had already used the elasticsearch library, which could also be used by the local version of the stack. The main disadvantage was the price, since there was only a 14 day trial version and the standard prices were higher than on AWS.
- Using ELK as a service on the cloud offered by AWS: Similar to the service offered by elastic.co, but the communication with Elasticsearch is achieved

through normal http requests. The pricing was better than that of elastic.co and a free instance running 24/7 with the AWS trial could be used.

I decided to go with the third option and use the ELK offered as a service by AWS since it required no maintenance, it was faster to get up and running, and it can be used non-stop with the free AWS trial and the price after the trial, while higher than just an EC2 instance, still seemed competitive enough for the needs.

User Interface/service delivery/webserver

The static web is hosted on S3 storage, and is developed using the following technologies.

- Html, Css, Javascript
- Bootstrap
- FIB Javascript Authentication API
- Kibana Iframes for charts visualization

There are the following files used for the static web.

- index.html: It contains all the html design and the visualization of the charts using Kibana iframes
- auth.js: This file contains the code of the authentication with the javascript FIB Auth API

8. Conclusions

The aim of this project was to create a proof of concept of a real-time twitter sentiment analysis architecture and its visualization frontend, built in the Amazon Web Services cloud. The initially requirements of the scope were fulfilled for the delivery of a functional proof of concept.

Initially planned was, in addition to the speed layer, the integration of a batch layer with a DynamoDB for more sophisticated analysis of the past processed tweets. Due to the set scope on developing a real-time architecture, the decision fell on taking out the batch layer. It is simple to integrate an additional batch layer later on, that retrieves the information after the sentiment processing and stores it in the DynamoDB without impacting the present architecture.

Future work would contain the integration of the news api for comparison insights of media reporting sentiments and the general opinion of twitter user. Extended use cases that are based on this first prototype need a reevaluation of the used services and components in sense of e.g. processed amount of tweets/articles [Sentiment APIs vs own-development], more sophisticated visualizations [Kibana vs own-development], or special user features for the web fronted [S3 vs dynamic web-frameworks].