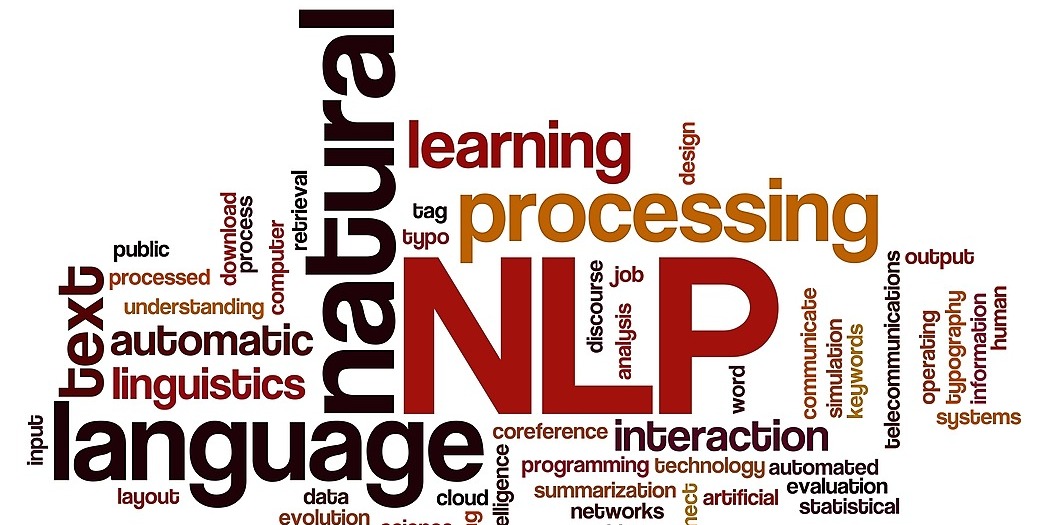
**Project of Intelligent Information Access and Natural Language Processing**

**Emotional City**



**Professors:**

Giovanni SEMERARO

Pierluigi CASSOTTI

**Student:**

Davide DI PIERRO 716234

**Summary**

[I. Introduction 5](#_Toc59294232)

[II. Twitter API 5](#_Toc59294233)

[II.I API v1.1 5](#_Toc59294234)

[II.II Tweets 6](#_Toc59294235)

[II.II.I Filter tweets 6](#_Toc59294236)

[II.II.II Locations 7](#_Toc59294237)

[II.III User 8](#_Toc59294238)

[III. Design and Implementation 8](#_Toc59294239)

[III.I Crowd Pulse 8](#_Toc59294240)

[III.I.I Available plugins 9](#_Toc59294241)

[III.I.I.I email-notify 9](#_Toc59294242)

[III.I.I.II social-twitter 9](#_Toc59294243)

[III.I.I.III social-facebook 10](#_Toc59294244)

[III.I.I.IV data-java 10](#_Toc59294245)

[III.I.I.V detect-language-optimaize 10](#_Toc59294246)

[III.I.I.VI geo-profile-google-maps 10](#_Toc59294247)

[III.I.I.VII geo-message-from-profile 11](#_Toc59294248)

[III.I.I.VIII index-uniba 11](#_Toc59294249)

[III.I.I.IX tag-babelfy 11](#_Toc59294250)

[III.I.I.X tag-me 11](#_Toc59294251)

[III.I.I.XI tag-open-calais 11](#_Toc59294252)

[III.I.I.XII tag-wikipedia-miner 11](#_Toc59294253)

[III.I.I.XIII categorize-wikipedia 11](#_Toc59294254)

[III.I.I.XIV tokenize-open-nlp 11](#_Toc59294255)

[III.I.I.XV lemmatize-morphit 12](#_Toc59294256)

[III.I.I.XVI lemmatize-stanford-corenlp 12](#_Toc59294257)

[III.I.I.XVII lemmatize-multi 12](#_Toc59294258)

[III.I.I.XVIII pos-tag-open-nlp 12](#_Toc59294259)

[III.I.I.XIX pos-tag-simple-it 13](#_Toc59294260)

[III.I.I.XX pos-tag-simple-en 13](#_Toc59294261)

[III.I.I.XXI pos-tag-simple-multi 13](#_Toc59294262)

[III.I.I.XXII rem-stop-word-simple 13](#_Toc59294263)

[III.I.I.XXIII infogram 13](#_Toc59294264)

[III.I.I.XXIV sentiment-sentit 14](#_Toc59294265)

[III.I.I.XXV sentiment-sentiwordnet 14](#_Toc59294266)

[III.I.II Used plugins 15](#_Toc59294267)

[III.I.II.I social-twitter 15](#_Toc59294268)

[III.I.II.II data-java 17](#_Toc59294269)

[III.I.II.III tokenize-open-nlp 18](#_Toc59294270)

[III.I.II.IV sentiment-sentit 20](#_Toc59294271)

[III.II Application 23](#_Toc59294272)

[III.II.I Database 23](#_Toc59294273)

[III.II.II Emotional City API 24](#_Toc59294274)

[III.II.II.I class Tweet 25](#_Toc59294275)

[III.II.II.II writeLogs 26](#_Toc59294276)

[III.II.II.III createTweet 26](#_Toc59294277)

[III.II.II.IV count\_tweets 27](#_Toc59294278)

[III.II.II.V time2int 29](#_Toc59294279)

[II.II.II.VI int2time 30](#_Toc59294280)

[III.II.II.VII string2int 30](#_Toc59294281)

[III.II.II.VIII senti\_tweets 30](#_Toc59294282)

[III.II.II.IX geo\_count\_tweets 32](#_Toc59294283)

[III.II.II.X geo\_senti\_tweets 33](#_Toc59294284)

[III.II.II.XI search\_tweets 34](#_Toc59294285)

[III.II.II.XII search\_geo\_tweets 35](#_Toc59294286)

[III.II.II.XIII backup 37](#_Toc59294287)

[II.II.II.XIV read\_backups 39](#_Toc59294288)

[II.II.II.XV get\_log 39](#_Toc59294289)

[III.II.III Test 40](#_Toc59294290)

[III.II.III.I main 40](#_Toc59294291)

[III.II.III.II count\_tweets 41](#_Toc59294292)

[III.II.III.III senti\_tweets 41](#_Toc59294293)

[III.II.III.IV geo\_count\_tweets 41](#_Toc59294294)

[III.II.III.V geo\_senti\_tweets 42](#_Toc59294295)

[III.II.III.VI search\_tweets 42](#_Toc59294296)

[III.II.III.VII search\_geo\_tweets 42](#_Toc59294297)

[III.II.III.VIII backup 43](#_Toc59294298)

[III.II.III.IX read\_backups 43](#_Toc59294299)

[III.II.III.X get\_log 43](#_Toc59294300)

[IV. Tools 44](#_Toc59294301)

# Introduction

**Emotional City** is an integrated platform that, through public sources (newspapers, social platforms, opendata, etc.) carries out an NLP pipeline that involves **sentiment analysis process** and **lexical semantic change detection.**

The goal is to collect all the tweets written in Apulia, and concerning Apulia, of the last eighteen months. From these tweets, it will then be necessary to perform sentiment analysis to collect statistics on topics related to our region. For the collection and some preliminary analysis, Crowd Pulse can help us as it is able to retrieve data from Twitter by filtering them by keywords, users or geolocation. It also has plugins that allow some preliminary operations such as tokenization, stopword removal, lemmatization and sentiment analysis with SentIt.

# Twitter API

## II.I API v1.1

All Twitter APIs that return Tweets provide that data encoded using JavaScript Object Notation (JSON). JSON is based on key-value pairs, with named attributes and associated values. These attributes, and their state are used to describe objects.

At Twitter we serve many objects as JSON, including **Tweets** and **Users**. These objects all encapsulate core attributes that describe the object. Each Tweet has an author, a message, a unique ID, a timestamp of when it was posted, and sometimes geo metadata shared by the user. Each User has a Twitter name, an ID, a number of followers, and most often an account bio.

With each Tweet we also generate "entity" objects, which are arrays of common Tweet contents such as hashtags, mentions, media, and links.

There are a set of **Data Dictionaries** for these fundamental Twitter objects. Reflecting the JSON hierarchy above, here are links and further descriptions of these Objects:

* [Tweet](https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object) - Also referred to as a ‘Status’ object, has many ‘root-level’ attributes, parent of other objects.
  + [**User**](https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/user-object) - Twitter Account level metadata. Will include any available account-level enrichments, such as [Profile geo](http://support.gnip.com/enrichments/profile_geo.html).
  + [**Entities**](https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/entities-object) - Contains object arrays of #hashtags, @mentions, $symbols, URLs, and media.
  + [**Extended Entities**](https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/extended-entities-object) - Contains up to four native photos, or one video or animated GIF.
  + [**Places**](https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/geo-objects) - Parent to ‘coordinates’ object.

The following JSON illustrates the structure for these objects and some of their attributes:

{

"created\_at": "Thu Apr 06 15:24:15 +0000 2017",

"id\_str": "850006245121695744",

"text": "1\/ Today we\u2019re sharing our vision for the future of the Twitter API platform!\nhttps:\/\/t.co\/XweGngmxlP",

"user": {

"id": 2244994945,

"name": "Twitter Dev",

"screen\_name": "TwitterDev",

"location": "Internet",

"url": "https:\/\/dev.twitter.com\/",

"description": "Your official source for Twitter Platform news, updates & events. Need technical help? Visit https:\/\/twittercommunity.com\/ \u2328\ufe0f #TapIntoTwitter"

},

"place": {

},

"entities": {

"hashtags": [

],

"urls": [

{

"url": "https:\/\/t.co\/XweGngmxlP",

"unwound": {

"url": "https:\/\/cards.twitter.com\/cards\/18ce53wgo4h\/3xo1c",

"title": "Building the Future of the Twitter API Platform"

}

}

],

"user\_mentions": [

]

}

}

## II.II Tweets

The following API endpoints can be used to programmatically retrieve Tweets:

* GET statuses/show/:id
* GET statuses/oembed
* GET statuses/lookup

### II.II.I Filter tweets

Filters have the goal to return statuses that match one or more filter predicates. Multiple parameters may be specified which allows most clients to use a single connection to the Streaming API. Both GET and POST requests are supported but GET requests with too many parameters may cause the request to be rejected for excessive URL length. Use a POST request to avoid long URLs.

The track, follow, and locations fields should be considered to be combined with an OR operator.

These are the filter parameters:

* name: Description
* follow: A comma separated list of user IDs, indicating the users to return statuses for in the stream.
* track: Keywords to track. Phrases of keywords are specified by a comma-separated list which will be used to determine what Tweets will be delivered on the stream. A phrase may be one or more terms separated by spaces, and a phrase will match if all of the terms in the phrase are present in the Tweet, regardless of order and ignoring case. By this model, you can think of commas as logical ORs, while spaces are equivalent to logical ANDs (e.g. ‘the twitter’ is the AND twitter, and ‘the,twitter’ is the OR twitter). The text of the Tweet and some entity fields are considered for matches.

Specifically, the text attribute of the Tweet, expanded\_url and display\_url for links and media, text for hashtags, and screen\_name for user mentions are checked for matches.

* locations: Specifies a set of bounding boxes to track. A comma-separated list of longitude, latitude pairs specifying a set of bounding boxes to filter Tweets by. Only geolocated Tweets falling within the requested bounding boxes will be included—unlike the Search API, the user’s location field is not used to filter Tweets.

Each bounding box should be specified as a pair of longitude and latitude pairs, with the southwest corner of the bounding box coming first. Bounding boxes do not act as filters for other filter parameters. They are in OR logic.

* delimited: Specifies whether messages should be length delimited.
* stall\_warnings: Specifies whether stall warnings should be delivered.
* language: This parameter may be used on all streaming endpoints, unless explicitly noted. Setting this parameter to a comma-separated list of [BCP 47](http://tools.ietf.org/html/bcp47) language identifiers corresponding to any of the languages listed on Twitter’s [advanced search](https://twitter.com/search-advanced) page will only return Tweets that have been detected as being written in the specified languages. For example, connecting with language=en will only stream Tweets detected to be in the English language.

### II.II.II Locations

Tweets can be associated with a location, generating a Tweet that has been ‘geo-tagged.’ Tweet locations can be assigned by using the Twitter user-interface or when posting a Tweet using the API. Tweet locations can be an exact ‘point’ location, or a Twitter Place with a ‘bounding box’ that describes a larger area ranging from a venue to an entire region.

There are two ‘root-level’ JSON objects used to describe the location associated with a Tweet: coordinates and place. Places are specific, named locations with corresponding geo coordinates.

Tweet location operators:

* **point\_radius**: The point\_radius: operator allows you to specify a circular geographic area and match Tweets containing Tweet-specific location data that fall within that area. To use, define a central lon-lat coordinate, and then set the radius (up to 25 miles). Any Tweet containing a geo Point that falls within this region will be matched. Additionally, Tweets containing Twitter Places will match where the geo polygon defined for the Place falls fully within the defined point-radius area. Places whose polygons fall outside the defined point-radius area to any extent will not match.

Usage resembles the following: point\_radius:[lon lat radius]

* **bounding\_box**: The bounding\_box: operator allows you to specify a 4-sided geographic area and match Tweets containing Tweet-specific location data that fall within that area. To use, define lon-lat coordinates that represent the opposite corners of the box, such that each side of the box is up to 25 miles in length. Any Tweet containing a geo Point that falls within this region will be matched. Additionally, Tweets containing Twitter Places will match where the geo polygon defined for the Place falls fully within the defined point-radius area. Places whose polygons fall outside the defined point-radius area to any extent will not match.

## II.III User

The following API endpoints can be used to programmatically follow users, search for users, and get user information:

* GET users/lookup
* GET users/search
* GET users/show

# Design and Implementation

At this stage we have to make decisions on how to extract data, filter it, and perform the necessary operations. Luckily, for our purposes we can use Crowd Pulse, a platform for semantic analysis of social content. It is a system rich of plugins that can be used for Emotional City. We will go here to list all possible plugins before listing those that may be of interest to us. Crowd Pulse has been used in a local virtual machine.

## III.I Crowd Pulse

Crowd Pulse is a platform for the semantic analysys of social contents. It allows to extract, filter, manipulate and analyze data retrieved from social (Twitter in our case). We can perform all these operations by means of a configuration file written in JSON inside the Crowd Pulse platform. Each configuration file is called **project**. Each project is composed by:

* Process: it contains information about the name of the project and logs’ path;
* Nodes: it contains plugins and the configurations of them;
* Edges: describe the sequence in which plugins activate;

The core of Crowd Pulse are the available plugins.

### III.I.I Available plugins

#### III.I.I.I email-notify

It reads the configuration from an email.properties file containing the following parameters:

* email.host, the SMTP server (e.g. "smtp.gmail.com");
* email.port, the port to connect to;
* email.username, the username to connect to the server;
* email.password, the password associated to the username;
* email.use\_ssl, whether to use SSL for connection;
* email.from, the email address to send messages from;
* email.subject (optional, defaults to "CrowdPulse Notification"), the subject of the email;
* email.body\_success (optional, defaults to "The pipeline "{{NAME}}" has completed successfully."), the body of the email when the plugin completes successfully (occurrences of "{{NAME}}" will be replaced with the name of the process set in the plugin);
* email.body\_error (optional, defaults to "The pipeline "{{NAME}}" has ERRORED!"), the body of the email when the plugin errors (occurrences of "{{NAME}}" will be replaced with the name of the process set in the plugin);
* email.notify\_success (optional, defaults to true), whether to notify a successful completion;
* email.notify\_error (optional, defaults to true), whether to notify an errored completion;

#### III.I.I.II social-twitter

This module contains several plugins (**extractor-twitter**, reply-extractor-twitter, profiler-twitter, twitter-profile-grapher) that need a twitter4j.properties file in the class loader accessible resources directory.

It must hold the following keys and related values:

* twitter.consumerKey: your Twitter consumer key;
* twitter.consumerSecret: your Twitter consumer secret;
* twitter.accessToken: your Twitter access token;
* twitter.accessTokenSecret: your Twitter access token secret;

#### III.I.I.III social-facebook

This module contains a collection of plugins (extractor-facebook, profiler-facebook, reply-extractor-facebook, facebook-profile-grapher) that need a facebook4j.properties file in the class loader accessible resources directory.

It must hold the following keys and related values:

* oauth.appId, your Facebook application ID;
* oauth.appSecret, your Facebook application secret;

#### III.I.I.IV data-java

Java data access layer for Crowd Pulse. The available plugins are:

* **message-fetch** to load messages stored on a database;
* message-rx-fetch to load messages stored on a database in a reactive way (experimental);
* message-filter to filter messages in a pipeline;
* **message-persist** to save messages in a database;
* profile-fetch to load profiles stored on a database;
* profile-rx-fetch to load profiles stored on a database in a reactive way (experimental);
* profile-persist to save profiles in a database;
* streamer to make the elements flow untouched in the stream;

The included packages are:

* entity: POJO classes of Crowd Pulse entities;
* repository: repositories for object persistence on MongoDB;
* plugin: shared IPlugin implementations to fetch, filter and persist Messages and Profiles;

#### III.I.I.V detect-language-optimaize

Crowd Pulse message language detector. The optimaize plugin uses a list of bundled common terms to detect the language of all streaming messages.

#### III.I.I.VI geo-profile-google-maps

Google Maps based Crowd Pulse profile geo-location fixer. To fix geolocation for profiles you need to create a geocoding.properties file and put it into the resources’ directory (must be accessible by the class loader).

This file must contain the geocoding.apiKey property, whose value must be a Google Geocoding API key. To get your key, do the following:

1. If you don't have on already, create a new project on the [Google Developers Console] ([*https://console.developers.google.com*](https://console.developers.google.com/)).
2. Go to "APIs & Auth".
3. Go to the "Credentials" sub-section and create a new Server Key. That's your key.
4. Go to the "APIs" sub-section and enable the "Geocoding API" from the Google Maps service.

#### III.I.I.VII geo-message-from-profile

Crowd Pulse message geo-location fixer. The plugin fromprofile uses the geo-location information stored in the author profile to associate messages with an approximate geo-location.

#### III.I.I.VIII index-uniba

Crowd Pulse message indexer. The index-uniba plugin needs an index-uniba.properties file in the class loader accessible resources directory. The file must contain the following properties:

* index.base is the base URL of the RESTful Web Service providing the indexing;

#### III.I.I.IX tag-babelfy

Babelfy Crowd Pulse message tagger. The babelfy plugin needs a babelfy.properties file with a babelfy.key property. To get your API key you simply need to register on [babelfy.org](http://babelfy.org/).

#### III.I.I.X tag-me

TagMe Crowd Pulse message tagger. The tagme plugin needs a tagme.properties file with a tagme.key property. To get your API key read the [Introduction](http://tagme.di.unipi.it/tagme_help.html#intro) on the [official Web site](http://tagme.di.unipi.it/). This plugin will be used and is responsible for entity-linking. It uses TAG.ME. Before this plugin, lemmatizer-stanford must be applied. You must specify a threshold for relevance (~0.15). This is preferred to Babelfy because we can easily set a threshold and we do not need any registration or properties file.

#### III.I.I.XI tag-open-calais

OpenCalais Crowd Pulse message tagger. The opencalais plugin needs a opencalais.properties file with a opencalais.key property.

#### III.I.I.XII tag-wikipedia-miner

Wikipedia Miner Crowd Pulse message tagger. The wikipediaminer plugin uses the [Wikipedia Miner](http://wikipedia-miner.cms.waikato.ac.nz/) service to perform message tagging.

#### III.I.I.XIII categorize-wikipedia

Crowd Pulse Wikipedia tag categorizer. The wikipedia plugin tries to associate Wikipedia categories to non-stop-word tags in the Messages flowing through the stream.

#### III.I.I.XIV tokenize-open-nlp

Apache OpenNLP based Crowd Pulse tokenizer. The tokenizer-opennlp plugin needs a series of tokenization models in the class loader accessible resources directory, in the form of: LANGUAGE-token.bin.

LANGUAGE is the two-characters representation of the language of the model.

Here's a list of available token models:

* [Italian](https://github.com/aciapetti/opennlp-italian-models/raw/master/models/it/it-token.bin) ([source page](https://github.com/aciapetti/opennlp-italian-models));
* [English](http://opennlp.sourceforge.net/models-1.5/en-token.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));
* [German](http://opennlp.sourceforge.net/models-1.5/de-token.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));
* [Danish](http://opennlp.sourceforge.net/models-1.5/da-token.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));
* [Dutch](http://opennlp.sourceforge.net/models-1.5/nl-token.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));
* [Portuguese](http://opennlp.sourceforge.net/models-1.5/pt-token.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));
* [Northern Sami](http://opennlp.sourceforge.net/models-1.5/se-token.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));

#### III.I.I.XV lemmatize-morphit

Crowd Pulse lemmatizer. The lemmatizer-it plugin needs two files in the class loader accessible resources directory:

* morphit contains a list of terms, lemmas and related POS tag (fetch it from the [official website](http://sslmitdev-online.sslmit.unibo.it/linguistics/morph-it.php)).
* tanl-morphit contains a mapping from TANL and MorphIT POS tags (included in this repo).
* Crowd Pulse lemmatizer implemented with Stanford CoreNLP.

#### III.I.I.XVI lemmatize-stanford-corenlp

Crowd Pulse lemmatizer implemented with Stanford CoreNLP. The lemmatizer-stanford plugin needs additional lemmatization models in the classpath (see [*http://nlp.stanford.edu/software/corenlp.shtml*](http://nlp.stanford.edu/software/corenlp.shtml)).

#### III.I.I.XVII lemmatize-multi

Multi-language Crowd Pulse lemmatizer. The lemmatizer-multi plugin redirects the lemmatization of messages to lemmatizer-it and lemmatizer-stanford according to the message language.

#### III.I.I.XVIII pos-tag-open-nlp

Multi language Crowd Pulse Part-of-Speech tagger. The postagger-opennlp plugin needs a series of POS tagging models in the class loader accessible resources directory, in the form of: LANGUAGE-pos-maxent.bin.

LANGUAGE is the two-characters representation of the language of the model.

Here's a list of available POS tagging models:

* [Italian](https://github.com/aciapetti/opennlp-italian-models/raw/master/models/it/it-pos-maxent.bin) ([source page](https://github.com/aciapetti/opennlp-italian-models));
* [English](http://opennlp.sourceforge.net/models-1.5/en-pos-maxent.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));
* [German](http://opennlp.sourceforge.net/models-1.5/de-pos-maxent.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));
* [Danish](http://opennlp.sourceforge.net/models-1.5/da-pos-maxent.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));
* [Dutch](http://opennlp.sourceforge.net/models-1.5/nl-pos-maxent.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));
* [Portuguese](http://opennlp.sourceforge.net/models-1.5/pt-pos-maxent.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));
* [Northern Sami](http://opennlp.sourceforge.net/models-1.5/se-pos-maxent.bin) ([source page](http://opennlp.sourceforge.net/models-1.5/));

#### III.I.I.XIX pos-tag-simple-it

Simpler Crowd Pulse Part-of-Speech tagger for Italian messages. The simplepostagger-it reduces complex POS-tags to the "n", "v", "a", "r" set.

#### III.I.I.XX pos-tag-simple-en

Simpler Crowd Pulse Part-of-Speech tagger for English messages. The simplepostagger-en reduces complex POS-tags to the "n", "v", "a", "r" set.

#### III.I.I.XXI pos-tag-simple-multi

Multi language simpler Crowd Pulse Part-of-Speech tagger. The simplepostagger-multi plugin reduces complex POS-tags to the "n", "v", "a", "r" set.

#### III.I.I.XXII rem-stop-word-simple

Text-based Crowd Pulse stop word remover for tags, tokens and categories. The simple plugin needs a series of stop words files in the class loader accessible resources directory. These files will be loaded if specified in the plugin configuration, e.g.:

"stop-word-step": {

"plugin": "simple",

"config": {

"apply\_to": ["tokens", "tags", "categories"],

"dictionaries": {

"all": ["stop-words-{{LANG}}", "generic-stop-words"],

"tokens": ["stop-words-{{LANG}}-tokens", "some-tokens"],

"tags": ["stop-words-{{LANG}}-tags"],

"categories": ["stop-words-{{LANG}}-categories", "exclude-categories"]

}

}

}

This plugin will be used. It executes the stopword removal for the tweets. In its configuration we can specify to which parts it must execute like tokens, tags or categories and the dictionary to be used to remove unrelevant words. We can also specify custom words to remove.

#### III.I.I.XXIII infogram

Crowd Pulse info-graphics generator. The infogram plugin needs a infogram.properties file in the class loader accessible resources directory. The file must contain the following properties:

* infogram.apikey is the API key for your Infogram account
* infogram.secret is your secret key for your Infogram account

The configuration for the plugin can specify a path where the generated infographs must be saved as PNGs. If no path is set, the files will be saved into the system temporary directory.

#### III.I.I.XXIV sentiment-sentit

SentIt based Crowd Pulse message sentiment analysis plugin. The sentiment-sentit plugin needs a sentit.properties file in the class loader accessible resources directory, with the sentit.key value holding the SentIt API key.

At the end of the execution, we can see the results through the View panel. We can view pie charts, histograms and word clouds based on the viewpoint we focus on.

#### III.I.I.XXV sentiment-sentiwordnet

SentiWordNet and MultiWordNet based Crowd Pulse message sentiment analysis plugin. The sentiwordnet plugin uses both MultiWordNet and SentiWordNet to provide a sentiment value for messages. Therefore, you need a few file in the class loader accessible resources directory:

* LANGUAGE\_index [MultiWordNet](http://multiwordnet.fbk.eu/english/home.php) files, where LANGUAGE is the two-characters code for the language you want to support (you can fetch English and Italian indexes [here](https://github.com/frapontillo/multiwordnet-simple)).
* sentiwordnet is the [SentiWordNet](http://sentiwordnet.isti.cnr.it/) file, containing mappings from WordNet synsets to sentiment values.

SentiWordNet is the result of the automatic annotation of all the synsets of WORDNET according to the notions of “positivity”, “negativity”, and “neutrality”. Each synset s is associated to three numerical scores Pos(s), Neg(s), and Obj(s) which indicate how positive, negative, and “objective” (i.e., neutral) the terms contained in the synset are.

The semi-supervised learning step consists of four substeps: (1) seed set expansion, (2) classifier training, (3) synset classification, and (4) classifier combination.

In Step (1), two small “seed” sets (one consisting of all the synsets containing 7 “paradigmatically positive” terms, and the other consisting of all the synsets containing 7 “paradigmatically negative” terms are automatically expanded by traversing a number of WORDNET binary relations than can be taken to either preserve or invert the Pos and Neg properties (i.e., connect synsets of a given polarity with other synsets either of the same polarity), and by adding the synsets thus reached to the same seed set (for polarity-preserving relations) or to the other seed set (for polarity-inverting ones). This expansion can be performed with a certain “radius”; i.e., using radius k means adding to the seed sets all the synsets that are within distance k from the members of the original seed sets in the graph collectively resulting from the binary relationships considered.

In Step (2), the two sets of synsets generated in the previous step are used, along with another set of synsets assumed to have the Obj property, as training sets for training a ternary classifier (i.e. one that needs to classify a synset as Pos, Neg, or Obj). The glosses of the synsets are used by the training module instead of the synsets themselves, which means that the resulting classifier is indeed a gloss (rather than a synset) classifier. SentiWordNet uses a “bag of words” model, according to which the gloss is represented by the (frequency-weighted) set of words occurring in it.

In Step (3) all WORDNET synsets (including those added to the seed sets in Step (2)) are classified as belonging to either Pos, Neg, or Obj via the classifier generated in Step (2). Step (2) can be performed using different values of the radius parameter, and different supervised learning technologies. We have set up our classifier committee as consisting of 8 members, resulting from four different choices of radius (k ∈ {0, 2, 4, 6}) and two different choices of learning technology.

In Step (4) the final Pos (resp., Neg, Obj) value of a given synset is generated as its average Pos (resp., Neg, Obj) value across the eight classifiers in the committee.

I will describe later why we do no choose this algorithm and we prefer SentIt.

### III.I.II Used plugins

#### III.I.II.I social-twitter

This plugin will be used to retrieve tweets. There are different filters available:

* Based on content (terms, hashtag, etc.);
* Based on user (sent and received);
* Based on geo-coordinates(**bounding\_box** approach);
* Based on filters (language, date, etc.);
* Combination of filters;

The plugin is named "extractor-twitter" and now I will report a simple exam on how to configure a simple extractor based on content (query) and geo-locations (location):

{ "process": {[…]},

"nodes": {

"message-extractor":{

"plugin": "extractor-twitter",

"config": {

“query”: [“berlusconi”, “dimaio”], // words of interest

"location": " 41.11148,16.8554,300", // point\_radius

"language": "it",

"since": "2020-11-01",

"until": "2020-11-06"

}

},

"message-persister": { […]}

},

"edges": {

"message-extractor": ["message-persister"]

}

}

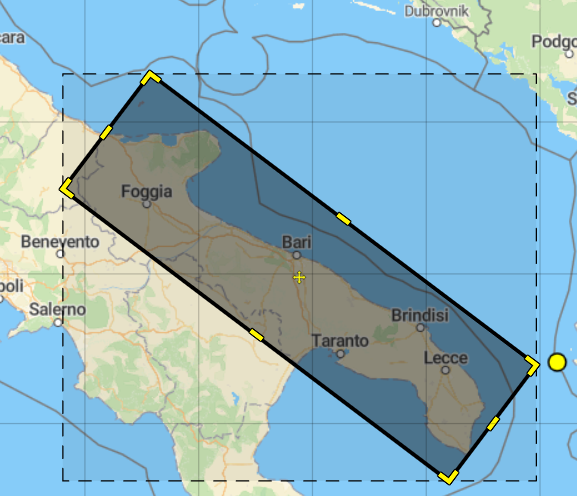
For our purposes we will use the **bounding\_box** approach because we can cover the Apulia region in a box. I used <https://boundingbox.klokantech.com/> to retrieve the coordinates of the box. Now I will report them, using the format used in Crowd Pulse (latitudine, longitude):

39.6176788304, 18.2602360711 - bottom left;

41.4837816668, 14.7748476736 - top left;

42.2822860976, 15.530690135 - top right;

40.4389399178, 19.0160785326 - bottom right;



For retrieving posts **regarding** the Apulia region (outside the region), we can query based on words. In this case, we can list the name of the region, the main cities and maybe the mayor of the main town. We can also provide a much longer list, but the API will reject too long requests and for our purposes this is sufficient. Here is a possible filter list: “puglia”, “bari”, “barletta”, “andria”, “trani”, “lecce”, “taranto”, “brindisi”, “foggia”, “decaro”.

It is true that if a post regards Cisternino and does not contain any words mentioned above, it would not be retrieved but I have to say that often in hashtags (or in the text) appear the region or the province, in the case in which the post is written outside the region.

For this purpose, combining the boundix\_box research with the hashtag query, we can collect a lot of data.

This is the extractor we will use for Emotional City. Remember that we want to collect of the last eighteen months.

{

"process": {

"name": "Project - Emotional City",

"logs": "/opt/crowd-pulse/logs"

},

"nodes": {

"message-extractor": {

"plugin": "extractor-twitter",

"config": {

"query": [

"puglia",

"bari",

"barletta",

"andria",

"trani",

"lecce",

"taranto",

"brindisi",

"foggia",

"decaro"

],

“location”: “39.6176788304, 18.2602360711, 41.4837816668, 14.7748476736, 42.2822860976, 15.530690135, 40.4389399178, 19.0160785326”,

"since": "2019-07-01",

"language": "it"

}

},

"message-persister": {

"plugin": "message-persist",

"config": {

"db": "emotionalcity"

}

}

},

"edges": {

"message-extractor": [

"message-persister"

]

}

}

#### III.I.II.II data-java

For storing the data, the message-persist plugin must be used. The only parameter in the configuration is the name of the database on which data will be stored. This plugin is already present in the file above:

"message-persister": {

"plugin": "message-persist",

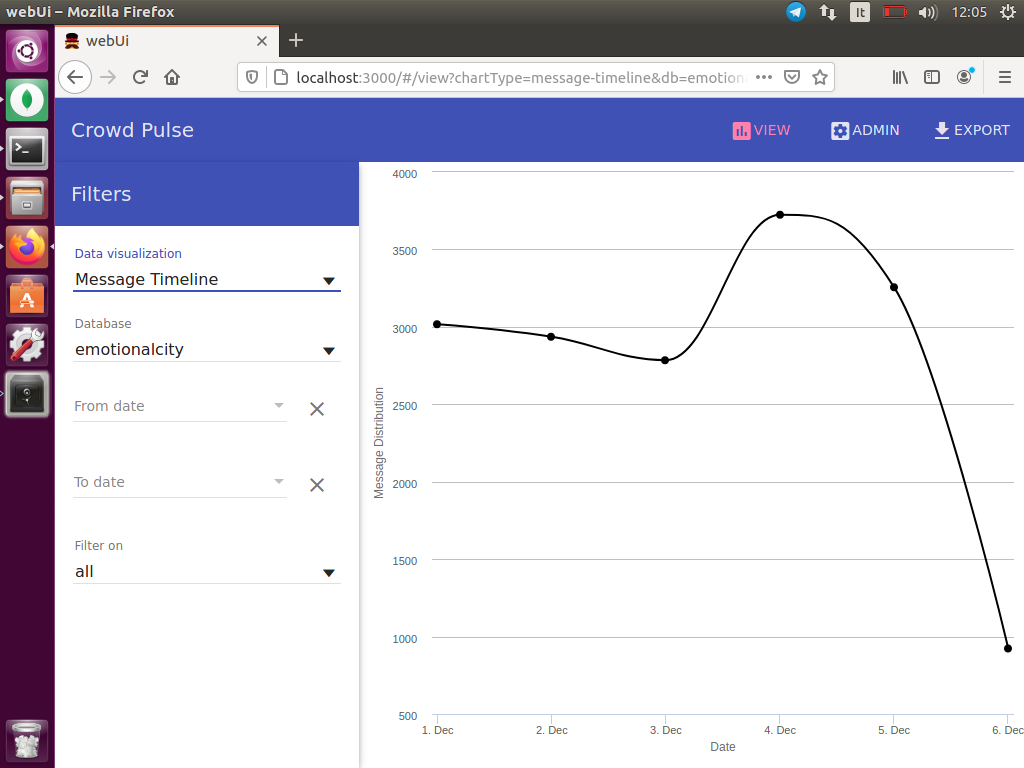
"config": {

"db": "emotionalcity"

}

}

At the end of the extraction and saving, we can see the amount of data collected per time period through Crowd Pulse. This is an example of data collected only in one day (just for example).



#### III.I.II.III tokenize-open-nlp

This plugin for tokenizing data previously retrieved. In its configuration we can specify the minimum number of characters a word must contain to be considered useful and if we want to consider mentions, urls or hashtags. In our case, first we retrieve data from the database emotionalcity, then we tokenize the tweets and finally we store the new data in the same database.

{

"process": {

"name": "Project - Emotional City processing",

"logs": "/opt/crowd-pulse/logs"

},

"nodes": {

"fetch": {

"plugin": "message-fetch",

"config": {

"db": "emotionalcity"

}

},

"tokenizer": {

"plugin": "tokenizer-opennlp",

"config": {

"minChars": "4",

"mentions": "true",

"urls": "true",

"hashtags": "false",

"calculate": "new"

}

},

"persistance": {

"plugin": "message-persist",

"config": {

"db": "emotionalcity"

}

}

},

"edges": {

"fetch": [

"tokenizer"

],

"tokenizer": [

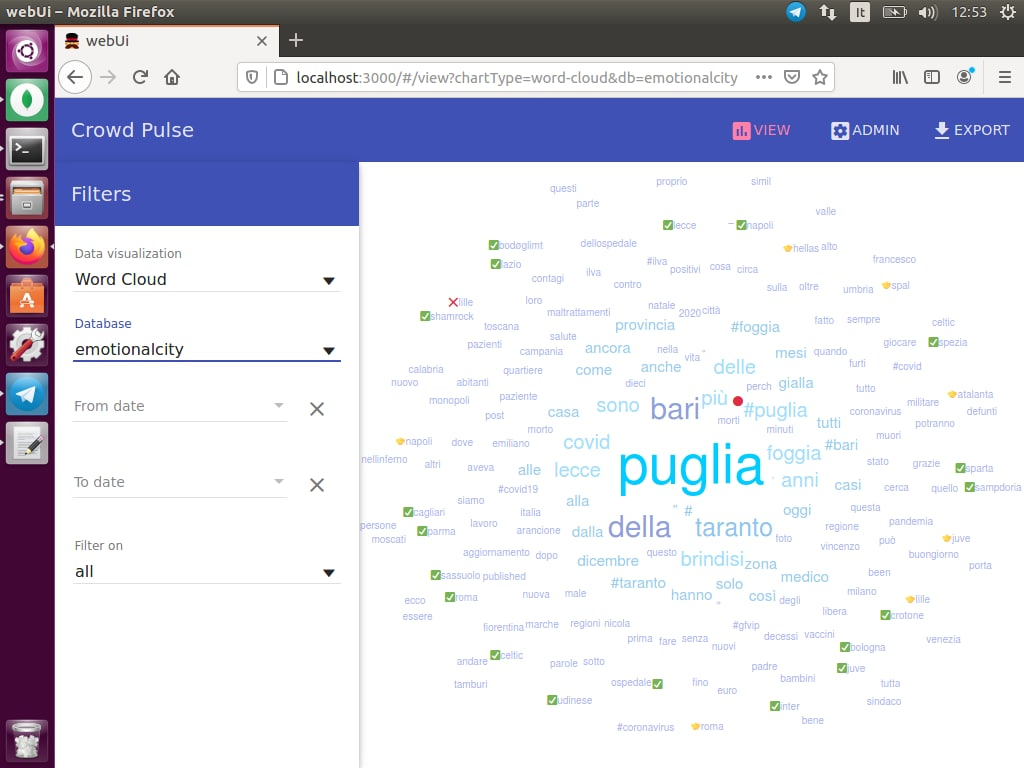
"persistance"

]

}

}

In our case, we remove mentions and urls but we maintain hashtags. I will report the Word Cloud of the tokenized tweets.



#### III.I.II.IV sentiment-sentit

This last plugin provides the sentiment analysis of the tweets. Sentiment analysis algorithms associate a polarity (negative, neutral, positive) to each tweet.

It performs a Machine Learning algorithm called SentIt that returns the polarity of tweets (1: positive, 0:neutral, -1:negative). This is preferred to the approach with SentiWordNet and MultiWordNet because it has been realized from the University of Bari (in particular by professors Novielli and Basile) and so we can have access without limits. Moreover, this system has been oriented to analyze data coming from social networks and does not provide a value for each category (positive, negative, obj) but we have a clear distinction among classes. In this way we are able to calculate statistics in an easier way. The plugin does not require any parameters.

{

"process": {

"name": "Project - Emotional City processing",

"logs": "/opt/crowd-pulse/logs"

},

"nodes": {

"fetch": {

"plugin": "message-fetch",

"config": {

"db": "emotionalcity"

}

},

"sentiment": {

"plugin": "sentiment-sentit",

"config": {

"calculate": "new"

}

},

"persistance": {

"plugin": "message-persist",

"config": {

"db": "emotionalcity"

}

}

},

"edges": {

"fetch": [

"sentiment"

],

"sentiment": [

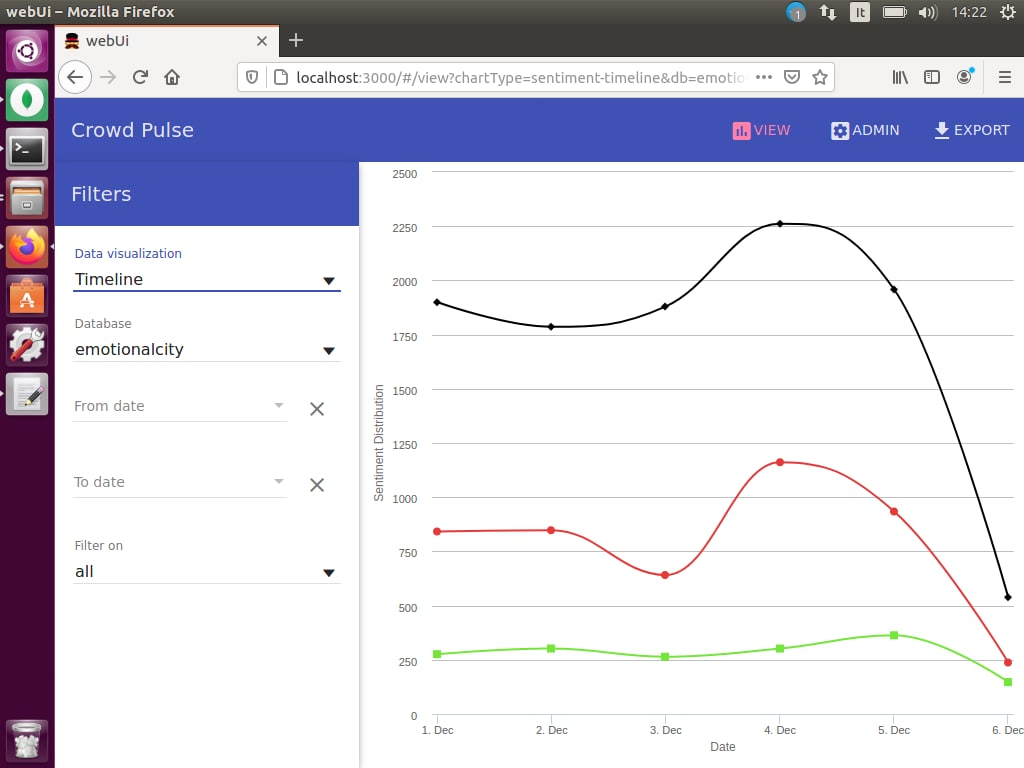
"persistance"

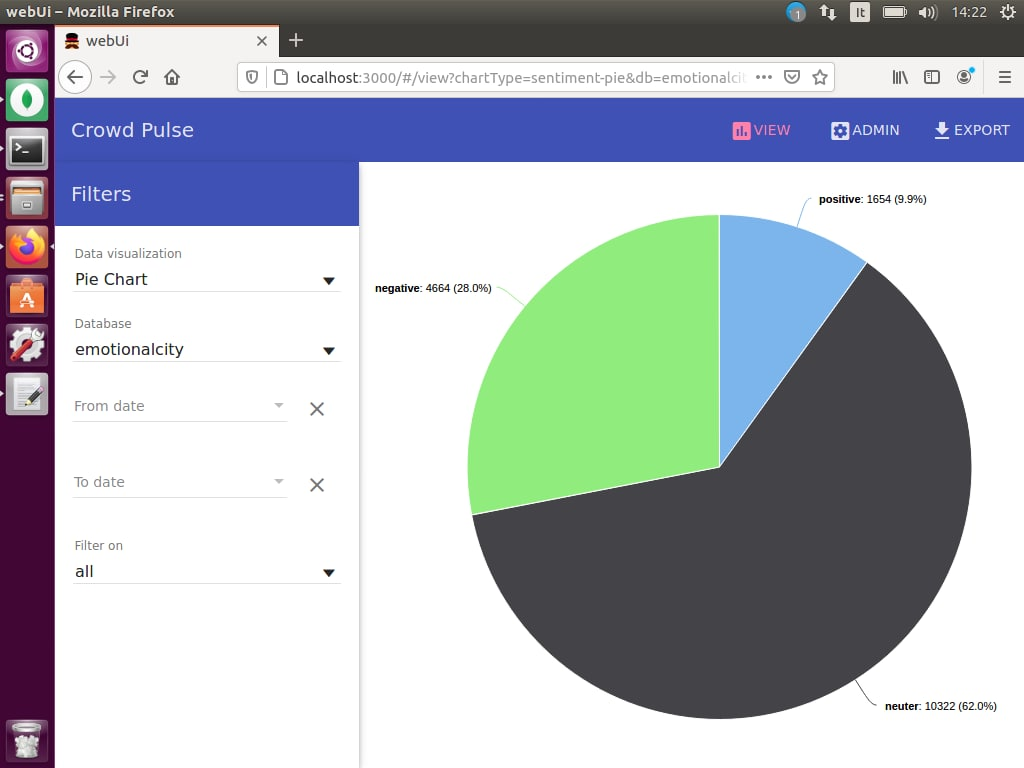
]

}

}

At the end of the SentIt algorithm, we can see statistics about the sentiment and visualize them with charts like pie chart or histograms.





## III.II Application

After the extraction of data, the Emotional City application must be realized. It consists, mainly, in retrieving information from the database filled with the data extracted through Crowd Pulse.

### III.II.I Database

MongoDB is a non-relational, document-oriented DBMS. Classified as a NoSQL type database, MongoDB moves away from the traditional table-based structure of relational databases in favor of JSON-style documents with dynamic schema. MongoDB stores data in flexible JSON-like documents, which means that fields can vary from one document to another and you can change the data structure over time.

The document model maps the objects of the application code, simplifying the work on the data. Ad-hoc queries, indexing and real-time aggregation provide effective ways to access and analyze data.

MongoDB is originally conceived as a distributed database; the high availability, horizontal scalability and geographical distribution are therefore native and easy to use. It is also free.

These are the main reasons for which we opted for this DBMS. This is an example of output produced by this model:

{

"home": "Uniba.it",

"link": "http://www.uniba.it",

"description": «Università di Bari",

"departments": [

{

"department": "Informatica",

"link": "http://informatica.uniba.it"

},

{

"department": "Matematica",

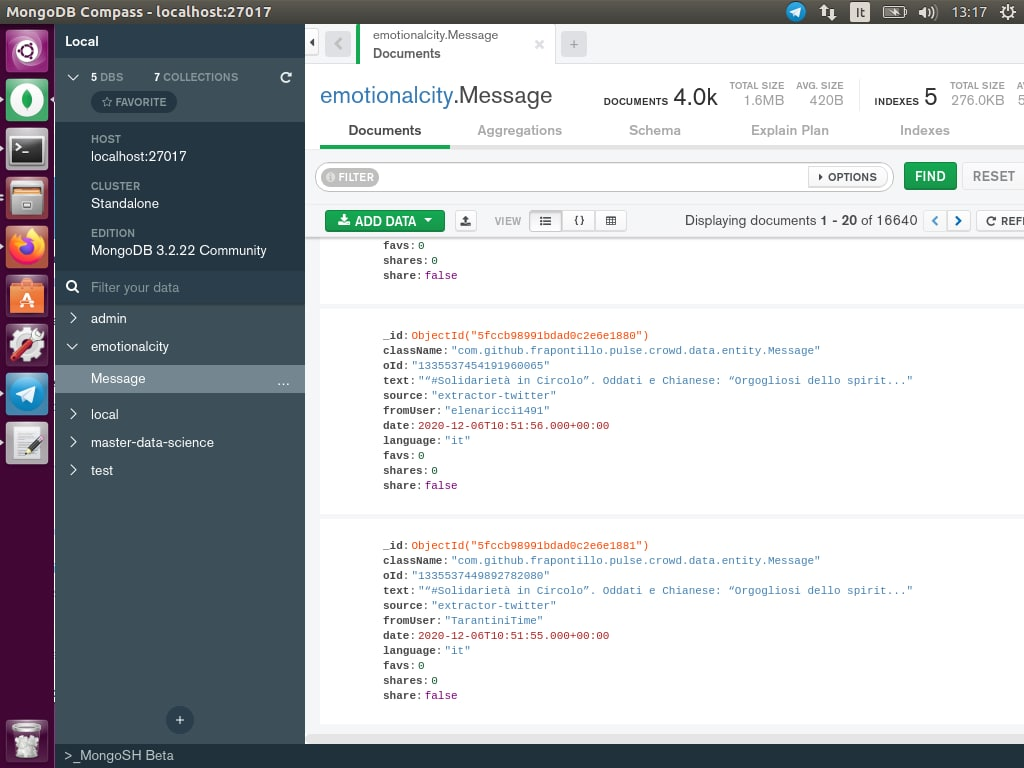
"link": "http://www.dm.uniba.it"

}

]

}

We can easily recognize that it is the same structure of the JSON file written for the configuration of the plugins. For seeing the data in the db we can use the tool Compass. Let’s have a look on some example data:



### III.II.II Emotional City API

Now that we have a database of tweets, we can start writing the Emotional City functions that will allow us to extract relevant information about Apulia.

For our case study, it was necessary to export the database from MongoDB and import on my local machine. This has been done through the command mongoexport and mongoimport. The files have been stored in the Google Drive folder of Pierluigi Cassotti, the supervisor of the case study. I used a python source code (called “onedrive.py”) to get the links to retrieve the dump of the database and the metadata. The dump is named “Message.bson” and contains the actual data while “Message.metadata.json” contains the metadata associated to Message (i.e. indexes). The supervisor gave me also the ids to connect to his Google drive folder, in order to make me access to it.

After uploading the data into my MongoDB system, I was able to implement functions to retrieve the useful data. For each function, I will describe the goal, the inputs, the type of the inputs, the output and the type of the output. All these functions will be integrated next into a system able to provides API to the Emotional City system. The name of the file collecting these functions is “twitter\_requests.py”.

#### III.II.II.I class Tweet

First, we need to create a class Tweet in order to return information about a tweet in a structured way, when necessary. The class has been specified simply through getters and setters of its elements and the function *str* which permits to print in a structured way a specific instance.

class Tweet:

def \_\_init\_\_(self, \_id, text, fromUser, date, sentiment):

self.\_id = \_id

self.text = text

self.fromUser = fromUser

self.date = date

self.sentiment = sentiment

def setClassName(self, className):

self.className = className

def setOId(self, oId):

self.oId = oId

def setCreatedDate(self, createdDate):

self.createdDate = createdDate

def setSource(self, source):

self.source= source

def setToUsers(self, toUsers):

self.toUsers = toUsers

def setLanguage(self, language):

self.language = language

def setLocation(self, location):

self.location = location

def setFavs(self, favs):

self.favs = favs

def setShare(self, share):

self.share = share

def \_\_str\_\_(self):

return "\_id : " + str(self.\_id) + ", text : " + str(self.text) + ", sentiment : " + str(self.sentiment)

#### III.II.II.II writeLogs

This function is responsible to write logs in order to tracks all the errors encountered in computations. A log file is created and has the same structure of the log generated by Crowd Pulse, so that we can read both logs using the same function.

"""

Input

logger: log handler - Logger

func : function that generates the log - string

param: wrong parameter that causes the error - string

"""

def writeLogs(logger, func, param):

if(logger is None):

raise Exception()

if(func is None):

raise Exception()

if(param is None):

raise Exception()

logger.setLevel(logging.DEBUG)

d = {'clientip': '127.0.0.1', 'user': 'root'} #custom info

fh = logging.FileHandler('emotionalcity.log')

fh.setLevel(logging.DEBUG)

logger.addHandler(fh)

logger.debug("%s [main] ERROR - Wrong or missing parameter %s in function %s", date.today(), param, func, extra=d)

#### III.II.II.III createTweet

This function creates a new tweet instance using the constructor of Tweet. It is responsible that the mandatory fields are present.

"""

Input

tweet : Tweet in the DB - JSON

logger: log handler - Logger

Output

newTweet: - Instance of the class Tweet - Tweet

"""

def createTweet(tweet, logger):

if(logger is None):

raise Exception()

if("\_id" not in tweet):

writeLogs(logger, "createTweet", "\_id")

raise Exception()

if("text" not in tweet):

writeLogs(logger, "createTweet", "text")

raise Exception()

if("fromUser" not in tweet):

writeLogs(logger, "createTweet", "fromUser")

raise Exception()

if("date" not in tweet):

writeLogs(logger, "createTweet", "date")

raise Exception()

if("sentiment" not in tweet):

writeLogs(logger, "createTweet", "date")

raise Exception()

\_id = tweet['\_id']

text = tweet['text']

fromUser = tweet['fromUser']

date = tweet['date']

sentiment = tweet['sentiment']

newTweet = Tweet(\_id, text, fromUser, date, sentiment)

if("classname" in tweet):

newTweet.setClassName(tweet['className'])

if("oId" in tweet):

newTweet.setOId(tweet['oId'])

if("createdDate" in tweet):

newTweet.setCreatedDate(tweet['createdDate'])

if("source" in tweet):

newTweet.setSource(tweet['source'])

if("toUsers" in tweet):

newTweet.setToUsers(tweet['toUsers'])

if("language" in tweet):

newTweet.setLanguage(tweet['language'])

if("location" in tweet):

newTweet.setLocation(tweet['location'])

if("favs" in tweet):

newTweet.setFavs(tweet['favs'])

if("shares" in tweet):

newTweet.setShares(tweet['shares'])

return newTweet

#### III.II.II.IV count\_tweets

This function divides the tweets in periods given start, end and time\_range. The latter is the dimension of every split. For each period the number of tweets is given. If the parameter only\_geo is true, only geo located tweets will be taken into account.

"""

Input

db: DB instance - Database

start : Unix timestamp in seconds - int

end : Unix timestamp in seconds - int

time\_range : Length of the interval in seconds - int

only\_geo : Consider only tweets that are geolocated - boolean

logger: log handler - Logger

Output

[d0,d1,...,dN]

N = (end-start)/time\_range

d0 : (i0,i1,count)

i0 : Interval start

i1 : Interval end

If only\_geo==False:

count : Integer - Number of tweets gathered in the interval (i0,i1)

Else:

count : Integer - Number of tweets geolocated, gathered in the interval (i0,i1)

WARNING The attribute in the Message Repository that refers to the tweet creation is date that is not a timestamp

(The attribute created\_date instead refers to the date when a tweet is stored in mongodb)

"""

def count\_tweets(db, start, end, time\_range, only\_geo, logger):

"""

i0 = calendar.timegm(datetime.utcnow().utctimetuple())

i1 = calendar.timegm(datetime.utcnow().utctimetuple())

count = 4

d0 = (i0,i1,count)

i0 = calendar.timegm(datetime.utcnow().utctimetuple())

i1 = calendar.timegm(datetime.utcnow().utctimetuple())

count = 100

d1 = (i0,i1,count)

result = [d0,d1]

return result

"""

if(logger is None):

raise Exception()

if(db is None):

writeLogs(logger, "count\_tweets", "db")

raise Exception()

if(start is None):

writeLogs(logger, "count\_tweets", "start")

raise Exception()

if(end is None):

writeLogs(logger, "count\_tweets", "end")

raise Exception()

if(time\_range is None):

writeLogs(logger, "count\_tweets", "time\_range")

raise Exception()

if(only\_geo == True):

tweets = list(db.Message.find({"$and": [

{"date": {

"$gte": int2time(start),

"$lt": int2time(end)

}},

{"location": {"$exists": True} }

]}))

else:

tweets = list(db.Message.find({"date": {

"$gte": int2time(start),

"$lt": int2time(end)

}} ))

n\_ranges = math.ceil((end-start)/time\_range)

ranges = np.zeros((n\_ranges, 3))

lower = start;

upper = start + time\_range

sum = 0

for i in range (0, n\_ranges):

ranges[i][0] = lower

ranges[i][1] = upper

lower = upper

upper += time\_range

for i in range(0, len(tweets)):

ranges[int((time2int(tweets[i]['date'])-start)/time\_range)][2] += 1

sum += 1

return (ranges, sum)

#### III.II.II.V time2int

This function returns the number of seconds passed from the 1970/01/01 until the time passed in output.

"""

Input

time: Timestamp - Unix timestamp

YYYY-MM-DD HH-MM-DD

Output

seconds: Distance in seconds from the 1970/01/01 - int

"""

def time2int(timestamp):

return int(time.mktime(timestamp.timetuple()))

#### II.II.II.VI int2time

This function is inverse to the previous one. It returns the timestamp given the time passed in seconds.

"""

Input

dist: Distance in seconds from the 1970/01/01 - int

Output

time: Timestamp - Unix timestamp

YYYY-MM-DD HH-MM-DD

"""

def int2time(seconds):

return datetime.fromtimestamp(seconds)

#### III.II.II.VII string2int

This function is the same as time2int but accepts the date in input as string.

"""

Input

date: Date in format YYYY/MM/DD - string

Output

seconds: Distance in seconds from the 1970/01/01 - int

"""

def string2int(date):

return int(time.mktime(datetime.strptime(date, "%Y/%m/%d %H:%M:%S").timetuple()))

#### III.II.II.VIII senti\_tweets

This function acts as count\_tweets but, for any period, it gives also the mean of the sentiment of the tweets.

"""

Input

db: DB instance - Database

start : Unix timestamp in seconds - int

end : Unix timestamp in seconds - int

time\_range : Length of the interval in seconds - int

only\_geo : Consider only tweets that are geolocated - boolean

logger: log handler - Logger

Output

[d0,d1,...,dN]

N = (end-start)/time\_range

d0 : (i0,i1,sentiment)

i0 : Interval start

i1 : Interval end

If only\_geo==False:

sentiment : Float - Mean value of tweets sentiment in the interval (i0,i1)

Else:

sentiment : Float - Mean value of geolocated tweets sentiment in the interval (i0,i1)

WARNING The attribute in the Message repository that refers to the tweet creation is date that is not a timestamp

(The attribute created\_date instead refers to the date when a tweet is stored in mongodb)

"""

def senti\_tweets(db, start, end, time\_range, only\_geo, logger):

"""

i0 = calendar.timegm(datetime.utcnow().utctimetuple())

i1 = calendar.timegm(datetime.utcnow().utctimetuple())

sentiment = 0.8

d0 = (i0,i1,sentiment)

i0 = calendar.timegm(datetime.utcnow().utctimetuple())

i1 = calendar.timegm(datetime.utcnow().utctimetuple())

sentiment = 0.4

d1 = (i0,i1,sentiment)

result = [d0,d1]

return result

"""

if(logger is None):

raise Exception()

if(db is None):

writeLogs(logger, "senti\_tweets", "db")

raise Exception()

if(start == None):

writeLogs(logger, "senti\_tweets", "start")

raise Exception()

if(end == None):

writeLogs(logger, "senti\_tweets", "end")

raise Exception()

if(time\_range == None):

writeLogs(logger, "senti\_tweets", "time\_range")

raise Exception()

if(only\_geo == True):

tweets = list(db.Message.find({"$and": [

{"location": {"$exists": True} },

{"date": {

"$gte": int2time(start),

"$lt": int2time(end)

}},

]}))

else:

tweets = list(db.Message.find({"date": {

"$gte": int2time(start),

"$lt": int2time(end)

}}))

n\_ranges = math.ceil((end-start)/time\_range)

ranges = np.zeros((n\_ranges, 4))

lower = start;

upper = start + time\_range

for i in range (0, n\_ranges):

ranges[i][0] = lower

ranges[i][1] = upper

lower = upper

upper += time\_range

for i in range(0, len(tweets)):

ranges[int((time2int(tweets[i]['date'])-start)/time\_range)][2] += 1

ranges[int((time2int(tweets[i]['date'])-start)/time\_range)][3] += tweets[i]['sentiment']

for i in range (0, n\_ranges):

if(ranges[i][2] > 0):

ranges[i][3] /= ranges[i][2]

return ranges

#### III.II.II.IX geo\_count\_tweets

This returns the number of tweets taken retrieved inside some geo coordinates and after a certain period.

"""

Input

db: DB instance - Database

time : Unix timestamp in seconds - int

geo : List of couples of two Doubles (latitude,longitude) - example [(90.,1.)] - List<(Double,Double)>

logger: log handler - Logger

Output

count - int

count : Number of tweets gathered from the location inside geo and after time

https://docs.mongodb.com/manual/reference/operator/query-geospatial/

The function should filtering those messages created after the UNIX timestamp time.

WARNING The attribute in the Message repository that refers to the tweet creation is date that is not a timestamp

(The attribute created\_date instead refers to the date when a tweet is stored in mongodb)

"""

def geo\_count\_tweets(db, time, geo, logger):

if(logger is None):

raise Exception()

if(db is None):

writeLogs(logger, "geo\_count\_tweets", "db")

raise Exception()

if(time is None):

writeLogs(logger, "geo\_count\_tweets", "time")

raise Exception()

if(geo is None):

writeLogs(logger, "geo\_count\_tweets", "geo")

raise Exception()

tweets = list(db.Message.find({"$and": [

{"location": {"$geoWithin": {"$box": geo} } },

{"date": {"$gte": int2time(time)} }

]}))

return len(tweets)

#### III.II.II.X geo\_senti\_tweets

This function acts as geo\_count\_tweets but it also calculates the mean of the sentiment.

"""

Input

db: DB instance - Database

time : Unix timestamp in seconds - int

geo : List of couples of two Doubles (latitude,longitude) - example [(90.,1.)] - List<(Double,Double)>

logger: log handler - Logger

Output

sentiment : Mean value of tweets sentiment of tweets gathered from a location inside geo and after time - float

https://docs.mongodb.com/manual/reference/operator/query-geospatial/

The function should filtering those messages created after the UNIX timestamp time.

WARNING The attribute in the Message repository that refers to the tweet creation is date that is not a timestamp

(The attribute created\_date instead refers to the date when a tweet is stored in mongodb)

"""

def geo\_senti\_tweets(db, time, geo\_box, logger):

if(logger is None):

raise Exception()

if(db is None):

writeLogs(logger, "geo\_senti\_tweets", "db")

raise Exception()

if(time is None):

writeLogs(logger, "geo\_senti\_tweets", "time")

raise Exception()

if(geo\_box is None):

writeLogs(logger, "geo\_senti\_tweets", "geo\_box")

raise Exception()

print(int2time(time))

tweets = list(db.Message.find({"$and": [

{"location": {"$geoWithin": {"$box": geo\_box} } },

{"date": {"$gte": int2time(time)} }

]}))

sentiment = 0

for i in range(0, len(tweets)):

sentiment += tweets[i]['sentiment']

return sentiment/len(tweets) if len(tweets) > 0 else 0

#### III.II.II.XI search\_tweets

This function return the tweets gathered between two periods of time and containing specific query words in the text.

"""

Input

db: DB instance - Database

start : Unix timestamp - int

end : Unix timestamp - int

query : words - List<str>

logger: log handler - Logger

Search in tweets the query tokens

Output

[t0,t1,...,tN] - Tweets gathered in the interval (start,end)

t0 : Instance of class Tweet - Tweet

WARNING IN MONGODB THE ATTRIBUTE TEXT IS INDEXED

https://docs.mongodb.com/manual/text-search/

WARNING The attribute in the Message repository that refers to the tweet creation is date that is not a timestamp

(The attribute created\_date instead refers to the date when a tweet is stored in mongodb)

"""

#\_id,className,oId,createdDate,text,source,fromUser,toUsers,date,language,location,favs,share,sentiment

def search\_tweets(db, start, end, query, logger):

if(logger is None):

raise Exception()

if(db is None):

writeLogs(logger, "search\_tweets", "db")

raise Exception()

if(start is None):

writeLogs(logger, "search\_tweets", "start")

raise Exception()

if(end is None):

writeLogs(logger, "search\_tweets", "end")

raise Exception()

if(query is None):

writeLogs(logger, "search\_tweets", "query")

raise Exception()

q = ""

for s in query:

q += s + " "

print(q)

tweets = list(db.Message.find({"$and": [

{"date": {

"$gte": int2time(start),

"$lt": int2time(end)

}},

{ "$text": { "$search": q } }

]}))

return tweets

#### III.II.II.XII search\_geo\_tweets

This function return the tweets given different filters: query words, minimum sentiment, maximum sentiment, geo position and maximum time.

"""

Input

db : DB instance - Database

query : words - List<str>

sentiment\_min : Minimum sentiment - float

sentiment\_max : Maximum sentiment - float

geo : List of couples of two Doubles (latitude,longitude) - example [(90.,1.),(90,0.2),..] - List<(Double,Double)>

time : Time - int

logger : log handler - Logger

Search in tweets the query tokens

Output

[t0,t1,...,tN] - Tweets gathered in the interval (start,end)

t0 : Tweet (instance of class Tweet)

If sentiment\_min is not None

If sentiment\_max is not None

search only where sentiment-max >= tweet.sentiment >= sentiment-min

Else

search only where tweet.sentiment >= sentiment-min

Else

If sentiment\_max is not None

search only where tweet.sentiment <= sentiment-max

If geo is not None search only where tweet.location is inside geo object.

If time is not None search only where tweet.date < time

WARNING IN MONGODB THE ATTRIBUTE TEXT IS INDEXED

https://docs.mongodb.com/manual/text-search/

"""

def search\_geo\_tweets(db, query, sentiment\_min, sentiment\_max, geo, time, logger):

if(logger is None):

raise Exception()

if(db is None):

writeLogs(logger, "search\_geo\_tweets", "db")

raise Exception()

if(sentiment\_min is None):

sentiment\_min = -1

if(sentiment\_max is None):

sentiment\_max = 1

if(geo is not None or time is not None):

if(geo is not None and time is not None):

tweets = list(db.Message.find({"$and": [

{"location": {"$geoWithin": {"$box": geo} } },

{"date": {"$lt": int2time(time) } },

{"sentiment": {"$gte": sentiment\_min } },

{"sentiment": {"$lte": sentiment\_max } }

]}))

else:

if(geo is not None):

tweets = list(db.Message.find({"$and": [

{"location": {"$geoWithin": {"$box": geo} } },

{"sentiment": {"$gte": sentiment\_min } },

{"sentiment": {"$lte": sentiment\_max } }

]}))

else:

tweets = list(db.Message.find({"$and": [

{"date": {"$lt": int2time(time) } },

{"sentiment": {"$gte": sentiment\_min } },

{"sentiment": {"$lte": sentiment\_max } }

]}))

else:

tweets = list(db.Message.find({"$and": [

{"sentiment": {"$gte": sentiment\_min } },

{"sentiment": {"$lte": sentiment\_max } }

]}))

listTweets = []

for i in range(0, len(tweets)):

found = False

for q in query:

if(q in tweets[i]['text']):

found = True

if (found == True):

newTweet = createTweet(tweets[i], logger)

listTweets.append(newTweet)

return listTweets

#### III.II.II.XIII backup

This function creates a dump of the database and saves a copy into the Google Drive folder.

"""

Input

db : DB instance - Database

time : Unix timestamp in seconds - int

logger : log handler - Logger

Use mongodump for backups, the function should filtering those messages created after the UNIX timestamp time.

WARNING!!

In this case use CREATED\_DATE

in the Message Repository use the attribute created\_date, the attribute is UNIX timestamp EXPRESSED IN MILLISECONDS.

Backup save path

emc-backups/year/month/day/ - Extract this information by the time parameter.

"""

def backup(db, time, logger):

#Configuration

CLIENT\_ID = '7419936f-f4e1-4800-9afa-6d90e09f03cb'

TENANT\_ID = 'c6328dc3-afdf-40ce-846d-326eead86d49'

AUTHORITY\_URL = 'https://login.microsoftonline.com/{}'.format(TENANT\_ID)

RESOURCE\_URL = 'https://graph.microsoft.com/'

API\_VERSION = 'v1.0'

SCOPES = ['Sites.ReadWrite.All','Files.ReadWrite.All']

sys.path.append('.')

from onedrive import upload

from onedrive import check\_hash

from onedrive import get\_headers

'''

if(logger is None):

raise Exception()

if(db is None):

writeLogs(logger, "backup", "db")

raise Exception()

if(time is None):

writeLogs(logger, "backup", "time")

raise Exception()

'''

timestamp = str(int2time(int(time/1000)))

year = timestamp[:4]

month = timestamp[5:7]

day = timestamp[8:10]

'''

EXPORT BSON

tweets = db.Message.find({ "createdDate": { "$gte": time\*1000} } )

with open('backups/Message.bson', 'wb+') as f:

for doc in tweets:

f.write(BSON.encode(doc))

'''

tweets = db.Message.find({"createdDate" : {"$gt": time}}) #createdDate is in milliseconds

col = getattr(db, "tweets")

collection = tweets

jsonpath = "tweets" + ".json"

jsonpath = join("backups/", jsonpath)

with open(jsonpath, 'wb') as jsonfile:

jsonfile.write(dumps(collection).encode())

upload("backups/", "emc-backups/" + year + "/" + month + "/" + day, get\_headers(CLIENT\_ID, AUTHORITY\_URL, SCOPES))

#check\_hash("backups/", "emc-backups/" + year + "/" + month + "/" + day + "/", None)

#os.chdir("backups/")

#os.remove("emc-backups/" + year + "/" + month + "/" + day + "/")

#### II.II.II.XIV read\_backups

This function reads the Google Drive backup folder in order to retrieve the files.

"""

List files in the folder with the name, size and the number of files inside if it is a folder

If year is None:

List years

If month is None:

List months

If day is None:

List days

Else:

List files in emc-backups/year/month/day/

Input:

year : Year - int

day : Month - int

month : Day - int

logger : log handler - Logger

"""

def read\_backups(year, month, day, logger):

from onedrive import list\_files\_folders

if(logger is None):

raise Exception()

if(year == None):

writeLogs(logger, "read\_backups", "year")

raise Exception()

if(month == None):

writeLogs(logger, "read\_backups", "month")

raise Exception()

if(day == None):

writeLogs(logger, "read\_backups", "day")

raise Exception()

list\_files\_folders("emc-backups/" + year + "/" + month + "/" + day + "/", None)

#### II.II.II.XV get\_log

This function opens a log and reads line matching some filters like type or component.

"""

Filters:

date Unix timestamp in seconds

log\_type = INFO,WARN,ERROR

component = es. opennlp

Input:

log\_path : log file path - string

log\_type : Type - string

component : Component - string

logger : log handler - Logger

"""

def get\_log(log\_path, date, log\_type, component, logger):

if(logger is None):

raise Exception()

if(log\_path is None):

writeLogs(logger, "get\_log", "log\_path")

raise Exception()

f = open(log\_path, "r")

file = f.read().split("\n")

for line in file:

if(date is None):

dateString = line[:19]

else:

dateString = str(int2time(date))

if(log\_type is None):

log\_type = line[0] #any substring of line is correct

if(component is None):

component = line[0] #any substring of line is correct

if(dateString==line[:19] and log\_type in line and component in line):

print(line)

### III.II.III Test

#### III.II.III.I main

First, we read the general information about the database like the name, the credentials and the name of the collection. All these properties are in the file named “twitter\_requests.properties” which is read before starting any operation. After that, we connect to the db and we create an object logger of type Logger to create eventual log messages.

os.chdir("C:\\Users\ddipi\Desktop\Emotionalcity")

f = open("twitter\_requests.properties", "r")

file = f.read().split("\n")

db\_url = file[0].split("=")[1]

db\_name = file[4].split("=")[1]

client = MongoClient(db\_url, unicode\_decode\_error\_handler='ignore')

db = client[db\_name]

logger = logging.getLogger('emotionalcity')

#### III.II.III.II count\_tweets

print("TEST count\_tweets :")

try:

(ranges, sum) = count\_tweets(db, 1607464906, 1607475302, 3600, False, logger)

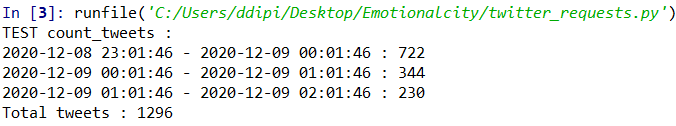
for i in range (0, ranges[:,0].size):

print(str(datetime.fromtimestamp(ranges[i][0])) + " - " + str(datetime.fromtimestamp(ranges[i][1])) + " : " + str(int(ranges[i][2])))

print("Total tweets : " + str(sum))

except:

print("Wrong parameters in count\_tweets")



#### III.II.III.III senti\_tweets

print("\nTEST senti\_tweets :")

try:

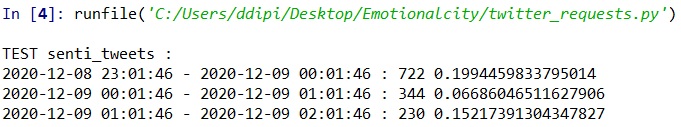
ranges = senti\_tweets(db, 1607464906, 1607475302, 3600, False, logger)

for i in range (0, ranges[:,0].size):

print(str(datetime.fromtimestamp(ranges[i][0])) + " - " + str(datetime.fromtimestamp(ranges[i][1])) + " : " + str(int(ranges[i][2])) + " " + str(ranges[i][3]))

except:

print("Wrong parameters in senti\_tweets")



#### III.II.III.IV geo\_count\_tweets

print("\nTEST geo\_count\_tweets :")

try:

count = geo\_count\_tweets(db, 1607466785, [(40.41388645, 18.1876629), (41.092715299999995, 16.87499235)], logger)

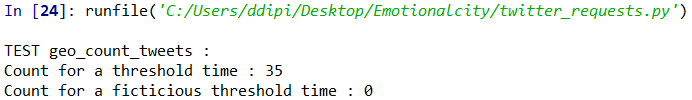
print("Count for a threshold time : " + str(count))

count = geo\_count\_tweets(db, 9999999999, [(40.41388645, 18.1876629), (41.092715299999995, 16.87499235)], logger)

print("Count for a ficticious threshold time : " + str(count))

except:

print("Wrong parameters in geo\_count\_tweets")



#### III.II.III.V geo\_senti\_tweets

print("\nTEST geo\_senti\_tweets : ")

try:

sentiment = geo\_senti\_tweets(db, 1607466785, [(40.41388645, 18.1876629), (41.092715299999995, 16.87499235)], logger)

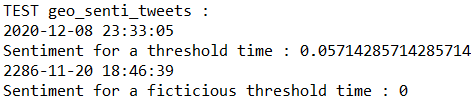
print("Sentiment for a threshold time : " + str(sentiment))

sentiment = geo\_senti\_tweets(db, 9999999999, [(40.41388645, 18.1876629), (41.092715299999995, 16.87499235)], logger)

print("Sentiment for a ficticious threshold time : " + str(sentiment))

except:

print("Wrong parameters in geo-senti\_tweets")



#### III.II.III.VI search\_tweets

print("\nTEST search\_tweets :")

try:

tweetList = search\_tweets(db, 1607464906, 1607475302, ["manchi", "fegato"], logger)

for i in range(0, len(tweetList)):

print(tweetList[i])

except:

print("Wrong parameters in search\_tweets")

#### III.II.III.VII search\_geo\_tweets

print("\nTEST search\_geo\_tweets :")

try:

tweetList = search\_geo\_tweets(db, ["Buonanotte", "#realjuve"], None, None, [(40.41388645, 18.1876629), (41.092715299999995, 16.87499235)], 1607466785, logger)

for x in tweetList:

print(x)

except:

print("Wrong parameters in search\_geo\_tweets")

#### III.II.III.VIII backup

print("\nTEST backup :")

try:

backup(db, 0, logger)

print(os.listdir("backup"))

except:

print("Wrong parameters in backup")

#### III.II.III.IX read\_backups

print(“\nTEST read\_backups :”)

try:

read\_backups(None, "2020", None, None, logger)

except:

print("Wrong parameters in read\_backups")

#### III.II.III.X get\_log

print("\nTEST get\_log : ")

try:

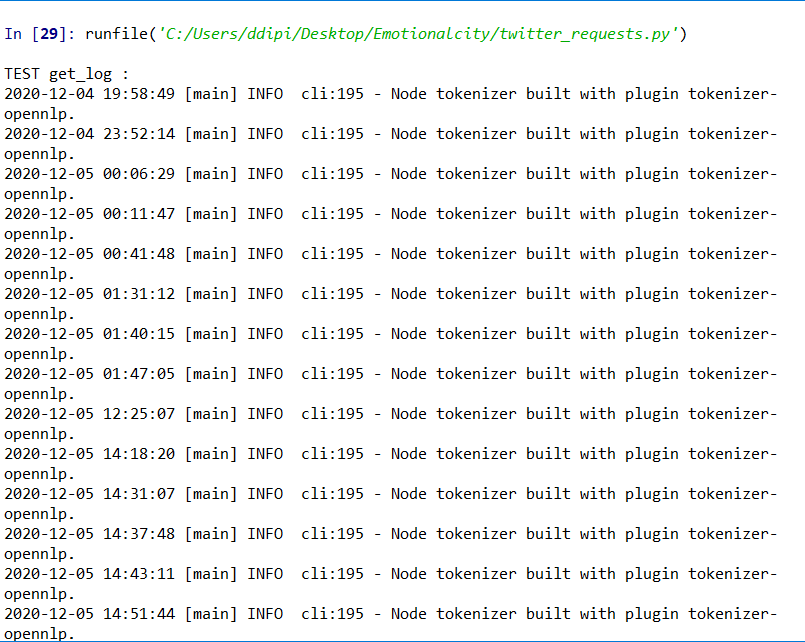
get\_log("C:\\Users\ddipi\Desktop\emotionalcity\log.log", None, "INFO", "opennlp", logger)

print("\n")

get\_log("C:\\Users\ddipi\Desktop\emotionalcity\log.log", string2int("2020/12/04 19:49:28"), "ERROR", None, logger)

except:

print("Wrong parameters in log")



After all the operations, we shut down the logger through logging.shutdown()

# Tools

For the overall development of the case study, we used the following tools:

* Crowd Pulse: <https://github.com/frapontillo/crowd-pulse>
* MongoDB: <https://www.mongodb.com/it>
* Anaconda: <https://www.anaconda.com/>
* Oracle VM VirtualBox: <https://www.virtualbox.org/>