Masaryk University
Faculty of Informatics

The Categorization of the Darkweb

Master's Thesis

Bc. Linda Hansliková

Brno, Spring 2020

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, Type-Script, 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 perform 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 nodes 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 [21]. 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 depicted web graph displays all its data at once without any labels or details. The result may be useful for viewing the internet as a whole. However, for our purposes this view was not sufficient. One of the requirements of this thesis was the possibility of the inspection of the relationships between nodes in more detail. Visible information about the pages was another requirement.

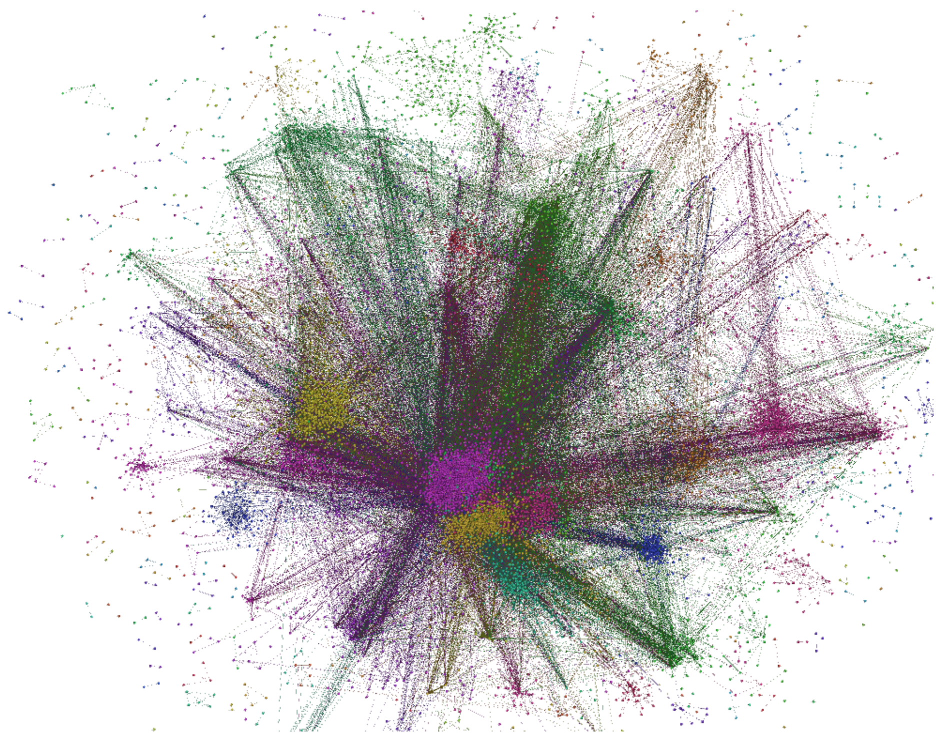

Figure 2.1: Web graph by Citeo [17].

The web graph therefore needed to be composed of a significantly smaller amount of nodes. This needed to be achieved in order for the view port not to be cluttered. This would result into obtaining the sought knowledge without hindrances. The majority of the nodes in our graph were not isolated [1]. The nodes were in fact part of a single connected component. As a result it was possible to partition the graph based on the density of its nodes into 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. Each subgraph of such a graph is a commu-

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

nity [20]. Each community can be portrayed as a meta node of the graph. This way the number of nodes in the graph can be reduced. The quality of such a partition can be measured using modularity [23]. L. Wenye and D. Schuurmans describe modularity in their work with the following words:

> 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 [22]

[23].

2.2 Louvain algorithm

A widely used algorithm for finding communities in graphs is LA [18]. 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 [26], of Pons and Latapy [24], and of Clauset, Newman and Moore [19] 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 describes 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

The scraped pages of the dark-web were stored in ElasticSearch. We created 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 readability and portability [1]. Python is open-source and is considered to have an extensive documentation and community available. Another advantage is the popularity of Python in the science community. There is a great amount of useful libraries for research purposes such as NetworkX[1] [10] or cylouvain[2] [3] because of this. 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 with helpful documentation and community.

It took approximately 60 seconds to retrieve about 90,000 pages from the database and circa 13 seconds to divide such a response into communities. We considered this wait time to be too long. Therefore we decided to cache the data of the first response. For that purpose Redis [14] was used. It is an open-source solution which we use as a

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

key-value store. It supports basic data structures[3] as values, but not custom objects. Since the API uses custom objects for communities, pages and links, an object serializer had to be leveraged along with Redis. We decided not to write our own but to utilise the Python pickle module[4] [11] (pickle). The reason behind this decision was pickle is convenient, simple, and fulfilled all our needs for serializing.

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 considerable community with convenient documentation. JS is not strongly typed and the code might therefore be complicated to read or navigate. For this reason the FE was written in TypeScript (TS) [16] which is a superset of JS with the advantage of being typed. There is a significant number of tools and libraries for implementing user interfaces (UI) in a clean and timely manner created for both JS and TS. One of such tools is the framework React.js [13] (React). React is one of the most favoured JS frameworks [9]. The advantages of using React are readable code and improved performance by managing the re-rendering of page elements.

To achieve a satisfying UX the app needed to be interactive and obtain new or modified data frequently. Repeated requests to the BE would mean longer wait time for the user. However, a proper mechanism for data-storing would present a convenient solution to this issue and make the requests unnecessary. A JS library which handles the app state and works well with TS and React is called Redux.js [15] (Redux). Redux is a single store approach. This ensures easy hydra-

3. Simple structures, e.g. strings, numbers or sets.
4. A module used for converting Python objects to streams of bytes and vice versa.

tion[5]. Redux also provides a custom set of TS typings and provides the developers with easy to use debugging tools. Another advantage of this library is the documentation.

The visualization of the web graph alone was realized using the react-d3-graph library [12] (RD3). RD3 is an implementation of the library d3.js [4] made more convenient for the use with React.js.

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. The sidebar contains several inputs and buttons in a column as can be viewed in figure 3.1. At the very top of the sidebar a drop-down button (DD) is present.

The DD sets the mode according to which the pages are grouped into communities. There are two modes available, link-mode and category-mode. If link-mode is selected, the pages are divided depending on the connections between them only. If category-mode is selected, the pages are divided into groups by categories. The pages in the groups are further divided into communities according to links between them, as if in link-mode.

For the purpose of filtering nodes according to a search phrase an input field with a submit button was placed underneath the DD. The last element shown is an indicator of the current level - how many times the user zoomed into a community. The indicator has a zoom-out button placed next to it.

The moment the data is retrieved from the BE the loader gets replaced for a graph. The graph-nodes represent either the communities the pages are partitioned into or the pages themselves. If the graph contains any isolated nodes (isolates) a mock-community is displayed containing all isolates. This community cannot be zoomed into. A community node is depicted as a pie chart. The individual sectors of the pie chart represent the categories of the pages belonging into the community [picture]. A single page is represented as a square [picture]. The colour of the square corresponds to the category of the page. It is possible for a level to depict communities and

5. The process of an object being provided with information

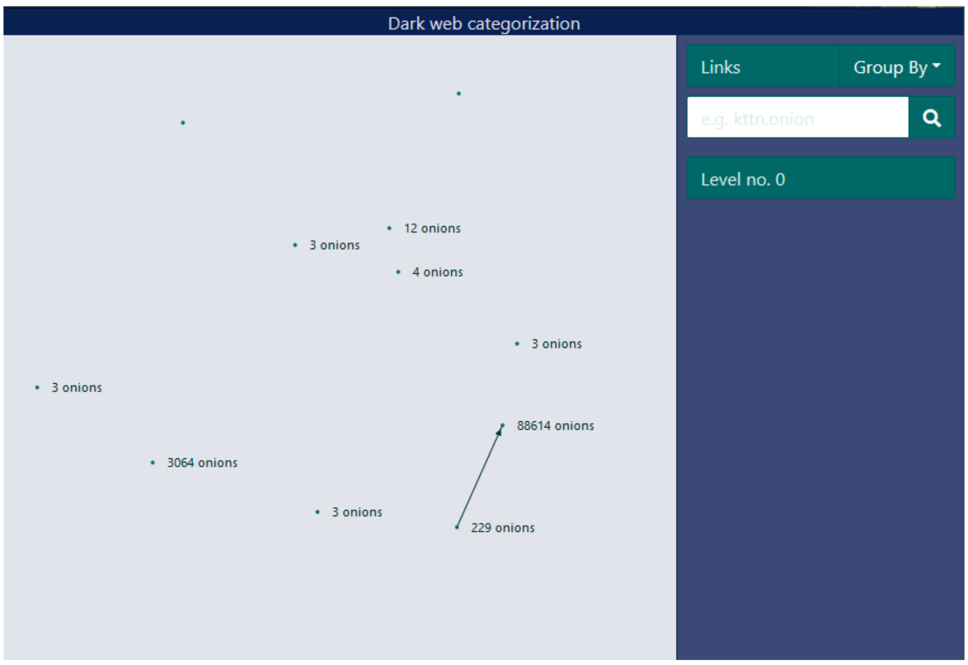

Figure 3.1: The basic view of the app with a selected community and its details.

pages at once. The links between the nodes visualize the links between pages or communities.

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 exactly 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 downloadable in a text file.

[picture]

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) of members belonging into the community. There are up to ten urls visible in the sidebar. The remaining urls, if any, are downloadable in a text file.

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

[picture]

In case of the need of further information a link, also present in the sidebar, can be clicked. If clicked, a pop-up window with detail-options is displayed. Details may include the title, category, links and page-content, depending on the user's selection. Urls are present by default. After selecting the needed options and clicking the download button, a text file with the desired information is downloaded. The page or community-members with their details are depicted in JSON format.

A graph node representing a community can be double clicked. After doing so, a new graph is shown. The data of this graph consists of the members of the clicked community. We call this process zooming. In case the parent community contains too many sub-communities, it is displayed as a single community-node. The zooming-in or -out of communities adjusts the zoom-level. This level is represented by a number and is visible below the node-filter. If the current zoom level is more than zero, i.e. at least one community-node was double clicked, a button for zooming out appears next to the level indicator. It is shown in figure 3.2.

If further zooming is not possible, the user reached the maximum level. Each community may have a different maximum level, depending on the number of its pages and its structure.

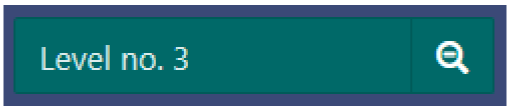

Figure 3.2: The level indicator with the zoom-out button. After the zoom-out button is clicked, the user is shown the communities of the previous level.

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[6] 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. One of them is the need of a TS file with the types (typing) for every used library. Typings for popular libraries are often downloadable as modules. If 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 were custom made. They 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. Types in this file are available without importing them to all files in the project.

Objects passed between functions also need to be typed. Those models are stored in the folder *models*. Each file contains one server model and one client model. The conversion between these models is conducted in specific helper functions. The advantage of this approach is the independence of client models from the BE 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 depending on the element they are meant to modify. These files were placed into the folder *styles*.

6. 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.

The remaining folders each represent a different part of the UI. The structure of their 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. 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*. These are utilised as action types in actions. An action is a 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. The purpose of an AC is to be injected into reducers, i.e. dispatch them.

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) is created. This id is used 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 of the resource. The second argument is an object describing further details of the request and is optional. Such an object may contain 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 a success AC with the acquired response is dispatched. If an error is caught during the fetching a failure AC is dispatched. The payload of this AC is an error-object with the id and error message if any.

The dispatching of actions enables the changing of the state via reducers situated in the *reducers* folders. The state is a single immutabe[7] object and is used in the whole application. A reducer is a pure func-

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

tion[8] receiving the current state and the dispatched action as its arguments. It then returns the newly computed state. A reducer creates a new state only if the type of the given action is recognized. If not, the previous state is returned unmodified. The state object received or returned by the reducer does not need to be the entire state (app-state). A reducer may be responsible only for a part of the app-state. However, the root reducer is responsible for the whole app-state.

The folders *components* contain files with React components. They define the skeleton of the UI with the specified behaviour. The folders *containers* hold files with React containers. A container is a file with access to the app-state. It is responsible for passing data to components.

8. 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) might be used in the future instead of LA. LeA is another algorithm for community detection on large graphs. The paper in which LeA is described claims LA has a major flaw which is eliminated in LeA [25]. It needs to be taken into consideration if it becomes adopted widely.

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] Most popular frameworks according to stack overflow. https://insights.stackoverflow.com/survey/2018#technology-_-frameworks-libraries-and-tools/. [cit. 2020-06-01].

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

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

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

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

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

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

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

[17] Web graph ny criteo. `https://labs.criteo.com/wp-content/uploads/2014/05/graphe_web-2.png?w=640/`. [cit. 2019-08-16].

[18] 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].

[19] 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].

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

[21] 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].

[22] 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].

[23] 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].

[24] 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].

[25] 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].

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