Masaryk University
Faculty of Informatics

The Categorization of the Darkweb

Master's Thesis

Bc. Linda Hansliková

Brno, Fall 2019

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Linda Hansliková

Advisor: RNDr. Martin Stehlík, Ph.D.

Acknowledgement

My thanks go to my adviser RNDr. Martin Stehlík, Ph.D. for allowing me to proceed with this topic, for his advice and patience.

Abstract

The future Categorization of the Dark Web abstract

Keywords

Web Application, Dark-web, Python, RestApi, Django REST, TypeScript, React, Redux, d3 Graph

Contents

List of Figures

1 Introduction

2 Analysis

The purpose of this thesis was to categorize the dark web and visualize the result. In order to do that, the dark web had been scraped and the acquired pages had been stored in an ElasticSearch database. This was done as part of a bachelor's thesis which was being completed at the same time as this thesis. In order to retrieve the data from the database and do various operations on them, a back-end (BE) needed to be created. One of the operations was the categorization of the acquired pages. For that, an appropriate topic modeling approach was required. We chose to adopt the Latent Dirichlet Allocation (LDA). To visualize the output a graph was used. However, the data set is rather sizable and to be able to provide the user with useful information, not all pages can be displayed at once. Therefore a proper way to divide the graph into several subgraphs had to be found. We decided to use the well known Louvain algorithm (LA). To display the graph in a comprehensible manner a front-end (FE) was created.

In this chapter we will describe LDA, which is the method used to categorize the scraped pages. Next we will talk about why it is necessary to divide immense numbers of data into clusters in order to display them as a graph. And lastly, we will detail communities and LA, the algorithm for dividing pages into communities.

2.1 Web graph

We display our data as a graph. More precisely a web graph [19]. A web graph is a graph representation of the web where nodes are portrayals of the pages and edges depict links between the pages. Web graphs tend to be built from an enormous amount of data. As such, they can be advertised in various ways. One of the visualizations is shown in figure 2.1. The web graph depicted displays all its data at once without any labels or details. The result might be useful for viewing the internet as a whole but for our purposes was insufficient since a user has had to be able to view the relationships between nodes in more detail along with information about the pages. The web graph needed to be composed of a significant smaller amount of nodes so that the view port was not too cluttered and the sought

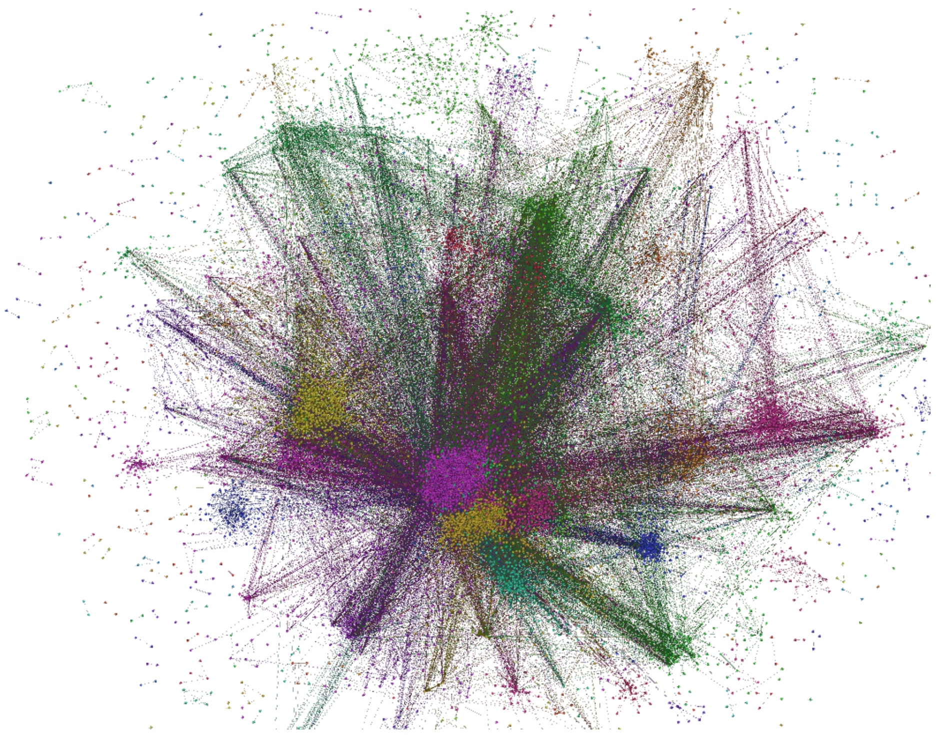
Figure 2.1: Web graph by Citeo.

knowledge could be obtained with as little hindrances as possible. Because the majority of the nodes in the graph were not isolated [1] and were in fact part of a single connected component it was possible to partition the graph based on the density of its nodes, the so called communities.

2.1.1 Community structure

If a graph can be partitioned into several subgraphs so that nodes from one subgraph are internally connected densely and are connected scarcely to nodes from other subgraphs, we can claim it has a community structure with each subgraph as a community [18]. Each community can be portrayed as a meta node of the graph and so pos-

1. An isolated node is a node with zero incoming and outgoing edges.

sibly reduce the number of nodes in the graph dramatically. The quality of such a partition can be measured using modularity. [21]

> For a candidate partition of the vertices into clusters, the modularity is defined to be the portion of the edge connections within the same cluster minus the expected portion if the connections were distributed randomly

[20]

2.2 Louvain algorithm

A profoundly used algorithm for finding communities in graphs is LA [16]. It is a greedy algorithm which maximizes modularity locally. Each node is assigned a community. Afterwards the node is taken out of its community and randomly appointed to the communities of its neighbours. After the node visited communities of all its neighbours it is left in the one with the maximum modularity value, which can also result in it remaining in its original community. Next, the algorithm runs on the newly gathered communities and tries to assign each whole community to its neighbouring communities in a similar manner. This is repeated until the modularity cannot be improved further.

LA is favoured for its simplicity, speed and accuracy. Since its discovery, in 2008, it was possible to detect communities in graphs with billions of nodes in a relatively timely manner. LA was compared to other algorithms for community detection, namely the algorithm of Wakita andTsurumi [24], of Pons and Latapy [22], and of Clauset, Newman and Moore [17] on graphs of sizes varying between 34 nodes and 77 edges to as much as 118 million nodes and 1 billion edges. The difference between the computing times of the previously stated algorithms grows with the size of the graphs and favours LA. In fact, it took 152 minutes for LA to detect the communities of the greatest graph whereas the computation time of the other algorithms was more than 24 hours. In terms of precision, LA was also the most precise one with slightly better modularities.

3 Development

This chapter will describe the design and implementation of the application which is composed of a representational state transfer (REST) application program interface (API) and a front-end (FE) web application.

3.1 API

Since the scraped pages of the dark-web were stored via ElasticSearch it was necessary to create a back-end application (BE) in order to perform various operations on the data-set before sending it to the FE. We decided to create a Python BE since the application had to be able to run on a UNIX system.

3.1.1 Technology overview

Python is a widely used interpreted programming language known for its readability and portability [1]. It is open-source and is considered to have an extensive documentation and community available. Another huge advantage is its popularity in the science community. Because of this there is a great amount of useful libraries for research purposes such as NetworkX[1] [9] or cylouvain [2] [3]. As we wanted to follow the REST architecture we decided to make use of the Django framework [8]. It is responsible for tasks such as running the server or managing web requests. Another advantage of Django is its Django REST framework (DRF). DRF offers a convenient way for creating restful endpoints and responses. [5]. Both frameworks are open-source again with extremely helpful documentation and community.

Because it takes approximately 60 seconds to retrieve about 90,000 pages from the database and circa 13 seconds to divide such a response into communities, caching had to be introduced. For that purpose Redis [13] is used. It is an open-source solution which we use as a key-value store. It supports basic data structures as values, e.g.

1. A library used for creating and working with graphs.
2. A library with a fast implementation of LA.

strings, numbers or sets but not custom objects. Since the API uses custom objects for both communities and pages, an object serializer had to be leveraged along with Redis. We decided not to write our own but to utilise the python pickle module [3] [10].

3.2 Front-end

For users to be able to see the data acquired from the BE in a reasonable way a FE application was created.

3.2.1 Technology overview

The probably most favoured programming language used for creating web applications [2] is called JavaScript (JS) [6]. It is an interpreted language supported by all modern browsers. It is open-source and as such disposes of a big community with splendid documentation. Because it is not strongly typed the code might be complicated to read or navigate. For this reason the FE was written in TypeScript (TS) [15] which is a superset of JS with the advantage of being typed. Both JS and TS come with a significant amount of tools used for implementing user interfaces (UI) in a clean and timely manner. One of the most favoured frameworks is React.js [12] which when used correctly results in readable code and improves performance by managing the re-rendering of page elements.

To be able to supply the FE with the needed data for user interactions a store had to be implemented. React.js and TS work extremely well with a state container called Redux [14] and because of that we decided to leverage it. It is centralized with only one store, easy to debug and again has a convenient documentation.

The doubtlessly most important part of this FE is the visualization of the graph built on the data sent from the BE. This feature is built using the react-d3-graph library [11] which is an implementation of the library d3.js [4] made more convenient for the use with React.js.

3. A module used for converting python objects to streams of bytes and vice versa.

3.2.2 User interface

After the application is loaded the UI is composed of a header with the name of the application "Dark web categorization" a loader and a sidebar in the right hand side with several inputs or buttons in a column as can be viewed in figure 3.1. At the very top a drop-down button is present which enables the user to filter the data. Underneath it an input field for filling in a search phrase with a submit button next to it for filtering out a single node or multiple nodes with which the search phrase matches is situated. The last element shown is an indicator of the current level the user is observing which at the beginning is zero.

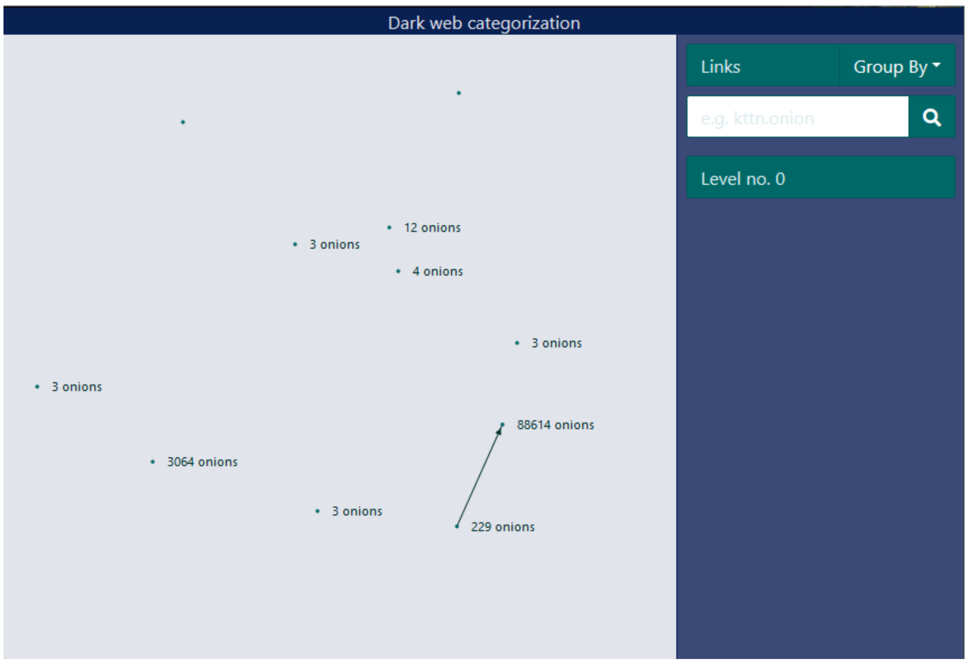

Figure 3.1: The basic view of the app with a selected community and its details.

The moment the data is retrieved from the BE the loader gets replaced for a graph which represents the communities the pages are partitioned into or the pages themselves and the links between them. Each node is displayed as a pie chart of categories of the pages belonging to the community represented by the node (CRN) or the page itself. There also might be a mock-community visible containing isolated nodes which cannot be zoomed into. It is possible for a level to depict communities and pages at once.

After single-clicking a node additional information is exampled in the sidebar.The details vary depending on whether the node is representing a community or a single page. The details of an individual page are as follows:

Url which also serves as a unique identificator of the page.

Category of the page. Each page belongs to one category.

Links to other pages displayed as a list of url addresses. There are up to ten links visible in the sidebar. The remaining links, if any, are downloadable in a text file.

Content of the page if it is available. If the content is too long to be disclosed in the sidebar it is again downloadable in a text file.

The details of a community consist of grouped information of its pages and include the following:

Category composition which is aggregated from the categories of all the pages of the community. Each category is represented by its name and the percentage of its relevance in the community.

Page url addresses (urls) belonging into the community. There are up to ten urls visible in the sidebar. The remaining pages, if any, are downloadable in a text file.

Urls count represents the number of all the pages belonging into the community.

The CRN needs to be double-clicked in order to zoom into a it to view its sub-communities. If the current level is higher than zero a button for zooming out appears next to the level indicator. It is shown in figure 3.2.

The user is able to zoom into the very last level (maximum level) where only individual pages are displayed. Each community may have a different maximum level, depending on the number of its pages and its structure.

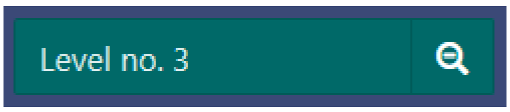

Figure 3.2: The level indicator with the zoom-out button. After the zoom-out button is clicked, the user is shown the communities of the previous level.

3.2.3 Implementation

The FE project consists of three folders and several configuration files. The folder *node_modules* contains imported libraries including React.js, Redux or d3. The next folder named *public* encloses a .ico file [4] and a html file which is the default entry point when the application is started. The last folder *src* contains the source code itself.

As previously mentioned, the FE is written in TS which has the advantage of readability and easy navigation. There are, however, also disadvantages and one of them is the need for every library used is a TS file to be typed. Fortunately typings are downloadable as modules for lots of popular libraries. In case a library has no ready-to-download typings own ones have to be written. In our case the typings for the library react-d3-graph was custom made and can be found in the folder *@types/react-d3-graph*. The file *commont.d.ts* holds types used heavily across the application e.g. Action. The reason for this is not to be liable to import every type every time it is used as types in this file are available for all other files.

Objects passed between functions in the application have also to be typed. There are three main models

The visual aspect is implemented using Less [7] which is a language extending CSS with improvements such as the possibility of using variables. The Less classes are divided into files according to the elements they are meant to modify and are placed into the folder *styles*.

The remaining folders each represent a different part of the UI and contain the functionality and the frame of it. The structure of the

4. A picture with the dimensions 16x16 pixels used by the browser to represent the web page or application. It is usually displayed in the tab in which the application is opened.

sub-folders is similar. Therefore it is sufficient to describe them as a whole. Folders named *utils* contain files with helper functions such as converters between server and client models. *Constants* contains folders with string constants or simple functions which return a string depending on the input of styles and routes. The rest of the folders represent some part of the Redux framework.

The most basic files which only include string constants are situated in the folders named *actionTypes* which are utilised as action types in actions which are simple objects containing a type and an optional payload. Actions themselves are returned by action creators (AC). AC are functions returning an action and can be found in folders called *actions*. They can be as simple as those present in the file *nodesActionCreators.ts*. But they can be more complicated such as the AC *fetchNodes.ts* and dispatch multiple simple ACs.

fetchNodes and the folder it is placed in share the same name. For easier testing purposes the main logic of this AC is put into a function which receives the simple ACs as dependencies. When this AC is called it first dispatches a simple AC to indicate the fetching has begun. After that an identificator (id) in order to create an error object in case of failure. Next, the fetching itself begins. The fetching in *fetchNodes* is realized with the library isomorphic-fetch. The fetch function of this library expects the first argument to be the url address to the resource. The second argument is an object describing further details of the request and is optional. For instance the request method, headers or the payload. If the request does not result in error the response status is checked. After the fetching is complete the node-mode is updated depending on whether communities were acquired or the maximum level was reached and the response contains only individual pages. Afterwards a success AC with the response is dispatched. If an error is caught during the fetching a failure AC with the previously created error id is dispatched.

The dispatching of actions enables the changing of the state via reducers situated in the *reducers* folders. The state is a single object for the whole application and is immutable [5]. A reducer is a pure func-

5. The object cannot be adjusted directly. Instead, a new modified object is returned and the original one stays unmodified.

tion [6] receiving the current state with the dispatched action as its arguments and returning the newly computed state. A reducer creates a new state only if it recognizes the type of the action. If not, the old state is returned. Because the state is immutable and every returned object would change the whole state the function *combineReducers* is leveraged. It expects an object containing reducers as its argument. It achieves performance adjustments so that the state is modified only when there were valid changes.

The folders *components* contain files with React components. They represent the skeleton of the UI with the specified behaviour. An important aspect

6. The return value of a pure function is only dependent on its input values. A pure function has no side effects.

4 Conclusion

4.1 Evaluation

4.2 Future work

The Leiden algorithm (LeA), which is another algorithm for community detection on large graphs, might be used in the future instead of LA. The paper in which LeA is described claims LA has a major flaw which is eliminated in LeA. It needs to be taken into consideration if it becomes adopted widely. [23]

Bibliography

[1] About python. `https://www.python.org/about/`. [cit. 2019-09-09].

[2] Active repositories per language on github. `https://githut.info/`. [cit. 2019-10-09].

[3] Cylouvain. `https://pypi.org/project/cylouvain/`. [cit. 2019-09-09].

[4] d3. `https://d3js.org/`. [cit. 2019-10-09].

[5] Django rest framework. `https://www.django-rest-framework.org/`. [cit. 2019-09-09].

[6] Javascript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript`. [cit. 2019-10-09].

[7] Less. `http://lesscss.org/`. [cit. 2019-12-09].

[8] Meet django. `https://www.django-rest-framework.org/`. [cit. 2019-09-09].

[9] Networkx - software for complex networks. `https://networkx.github.io/`. [cit. 2019-09-09].

[10] Pickle - python object serialization. `https://docs.python.org/3/library/pickle.html`. [cit. 2019-09-09].

[11] React-d3-graph. `https://goodguydaniel.com/react-d3-graph/docs/`. [cit. 2019-10-09].

[12] React.js. `https://reactjs.org/`. [cit. 2019-10-09].

[13] Redis. `https://redis.io/`. [cit. 2019-09-09].

[14] Redux. `https://redux.js.org/`. [cit. 2019-10-09].

[15] Typescript. `https://www.typescriptlang.org/`. [cit. 2019-10-09].

[16] Vincent Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics Theory and Experiment*, 2008, 04 2008. [cit. 2019-08-16].

[17] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70:066111, Dec 2004. [cit. 2019-08-28].

[18] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):4–8, Feb 2010. [cit. 2019-08-16].

[19] Jean-Loup Guillaume and Matthieu Latapy. The Web Graph: an Overview. In *Actes d'ALGOTEL'02 (Quatrièmes Rencontres Francophones sur les aspects Algorithmiques des Télécommunications)*, Mèze, France, 2002. [cit. 2019-08-16].

[20] Wenye Li and Dale Schuurmans. Modular community detection in networks. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1366–1371. AAAI Press, 2011. [cit. 2019-08-28].

[21] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006. [cit. 2019-08-16].

[22] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. *Journal of Graph Algorithms and Applications*, 10(2):191–218, 2006. [cit. 2019-08-28].

[23] Vincent A. Traag, Ludo Waltman, and Nees Jan van Eck. From louvain to leiden: guaranteeing well-connected communities. In *Scientific Reports*, 2018. [cit. 2019-08-16].

[24] Ken Wakita and Toshiyuki Tsurumi. Finding community structure in mega-scale social networks. *CoRR*, abs/cs/0702048, 2007. [cit. 2019-08-28].