Topic-zoomer: a URL categorization system

Luca Giacomelli
DISI - University of Trento
Student id: 179954
luca.giacomelli-2@studenti.unitn.it

Pierfrancesco Ardino
DISI - University of Trento
Student id: 189159
pierfrancesco.ardino@studenti.unitn.it

ABSTRACT

Smartphones and tablets pervade our lives on a daily basis and people use them in most curious ways.
The key idea behind this work is to take advantage of the traffic they generate while they are connected to the Internet, extracting *topics* from what people search on the web with their mobile devices. This use case provides also the geographical coordinates of the users based on the telephone cell they are connected to.
For the purpose of this work the approximate location of the cell is enough, even though it implicitly defines a limit on the precision while selecting an area inside a map.
This paper presents Topic-zoomer, a URL categorization system.

Keywords

Data mining, URL categorization, topic extraction, geolocalized URLs, Latent Dirichlet Allocation

1. INTRODUCTION

Nowadays extracting topics from a text is a very common goal in data mining and the problem of categorizing URLs is highly connected to it.
Generally speaking, the main purpose is to find out the topics of a set of documents. Applying this concept to a more specific problem, Topic-zoomer is able to find the topics of a dataset based on the documents' geographical location.
Starting from a geotagged dataset, the tool is able to restrict its space domain only to pages which are inside a certain area. In this way, it can discover the topics of the various regions.
This work can be employed in many different fields. For example, it can be useful for public institutions such as municipalities; they could understand the citizens' needs in order to improve the functionality of the city's facilities and their territory coverage.
However, the main problem of a real-life usage of this kind of analysis can be the data source. Internet Service Providers (ISPs) can easily extract and provide geotagged URLs based on the HTTP requests passing through their routers. Furthermore, the anonymity of the users generating such traffic must be preserved.
This kind of problems is a challenge for traditional programming paradigms, since the amount of data can be huge and it may also involve streaming processing.

2. RELATED WORK

URL categorization is a task that many people and companies are addressing in the last few years, pushing even to the categorization of the whole World Wide Web. Some of the companies involved in this kind of analysis are Cyren[1] and BrightCloud[2]. Moreover, a huge number of academic researches have been published and also some patents[3] have been filed on the subject. However, the specific case of geotagged URLs has not raised so much interest neither in the academic nor in the developer communities.
Furthermore, when approaching problems related to Big Data or Data Mining, the challenge is to deal with huge amounts of data that cannot fit in a single machine or use smart solution to avoid useless computation. This implies that merely sequential solutions are not admitted, since they can be costly in terms of time. Instead it can be possible to use computations already done with the aim of reducing the amount of time needed for a run and at the same time trying to not losing accuracy on the results.

3. PROBLEM DEFINITION

The task Topic-zoomer aims to solve can be better described starting from its *input* and *output* data.
The *input* is a CSV file containing rows of the following form:

$$< latitude, longitude, \{url0\,|url1\,|...\} > \qquad (1)$$

The URLs are analyzed in order to identify the topics that are represented as a vector of words that very often appear together.
Let A be an algorithm to compute such topics (eg. TF/IDF, frequent itemset or LDA), the algorithm L gives as a result the topics of the dataset in general.
L also allows the user to restrict the search in the dataset using the following parameters:

- A: the map area identified by top-left and bottom-right corners. This is the region of interest for retrieving the topics.
- S: the size of the grid to be created inside the area A (ie. the square side length).
- k: the number of topics to search inside each square of the grid.

[1]http://www.cyren.com/url-category-check.html
[2]https://www.brightcloud.com/tools/change-request-url-categorization.php
[3]https://www.google.com/patents/US8078625

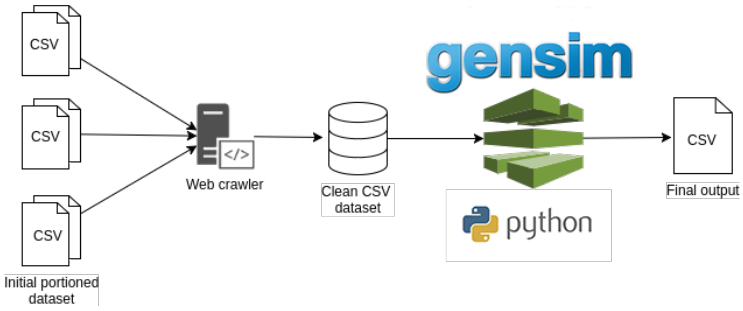

Figure 1: The schema representing the overall solution.

In this way A is divided into squares of size $S \times S$ and the k topics are identified inside each square.

For this tool A is not directly given as input. In fact, the top-left and bottom-right corners are computed from the dataset taking the top-left and and bottom-right corners of the dataset. Also the number of k is fixed to 3.

4. SOLUTION

This section describes in details all the steps needed to realize this tool, from the initial web page download to the data analysis done with python. The order by which they are presented reflects the one that was followed during the experiments. Furthermore, Figure 1 represents the architecture of the overall solution.

4.1 Data collection and preprocessing

The first task was to transform an initial dataset with the form shown in 1 into another one of the form

$$< latitude, longitude, page\_text > \quad (2)$$

A *crawler* has been developed in order to follow the links in the dataset and download the web pages linked to them. This may seem a trivial operation at a first glance, but it is not really true. In fact, the initial dataset contained a lot of URLs pointing to images, videos, configuration files for mobile phones applications and many other elements that are far from being a text web page. To retrieve only sensible information, the crawler rejects all the requests not containing *text/html* in the responses' header. This is the one which is usually populated by a web server when a client visits one of its pages. Last but not least, the crawler has been parallelized between multiple processes to minimize the download time. Multiple processes are preferred to multiple threads because Python, the language used to develop the crawler, is known to be subject to the Global Interpreter Lock[1] (GIL). Finally, after this crawling phase, a new CSV file containing rows as described in 2 is generated.

4.2 Algorithm selection

After downloading the web pages, the dataset has to be cleaned.

Among all the ways to extract topic such as TF-IDF or a simple Näive word count, the *Latent Dirichlet Allocation*[2] (LDA) algorithm was chosen as L to find the topics of the dataset in general. LDA is a generative and probabilistic model for collections of discrete data such as text corpora. It is a Bayesian hierarchical model, where each topic is modeled over an underlying set of topic probabilities. LDA is a good candidate when choosing between different methods to classify text. From a higher perspective LDA can be seen as a clustering technique, whose goal is to divide the document corpora given as input into k clusters, which are the topics to be retrieved from the text. In order to compute LDA, *tokenized pages* are first of all used to create an *id-term dictionary*, which is then used to compute a *document-term matrix*. Then the LDA model is computed using `ldamodel` from the `gensim` module. This Python module allows LDA model estimation from a training corpus. The model can also be updated with new documents in order to support online training. This function will be used later in one of the two recomputation methods developed for this tools.

4.3 Process overview

The process of finding the most important topics for each area $S \times S$ after the crawling phase is composed as follows:

1. Provide Topic-zoomer with the parameters, S, the location of the dataset and *recType*, which is the kind of recomputation to be used (0 for no computation, 1 for using `ldaUpdate` and 2 for using merging technique).
2. Load the dataset (in CSV format) generated by the pre-processing phase described in Section 4.1.
3. compute the top-left and the bottom-right corners from the dataset.
4. Create squares of dimension $S \times S$ (ie. compute the inner grid) using S and the two corners computed in 3.
5. Check if the new S is a multiple of the old S saved from previous computations. Remember that information of a computation are saved only if *recType* is different from 0. If the new S is a multiple of the old S it means that it is possible to use the recomputation method defined by *recType* (see Section 5.5 for a more detailed explanation). If the new S is a multiple of the old S then the process skips directly to point 12. If the new S is equal to the old S the software terminates its execution.
6. For page in the dataset remove punctuation and stopwords[3] for the language of interest[4] in order not to consider the most common short function words.

[4]In this specific case, Italian and English stopwords are re-

Algorithm 1 Sequential topic computation

```
1: procedure SEQUENTIAL_COMPUTATION(size, dataset)
2:     dataset ← read(dataset)                                              ▷ Load dataset
3:     oldModel ← read(oldModel)                                            ▷ Load old dataset
4:     if (size % oldModel.size == 0 && size!=oldModel.size) then           ▷ Check for recomputation
5:         base, height = calculate_New_Size(size, oldSize)                 ▷ Calculate the new dimensions of the grid
6:         for oldCorpus in oldModel.corpus do
7:             corpus[b][h] ← update_corpus(oldCorpus)                      ▷ update corpus for each new square
8:         end for
9:         if (recType == 1) then
10:            model ← ldaModelUpdate(numTopics, NumWords, corpus, oldModel.dictionary)   ▷ Recompute using
    ldaModelUpdate
11:        end if
12:        if (recType == 2) then
13:            model ← topicMerging(oldModel, corpus[b[h])                  ▷ Recompute merging the most relevant topics of each
    previously computated block
14:        end if
15:    else if size!=oldModel.size then
16:        base ← round(dataset.base/size)                                  ▷ Calculate base of inner grid
17:        height ← round(dataset.height/size)                              ▷ Calculate height of inner grid
18:        clean_dataset, dictionary ← create_dictionary(dataset)           ▷ Clean and tokenize dataset
19:        for clean_page in clean_dataset do
20:            b, h ← getCoordinates(clean_page)                            ▷ Assign each text to the correct block of the b × h inner grid
21:            text[b][h] ← clean_page
22:        end for
23:        for b in base;h in height do
24:            corpus[b][h] ← create_Corpus(text[b][h], dictionary)         ▷ Create dictionary and corpus for the ldamodel
25:            if (corpus[b][h]! = empty) then
26:                model[b][h] ← ldaModel(numTopics, NumWords, corpus[b][h], dictionary)▷ Calculate the LDA model for
    each block
27:            end if
28:        end for
29:    end if
30:    if size!=oldModel.size then
31:        if (recomputation! = 0) then
32:            save(model)                                                  ▷ Save model for reuse
33:        end if
34:        printToCsv(model)                                                ▷ Save results in a csv file
35:    end if
36: end procedure
```

7. Create the *id-term dictionary* from the whole cleaned dataset.

8. Assign each pages to the respective square in the grid.

9. Create a corpus for each square in the grid. This step is fundamental, as each page of the square has to be converted into a *word-term matrix*.

10. Train the LDA model of each square with the proper k parameter and chosing between the *single-core* or *multi-core* implementation.

11. Save results for future recomputation if *recType* is different from 0.

12. Save the results in a csv file.

All this steps will be explained in detail in the following sections.

moved due to the source of the dataset used for the experiments.

5. IMPLEMENTATION

In this section will be explained how this version of topic-zoomer was implemented. This project was written in Python using mainly functions from Gensim Library. Other python packages that were used are NLTK (Natural Language Tool Kit) and Stop words, that simply contains a list of stop words in different languages.

5.1 Pseudocode

Algorithm 1 shows the pseudocode of the project. As stated in section 4.3, there are four main parts: dataset pre-processing, the avoid-recomputation, computation and printing. Now each part will be explained in detail.

5.2 Pre-processing

The first thing to do after reading the dataset is to make it usable.
Since the goal is to find relevant topics in every block of size $s \times s$, the first step is to divide the dataset into blocks. Figure 2 shows how the total area(in red) is divided in blocks. In mathematical terms, if *size* is the size of the block, *base* is

Algorithm 2 Create Dictionary

```
1: procedure CREATE_DICTIONARY(dataset)
2:     global_text[], global_dictionary[]                    ▷ Create local variables
3:     for page in dataset do
4:         text = toLowerCase(page.text)                     ▷ convert upper case to lower case
5:         text = regexpFilter(text, [a-zA-Z]+)              ▷ keep only words with at least a letter
6:         text = tokenize(text)                             ▷ Tokenize the document
7:         text = removeStopWords(text)                      ▷ Remove stop words
8:         global_text.append(page.x,page.y,text)            ▷ Save each entry with its coordinates
9:         global_dictionary.append(text)
10:    end for
11:    dictionary = dictionary(global_dictionary)            ▷ turn our tokenized dataset into a id <-> term dictionary
12:    return dictionary,global_text                         ▷ return the dictionary of the whole dataset and the cleaned text for each entry
13: end procedure
```

the base of the dataset and *height* the height, then the inner grid will have dimension equal to $(base/size) \times (height/size)$, with both divisions rounded up to the nearest integer value.
The second part of this pre-processing regards the text documents of the dataset. The dataset is cleaned using the *create_dictionary*() function, showed in algorithm 2.

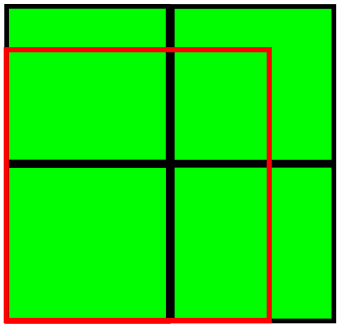

Figure 2: How the total area is divided in blocks

The *create_dictionary*() function starts with creating two variables, one for storing each entry of the dataset after it has been cleaned and one for storing all the text and creating the dictionary for the dataset.
Then each entry of the dataset is cleaned in the following way:

1. Convert the the text into lowercase characters.
2. Use a regular expression filter so only words are kept.
3. The text is tokenized, that means it becomes a structure of n tokens, each of them is a word.
4. Stop words are removed. Stop words are extremely common words that do not have any meaning for this purpose. In this function are removed italian and english stop words.
5. The cleaned text is stored with its coordinates, so in the future will be possible to assign it to its correct block. It is also stored in another variable, in order to create the dictionary of the dataset.

After all the dataset is cleaned, a dictionary is created, that means that at each different token is assigned an id.
The *create_dictionary*() function returns both the dictionary and the text. The final step is assigning each cleaned text to its correct block. The *create_dictionary*() function returns the cleaned text in the form $< x\_coordinate >;< y\_coordinate >;< cleaned\_text >$. So the task consists in iterating and, after reading the coordinates of each text, appending the text to the correct block. The final result will be a dictionary for the whole dataset and a $b \times h$ matrix with all the text.

5.3 Computation

The computation is done using the ldaModel function from the Gensim Library.

$$gensim.models.ldamulticore.LdaMulticore(num_{topics} = topics_{no}, corpus, id2word = dictionary, workers = cpu_{cores-1}, passes = 50)$$

This function needs the previously calculated dictionary and a corpus, and returns a model with num_{topics} topics. For calculating the corpus the function *create_corpus*() was used. This function receives a dictionary and a tokenized text as input and returns a document-term matrix. In other words, using the dictionary and the tokenized list it returns a matrix where on one axis there are id index of the dictionary, on the other there are the tokens. Then they are counted and each cell represents the number of times each term has appeared.
In this project the number of topics is fixed at three, with three words for each one. This is because with more words or more topics the model it is not accurate and the results are not meaningful.
The number of passes indicates the Maximum number of iterations allowed to LDA algorithm for convergence. It is set to 50 because it is a good compromise between performance and accuracy. The multicore version was used because it has better performance than the single core version. The number of workers(threads that the ldaModel function will use), as depicted in the ldaModel documentation, is the number of physical cores minus 1. This means that if the cpu has more than two cores then the multicore version will be used otherwise it uses the single core version.

5.4 Printing

The last thing to do is printing the results.
The solution chosen for dislaying the outcome is a csv file. In each row are written the coordinates of the block and three topics. It is structured in this way:

< Top-Left corner >; < Bottom-Right corner >; < Topic with id 1 >; < Topic with id 2 >;< Topic with id 3 >

If a block is empty it is not printed. After printing the software terminates its execution.

5.5 Avoid-recomputation

These kind of tools in order to be usable needs not only to be accurate, but they have to perform well also with respect to time and memory usage.
A tool that performs well in term of accuracy but needs an high amount of memory could be really expensive and one company could prefer less expensive tools, thus it is necessary to find a good compromise between performances and costs.
Use previous computations to quickly compute the topics for each of the new cells is a key point for this project and in general for these tools to be usable. As said before, two different recomputation methods have been developed and implemented for this project. Both are based on cell rescaling and merging of the cells in order to provide the zoom out feature.
It was decided to use the following simplification: resizing can only be done when the new size of the cell is a multiple of the old size of a cell. For example if the cell size of the previous computation was 2 and the new size is 3 then it is not possible to use recomputation and so a new computation will be done, otherwise the cell are rescaled and merged avoiding useless computation.

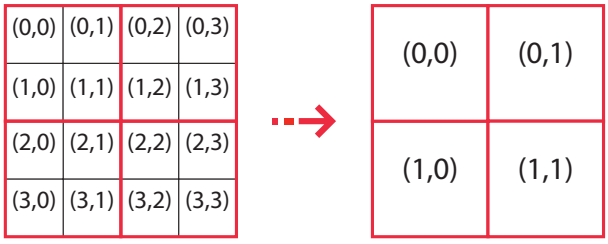

Figure 3: An example of cell rescaling.

For instance, figure 3 shows the rescaling for a 4x4 map to a 2x2 map.

5.5.1 *Recomputation with ldaUpdate*

The first recomputation method implements the *ldaUpdate* function that updates an already computed model with new document. Taking as example the grid in Figure 3, at the beginning the ldaModel is computed for each cell in the grid and saved in a stable storage for future computations. The idea is to reuse and update one of the models. Cells (0,0), (0,1), (1,0), (1,1) of the grid on the left will compose the (0,0) cell of the 2x2 grid, so the model computed for the cell (0,0) can be used and updated with the corpus of the remaining cells. In general one of the cells that will compose the new cell is used as base model, while the corpus of the others will be used to update it. Algorithm 3 shows the operation performed for the recomputation.

5.5.2 *Recomputation with topic merging*

The second recomputation method counts how many times the topics of the old cells appear in the new corpus, returning the most probable topics. Again, taking as example the grid in Figure 3, the topics of cells (0,0), (0,1), (1,0), (1,1) of the grid on the left are inserted into an array, while the corpus of these cell will compose the corpus of cell (0,0) on the right grid. Then the *topic_update* function will compute how many times each topic of the array appears in the new corpus. Algorithm 4 shows the operation performed for the recomputation.

6. RESULTS

The final step consists of measuring the overall performances.
For doing so, this software was executed with squares of side equal to 4. Then with side equals to 8, using all the possible techniques (normal computation and the two types of recomputation). The table 1 shows the execution time for each technique:
Looking at this table it is easy to see that the proposed

Table 1: Execution time

Square's side	Computation technique	Time
4	Normal computation	4045 s
8	Normal computation	2662 s
8	"Merge" recomputation	11 s
8	"LdaUpdate" recomputation	640 s

recomputation algorithms are faster than the normal algorithm.
After looking at the performance it is time to compare the quality of the results using the proposed algorithms.
From the quality point of view there were some problems related to the dataset (like broken links, links to pages with irrelevant or absent content etc.) but we can see that the results are coherent and the recomputation techniques give results that are meaningful using a faster computation.
Just for giving an insight, choosing a step equal to 8 the results related to the first square are *diritti rai http ; type itemid x ; x f mm ;* and with the two recomputation techniques the following topics were obtained *rai;diritti;type;* and *f mm type ; x dl diritti ; test page d ;*.
In conclusion, this method does work but there is still room for further improvements like an adwanced crawler that filters all the pages retriving only meaningful information useful for this type of analysis.

7. CONCLUSIONS AND FUTURE WORK

The algorithms implemented in this projects are three, as explained in sections 5.3 and 5.5.
The first is an implementation of the LDA and the other two are designed for avoiding recompuations in certain cases (when the new side is bigger and a multiple of the previous side).
The first recomputation method is based on the LDA update function of the GENSIM library, it basically merges the squares into a bigger one, avoiding the recomputation of the area given by the formula $number\_of\_new\_squares \times area\_old\_squares$. This gives quite the same results of the LDA model.
The second recomputation method takes the probability for the most probable topics for each old square and averages

Algorithm 3 Recomputation with *ldaUpdate*

```
1: procedure RECOMPUTATION_LDAUPDATE(dataset)
2:     to_merge = newStep / oldStep,
3:     y = 0
4:     y_old = 0
5:     while y_old < height_old do
6:         x_old = 0
7:         x = 0
8:         while x_old < length_old do
9:             corpus_to_merge = []
10:            for i in range (0,to_merge) do
11:                for j in range (0,to_merge) do
12:                    if not ldaModelMatrix[x][y] and corpus_old[y_old +i][x_old + j] then
13:                        ldaModelMatrix[y][x] = ldaModelMatrix_old[y_old + i][x_old + j]      ▷ Use an old ldaModel as base
14:                    else
15:                        corpus_to_merge += corpus_old[y_old + i][x_old + j]                  ▷ Merge the corpus of the cells
16:                    end if
17:                end for
18:            end for
19:            if ldaModelMatrix[y][x] then
20:                ldaModelMatrix[y][x].update(corpus_to_merge)          ▷ Update the base model with the merged corpus
21:            end if
22:            x_old += to_merge
23:            x +=1
24:        end while
25:        y_old += to_merge
26:        y +=1
27:    end while
28: end procedure
```

them for finding the most probable new topics. Obviously this method is less reliable but really fast.
Section 6 shows the performance of these algorithms. Unfortunately, due to the problems of some links of the dataset, an in depth analysis of the results from the quality and reliability point of view was not feasible but there is reason to believe that with a better dataset or crawler the aforementioned algorithms will work.
This project could be improved in many different ways, left as future works, like the implementation of a GUI for choosing in a user-friendly manner the square's area or an area different from a square. The possibility of chosing different areas will lead to new aprroaches for the avoiding recomputaion techniques, that will be more similar to the first recomputation method.
Another improvement will be the creation of a crawler designed for searching only on certain web pages related to the area of interest of the final user.
In conclusion it is possible to assert that this project is suitable for finding topics in areas divided in squares and a good starting point for user-defined areas, in both cases using geotagged URLs.

8. REFERENCES

[1] D. Beazley. Understanding the python gil. In *PyCON Python Conference. Atlanta, Georgia*, 2010.

[2] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[3] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

Algorithm 4 Recomputation with *topicMerging*

```
1: procedure RECOMPUTATION_TOPICMERGING(dataset, topics)
2:     to_merge = newStep / oldStep,
3:     y = 0
4:     y_old = 0
5:     while y_old < height_old do
6:         x_old = 0
7:         x = 0
8:         corpus_to_merge = []
9:         while x_old < length_old do
10:            corpus_to_merge += corpus_old[y_old + i][x_old + j]
11:            for i in range (0,to_merge) do
12:                for j in range (0,to_merge) do
13:                end for
14:            end for
15:            ldaModelMatrix[y][x] = topic_update(corpus_to_merge, dictionary, topics)        ▷ Compute the new topics
16:            x_old += to_merge
17:            x +=1
18:        end while
19:        y_old += to_merge
20:        y +=1
21:    end while
22: end procedure
23: function TOPIC_UPDATE(corpus, dictionary, topics)
24:    topic_occ = {}
25:    for  topic in topics do
26:        for page in corpus do
27:            for token in page do
28:                if dictionary[token[0]] == topic then
29:                    if topic not in topic_occ then
30:                        topic_occ[topic]=token[1]
31:                    else
32:                        topic_occ[topic] += token[1]
33:                    end if
34:                end if
35:            end for
36:        end for
37:    end for
38:    total_topics = sum(v for k,v in topic_occ.items())
39:    return RankedAndSortedTopics(topic_occ, total_topics)
40: end function
```