Universitat Rovira i Virgili

CloudButton challenge

Distributed Systems

- Gil Arasa Verge
- Jordi Bayarri Planas
- Enrique Martínez Martínez

13-6-2021

Contents

INTRODUCTION

This project encompasses all the theory taught during the course and is a good opportunity to challenge our abilities and learn.

In this document we explain the most interesting parts of the project, including the architecture, organization, and the decisions that we had to go through. We used one Jupyter Notebook for each part of the task to clearly separate the different stages.

Thanks to lithops we got the following advantages of indirect communication:

Scaling transparency is provided by the Lithops partitioner when we parametrize the size of each chunk in order to automatically decide how many nodes in the system we need to process in parallel the load and in such a way that map functions won't process the same data twice separately (concurrency transparency).

Location transparency is directly provided by IBM Cloud Functions because we do not specify in which nodes to run the computation and we only give it instructions with Lithops to a single endpoint (API). We do not have to care about provisioning the underlying system in any way.

Fault tolerance is nonexistent because we must have the Lithops client running on the thin client all the time during the compute (losing connectivity from the client side will cancel everything).

STAGE 1. SCRAPING

Data crawler. Massively parallel functions crawling data and storing it in Cloud Object Storage. Obtain information from web pages or tweets and create a dataset of text data. Use FaaS backend in lithops to launch crawling process over serverless functions.

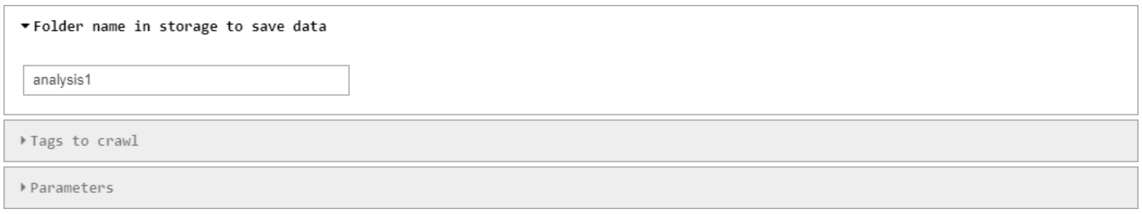
In the first stage we had tried a lot of methods and libraries (tweepy, twint) to crawl tweets but most of them had some limitations (rate limits even using API keys) or required a manual work (scrapy). Because the purpose of open source software is mainly not reinventing the wheel and not maintaining all the code by ourselves, we decided to adopt a Twitter scrapper found on Github:

https://github.com/JustAnotherArchivist/snscrape

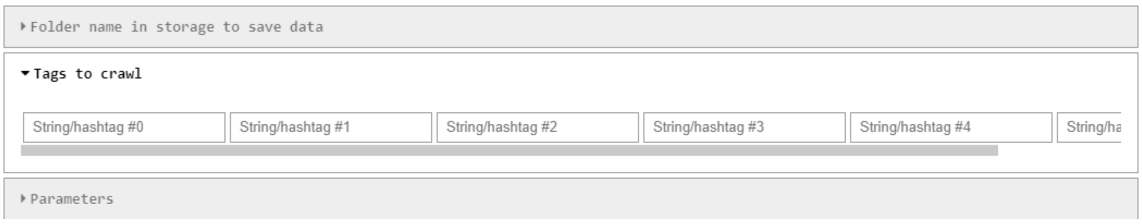
This code is inside "custom_snscrape" folder, customized to be imported as a library with just the Twitter scraper.

We used Jupyter Notebook widgets to write the hashtags or strings that we want to crawl easily and interactively.

The first widget is used to specify a parent name to the crawl we are going to run. This name is used to save the data in the cloud object storage as a folder name to make it easily accessible later or to add more hashtags in the future. Basically, this allows to preserve the data on different projects and to modify them or expand them while working on other projects in parallel.

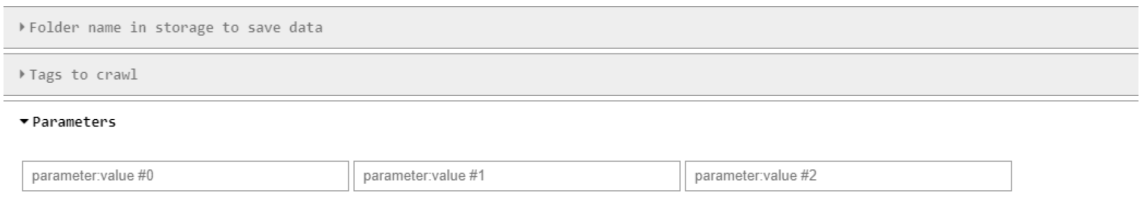


The second area of widgets are the hashtags. Note that there are only 6 hashtags widgets, but it is possible to add more easily modifying its cell code.



The last area of widgets is used to specify search parameters to the query. The format of query parameters is: *parameter:value* (ex. *near:Tarragona*, *until:2012-12-31*).

Any parameters from the Twitter's Advanced Search can be used (twitter.com/search-advanced).


▸Folder name in storage to save data

▸Tags to crawl

▾Parameters

To avoid repeated Tweets while running parallel crawl functions, we have used a query transformation with a special format. For example, if we want to search the hashtags #covid, #sars and #coronavirus, the final queries will be:

“#covid -#sars -#coronavirus, #sars -#coronavirus, #coronavirus”

The above statement only applies to “Tags to crawl” or the hashtag section in the accordion. This is due Twitter exposing a Cursor (a parameter from the previous query is required to be sent to continue search with the same query) in their search API, which by nature is better for them to scale but is worse for scrapers because it makes search sequential for the used terms which implies that we are not able to parallelize the search for each term without getting the same results or to avoid checking for duplicates. However, we thought of that method so that different terms or hashtags will be scraped in parallel by different cloud functions, converting the problem from sequential to parallelizable taking the most data as fast as possible.

There’s a way that the API can be used to in completely parallel even if it’s using cursors for the results. The advanced search terms for date specification can be distributed over the time-lapse required so that each single FaaS function for example scraps a span of 5 unique days in a year which implies that 73 functions would be scraping in parallel over a single year. This also would give us data uniformly distributed in time. More about this in one of the paragraphs below.

We used the map function of the Lithops function executor to run in parallel all the queries. The results are stored on the cloud object storage with the following format (same example as before without introducing intervals dates):

analysis1/#coronavirus

analysis1/#sars

analysis1/#covid

If you decide to introduce interval dates:

analysis1/#coronavirus since:date1 until:date2

analysis1/#coronavirus since:date2 until:date3

analysis1/#coronavirus since:date3 until:date4

…

To increase the parallelism and be able to get a big amount of data, we added an algorithm that modify the list of queries separating the search between certain dates. We can modify the initial date; the number of intervals and the time of each interval will cover. The best way of understanding it is looking for an example:

We want to search for the hashtag #covid. If we select 1-1-2021 as initial day, an interval of 15 days and 4 numbers of intervals the final query would be:

['#covid since:2021-01-01 until:2021-01-16',

'#covid since:2021-01-16 until:2021-01-31',

'#covid since:2021-01-31 until:2021-02-15',

'#covid since:2021-02-15 until:2021-03-02']

Using all the query modifications together we can throw a big number of functions parallelly to the cloud and get big amounts of data, basically adapting a sequential problem to be a parallel one as a perfect use case to run in a FaaS.

Tweets are saved as a string and separated by newlines inside each hashtag object.

We limit the number of tweets that the crawl function processes to know how many we will end up scraping and to avoid long lived functions (when they finish, they're done, and we don't want them being killed by timeout or filling all the space in the Cloud Object Storage).

Number of functions: 292

Max tweets per function (if the date range has that number of tweets): 300

Total scraping time: 2 minutes

Scraped size: 233 MiB

Time per function: 7 seconds average.

STAGE 2. PREPROCESSING

Data preprocessing stage to produce structured data in csv format also stored in Cloud Object Storage.

Here we read stage 1 saved data in the folder with the project's name using the Lithops partitioner.

analysis1/#coronavirus

analysis1/#sars

analysis1/#covid

The partitioner will divide in chunks each of the hashtags in the analyzed folder al generate a map function for each. After all map functions are completed, a reduce function will join all the results and append them into an output file saved in the Cloud Object Storage at the root of the bucket with the project's name.

The partitioner is configured to divide each hashtag's tweets in chunks of 1 MiB.

Increasing this amount will make the processing take longer and may trigger function timeouts if the limits are not raised. However, we also do not want limits to be triggered because these functions are supposed to be short or fast enough.

Reducing the chunk size would increase the total processing speed, parallelizing more, meaning results would be available faster but at the cost of an increase of total resource consumption.

Diving in the processing functions:

The map function first tries to guess the language of the tweet's content using a small and fast model for language identification.

Using the guessed language and the one provided by twitter all tweets with non-supported languages for the multilingual sentiment analysis neural network are discarded.

We were compelled to do this because sometimes twitter mislabeled some Catalan tweets as other languages. Our native sentiment analysis doesn't quite support Catalan as well as other languages (English, Dutch, German, French, Spanish and Italian) but because it is a Romance language and three of the multilingual model languages are Romance languages we consider it may be able to extrapolate it's knowledge enough. We could also have fine-tuned the model, but it was out of the scope of the challenge.

Language	**Accuracy (exact)**	**Accuracy (off-by-1)**
English	67%	95%
Dutch	57%	93%
German	61%	94%
French	59%	94%
Italian	59%	95%
Spanish	58%	95%

Source: huggingface.co/nlptown/bert-base-multilingual-uncased-sentiment

Total stage 2 time: 395.43 seconds which is approximately 6 and a half minutes.

Number of functions: 1460 map functions and a single reduce function.

Output file size: 21 MiB, 83675 tweets.

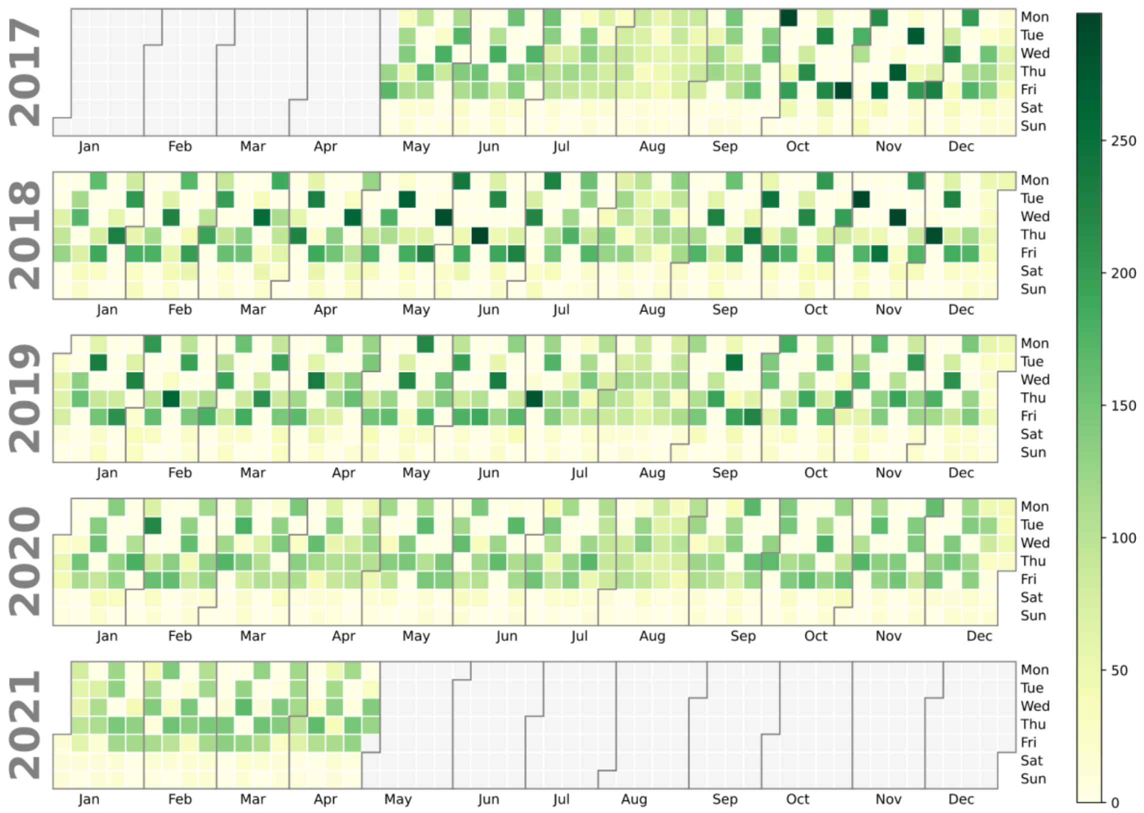
STAGE 3. ANALYSIS

Demonstrate simple queries in Python notebooks over your data. In particular, apply sentiment analysis to data in different dates and periods and generate simple plots. Enable search queries over the data.

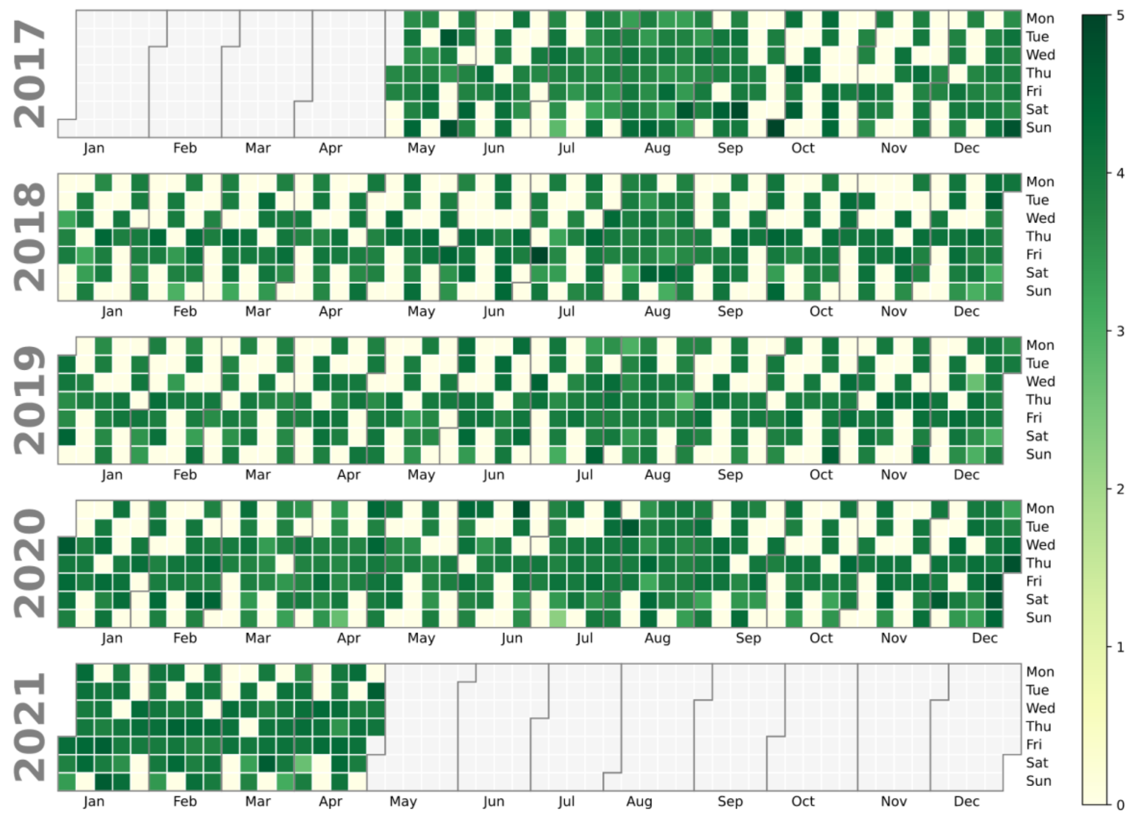
In the latest stage we read the pre-processed file from Cloud Object Storage into a Pandas DataFrame. The plots can be viewed within HTML files in the `notebooks/results` folder, and some of them are interactive.

Using the calplot library, we generate two kinds of calendar heatmaps: one with the number of tweets sent and another with the sentiment over time with the #H2020 hashtag.

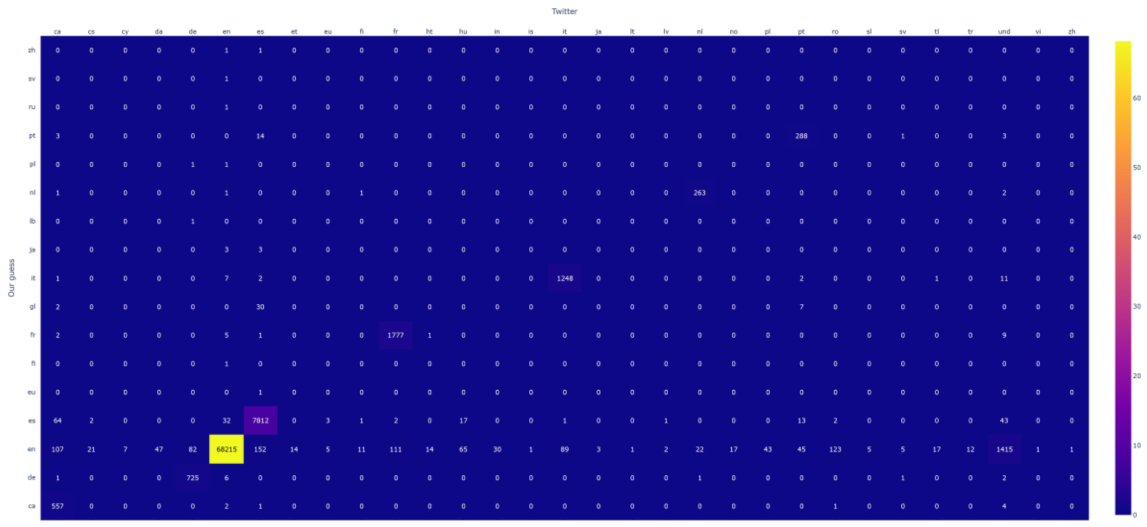
Number of tweets (the scores represent the number of tweets sent that day)



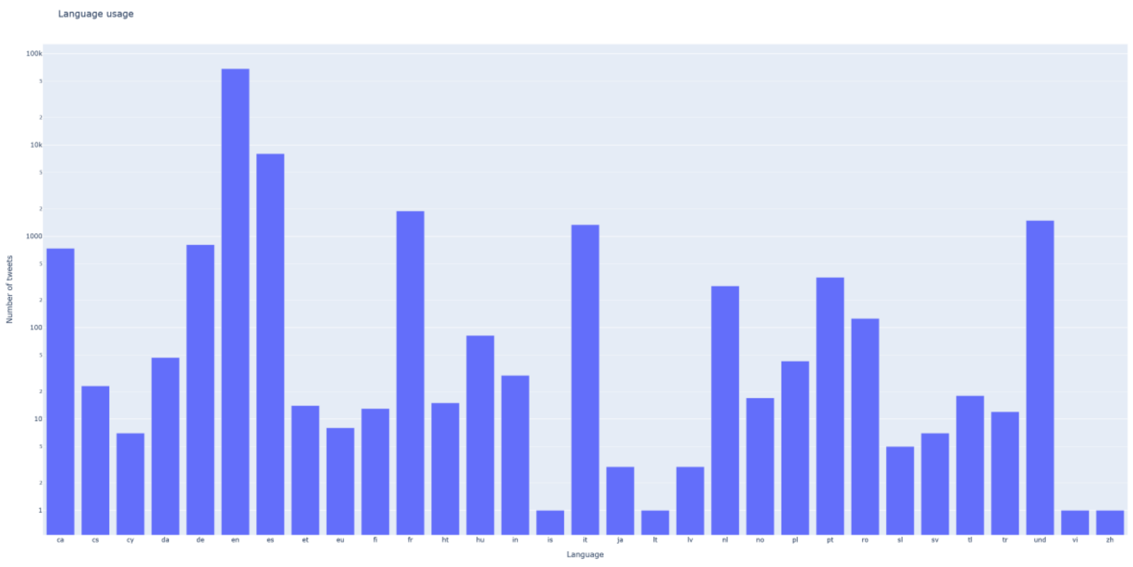
Sentiment (higher scores represent a more positive sentiment)



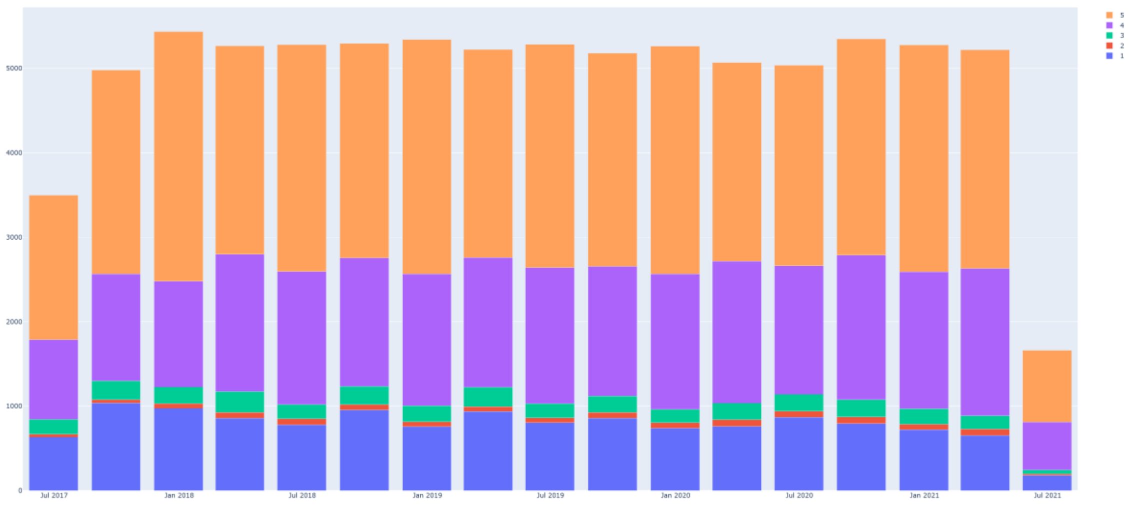
As we said earlier, some tweets aren't labeled with the actual language they're written in, so we also try to guess the language. The following plot shows that most tweets are not mislabeled. We have also been able to guess the language of some tweets that Twitter couldn't label (shown as 'und'). This is an interactive plot, and it can be found in the LanguageGuessing.html file.



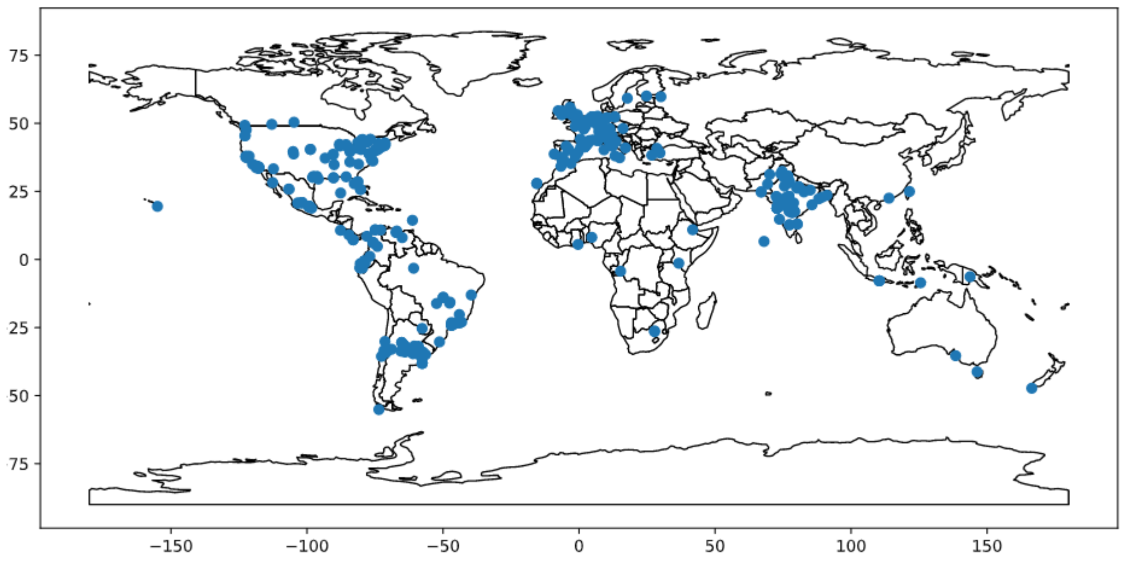
The next plot shows the language usage in the tweets (using Twitter's data). English is the most used language, followed by Spanish, French and Italian. Tweets classified as 'und' can be assumed to be mostly written in English, according to the language guessing model that we used.



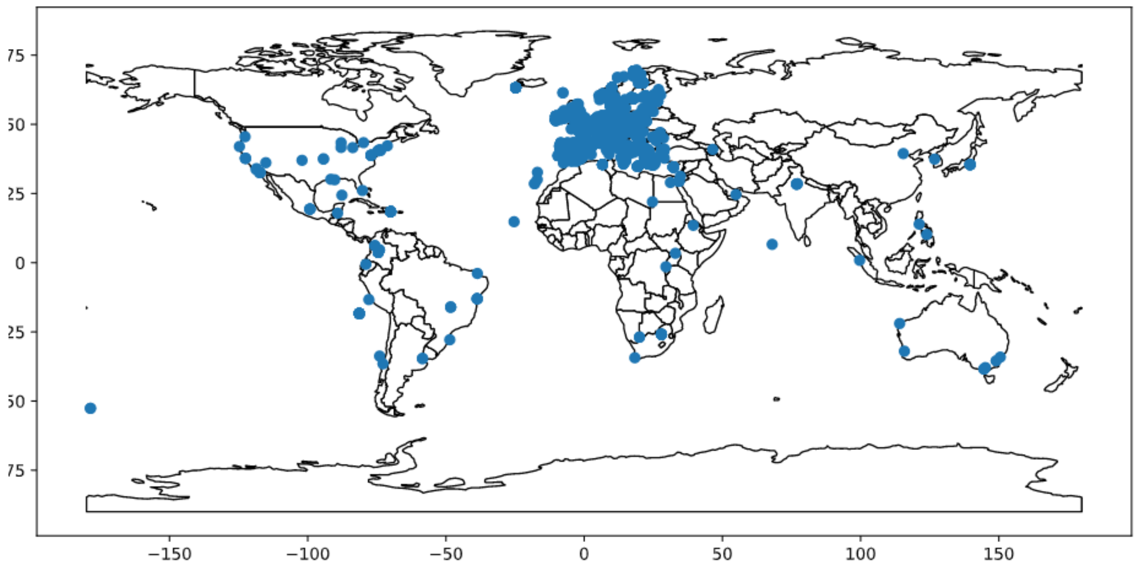
We have also used a stacked bar chart to show the sentiment, which is overwhelmingly positive as we saw earlier in the calendar heatmap. #H2020 is a very positive hashtag.



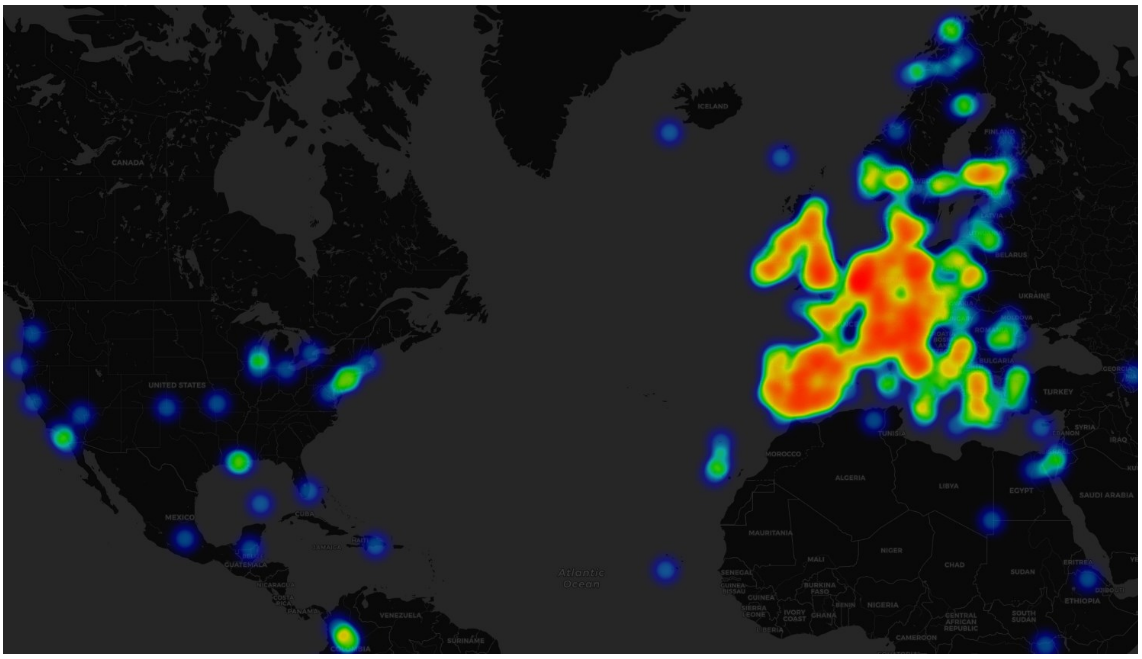
We have also plotted the locations of the tweets that we have crawled. We made 3 plots using these locations.

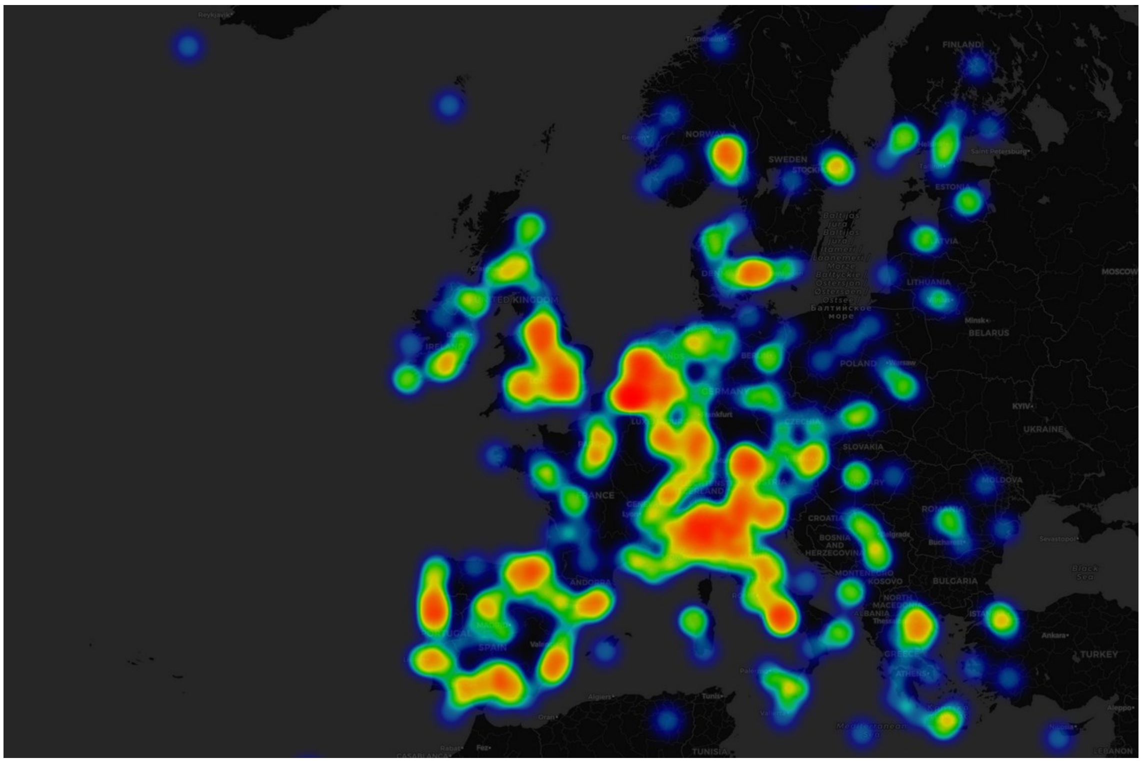


^ #coronavirus

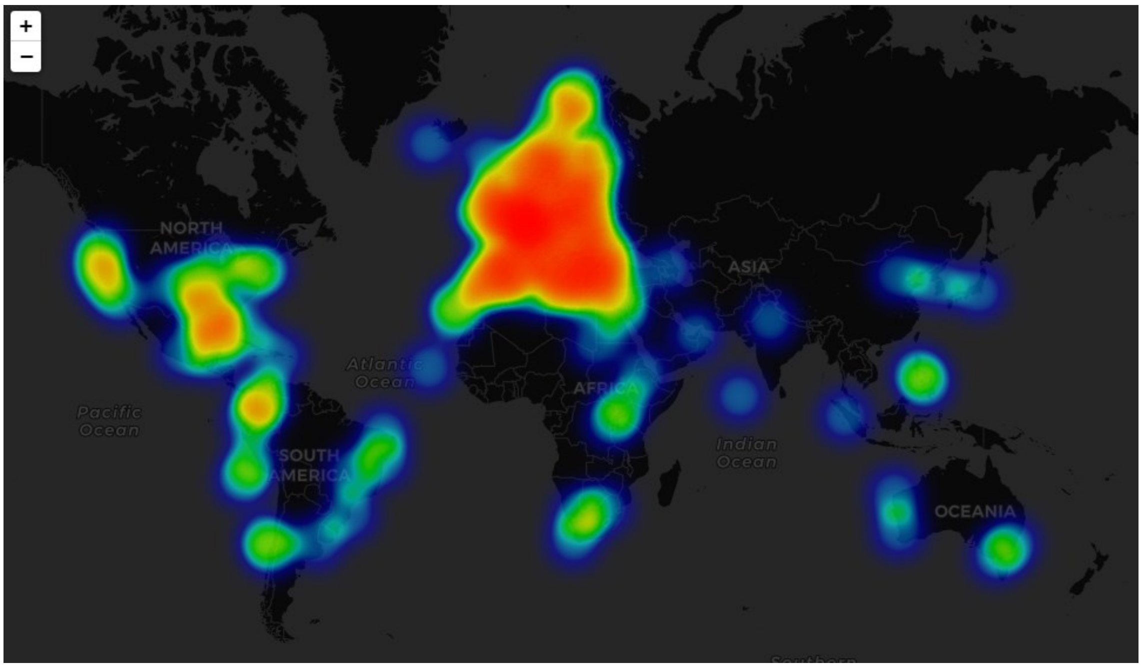


^ #H2020

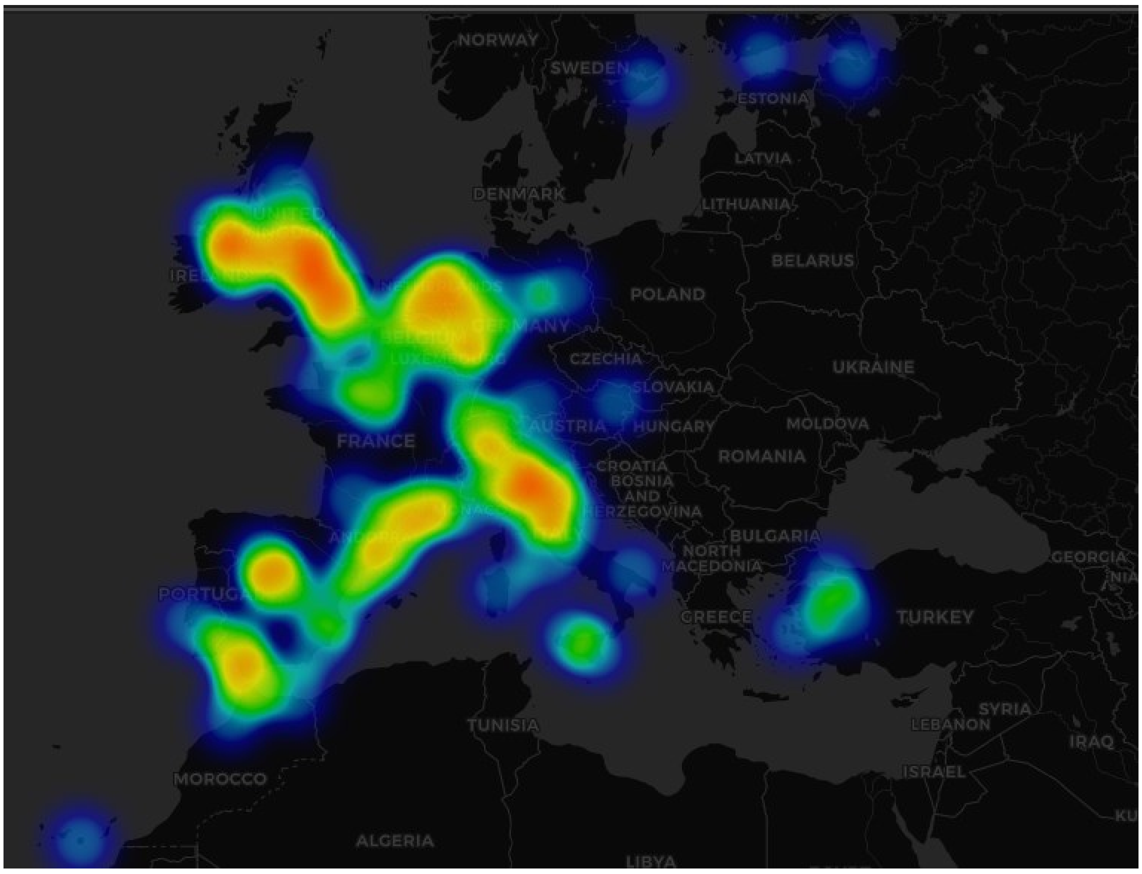

^ #H2020, number of tweets



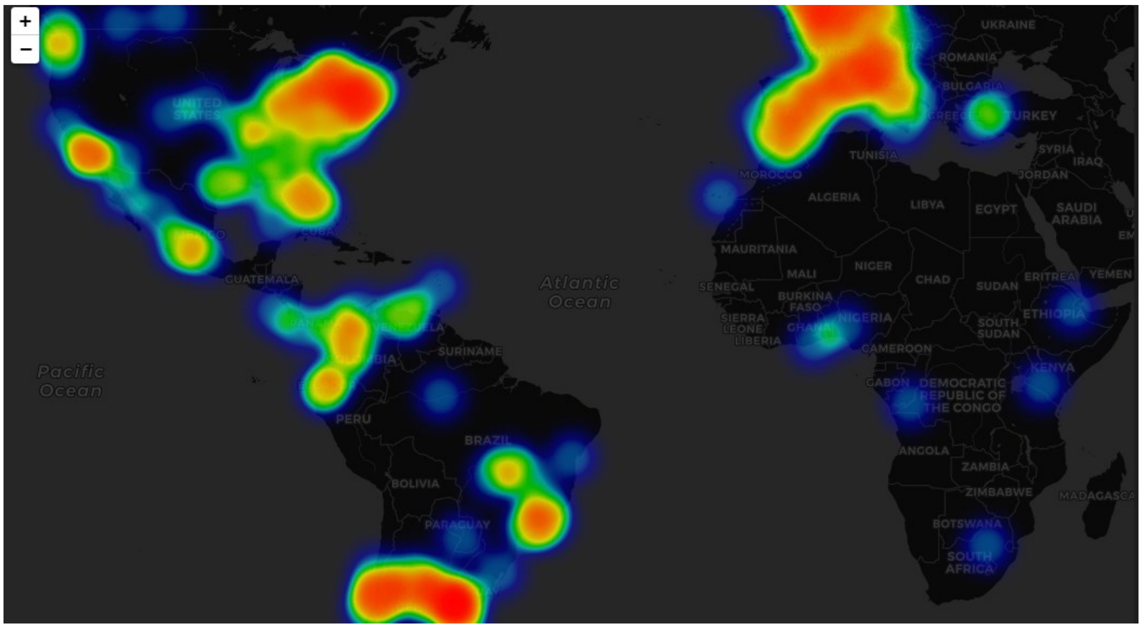
^ #H2020, number of tweets



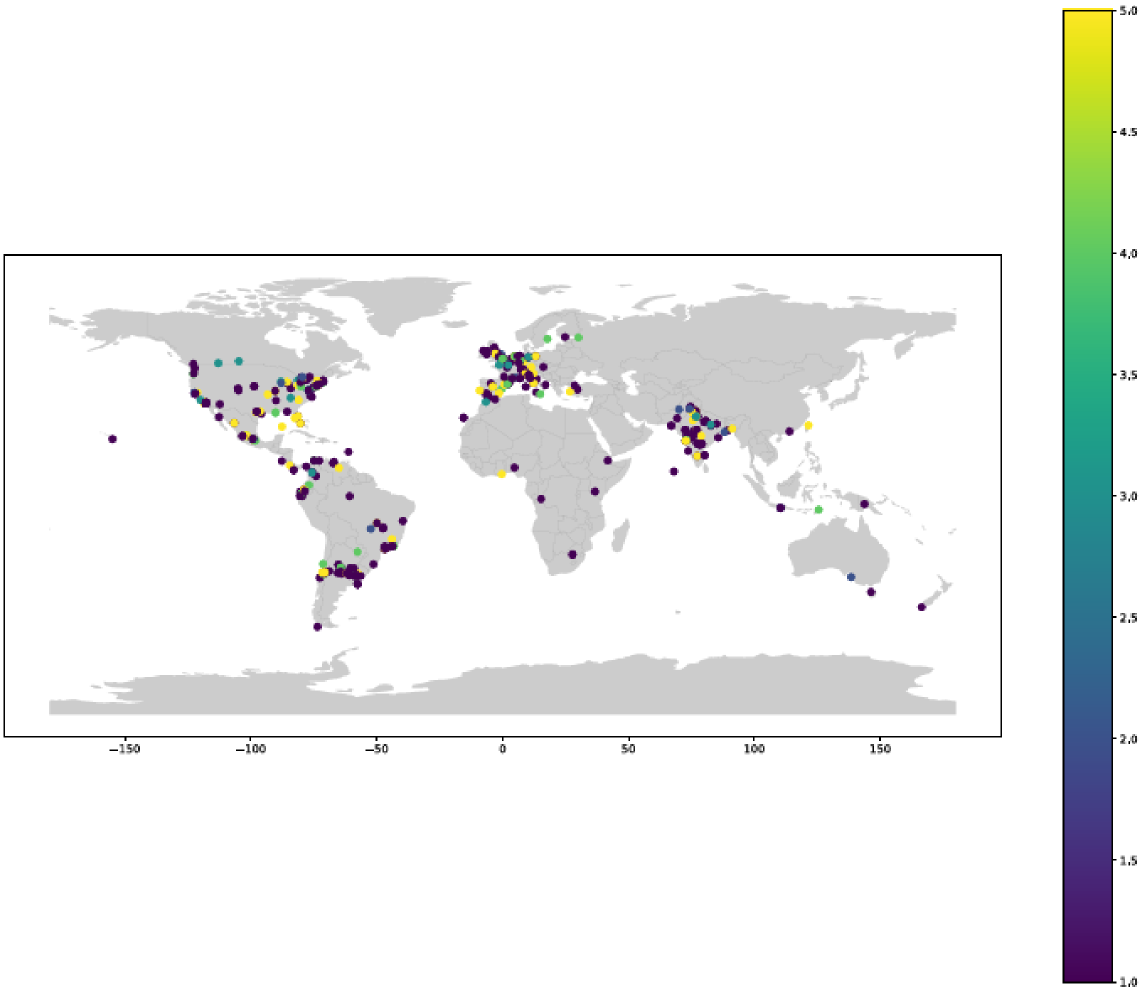
^ #coronavirus, number of tweets



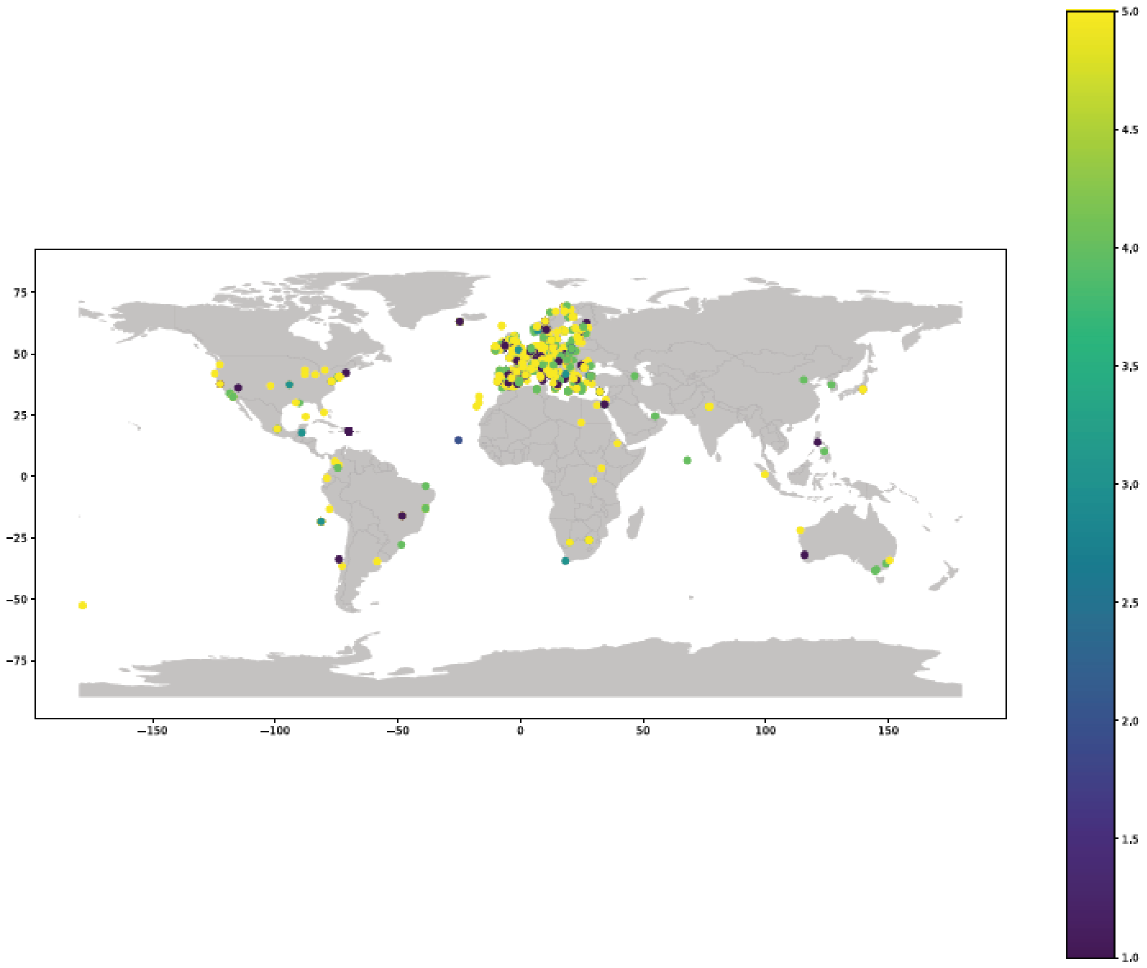
^ #coronavirus, number of tweets



^ #coronavirus, number of tweets



^ #coronavirus, mostly negative sentiment



^ #H2020, mostly positive sentiment

CONCLUSION

Certainly, the most difficult part has been having Lithops scale a medium sized deep learning model in such a way that timeouts didn't fire (we have been able to pull through thanks to tweaking the partitioner) although we have noticed that Lithops also supports VMs in the Virtual Private Cloud to do this type of expensive computations or long running jobs. The requirement of consistent client connectivity during the compute is also a limiting factor because a thin client may not be able to ensure connection to the network by some reason and thus the long running jobs are expected to never end correctly (this happened to us more than four or five times when using thousands of functions).

We are extremely grateful to our teachers, organizers and all the people involved in this project. We would also want to highlight the important role of Lithops during this project since we understood the workflow and explored all the functionalities it has become an essential tool for our understanding of cloud computing.