URL Categorization

Project for "Big Data and Social Networks", "Data mining" courses

Dmytro Koval
University of Trento
Mat. 188892
dmytro.koval@studenti.unitn.it

Ivan Chernukha
University of Trento
Mat. 188890
ivan.chernukha@studenti.unitn.it

ABSTRACT

This work presents experimental study of large-scale topics extraction from web pages visited by mobile users from certain geographical locations. We propose novel approach for detecting topics on the large area of the map using big data and data mining techniques. In particular, the implementation [1]is done using Apache Spark and MongoDB.

CCS Concepts

•**Information systems** → **Data mining;** *Document topic models;*

Keywords

Big data, Topic detection, Apache Spark, url categorization

1. INTRODUCTION

According to IBM, there was 2.5 quintillion bytes of data created daily (it used to be in 2015, now this figure is even bigger), and if a company, organization or government wants to harness the power of information - they should have the means to process that data.

One example of that sort of information is knowledge of what topics people are interested in different geographical regions. This is particularly what helped Barack Obama to be re-elected for his second presidential term in 2012. One part of his success was the way that his analytical team used Big Data in order to analyze what are the biggest concerns of people in different geographical regions. This particularly helped to target the efforts of persuading people to vote in swing states and communities - by knowing from web search history, Facebook and twitter trends what is important for people in the given area and what arguments will work better to persuade them.

That is a very smart example of targeted marketing. In the modern advertising data is everything, thanks to big data our ads have became much more personalized and this allows to show ads to the very ”right” audience, at the very ”right” time, thus saving the advertising money and maximizing the return on investment.

Such a technique of discovering the topics trending in geographical areas can also be used for analyzing the potential or existing market when taking decisions in companies, for marketing campaigns etc. As an example, one could use this method to figure out which hashtags are trending in Twitter and Instagram in certain areas at a given moment and tailor the marketing campaign in a way to leverage the viral nature of certain topics and hashtags to effectively promote brands, products or services.

For our project we've had a set of URLs accessed from mobile devices at the given GPS locations. In other words, as input data we knew the URLs visited from smartphones all over the Italy and few other European and African countries with the geographical coordinate record for every URL. By downloading the webpages those URL point to and analyzing their textual content we could figure out what is the topic of the page,

Sounds promising, but this includes some challenges though. The example dataset we've had for this project included 2,5 millions geopoints, each of those could have 1 to N URLs accessed from that spot and N tended to be in dozens sometimes. So if one would want to fetch all of those URLs with one webpage being loaded 5 to 25 seconds with the modern internet speeds it could take 10(mean time per URL in sec) * 8 (mean URLs per geopoint) * 2 500 000 geopoints = 200 millions seconds = 2314,8 days just to download that webpages if you do it one by one. In the end you'd find out that 60% of those URLs are media files (images, css files), other 39% are API requests that are either expired or require additional security tokens, so they mostly return ”access forbidden”, another 0.365% are webpages that don't exist anymore and only 0.635% left are actual data. Among example 408000 URLs only 2591 actually appeared to be real HTML pages with meaningful content, so this is the first challenge of filtering the data to suit our needs.

Another challenge was in general the amount of data appearing in this kind of problem - normally because the more data you have about what web pages people visited from their mobile devices, the better you can understand trends in given areas, and in our case as described in previous filtering challenge we've had 2 500 000 geopoints in our example dataset that takes weeks/months to process with

[1]https://github.com/merryHunter/url-topic-mining

ACM ISBN .
DOI: 10.1145/1235

a normal computer that is relatively powerful.

We've tackled that challenge in two ways:

1. Improved data mining algorithm that used caching instead of pre-computing
2. Parallelizing some computations with Apache Spark in order to allow running computations in cluster of many computers to speed up the process when time is more scarce resource than money needed to establish or rent a computer cluster.

One more challenge is transformation of geographical coordinates required to solve this kind of problem. Because the Earth is a sphere and not just plain, it is problematic to split the given area into squares of size SxS, because of the peculiarities of a used geographical coordinates model. For example, 1 degree of longitude at the equator = 111 km, at 45° of latitude - 78 km and at 75° - only 28.9 km. Because of this we need to use a special function to calculate other coordinates

2. RELATED WORK

After thorough search we discovered that, as a matter of fact, in the scientific world there was no one who did the research about exactly running topic detection algorithm on massive sets of URLs pointing to web pages associated with geocoordinates and determining the topics of interest in geographical area using this data.

But there were many researches that tackled the problem of analyzing big sets of web pages for different purposes, these include Topic Initiator Detection on the World Wide Web by X. Jin [5] that tackles the search of Topic Initiator among the web pages that discuss certain topic. Another examples include Topic Detection and Tracking for Chinese News Web Pages by Jing Qiu [9] that deals with mining online news to keep track of successive development of specific event.

Then, what is especially relevant to our problem - there are recent researches that study the use of social media in order to research people's areas of interest, potential problems and the attitude to specific ideas or events that are being discussed in society.

For example, UK governmental Department for Work & Pensions published a paper The Use of Social Media for Research and Analysis: A Feasibility Study [4] where they describe the use of Google Trends, Google Search, Twitter and Facebook, Wikipedia to research the public opinion and awareness on policies developed by government. They claim: "Firstly, these media can provide indications of information seeking behaviour (which may indicate public awareness of and attention to specific policies, as well as providing an idea of the sources where they get information from). Secondly, they can provide indications of public opinion of specific policies, or reaction to specific media events.". Another relevant paper is "Forecasting Significant Societal Events Using The Embers Streaming Predictive Analytics System" led by A. Doyle [2] describes the development of "a large-scale big data analytics system for forecasting significant societal events, such as civil unrest events on the basis of continuous, automated analysis of large volumes of publicly available data". Their system is the result of long and thorough work of many researchers and programmers and represent a really frontier example of the use of Big Data to tackle this class of problems, including many independent components strung together in a pipes-and-filters architecture and machine learning models for processing the data. "The system processes a range of data, from high-volume, high-velocity, noisy open-source media such as Twitter to lower-volume, higher-quality sources, such as economic indicators." - A. Doyle

Another widespread way to achieve similar goal as our project has is to use Google Trends, Google Search and Google Keyword Tool to examine the important topics raised in certain areas. For example, in Analysis of the Capacity of Google Trends to Measure Interest in Conservation Topics and the Role of Online News [8] paper Le T. P. Nghiem investigates how the public interest to a set of topics like climate change, ecosystem services, endangered species etc. evolved over time.

In the industry, similar analytics is used for researching the potentially profitable niche market [6] and augment a whole set of marketing efforts. [1] Interestingly, Jim Messina, who used to work as Barack Obama campaign manager and stood behind big data analytics for that campaign, is now running consulting business in his company The Messina Group [2], that according to their website, provides data analysis services, "Advising organizations and political campaigns who want to build movements, services, and products that succeed by being data-driven, digitally savvy, and grassroots focused". So as a matter of fact we don't have the opportunity to compare our algorithm to someone else's because we couldn't find any research article describing how to solve exactly the same problem that we have, but the cited earlier papers on similar topics confirm the importance of the research in a given area.

Another interesting example that we have had inspiration from is the service Trendsmap [3] that provides Realtime Local Twitter Trends interactive map - a very similar idea to the one in our project, but based on Twitter posts rather than web pages.

3. PROBLEM STATEMENT

Suppose we are given a dataset U of URLs with records in the following format: latitude *lat*, longitude *lon*, list of URL *urls*, where each URL u was visited by a user from that geographical location represented by a pair (lat, lon). Let's name each record a geoentry g and each pair of latitude and longitude as *location*. Having an area A of the world map defined by a top left and bottom right *locations*, we would like to create a grid on the map with a certain step S and identify major topics T of geoentries G located in each of the square q of the grid Q. T is presented by a set of keywords w. Basically, if we can fetch web pages those URLs are pointing to, the problem can be solved in two steps: (i) identifying topic of each web page in the square (ii) computing top-k topics of the square based on previous step.

What may sounds simple, is not simple at all. Very important problem here is *how* to locate geoentries inside each square? Considering we might have millions of records, simple rectangle comparison is not appropriate. Assuming we managed that, then the first step can be accomplished by using TF-IDF or LDA [3] (we will use the last one).

[2]http://themessinagroup.com

[3]http://trendsmap.com

What concerns the second step, we would like to re-use computations in order to avoid rerunning of the algorithm. In other words, having computed topics T_{comp} of web pages, we need to wisely choose $T_{top-k} \subset T_{comp}$.

4. SOLUTION

As it was described in problem statement part, for an input in this problem we have a huge set U of pairs <URLs, Location> that we will use to download corresponding web pages and examine their text to determine the topics of importance in a given area.

4.1 Data ingestion

The dataset was stored in a set of files in this format: $< lat, lon, [url_1, url_2...] >$. In order to be able to work with this data, we needed to import this into MongoDB as a collection containing three fields. At first we encountered problems with parsing URLs which were wrapped by curly braces, contained quotation and separator(commas) symbols. We've solved this way using *grep, sed* UNIX commands, afterwards run *mongoimport* command for importing.

4.2 URL filtering

When we started to fetch the pages by given URLs, we found out that many URLs lead to *.json files with no useful textual content, many are images, *.css and *.js files. Some of the URLs are outdated and no longer accessible, and if you try to download them all one by one it takes too long to find out there was only 0.635% of useful html web pages in there.

At first we did a simple check for a file extension at the end of the URL and refused to load URLs that end with ".jpg", ".png", ".gif", ".json", and ".xml" that are not web pages and do not have any useful meaning to help determine the topics of interest for a given area. This alone allowed us to reduce the number of requests by

Then for all the other URLs left instead of fetching them one by one we used Async Http Client Java library [4] to run the requests in parallel and thus not waste too much time on waiting until the server responds. Additionally, we decided to make HEAD requests to URLs first because many of the web resources despite having no file extension in the end might still appear to be images, videos, JSON/XML files, virtually anything. So instead of trying to download resources that potentially are not html web pages we are interested in, we first made faster HEAD requests with timeout of 5 seconds not to waste time on not available anymore resources and checked if MIME type of the resource returned is *"text/html"* and only then ran another GET request to fetch the page. All this optimization allowed to reduce the time needed to download the pages from the dataset dramatically.

This shows how much data filtering is important in Data Mining and Big Data projects, without proper data filtering we could have lost a lot of computational time downloading useless resources.

Next thing we have done is running those asynchronous requests in a cycle along all of our dataset of 20 000 000 URLs and for those URLs that appeared proper html pages we wrote the resulting downloaded webpage into a file whose name was the computeHashCode(String URL) if the length of its text was more than 140 symbols after trimming HTML tags and keeping only useful text (there were still plenty of outdated requests that returned "Access denied" so the limit of 140 chars allowed to keep meaningful pages). At this point of precomputing the entire initial dataset was filtered and became just 0.6535% of initial size (this percentage was the case for an example set of 408 000 URLs). The pages were downloaded and were ready to run the topic detection algorithm over them.

Possible improvement: for a Big Data part of the project we could have leveraged the power of cluster to run even more asynchronous requests in parallel but we'd then need the mechanism for effective storing the resulting downloaded web pages among the cluster. This could have been either HDFS shared between Spark slaves and master or the MongoDB with replication and sharding among Spark slave machines. In our project we tried to tune MongoDB for that purpose but it tended to use more time than we had available to set it all up, so we postponed this improvement until submitting the project for evaluation.

Another general optimization idea for the future could be to assign a timestamp to a downloaded webpage in a MongoDB document in order to track when the page was added to the dataset and (i) augment dataset with new pages as more information about requests becomes available over time without re-downloading the entire new set; (ii) clean old records over time to allow detecting the topics of interest in a certain period of time (e.g. during elections the topic will most likely be elections, when during soccer world cup finale it tends to be soccer etc.)

4.3 Dividing the world into the grid

Having a huge dataset of URLs with geolocations possibly all over the world, we realized it would be quite an immense waste of computational time if we walk through all those 2.5 million geoentries (Location and the set of URLs that was accessed from that one) just to filter the ones that belong to the initially required area A of the map.

As we need to walk through that geoentires anyways, why not doing it only once and caching the results for the later use? This is how the idea of precomputing originated for this project. We imagined that we could split the entire world into the smallest squares of size 16x16 km (there is an engineering limitation of geohashing algorithm that induced this size - explained later) and then aggregate those smallest squares in groups of 4 into one bigger every time, thus one square of size 64x64 km would have 4 squares of size 32x32 km inside, one 32x32 would have 4 of size 16x16 etc. We imagined the highest level would be 2048x2048km that would cover almost all the Western Europe in case analyst is interested in knowing what are the topics common in such a big region. Thus we would have 8 levels of the grid size: squares of 16, 32, 64, 128, 256, 512, 1024, 2048 km and this limitation is required for optimal computation speed.
Considering the tradeoff between ability to make the squares of custom size (such as 6, 12, 160km etc.) for the sake of convenient analytics and the performance improvement promised by the proposed kind of precomputation, this limitation looks bearable - there is always the possibility to see the topics for the squares at the lower level and combine them manually to achieve the aggregated topics for a desired

[4]https://github.com/AsyncHttpClient/async-http-client

area for the analyst using the program.

This is the example of how those squares would look like and how they would be aggregated into bigger ones (fig. 1).

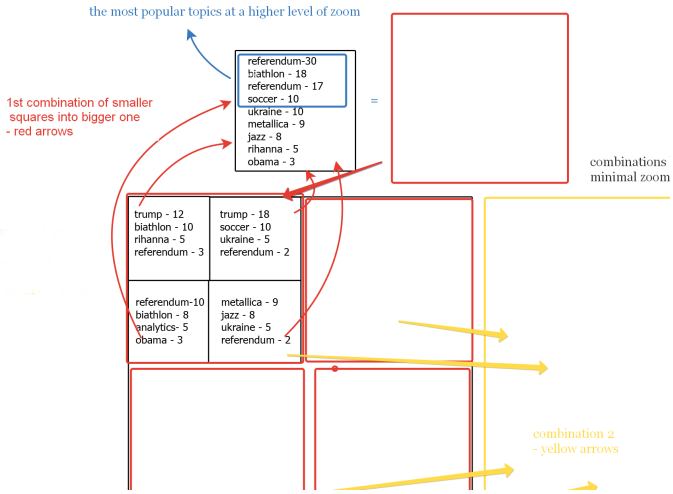

Figure 1: Squares and aggregation.

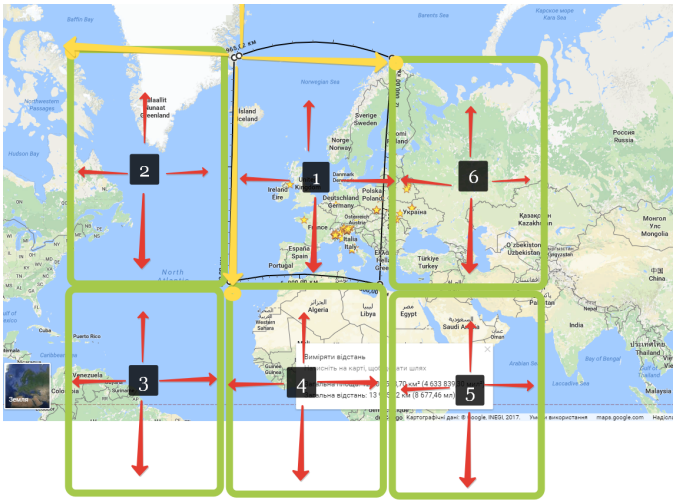

Figure 2: Top level squares.

For this project we've decided that top level (2048x2048km) squares would have their IDs starting from 1 to N depending on how many squares would fit into the world map(fig. 2).

Using geohash in order to speed up the process of squares lookup.

At the next step and later throughout the program there will be the need to quickly find the existing square in the database having the point that belongs to this square. The most straightforward algorithm is to walk through all the squares and check by coordinates if the point lies within the limits of lines $y = (-x; x); x = (-y; y)$, but when we'll have 16 384 squares of the smallest size in every top level square (and those will be around 200), needing to perform

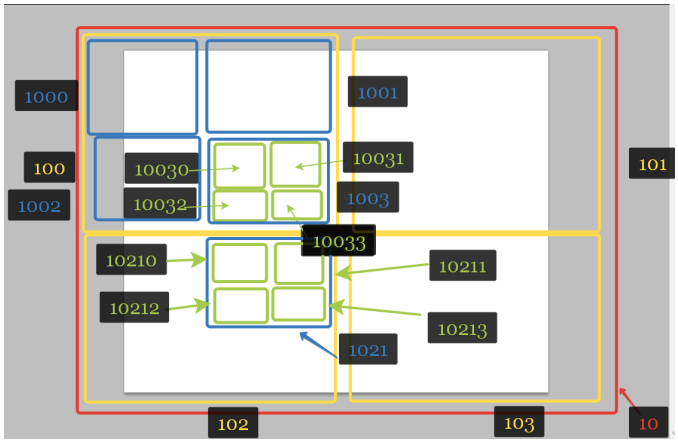

Figure 3: Id naming model.

this lookup operation 2 500 000 times for every geoentry this straightforward process becomes extremely expensive. Apparently, we needed a better mechanism for squares lookup. The solution appeared to be geohash [5] - a geocoding system used to compress the pair of (lat; lon) coordinates into a single string. According to Wikipedia "it is a hierarchical spatial data structure which subdivides space into buckets of grid shape". Geohashes offer properties like arbitrary precision and the possibility of gradually removing characters from the end of the code to reduce its size (and gradually lose precision). For example, the coordinate pair 57.64911,10.40744 (near the tip of the peninsula of Jutland, Denmark) produces a slightly shorter hash of u4pruydqqvj. This structure can be used for a quick-and-dirty proximity search: the closest points are often among the closest geohashes.

This is exactly what we needed for our problem! Geohash with 4 characters produces ±20 km error. It means any point within 20km will have the same 4 characters long geohash that the pivot point. This is acceptable for squares of size 16x16km, using 4 characters of geohash in practice we get at maximum 8 overlapping squares per geohash, and the algorithm for lookup looks like this:

Algorithm 1 Select square containing geolocation

Input: Location l
Output: Square containing l
pointGeoHash = $getGeoHash(l)$
squares.getSquaresWithGHash(pointGeoHash)
▷
% traverse the returned squares and use the conventional method of coordinates comparison to place the geoentry with corresponding location into the right square. %

It's still 8 comparisons in the worst case, but already much better than 16 384 * 200. Additionally, if we decide to make minimal squares of size 2km for higher precision we can use geohash with 5 characters. Any smaller size doesn't make sense for our algorithm as the next geohash error is +- 0.61km that will produce too many overlapping squares.

[5] http://en.wikipedia.org/wiki/Geohash

Algorithm of splitting the world into top level squares. Starting from the point on the map that we want to be the top left corner of the first top level square, we'll create the square of size 2048x2048. In order to achieve that, we compute the coordinates of bottom right point that is $d = \sqrt{qSide^2 + qSide^2}$ (based on Pythagorean theorem, where $qSide$ in this case = 2048km) away from top left point 45° in southeast direction, and having both coordinates that define a square, we assign the square an ID=++lastUsedTopLevelSquareId where numeration starts from 1.

After creating the first top level square we create four its neighbors:
1) Using the same coordinate computing function find the point that is $qSide$ away from $topLeftpoint$ of the current top-level square in western direction (yellow points in fig. 2 represent 3/4 of those points).
2-4) Same in northern, eastern, southern directions.
5-8) Create the corresponding squares starting from the computed coordinate of size $qSide$.
9-12) Call the same function recursively so that every new created square will attempt to traverse and create its own neighbors.

Apparently, this recursion will eventually redundantly traverse the squares that have been already created, for that case it's important to check if the square already exists before creating it, for that purpose for top level squares we need to compute geohash of length 1 with ±2500 km error and then see if such square exists using the coordinates. If it exists, return from function. That's the condition that will eventually stop the recursion.

After the last call of this recursive function is over, there's the need to traverse the entire grid of top level squares in the same way to establish pointers connection between them, every top level square has links to bottom, upper, left and right square, and those should be filled with this final function call: calling the function we should always pass the previous square from where the traverse function was called and the direction in which it was called (left, right, top, down). Based on this information the neighbor square discovered by this final traverse will be able to set the pointer to the neighbor where it was traversed from. These connections will be needed later in order to move between adjacent squares and display the topics discovered for those.

Now as we have all the top level squares we need to split those into all the child squares of size 16, 32, 64, 128, 256, 512, 1024. They're different from top level squares in the way how they are connected and identified. To tackle this need effectively we came up with idea to have the special way of assigning ID numbers for those squares: except the top level squares all the inners posses this format:

X1..7[0..3] where X - the id of a parent square, 0, 1, 2 or 3 is the id of the child square, numeration is row-by-row starting from top left sub-square (fig. 3). For instance, for id 10001: child id is 1, parent id is 1000.
There's also a function to split the squares into subsquares and it's also recursive. We call this function for every top level square when creating it.

After top level and all the inner squares are created, the world is partitioned, we're ready to compute the topics.

NB: existing problem here is the function that computes new coordinates based on bearing (direction of movement in degrees from starting point) and distance does compute the longitude far from precisely anywhere far from the equator. If the Earth would be flat it'd work perfectly, but as the Earth is an ellipsoid the top level squares created anywhere far from equator tend to be extremely distorted. It could be fixed having the proper function to compute the new coordinates based on bearing and distance, but 3 days spent on this issue alone didn't bring us any closer to success, though we realized we need the help from geography, geometry or aviation expert to get this done for a real full-scale software product.

The solution is somewhere behind normalization of longitude coordinate depending on the current latitude, as longitude density changes drastically with the change of latitude. The formulas we tried didn't really work and our background didn't allow us to solve this problem.

Another alternative could be using the Python geopy library [6] (again, 3 days of search didn't give any insight on existing Java library to solve this problem), but we'd need some more time to create an interface from our Java program to Python program, so at the moment we've decided to concentrate on other parts of this project.

One more idea about top level squares - create them manually as there are not that many and avoid oceans, poles, area with no mobile internet and traffic to speed up the program abandoning processing of the areas where by definition can not be any web page accesses using conventional cell phone.

For example, in the dataset we've been provided there are absolutely no requests coming from both Americas, nothing from China, Russia, Norway. Most of the requests are from Italy, a little bit through Europe and Africa. So depending on the dataset we could also reduce the load on system by creating the big squares manually in the areas where it makes sense.

Algorithm 2 Recursive partition square into children

```
function partitionSquareIntoChildren(Location topleft, int parentSide, long parentId)
        ▷ % minimal square size, no need to split squares beyond this size %
    if parentSquareSide == 16 then
        return
    end if
    Square q0 = new Square(topleft, parentSide/2)
    q.setId(parentId*10 + 0 )
    Square q1 = new Square(topRight, parenSide/2)
    q.setId(parentId*10 + 1 )
    Square q2 = new Square(bottomLeft, parentSide/2)
    q.setId(parentId*10 + 2 )
    Square q3 = new Square(bottomRight, parentSide/2)
    q.setId(parentId*10 + 3 )
    for i = 0 to 4 do
        partitionSquareIntoChildren(qi)
    end for
end function
```

4.4 Assigning URL sets to the squares

As we now have the world split into squares it's time to place the URLs accessed from certain locations into the squares on the world map grid that those locations belong

[6] https://pypi.python.org/pypi/geopy/1.11.0

to. This is one loop that goes all along the geoentries and places them into squares of the grid:

Algorithm 3 Placing URLs into the square

```
for i = 0 to geoentries.length do
    eHash = getGeoHash(geoentries[i].location)
    squares = getSquaresByGeoHash(eHash)                ▷
% now use the same algorithm presented earlier to choose
the exact square based on coordinates %
    selectedSquare.putURLs(geoentries[i].URLs;
end for
```

Side note: we're having the web pages previously downloaded and saved as files and it is only 0.635% of URLs in the dataset that lead to web pages. When we process URLs later to determine the topics, we check if the URL has the filename with the corresponding URL hash in the folder with downloaded webpages. We could also do it in this function instead to only write useful URLs that lead to webpages into squares.

4.5 Running topic detection for the smallest squares

Now as we have all the squares ready for processing, we fetch the lowest level squares (16x16) from the database and one by one start calling LDA function(MALLET library [7]) to compute their topics. For each square's web page(text is tokenized and lowercased, stopwords removed, however wider list of italian stopwords is required for proper work) we run LDA and model only 1 topic, $alphaSum = 1.0, beta = 0.1$. Afterwards, for each output we select top-10 keywords (defined from now as T_{small}) and its' occurrences(r_i), therefore compute rating of each keyword by the following principle: if any other list of keywords contains certain keyword w, then give to w a 'reward': set it's rating to $max(r_i, r_j) * 1.1$. The intuition behind this is quite simple: if w is contained by a different distributions, then this word is likely to be important and representative. Finally, we need to normalize and rescale each rating by the eq. (1)

$$r_{new} = 10 * \frac{(r - r_{min})}{r_{max} - r_{min}} \tag{1}$$

4.6 Topic aggregating

Another major improvement we have considered for our program is to benefit the aggregation function to combine topics of 4 child squares into one parent square. This would allow significant speed-up as we wouldn't need to rerun LDA every time we compute the topic for squares of bigger size than the smallest one. Finally, having computed topics T_{small}, we are ready to compute T_{top-k} for a certain square. For the smallest squares, this task is trivial - we simply choose those keywords, which have the largest rating in between the topics. For the all squares containing smallest ones, we define two statistics: (i)UR_i - normalized ratio of URLs number U_i in a child square q_i eq. (2); (ii) GR_i - normalized ratio of geoentries G_i in a child square q_i eq. (3). This is important, because we would like to regularize our model and penalize those topics, which were inferred from small number of URLs(cannot represent large area), small ratio of geoentries(meaning that one person could visit a large number of web pages, but it is not representative for the whole area). Normalization and rescaling is needed because of different distribution of number of URLs and geoentries among squares.

$$UR_i = \frac{U_i - U_{min}}{U_{max} - U_{min}} \tag{2}$$

$$GR_i = \frac{G_i - G_{min}}{G_{max} - G_{min}} \tag{3}$$

U_{min}, G_{min} - corresponding minimal and maximal values of number of URLs and geoentries between four squares.

The eq. (4) describes how to calculate rating of j keyword in square q_i for aggregation between four squares. Here α and β are free parameters in range $[0 : 10]$, which set 'reward' for keywords from area with high geoentries ratio and great URLs number; k is free parameter of significance in range $[1..N]$. After this we again apply normalizing and rescaling to ratings, but to range $[1 : 100]$.

$$r^*_{ij} = k * r_{old} + \alpha * UR_i + \beta * GR_i \tag{4}$$

4.7 Finding squares corresponding to the request

According to the project requirements the input:

- A set U of pairs <URL, Location>
- An area A of the map determined by a top left and a bottom right point)
- A step S (e.g. 2km)

Output:

- Create a grid by dividing the area A in squares of size SxS and identify and show the top-k popular topics for each such square.

The final function we need to write is the one that returns the required grid considering that we have all the squares and their topics already precomputed in the database. Take to notion: our grid has certain engineering limitations (discrete step, where squares begin and end is predefined).

In order to achieve that, we display the squares starting from the one whose topleft location is the closest to the one required, displaying squares of size closest available to the step and showing as many squares as we can fit the squares of a given size into the given area.

Additionally, for this function to work we need the way to move between neighboring squares. Here we will also utilize that special way of assigning IDs to the squares.

There are 4 helper functions to move up, right, down and left. We'll present the *moveDown* function here to show how the lower square ID is computed. (see Algorithm 4)

Apparently, this movement function works for all the squares of level lower than the top one. In case we need to move between top level squares we'll utilize the pointer links between neighbors we created when splitting the world into top level squares.

Now, when all utilitarian functions are ready, here is the pseudocode for the final function that receives area A and step S as an input and returns the information about required squares:

1. Find the square size that is closest to the step among available ones

Algorithm 4 Algorithm for computing ID for lower square neighbor

```
function moveDown(long squareId)
    idLastDigit = squareId%10
    if idLastDigit == 0 then
        newId = (squareId / 10) * 10 + 2
    else if idLastDigit == 1 then
        newId = (squareId / 10) * 10 + 3
    else if idLastDigit == 2 then
        newId = moveDown(squareId / 10) * 10 + 0;
    else if idLastDigit == 3 then
            ▷ % same recursion as in previous case but take
1st child of neighboring parent %
    end if
    return newId
end function
```

qSide:= findClosestToTheGivenStep(step)
(for example, step 25 will return the closest available qSide=16, step 38 will give qSide = 32)

2. topLeftSquare:= Select square containing geolocation(topLeft);
(this function is described in "Algorithm 1" section)

3. Calculate bottomLeft point of the input area A:
bottomLeft(bottomRight.latitude ; topLeft.longitude)

4. Find topright the similar way.

5. Compute distance in km between topleft and topRight = width of the area in km

6. widthInSquares:= widthInKm / qSide;
(e.g. for width of 600 and qSide of 512 it'll display 1 quad per row)

7. Compute distance between topLeft and bottomLeft = height in km

8. heightInSquares:= heightInKm / qSide;

9. Now as we have the topLeft square and size of grid in squares, we simply move between squares row by row with two nested for loops and display or return the squareId, it's location points and the topics computed for a given square. This can later be used to visualize the information on a digital map.

4.8 Parallelized version of the program

One of the most expensive computations in our algorithm in presence of abundant input data is computing the topics for squares at the lowest level (those of size 16x16km) - operation *computeTopicsSmallest*(). In this part of the algorithm we process the textual content of all the web pages accessed by mobile users within this square with LDA algorithm and write the resulting topic vectors (most important words from the content identified by LDA along with their occurrences number) into the Square data structure to the database for later display and use to compute aggregated topics.

Considering that the textual content is already pre-downloaded, this atomic operation takes the longest time in the existing algorithm, and it makes sense to leverage the power of parallelization to speed up this process. What we did here is fairly simple - in the loop by all of the lowest level squares for which we compute the topics at first, we parallelized this atomic operation of running LDA on webpages present in the square, so that we had squares processed separately on different workers. From perspective of Spark terminology, we created JavaRDD from list of all Squares with minimal side and map to each of them a function, which computes topics for their web pages. The performance gain is described in *Experiment II*.

Potential parallelization can be done also in webpage fetching step as described earlier, but there's the need for proper configuration of MongoDB or HDFS that we didn't manage to achieve yet.

5. EXPERIMENTS

We took subsample of a dataset U (51000 records), fetched valid web pages and partitioned map into smaller squares for only 1 top-level square of size 2048km. The last step of precomputing is assigning URLs to smallest squares table 1.

5.1 Experiment I - Sequential performance

We set up a baseline straightforward algorithm that computes LDA from scratch for all the URLs in every new size of square, aggregation algorithm is the one developed by us that computes the topics for quads of level 2+ based on the $n-1$ level squares topics, where n = [2..8].

All performance tests with step of size less than 512 were executed for area A with topLeft location 47.185257, 8.206737 and bottomRight 43.171934, 18.449864. For the step size equal 512; topLeft location 47.185257, 8.206737, bottomRight 36.769755, 21.896536 table 2. As one can observe, proposed approach allowed to reach more than **2.x** speed-up comparing to the baseline. We would like to present a sample output from the test run on *step* = 256 for quad id 1001(fig. 4 fig. 5). We highlight that it is impractically to compare outputs in terms of Jaccard similarity etc. because of algorithms specificity, however we still need to have reliable comparison measure for different variations of devised algorithm.

5.2 Experiment II - Spark performance gain

For this experiment we've benchmarked the performance of function that runs LDA for all the squares of the map at the lowest level. For this experimental case we've used the square of size 2048x2048 that had 16384 smallest 16x16 squares inside and only 469 of those were filled with geopoints from the dataset.

Then we ran this function on Spark standalone server with a local slave that had 1 configured core (1 possible worker and thread correspondingly). It took 20.29 seconds. After that we ran it in the same configuration but with 2 cores engaged and it sped up to 12.37 seconds for that operation.

We also tried this with two slave laptops and 4 cores in total but that appeared slower as it took more time to send data over the network from master node to remote slave, resulting in the same time as single core local slave takes to compute it. In theory it should run faster if there would be replicated MongoDB on both machines so that remote slave would fetch data from its own local copy. But having our time limits we didn't manage to run MongoDB in that mode so the only result we have is this single-slave with 1 and 2 workers. Still there's the evidence of performance gain.

Figure 4: Topics inferred by aggregation.

Table 1: Precomputing performance

Procedure	Time sec
partitionMapIntoSquares	5.6
partitionUrls	38.8
computeTopicsSmallest	**23.8**
computeTopicsSmallest P1	20.3
computeTopicsSmallest P2	**12.8**

Figure 5: Topics inferred by rerunning.

Table 2: Topic detection performance

Model	Step	Time sec
Topic aggregation	36	**10.0**
LDA rerunning	36	**33.2**
Topic aggregation	64	8.17
LDA rerunning	64	26.7
Topic aggregation	128	8.5
LDA rerunning	128	22.2
Topic aggregation	256	7.5
LDA rerunning	256	16.7
Topic aggregation	512	**36.0**
LDA rerunning	512	**63.0**

6. CONCLUSIONS

As a result of 1.5 months long project we've developed a complex modular software product concept that allows having the dataset U of pairs <URL, Location> representing the URLs that mobile Internet users accessed at a recorded geographical location, to get a worldwide statistics about what topics are popular in a given geographical area based on the text of the webpages accessed by those mobile users.

This is achieved by splitting the world map into the grid using the geographical coordinates, then spreading the geocoded URLs from the dataset among created squares - particles of the grid, and running LDA algorithm to detect the topics in the every present square of the smallest size (16x16).

Then those smallest squares topics are aggregated into bigger ones of size 32x32, 64x64 ... 2048 x2048 with another custom algorithm described in this report. Having this, we're solving the initial problem of the need to display the topics popular in a given geographical area with a certain step.

From what is done at the moment it's now easy to implement the UI for convenient visualisation of the topics on a digital map: the API can return the information about squares coordinates and the corresponding topics, it supports discrete movement of the map between squares in four direction: north, south, east and west (one move = side size of a currently displayed squares in a grid) as well as zooming in and out.

The developed concept is implemented using Spark framework, MongoDB NoSQL database and Java programming language.

Almost surprisingly, the longest operation in this problem - downloading the web resources by given URLs and the most important performance gain was achieved in this area filtering the input dataset. We've encountered difficulties with the function that calculated new geocoordinates given initial point, distance and direction in degrees, still need more work in this field or involvement of the expert. We probably could have achieved better parallelization if we use Spark at the step of downloading the web pages, but we needed HDFS or MongoDB replica in order to achieve that. This is going to be our next step in this project.

These experiments could be richer if we would have more URLs pre-downloaded, but erroneously we failed to do that in time we had. This is going to be our next step - to test this concept on much bigger sets of data when we finally can leave the machine to download resources for a few days without rushing to work on other parts of the project.

Overall we consider that a very decent amount of work was done within the time provided and we've really developed much deeper understanding in Data Mining and Big Data principles, so the goal of the course project was achieved, now there is a huge interest in developing this concept further to make it closer to production prototype.

7. REFERENCES

[1] A. Agius. 7 ways to use google trends you've never thought of before, 2015.

[2] D. Andy, K. Graham, S. Kristen, A. Chris, Z. Ilya, L. Zunsik, M. Sathappan, B. Patrick, S. Nathan, Z. Liang, L. Chang-Tien, K. R. Paul, F. Youssef, and R. Naren. Forecasting significant societal events using the embers streaming predictive analytics system. *Big Data*, December 2014.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[4] J. Bright, H. Margetts, S. Hale, and T. Yasseri. The use of social media for research and analysis : a feasibility study. In *DWP ad hoc research report*, volume 13. Department for Work and Pensions, UK government, December 2014.

[5] X. Jin, S. Spangler, R. Ma, and J. Han. Topic initiator detection on the world wide web. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 481–490, New York, NY, USA, 2010. ACM.

[6] L. Li. How to research a profitable niche market: Law of attraction case study, 2013.

[7] A. K. McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[8] L. T. P. Nghiem, S. K. Papworth, F. K. S. Lim, and L. R. Carrasco. Analysis of the capacity of google trends to measure interest in conservation topics and the role of online news. *PLoS One*, November 2016.

[9] J. Qiu, L. Liao, and X. Dong. Topic detection and tracking for chinese news web pages. *International Conference on Advanced Language Processing and Web Information Technology*, July 2008.