Dynamic Twitter geographical categorization

Mattia Affabris, 183200 - Alessandro Pezzè, 182501
[name].[surname]@studenti.unitn.it

June 21, 2017

CONTENTS

LIST OF FIGURES

***Abstract*—This document describes the overall concept, pragmatic execution, and related findings of a map-reduce implementation for the categorization of Twitter[1] tweets within dynamic geographical boundaries. Data is obtained from a geolocated stream of tweets; for each tweet a set of topics is extracted; an algorithm is eventually ran to generate a query-able map. A public API then exposes a second separate algorithm, capable of finding the most relevant topic in a portion of the overall map. These two algorithms are finally analysed and compared with regards to their performance and efficiency, both relative and absolute.**

I. Introduction

Nowadays the ability to efficiently retrieve large amounts of data from even larger datasets allows for smarter, faster, and overall better decision making in countless disciplines.[1] Coupled with the ability to programmatically analyse the human language, and derive meaning from complex domains such as written discourse, human behaviour may be efficiently categorized for a variety of purposes: data retrieved from something as simple as a Twitter map can yield significant results, if and when used correctly. With enough data, and precise enough human language understanding algorithms, one could not only arguably describe a population's current social situation, but also to a certain extent predict its variations in the near or far future.[2]

To avoid retreading on already well known theory, in this document it is assumed from here on that any required human language understanding systems are already in place, and that the challenge is thus merely twofold: data aggregation, in the sense of collecting and processing large amounts (volume) of possibly inconsistent (variability), real-time (velocity), dispared (variety), and correct (veracity) topics from Twitter tweets; [2] efficiency, in the sense of delivering the results of any computation within or sufficiently close to a given ideal set of constraints.

The aim of this project is thus to develop two disjointed and independent algorithms:

- one to bundle within a general pre-determined area a set of topics extracted form multiple tweets,
- one to extract the most relevant topic from an arbitrary subset of the initial area.

II. Related work

The process of acquiring, analysing, and aggregating large amounts of data is too common of a domain to be thoroughly referenced. Limiting the analysis only to Twitter related projects though yields a number of similar related works.

- *Twitter Data Clustering and Visualization* [3] aims to

  > *[...] demonstrate Twitters' capacity of identifying opinions, trends and events in a specific geographic location. Users' tweets are gathered, processed, analyzed and visualized in a 3D geospatial representation [...]*

  Their solution leverages Hadoop, C++, and Matlab, and is focused on the end-user experience rather than the underlying algorithms' efficiency, scalability, and/or versatility: the end result is a set of clustered tweets statically placed within a 2D or 3D map.
- *Geo-spatial Event Detection in the Twitter Stream* [4] focuses on real-time analysis of Twitter streams, in an effort to identify hot-spots of activity in a 2D geographical space. More importance is given to the analysis of the stream, and less to the geographical positioning of the hot-spots and the underlying execution parameters.
- *TwitInfo: Aggregating and Visualizing Microblogs for Event Exploration* [5] concentrates on aggregation based on specific events, rather than geographical boundaries:

  > *TwitInfo allows users to browse a large collection of tweets using a timeline-based display that highlights peaks of high tweet activity.*

  Search queries are thus event-based rather than map-based, yielding related sub-events and overall sentiment, rather than geographically bounded topics.

III. Problem statement

A. Topics extraction

As stated in Section I, for simplicity purposes a tweet's *topic(s)* is here defined as all those Hashtag(s), User Mention(s), and Symbol(s)[3] found within its body only. As such, there is no specific need implement a human language understanding algorithm. Note that if desired or required for further works, such an implementation can be easily integrated in the project's flow.

B. Mapping algorithms

Given an overall rectangular area (`map M`) and a set of geotagged tweets[4] (`data U`) placed within `map M`, compute the popular topics. As one changes the selection boundaries of a rectangular area (`selection C`), the computation must be done on the subset of `data U` made of only those Tweets that are placed within `selection C`. Ideally, the computation of the `selection C` must be significantly faster than the computation of the initial `map M`.

Two cascading algorithms are proposed to solve this problem statement: the *bootstrap algorithm*, and the *selection algorithm*. See Figure 1 for a visualization of the input and output flow.

Bootstrap algorithm:

- input
  - a set of geotagged *Tweets* (`data U`)
  - a rectangular area (`map M`)
  - a geographic distance (`step S`)
- output
  - a grid (`grid G`) of `SxS` squares (`tile T`)
  - for each `tile T` an enumeration of *topics*

[1] https://twitter.com/

[2] For some engaging fictional literature on the subject, see the fabricated concept of "Psychohistory" created by Isaac Asimov for his Foundation cycle of novels.

[3] https://dev.twitter.com/overview/api/entities-in-twitter-objects

[4] https://support.twitter.com/articles/78525

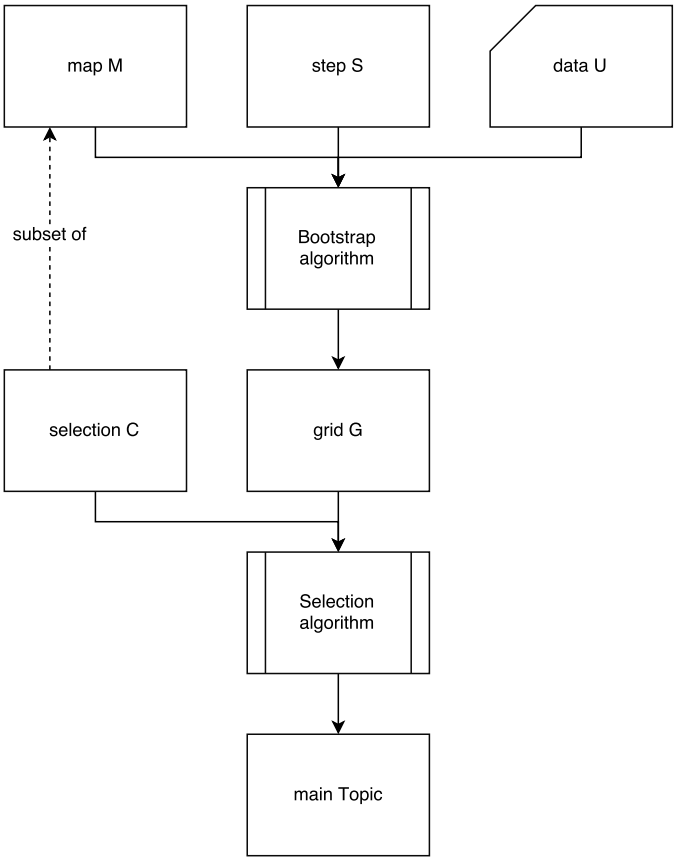


Figure 1. Section III-B: overall project input/output flow

Selection algorithm:

- input
  - a rectangular area (`selection C`)
  - the `grid G`
- output
  - the main *topic* of `selection C`

C. Considerations

Both the `map M` and the `selection C` rectangle areas are uniquely identified by two pairs of coordinates, one defining the lower-left vertex, and one the upper-right vertex.

The minimum `tile T` size is not set, but an inherent limitation is in place due to the precision of the data retrieved through Twitter's APIs: more specifically, coordinates are Lat/Lng pairs with up to 5 precision digits (i.e. 1.1 m of accuracy). [5] [6]

When running the *selection algorithm*, every `tile T` included in the `selection C` is used for the calculations, even if only a portion of the tile resides within the selection. This simplification is inherent in the problem's description. Consider for example a `step S` small enough such that every tweet resides in his own tile: the computation made on the `grid G` would thus be identical to the one made by the *bootstrap algorithm* on the initial `map M`, negating the required speed advantage of the second algorithm. As such, further subdividing the `tile T` to account for partial inclusion is the same as specifying an initial smaller `step S`. Note though that this over-inclusion becomes less impactful on the result's accuracy the larger the requested area, and the smaller the `step S`. See Figure 2 for a visualization.

[5] Note that most commercial GPSs are only accurate up to 4 precision digits (i.e. 11 m of accuracy).

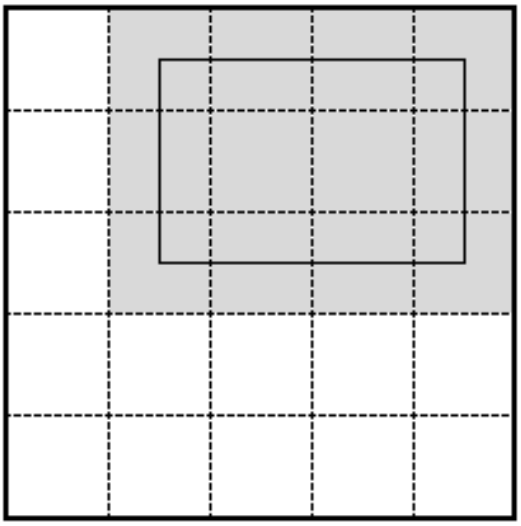


Figure 2. Section III-C: over-inclusion simplification

IV. Solution

A. Data retrieving

The dataset is built by leveraging a websocket connection to Twitter's stream APIs[6], allowing non-restricted real-time access to every sent tweet. The APIs are queried by a custom node.js script, in order to filter only those Tweets geolocated within the `map M`. The resulting data is then stripped of all unimportant information, leaving only each tweet's geolocation and Hashtag(s), User Mention(s), and Symbol(s)[7].

Twitter's stream APIs provide two kinds of geolocated tweets:

- automatic geolocatation through the device's GPS;
- manually specification of a geographic landmark.

The more common latter option is returned by the APIs as a bounded rectangle: in the custom script's implementation, the centre of the rectangle is taken as the tweet's pinpoint geolocation. Given how this rectangle is unbounded in size (e.g. it may specify "Italy", meaning its totality), only tweets whose location rectangle's side is smaller than 111 km are kept[8]: this filtering removes any and all tweets geolocated too

[6] https://dev.twitter.com/streaming/overview

[7] https://dev.twitter.com/overview/api/entities-in-twitter-objects

[8] This is equivalent to validating a geolocated tweet by verifying it has a minimum of 3 precision digits.

generally, and at the same time keeps those tweets located around large relevant areas (e.g. major cities).

Additionally, another smaller node.js script was written in order to generate fake tweets. This random generator was used during testing and development, in order to have a consistent and easily analysable set of data.

B. Framework

The development framework consists of Python 2.7, although Python 3 is supported as well. Some external dependencies are also required:

- colorclass, for color formatted console outputs;
- redis, to interface with the Redis key-value database;
- pyspark, to interface with Spark functions.

The pyspark module is injected globally within the Python framework when the program is invoked by the `spark-submit` command (shipped with Spark), meaning no installation is required.

To simulate a real distributed system, a number of virtual machines were spawned and managed using Docker[9]. Due to hardware limitations, each machine used only a single separate CPU core, and a 1 GB portion of RAM, limiting the count to 1 master and 3 workers. Note that since Docker effectively separates each instance into a self-enclosed (virtual) machine, an escalation to a real distributed system solution is only dependant on hardware availability. Creating a larger virtual distributed system is on the other hand more complex, due to mainly memory requirements and resources allocation; a number of projects exist to ease this expansion, e.g. Docker Swarm[10].

C. Redis

> *Redis is an open source (*BSD *licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries. Redis has built-in replication, Lua scripting,* LRU *eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.* [7]

Redis' in-memory paradigm allows for extremely fast and efficient performance, with direct key access close to O(1) speeds. [8]

The Redis store is implemented as a remote database, distributed across a separate docker machine. Since everything is stored on the same local Docker environment, there is no need to establish a secure tunnel between the main program (virtual) machine(s) and the database (virtual) machine. Note that any further expansions comprising separate hardware machines, especially in a non-LAN environment, would require expanding Redis' implementation to include secure connections.

[9] https://www.docker.com/

[10] https://github.com/docker/swarm

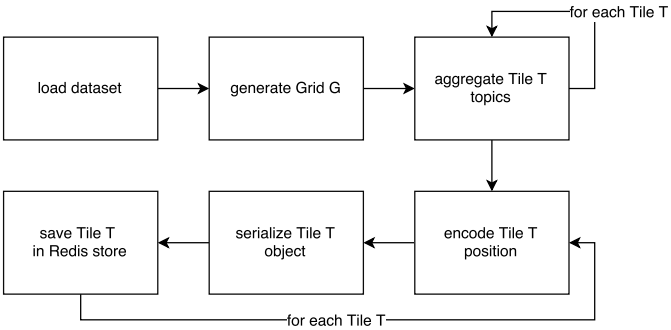


Figure 3. Section V-A: *bootstrap algorithm* flow

D. Spark

> *Apache Spark is a fast and general engine for large-scale data processing.* [9]

Spark is a software that, once installed on multiple (possibly virtual) machines, allows the parallelisation of a job's workload: more specifically a master machine automatically delegates the different jobs to the available worker machines. In this project's implementation, this creation and run process is automated by a custom `docker-compose` script: when ran it automatically instantiates all the machines, registering the three workers to the single master node. The program's code is then ran through a shell on the master machine.

PySpark is a specific Python implementation of Spark, that allows the leveraging of the map-reduce pattern in distributed systems solutions. This Spark Python API exposes two functions `reduceByKey()` and `map()`, that are automatically handled when the script is ran in a distributed environment.

E. Bootstrap algorithm

See Figure 3 for a visulization of the algorithm's flow.

- input
  - a set of geotagged *Tweets* (`data U`)
  - a rectangular area (`map M`)
  - a geographic distance (`step S`)
- output
  - a grid (`grid G`) of `S`x`S` squares (`tile T`)
  - for each `tile T` an enumeration of *topics*

This computation is ran only once during the program's bootup, and provides the `grid G` on which all other executions of the *selection algorithms* are ran.

The dataset is initially loaded in memory from a JSON file, either directly from the filesystem, through HDFS[11], or from a remote server. This dataset is serially pre-processed by MapReduce calls: all of the computation can thus be easily ran in parallel on multiple machines simply by connecting an unbounded number of Spark worker machines to the main master node.

An empty `grid G` that covers the `map M` is generated, formed by all the `tile T` that contain at least one tweet.

[11] `hdfs dfs -copyFromLocal <source> <dest>`

These empty `tile T` are then filled with the aggregate of all the *topics* of those Tweets geolocated within each `tile T`: more specifically, the enumeration consists of a set of pairs `(topic, occurrences)`.

Algorithm 1: *bootstrap algorithm*

```
Function Main(bl, tr, step):
    tiles = getTiles(bl, tr, step)
    tiles = tiles.map(t: getTopic(t))
    tiles.filter(Null)
    return tiles

Function getTiles(bl, tr, step):
    tiles = []
    tilesXs = []
    tilesYs = []
    coord = bl.x
    while coord < tr.x do
        tilesXs.append(coord)
        coord = coord+step
    coord = bl.y
    while coord < tr.y do
        tilesYs.append(coord)
        coord = coord+step
    foreach x in tilesXs do
        foreach y in tilesYs do
            tile = ((x, y), (min(tr.x, x+step), min(tr.y,
              y+step))
            tiles.append(tile)
    return tiles

Function getTopic(tile):
    tile = tile.filterData(tweets)
    tile.topics = tile.topics.map(t: (t.keyword, t.weight))
    tile.topics = tile.topics.reduceByKey(p, n: p+n)
    if tile.topics not Empty then
        tile.main = tile.topics.getMainTopic()
        return tile
    else
        return Null
```

Ultimately, the `tile T` are saved in a Redis store: the Redis key of each `tile T` represents its position, and is uniquely encoded in a specific `<bottom-left-x>X<bottom-left-y> String` format, allowing subsequent requests for that tile's data to be performed in Redis' O(1) speed (see Section IV-C). The `tile T` objects are serialized in the Redis store using Python's internal JSON serialization function. The serialization is necessary due to Redis' own limitations, and the `json.dumps()` function was found to be the faster among most other available serialization dumps. [10]

F. Selection algorithm

See Figure 4 for a visulization of the algorithm's flow.

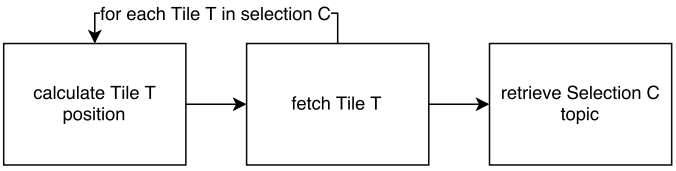


Figure 4. Section V-B: *selection algorithm* flow

- input
  - a rectangular area (`selection C`)
  - the `grid G`
- output
  - the main *topic* of `selection C`

The algorithm that retrieves the most relevant topic within the user-defined `selection C` is exposed by the `categorize` Python module. The function is invoked by passing the bottom-left and top-right vertices of the desired `selection C` rectangle area.

The function's main workload is fetching all and only those tiles that reside within the `selection C`, with any `tile T` only partially residing within it also being added (see Section III-C): this is done through basic geometric considerations, knowing the `tile T`'s size `S` and the `selection C` boundaries. Once a defined set of tiles is fetched, their data can be retrieved from the Redis store directly through the key, in an O(1) call (see Section IV-C). The main topic of the entire `selection C` is then extracted by a MapReduce count of all the main topics of all the fetched tiles.

Algorithm 2: *selection algorithm*

```
Function Main(bl, tr):
    area =
    area.coords = (bl, tr)
    i = bl.x - (bl.x % S)
    while i < tr.x do
        j = bl.y - (bl.y % S)
        while j < tr.y do
            key = createKey(i, j)
            value = Redis.get(key)
            if value not Null then
                area.data.append(value)
            j += S
        i += S
    area.topics = area.data.flatten(t: t.topics)
    area.topics = area.topics.reduceByKey(p, n: p+n)
    if area.topics not Empty then
        area.main = area.topics.getMainTopic()
        return area
    else
        return Null

Function createKey(x, y):
    return str(x)+"X"+str(y)
```

```
{
    duration: 0.1664748191833496,
  - main: [
        "andrea albanese",
        68
    ],
  - topics: [
    - [
        "andrea albanese",
        68
    ],
    - [
        "smmdayit",
        62
    ],
    - [
        "50twittertips",
        46
    ],
    - [
        "travel",
        40
    ],
    - [
        "reviews",
        33
    ],
    - [
        "uominiedonne",
        31
    ],
    - [
        "iusve - università",
        24
    ],
    - [
        "verona",
        22
    ],
    - [
        "iusve",
        19
    ],
```


Figure 5. Section IV-G: sample JSON output

G. User interaction

A single API endpoint was developed[12], allowing users to input the requested area `selection C` without requiring a direct access to Spark's console. A JSON object is returned with the `selection C`'s main topic, and additional meta information. Any further expansion may thus leverage this object to build a more complex GUI. See Figure 5 for an example: in this case, the *bootstrap algorithm* was initially ran on a set of all tweets geolocated within "Italy", in a 24 h time period, with `tile T` of size 1°; the user request (i.e. the *selection algorithm*) was ran with "Trentino Alto-Adige"'s geographic boundaries (45°N 10°E to 47°N 12°E).

Note that although the user can only select a rectangle area, different more complex shapes may be built by superimposing and/or aggregating rectangles of different sizes and in different positions. See Figure 6 for an example of a circular-like shape made of multiple `selection C`.

[12] e.g. http://10.0.75.1:5000/?x0=10&y0=45&x1=12&y1=47

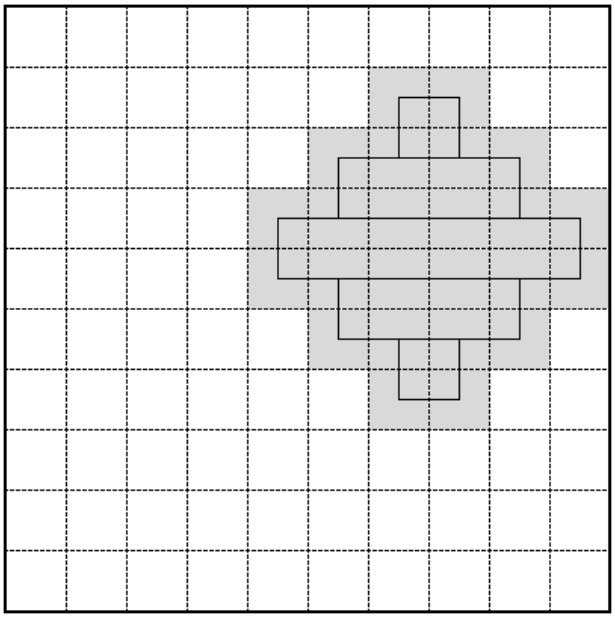


Figure 6. Section IV-G: complex `selection C` shape

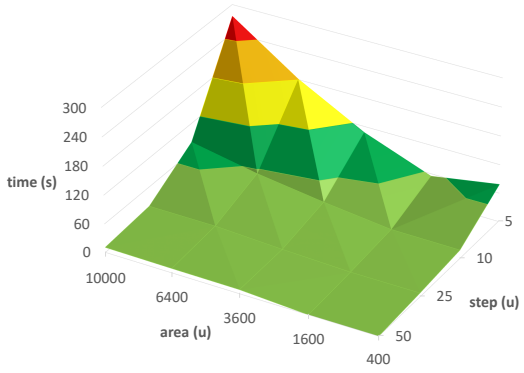


Figure 7. Section V-A: *bootstrap algorithm* tests results

V. Testing

All testing was done on `data U` consisting of 100 000 randomly generated fake tweets.

A. Bootstrap algorithm

See Figure 7. This algorithm becomes increasingly slower as the `map M` increases in size, and the `tile T` decreases in size: the main factor affecting the algorithm's speed is thus the total number of `tile T` that ultimately form the `grid G`. Since this same amount is also directly related to the accuracy

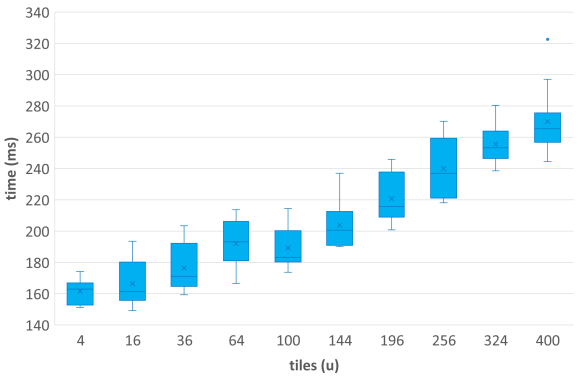


Figure 8. Section V-B: *selection algorithm* tests results

of the results returned by the *selection algorithm*, a trade-off must be found between initial bootup time, and accurate final results.

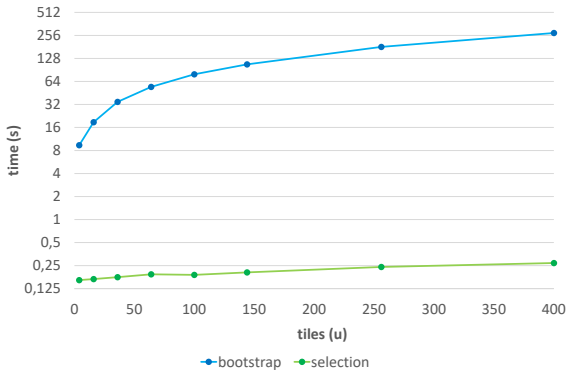
This algorithm's specific bottleneck is the function that filters the tweets present in each of the `grid G`'s `tile T`.

B. Selection algorithm

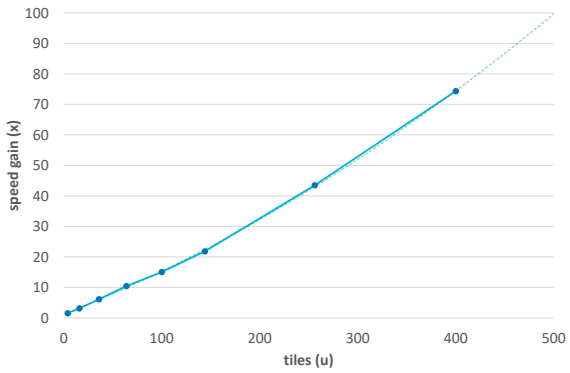
See Figure 8. This algorithm becomes slower as the requested `selection C` increases in size (or more specifically, the amount of `tile T` of the `grid G` within the `selection C` increases). Note that the total amount of `tile T` that form the entire `grid G` is notably non-relevant, since only the `tile T` within the `selection C` are analyzed by the algorithm, and the Redis store allows for `O(1)` access speeds.

C. Comparison

See Figure 9 (note that the time scale in chart Figure 9(a) is logarithmic). As expected, the *bootstrap algorithm* is noticeably slower than the *selection algorithm*. This is per specs, since the bootup calculation is done only once (by e.g. a server), whilst the algorithm exposed to the end-user through the APIs is as responsive as possible. The second algorithm becomes more and more efficient in comparison to the first as the amount of `tile T` involved increases, up to 74 x times faster with 400 `tile T`. A polynomial (second order) fitting of the resulting efficiency curve also shows a rapidly increasing effect, reaching a 100 x faster speed with 500 `tile T`.



(a)



(b)

Figure 9. Section V-C: algorithms comparisons tests results

VI. Conclusion

This paper presented an efficient, scalable, and versatile pair of cascaded algorithms that allow the retrieval, analysis, and visualization of a geographic area's trending topics, based on each and every tweets geolocated within that area. A slower, more resource intensive first booting algorithm directly allows user queries to be both immediate (less than 500 m sec) and accurate; in addition, no limits are placed on the size, shape, or position of the requested geographic area.

Additionally, the presented solution may be expanded in multiple directions:

- the code may be expanded in order to implement more functionalities;
- performance may be increased by switching from a small number of virtual machines to swarms of distributed hardware;
- the available APIs may be leveraged to build a working GUI;
- the Docker framework allows for immediate portability on different machines.

REFERENCES

[1] Data, data everywhere, *The Economist, Feb 25th 2010*

[2] Big Data for Development: A Review of Promises and Challenges, *Martin Hilbert, University of California, California*

[3] Twitter Data Clustering and Visualization, *Vrije Universiteit Brussel, Brussels, Belgium*

[4] Geo-spatial Event Detection in the Twitter Stream, AGT *International, Berlin, Germany*

[5] TwitInfo: Aggregating and Visualizing Microblogs for Event Exploration, MIT CSAIL, *Cambridge, Massachussetts*

[6] Measuring accuracy of latitude and longitude?, *StackOverflow*

[7] REDIS, *Redis.io*

[8] How does redis claim O(1) time for key lookup?, *StackOverflow*

[9] Spark, *Apache Spark*

[10] Pickle vs. JSON, *Konstantin Blog*