

Hasan Mikail Akkaya

17.05.2019

### **Web Crawling and Network Analysis for Social Science: A Case Study**

Every year hundreds of art exhibitions happen all over the world. Have you ever wondered who the artists that are featured in them are, what are the connections between these exhibitions, which exhibitions are the most popular or what venues host the most exhibitions? These questions could be answered by extensive surveys and field research, but there is an easier way to answer them. All these questions can be answered with the help from the data freely available on the internet. But the problem is that all this data is in an unstructured format scattered around the web. In this paper I will try to answer these questions using data openly available on the web and using network analysis techniques.

Social network analysis (SNA) has become a preferred method for social scientists over the last decade, when it comes to quantitative research. SNA encompasses a large set of methodological, statistical, and theoretical approaches that are developed for the analysis of relational data (Friemel 2017). With the internet age, the rise of “big data” has given social scientists a huge opportunity to collect data for their research. Internet is full of valuable data ready to be scraped and analyzed, but usually the researchers who are interested in that data are faced with a barrier caused by the lack of technical expertise. There are many tools developed to gather, clean, analyze and visualize data but many of them are out of reach for people without programming skills.

This paper intends to define a methodology to gather semi-structured data from internet, clean it, visualize and analyze it with the purpose of social research. The tools proposed for this task and the skills required to use them fall out of the scope of an ordinary social scientist, so this

paper takes that in to consideration and avoids all unnecessary technical jargon and explains the necessary tools from the bottom-up. While doing this, this paper will also provide a case-study by applying the proposed methodology to a real-life research topic.

There are 3 steps to be covered; (i) gathering the data (web-scraping), (ii) storing and cleaning the unstructured data, (iii) visualizing and analyzing the data. All the proposed tools are open-source and free to use for any research project. As this research endeavor is concerned with manipulating data, we need a programming language. Python 3 was chosen for this task, mainly because it is open-source and it has many powerful tools for scientists (“3.7.3 Documentation” n.d.). It is also a user-friendly programming language that focuses on code readability so it is a good starting point for non-programmers to start coding.

Web mining or web scraping is a powerful tool to gather publically available data from the internet. Many techniques and tools can be employed to achieve this task. The most reliable way is to use the API (application program interface) of the targeted websites. API's are a set of procedures and functions allowing a computer program to directly access the features or data of an another program. In the context of web mining, API's are a set of functions we can use to gather data from a website's database directly, without the need of a web browser. While API's are the most convenient way of scraping data, not all web sites provide an API to use their services or limit the API requests so it is not feasible for big scale data-mining. Other method of data mining from the web is web-crawling or scraping. Every time we enter a URL to a web-browser, the browser sends a request to that website's server and gets a response, which it displays for us to view the page we requested. (“How Web Browsers Work | How a Web Browser Works | InformIT” n.d.) This response is actually a set of instructions for the web browser, letting the browser know what will be displayed on the page or where it will be

displayed. These instructions are mainly written in HTML or XML. We can replicate this request and response process programmatically too, and extract the data we are interested in from that HTML or XML code. That is basically what web scraping is, requesting a webpage and pulling data from that page. But doing that one page a time would not be very time efficient. That's when web crawling comes in use. Web crawlers are programs that scrape websites but also find follow-up links in those websites to make further requests and continue gathering data.

Scrapy is a free, open-source and a fully-fledged web-crawling framework written in Python. ("Scrapy 1.6 Documentation — Scrapy 1.6.0 Documentation" n.d.) Its architecture is built around "spiders", which are self-contained crawlers that are given a set of instructions to scrape and parse websites. It has many built-in features like; generating feed exports to various file formats, support for selecting and extracting data from sources either by XPath or CSS expressions and also many community built add-ons like proxy or user-agent rotating tools. Scrapy can be run locally on any machine but also its architecture makes it very easy to deploy it on a cloud computing service. While being easy to work with, Scrapy also provides all the necessary tools to make a big scale crawler.

Even though Scrapy provides the tools to parse a website, which in basic terms means to go through the response and extract only the needed data, there are better tools to perform this task. BeautifulSoup 4 was chosen to do this task. BeautifulSoup is a Python library for pulling data out of HTML and XML files (Richardson, n.d.). It provides the tools for navigating, searching, and modifying the web page. So in our methodology, Scrapy will crawl the web and it will use BeautifulSoup 4 to extract the data we are interested in.

Scrapy architecture is set in a multistage scheme in order to maximize the efficiency of requesting websites, parsing the responses and manipulating or storing the items retrieved. When

a crawl is started the “spider” yields the initial requests and those requests are sent to the “scheduler”. The “scheduler” makes sure that requests and other processes are executed according to the settings file, in order to not overload the system and also the server the websites are requested from. When the time comes for a request it is sent to the “downloader”, which sends those requests to the internet and downloads the response. That response is sent back to the “Spider” and is parsed. Parsing is done according to the instructions provided under the parse methods in the spider code. There are two main jobs done while parsing, one is extracting the relevant data and sending it to the item pipeline, and the other is extracting the appropriate follow-up URL`s and yielding new requests for them. Every time a new item is yielded it goes through the “item pipeline”, which is a set of instructions how the items are to be processed. There can be multiple item types, for instance in our example there are three types of items artists, events and venues. All these three items contain different fields and are processed accordingly. Since we want to store the scraped data, in the end of the item pipeline we store these items in a file.

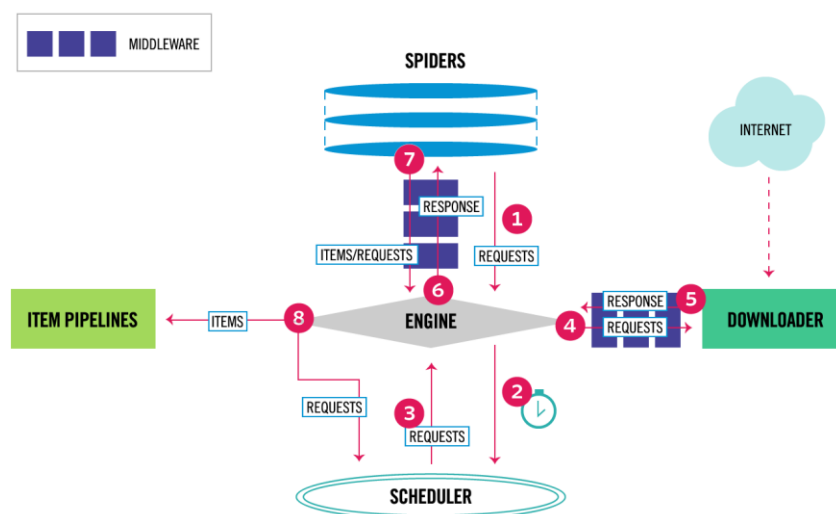


Fig. 1. Scrapy Architecture (“Architecture Overview - Scrapy 1.6.0 Documentation” n.d.)

There are many ways to store data using Scrapy with Python. Storing the data in CSV or JSON files is the easiest method, since Scrapy already has built in functions that allow us to write the items directly to a file with a single line of code. But since we want to gather and store a lot of data, it is more appropriate to use a database solution for storing data. There are many database structures we could use with Python, many of them are easy and free to use, but since our aim is to gather data for network analysis, it is much more convenient to use a graph database model. Graph database technology is an effective tool for modeling data when a focus on the relationship between entities is a driving force in the design of a data model (Silvescu, Caragea, and Atramentov, n.d.). Neo4J is one of many choices we have when it comes to graph databases but it was chosen because of its ease of use with Python and it is open-source. A Neo4J database instance is called a graph and it consists of nodes and relationships. In the context of graph theory, “a graph is a data structure composed of edges and vertices” (Rodriguez and Neubauer 2010). In the context of social science these are called nodes and edges, and in Neo4j these are called nodes and relationships. Every “node” has a unique id, a label and it can have endless properties with corresponding values. Nodes are connected to each other with “relationships” which are directed edges with a starting node, a type and an end node. Neo4J comes with in a software package called Neo4J Desktop, it consists of various tools and add-ons to create, manipulate and explore databases. To store data in a Neo4J database, first we need to create and then run that database. By default the database then is hosted on a local server with an address and we can access that database with a username and password either through a web-browser or other Neo4J add-ons. In order for our spider to communicate with Neo4J we need a driver. Neo4J Python Driver is the officially supported driver, which uses the binary protocol to connect to the database. But there are better alternatives. Py2neo is a client library, a community developed Python module which is a comprehensive toolkit for working with Neo4J within a Python application. (“The Py2neo v4 Handbook — The Py2neo v4 Handbook” n.d.) It has many functions to connect to a Neo4J database, to create nodes, relationships and commit them to a database. Py2Neo will be used in the items pipeline of our Scrapy spider, to store the nodes and relationships created from the

items. While our Scrapy crawler is running, all the retrieved data will be stored in the Neo4J graph database as soon as it is parsed, so if the spider is stopped at any time no data would be lost. While the crawl is in progress we can use Neo4J Browser or the tool provided in Neo4J Desktop called GraphXR to explore the data in real-time. After the Crawl is completed, all the data will be stored in our graph database. Using built in functions of Neo4J we can then export that data to a GRAPHML file, which is a format supported by Gephi, the software we will use for network analysis. Gephi is an open source network exploration and manipulation software (Bastian, Heymann, and Jacomy 2009). It is very powerful in terms of network visualization since it has many built in algorithms that are used to explore and visualize data with a network structure. Many types of data can be imported in to Gephi, one of the easy ways of doing it is by using GraphML, graph markup language. GraphML is an XML-based file-format that is used to store social networks (Brandes et al. n.d.). Other convenient method is to store and import network data in to Gephi is to use csv files. Gephi needs two types of files when working with csv, one is a node file and the other one is an edge file. There are some mandatory headers that these files have to contain so Gephi can properly import them. Gephi uses these columns to understand how the network elements relate to each other. Every node and edge can have as many other attributes as the network requires, but every node has to have an id and a label, which indicates the node's name and id number, and every edge a source id, target id and a weight, which indicates the starting node and the end node of that relationship and also the weight that edge has.

### **The Case Study**

To start scraping a website, first we need to understand how the website and its pages are structured so we can write an appropriate crawler for that website. There are two parts to this. One is to understand how the pages and their URL's are structured, so we know which pages to request and avoid unnecessary request which can slow down our crawl process or pollute our

data with irrelevant entries. Other part is to understand how the pages are structured themselves, in other words, which pages contain what kind of data, and where that data is stored in those pages. To understand these, first we visit the webpage and look for the data we are interested in. Mainly we are looking if the pages are categorized somehow. Page headers, menus, category pages or URL addresses help us with this task. For our example, in <https://kunstaspekte.art/> there are few types of pages. We are interested in the artist, venue and event network of biennials and other art related events so we try to find the web pages in this website that contain the info about these items, and try to find similarities between the pages of respective categories. In our example, if you surf the website for a while you will notice that the whole website is structured in an orderly manner, as many websites do. Every “person”, an artist, has their own page with a URL starting with [https://kunstaspekte.art/person/<artist\\_name>](https://kunstaspekte.art/person/<artist_name>) containing data about that artist, like their name or bio and also links to other pages that are related to them, like venues they had an exhibition in or events they participated. Every “venue”, a physical location, has its own page starting with [https://kunstaspekte.art/venue/<venue\\_name>](https://kunstaspekte.art/venue/<venue_name>) containing data about that venue, like its name, city, gps coordinates, street address, website, email and also links to other pages that are related to it, like venues they are cooperating with, events that were held there or artists that had an event in that venue. And lastly every “event” has its own page starting with [https://kunstaspekte.art/event/<event\\_name>](https://kunstaspekte.art/event/<event_name>) containing data about that event, like its name, start/end dates, press release and also links to other pages that are related to it, like the venue it was held in or participants of the event. As you can notice the whole website is structured similar to a network graph, there are nodes; artists, events and venues which all have some attributes related to them like name, URL, address, city, etc. and also all these nodes are some-how in

relation with each other, every venue has some events that were held there, every event has participants or every artist has venues they had an exhibition in or an event they appeared in.

After exploring the website, we need to decide which pages are to be scraped and what data from these pages we want to extract. There are three types of nodes in this website we are interested in, artists, events and venues. Every one of these “nodes” should have some attributes and lists containing other nodes they are connected to. We also need a way to identify these nodes, we could assign them a unique id number at scrape time but it would be easier to use some attribute of these nodes that is already unique and we are sure that all of the nodes will have that attribute. In our case we know that every node has its own web page, so we can use their URL`s as identifiers. So every node will have an attribute called “url” and we will store their web address in that field. This will also help us creating the relation lists, since the relations provided in this website are in form of URL links anyway. For instance for every artist page, there is a URL list of venues they had an exhibition in and a URL list of events they have participated in. If we use the URL`s of every page as their identifier we will be sure that every node has an unique identifier and also the relation lists we will use later to create “edges” contain a proper source identifier and a target identifier. After the inspection of what data we want to scrape and what pages we will scrape the data structure is as follows;



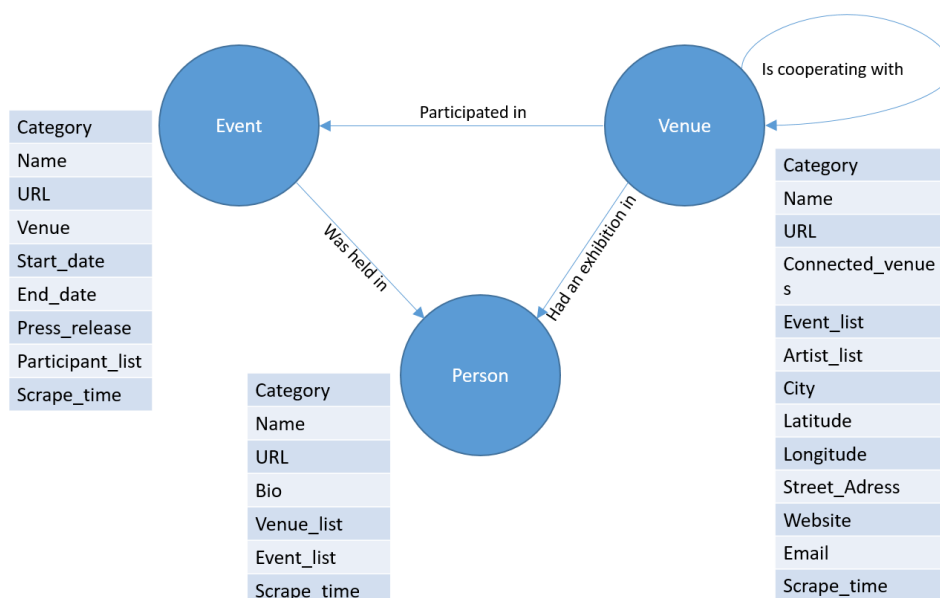


Fig. 2. Data structure of nodes and edges

After deciding what pages and what data we want to scrape from those pages we also need to decide how our crawler will navigate through the website. The way Scrapy works is that there has to be an initial request method of our spider class called “start\_requests” that specifies the URL or the URL list our spider will request initially, and then according to the instructions specified with the parse method that “start\_request” call uses, it will find other links to continue crawling. The spider will only be done when there are no new requests yielded so we need to make sure that our spider will scrape all the pages we are interested in and not skip any. We could always give the spider the homepage of the website as the initial URL and tell it to scrape all the links it finds, it would make sure that there are no pages we miss, but that would lead Scrapy to make lots of unnecessary requests which is not ideal. Bear in mind that Scrapy by default has a function that omits duplicate requests so even if we tell Scrapy to follow all the links it finds it will visit every page only once. In order to avoid making unnecessary requests we need to specify a more structured way for Scrapy to crawl the website. First of all we know that

we only are interested in event, artist and venue pages, so when yielding new request we will use only the links from the relationship lists we gather, and also to make sure, for every page we parse, first we will check if that page is an artist, event or an exhibition page, if it's not we will omit it. To start the crawl we will use the <https://kunstaspekte.art/artists-overview> page, where there is a list of every artist in this website. This makes sure that we don't omit any artist and we know that every artist has some connection to an event or venue so we know that all those nodes will be scraped as well. The way this is done will be explained in detail with the code itself.

## The Code

Since this is a Python project first we need to install Python for our system, it can be downloaded from the following link. <https://www.python.org/> It runs on all platforms, you just need to download the correct installation files for your OS. Python programs can be written on any text editor and if the file is saved with a <filename>.py extension, and it can be run using the command prompt but it is much easier to work with Python using an IDE (Integrated Development Environment). For this task we will use PyCharm Community Edition, which is a free development software that enables developers to write, test, debug and deploy code faster. ("Resources - Documentation | PyCharm" n.d.) It can be downloaded from the following link. <https://www.jetbrains.com/pycharm/download> PyCharm is not necessary to run Python scripts but it helps a lot, especially for new programmers when dealing with dependencies, libraries, environments and many aspects of Python. After installing Python and Pycharm, create a new project on PyCharm, this will pop-up a window with project settings. We need to name our project and also choose an environment for the project. PyCharm creates a virtual environment

with its own interpreter for every project to avoid loading modules on your system and it makes sure that your project is self-contained, all in one folder. We create a new project with virtual environment in a folder we specified and with Python 3 as the interpreter for that project. Then we will be welcomed with an empty project folder. First we need to install the modules we will use in this project, also called dependencies. For this we will use “pip”, it is a package-management system used to install and manage software packages written in Python. To run pip commands we use the terminal that is located in the bottom of the PyCharm interface. This terminal is just like the system terminal in the command prompt of the computer but it runs only in the virtual environment of the pycharm project we created. The modules we will need are Scrapy, bs4 and py2neo. To install these we run these commands in the Pycharm terminal

```
In [ ]: pip install scrapy
```

```
In [ ]: pip install py2neo
```

```
In [ ]: pip install bs4
```

Fig. 3. Dependency installation commands

These commands will tell PyCharm to collect and install the specified modules. After succesfull installation we will use a scrapy command in the terminal to start a new scrapy project with a name of our choice.

```
In [ ]: scrapy startproject project_name
```

Fig. 4. Initiating a new project

This will create few folders and python files necessary for our spider to work, file structure will be as follows;

```

project_name/
  scrapy.cfg          # deploy configuration file
  project_name/       # project's Python module, you'll import your code from here
    __init__.py
  items.py            # project items definition file
  middlewares.py      # project middlewares file
  pipelines.py        # project pipelines file
  settings.py         # project settings file
  spiders/            # a directory where you'll later put your spiders
    __init__.py

```

Fig. 5. Project directory structure

Now that our project is created we can start writing our spider. Scrapy is structured that there can be multiple spiders in one project but we will be using just one. To start writing our spider we need to create a new python file in the “project\_name/project\_name/spiders/” directory that will contain our spider script, let’s call it artist\_spider.py.

This artist\_spider.py is the place where our spider is going to live. It will have a class that defines our spider and that class will contain all the methods that spider will use. There is a mandatory variable of this spider class that should contain its name, which will be used to initiate the spider. Two methods are also mandatory, start\_requests method and parse method. Start\_requests contains instructions to be used when the crawls is started, in other words the initial URL’s to be scraped. The parse method contains the instructions to be used when parsing those initial requests.

For our project we will add few more methods;

- **start\_requests:**
  - This will run once when the spider is initialized.
  - It will use the artist list in the <https://kunstaspekte.art/artists-overview/<letter>>
  - This page is structured in a manner that if you type a letter in the end of it, it gives you a response containing a list of only links of all the artists starting with that letter
  - So this “start\_requests” method iterates through all the letters in alphabet and yields a response with a URL ending with that letter, using the “parse\_initial” parser
  - Example of the artist url list <https://kunstaspekte.art/artists-overview/a>
- **parse\_initial:**
  - This method is only used for the responses received from the “start\_requests” method requests
  - Since all those artist list pages have only links of artists, this parser collects all the URLs in the page and yields new requests using “parse” parser
- **parse:**
  - This method is an intermediate parser
  - It checks the header of the page, which is either “event” for event pages, “venue” for venue pages and “person” for artist pages
  - Then it send that page response to the appropriate parser
  - If the response header is not one of event, venue or person pages it dismisses that response

- **event\_parse:**
  - This parser is used for event page responses
  - It goes through the html response and finds all the relevant data such as;
    - Category
    - url
    - name
    - venue
    - start\_date
    - end\_date
    - press\_release
    - participant\_list
  - For all the data it uses “try” function which tries to find the data, if it cannot, it skips that field
  - For every time this parser is run it creates a new event item, fills it up with that events data and sends that item to the items pipeline
  - For every URL found in the participant\_list it also yields a new request using the “parse” parser
- **venue\_parse:**
  - This parser is used for venue page responses
  - It goes through the html response and finds all the relevant data such as;
    - Category
    - url
    - name
    - connected\_venue\_list
    - event\_list
    - artist\_list
    - city
    - coordinates
    - street address
    - website
    - email
  - For all the data it uses “try” function which tries to find the data, if it cannot, it skips that field
  - For every time this parser is run it creates a new venue item, fills it up with that events data and sends that item to the items pipeline
  - For every URL found in the connected\_venue\_list, event\_list and artist\_list it also yields a new request using the “parse” parser
- **person\_parse:**
  - This parser is used for person page responses
  - It goes through the html response and finds all the relevant data such as;
    - Category
    - url
    - name
    - bio
    - event\_list
    - venue\_list
  - For all the data it uses “try” function which tries to find the data, if it cannot, it skips that field
  - For every time this parser is run it creates a new person item, fills it up with that events data and sends that item to the items pipeline
  - For every URL found in the event\_list and venue\_list it also yields a new request using the “parse” parser

*The spider code is appended at the end of the document for reference. [Code 1.]*

When the crawl is running the spider sends all the items created to the items pipeline.

Every type of item has different attributes and is processed accordingly. The pipeline script of this crawler does few things. Initially when the crawl is started it initiates the database

connection with Neo4J. Later when the crawl is in process, for every item sent to the items pipeline it processes that item and sends it to the database. All the items created in the spider are formatted so the nodes are created right away. But since we are making a network graph we also need the relationship information between these nodes. This is the other process done in the items pipeline. The `process_item` method in the pipeline, checks every item for its category, creates a node for that item and stores the data of that item in that node. After that for every list in that item, it creates a relationship between that node and the entries in that list. We have used the URL's of the items as their identifiers, because we know that every item will have a unique URL, this comes handy in this stage because some of the target nodes of a relationship are not created yet, and we have only their URL. So we create an empty node with that URL, later when the spider scrapes that URL and sends its data to the items pipeline, our spider will update that node with the correct data.

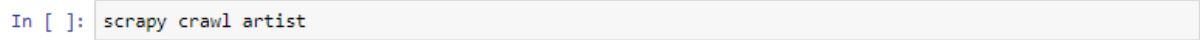
These are the main two stages of our spider that make it work. There are few more scripts that are necessary for its operation. `Middlewares.py` defines the operations that are done between the spider and pipeline, for our application we don't have to change anything in that script, but we can define various functions in this script that will be executed at specific times, like when the spider is initiated, every time a request is made or every time there is an exception error. `Items.py` is the script where we define the types of items used in our spider. For our case there are three types of Items, we create a class for each and define attributes that those items can have. This lets know our spider what data our items can store. Lastly there is the `settings.py` script, this script is basically used to adjust the settings of our spider. We can change the download delay, concurrent request amount, proxy settings, user agent settings and so on. All these scripts are created automatically when we initiate a new project and are filled with

necessary code. The only new file we have to create is our spider script which was explained above. Also we have to modify the items.py script to define our item types. And lastly we have to add instructions to the pipelines script, telling Scrapy what to do with those items and how to store them, as mentioned above.

*The pipelines.py code is appended at the end of the document for reference. [Code 2.]*

*The items.py code is appended at the end of the document for reference. [Code 3.]*

Now that we have written our spider, we can start crawling. To initiate a crawl we use the following command in the terminal.



```
In [ ]: scrapy crawl artist
```

Fig. 6. Command to initiate the crawl

This will initiate the spider named artist, in our case our only spider, and start crawling according to its instructions. Scrapy will go through every URL it finds and fits its criteria and will visit those pages, finding more pages. Depending on the website being scraped this can take few minutes or days. In our case, <https://kunstaspekte.art/> it took around 46 hours to scrape the whole website. Since it can be a long process, Scrapy allows for the crawl to be stopped and continued afterwards. To do this, you need to press ctrl+c keys once, and after Scrapy is done with the ongoing processes, it will halt the crawl. After our crawl, all the data is stored in the Neo4J database, ready to be analyzed.

Neo4J comes with a database browser where we can query for nodes and relationships to see what is in our database. There are also various helpful stats, like amounts and types of nodes and relationships. For our scrape I have collected, there are;

- 110809 event nodes
- 87383 person nodes
- 10717 venue nodes
- 92993 “has exhibition in” relationships

- 1938 “is cooperating with” relationships
- 591995 “participated in” relationships
- 110827 “took place in” relationships
- In total 208909 nodes and 797753 relationships

As you can tell, this is a huge amount of data that we collected. This illustrates the power of web scraping in data gathering. In less than 48 hours we collected more than 1 million data points that take up around 800MB of computer space. But unfortunately this ease of collecting data comes with a disadvantage. Our data set is now too big for us to analyze at once. So before analyzing them we will have to “clean” the data from the unnecessary nodes or relationships. The cleaning process can be done either in the Neo4J browser or we can export our database using a Python script using py2neo and dump the data we are not interested in while doing that. But in order to see the data and clean it accordingly we will do this process in Gephi.

To export our database to Gephi we will use an intermediary file format, GRAPHML, this file format is used to store and transfer graph based networks. Both Gephi and Neo4J supports this format. To export the data from Neo4J we will use an extension called APOC built for Neo4J that can be installed directly from our database settings menu in the Neo4J Browser. It consists of many useful network algorithms written in ready to use queries. After installing this extension we will use the following cypher query in the database browser that instructs our database to match all nodes and relationships from itself and store them in a GraphML file in a specified location;

```
1 // Export entire database
2 CALL apoc.export.graphml.all('/tmp/complete-graph.graphml', {useTypes:true, storeNodeIds:false})
```

Fig. 7. Command to export the database



After exporting our complete graph to a single file, we open that file in Gephi. Gephi is a powerful tool to explore, analyze and visualize network graphs. It has many built in features to classify or measure the attributes of network elements and also to visualize them in a pleasing and easy to understand way using those attributes. But Gephi struggles with graphs as big as the one we have, consisting of over a million of elements. That's why we will first analyze our dataset, create some attributes to the nodes we are interested in and delete or filter the nodes or relationships we are not interested in visualizing.

There are few attributes we want to calculate before deleting or filtering any network elements. First let's go over what we have in our network and which elements are worth visualizing. We don't want to visualize all of them, but we need the information that those elements we will not visualize are providing. The most often applied centrality measures are degree centrality, closeness centrality, and betweenness centrality. (Friemel 2017) First we want to calculate the degree of every node. Degree is the amount of incoming and outgoing relationships to that node. This attribute will be used to drive the size of the nodes, so we can visualize how well the node is connected to other nodes in our network. This will make the events that have more participants appear bigger, artists that have participated in more events appear bigger or venues that held more events appear bigger. A more sophisticated centrality measure is closeness which emphasizes the distance of a node to all others in the network by focusing on the geodesic distance from each node to all others. (Umadevi 2013) Lastly we will calculate the betweenness centrality. The betweenness centrality measures all the shortest paths between every pairs of nodes of the network and then count how many times a node is on a shortest path between two others (Grandjean, n.d.). These three measures are easily calculated in Gephi using the statistics tab. We will be using these measures to visualize our graph.

Next, since we have populated our dataset with the measures we wanted, we can remove the network elements we are not interested in visualizing. In the data laboratory tab, we have full control over the dataset we are working on. Here we can choose and delete all the nodes that have a degree of 2 or less. This will help us to lighten the amount of nodes to be visualized. In this window we can filter out and remove any data we want, but if we want to remove some data temporarily it is best practice to use the filter option in the overview tab. This allows us to choose attribute ranges and filter out some network elements without actually deleting them from our dataset.

Next step is spatialization, which is the step we will use some network algorithms to position our network elements on our map in a meaningful way. Layout of our network map should mean something, every element on the map should be in that position for a reason. We will achieve this with some algorithms designed just for this task. First one is the Fruchterman Reingold algorithm. (“Graph Drawing by Force-directed Placement - Fruchterman - 1991 - Software: Practice and Experience - Wiley Online Library” n.d.) This algorithm is called a force-directed layout algorithm, it treats all the networks elements as particles that have a either an attractive or a repulsive force acting on each other. While the algorithm runs, network elements push and pull on each other depending on their hierarchy in the network. This algorithm will cause the closer related nodes to come closer and the unrelated ones wonder away from each other. The other algorithm we will use is the Force Atlas 2, this is a similar a force-directed layout algorithm that treats the network elements as particles that have forces acting on each other. It gives us more settings like preventing overlapped nodes or dissuading hubs. Other layout algorithm we will use will be the Geo Layout algorithm. (Grandjean, n.d.) This one is particularly useful, since we have scraped the geographical data of the exhibition venues. Every

venue node, has a latitude and longitude attribute, and these attributes will be used to position our network elements on a world map.

### Works Cited

- Friemel, Thomas N. 2017. "Social Network Analysis." In *The International Encyclopedia of Communication Research Methods*, edited by Jörg Matthes, Christine S. Davis, and Robert F. Potter, 1–14. Hoboken, NJ, USA: John Wiley & Sons, Inc.  
<https://doi.org/10.1002/9781118901731.iecrm0235>.
- "3.7.3 Documentation." n.d. Accessed May 22, 2019. <https://docs.python.org/3/>.
- "How Web Browsers Work | How a Web Browser Works | InformIT." n.d. Accessed May 11, 2019. <http://www.informit.com/articles/article.aspx?p=680307>.
- "Scrapy 1.6 Documentation — Scrapy 1.6.0 Documentation." n.d. Accessed May 22, 2019.  
<https://docs.scrapy.org/en/latest/>.
- Richardson, Leonard. n.d. "Beautiful Soup Documentation," 68.
- "Scrapy\_architecture\_02.Png (1400×940)." n.d. Accessed May 22, 2019.  
[https://docs.scrapy.org/en/latest/\\_images/scrapy\\_architecture\\_02.png](https://docs.scrapy.org/en/latest/_images/scrapy_architecture_02.png).
- Silvescu, Adrian, Doina Caragea, and Anna Atramentov. n.d. *Graph Databases*.
- Rodriguez, Marko A., and Peter Neubauer. 2010. "Constructions from Dots and Lines." *Bulletin of the American Society for Information Science and Technology* 36 (6): 35–41.  
<https://doi.org/10.1002/bult.2010.1720360610>.
- Bastian, Mathieu, Sebastien Heymann, and Mathieu Jacomy. 2009. "Gephi: An Open Source Software for Exploring and Manipulating Networks." In *Third International AAAI*

*Conference on Weblogs and Social Media.*

<https://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154>.

Brandes, Ulrik, Markus Eiglsperger, I Herman, and M Kauffmann. n.d. “The GraphML File Format.” Accessed May 10, 2019. <http://graphml.graphdrawing.org/>.

“The Py2neo v4 Handbook — The Py2neo v4 Handbook.” n.d. Accessed May 22, 2019. <https://py2neo.org/v4/#>.

“Resources - Documentation | PyCharm.” n.d. JetBrains. Accessed May 22, 2019. <https://www.jetbrains.com/pycharm/documentation/>.

Umadevi, Dr V. 2013. “CASE STUDY – CENTRALITY MEASURE ANALYSIS ON CO-AUTHORSHIP NETWORK.” *Journal of Global Research in Computer Science* 4 (1): 67–70.

“Graph Drawing by Force-directed Placement - Fruchterman - 1991 - Software: Practice and Experience - Wiley Online Library.” n.d. Accessed May 22, 2019. <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211102>.

Grandjean, Martin. n.d. “Gephi Introduction,” 12.

```

import scrapy
from ..items import PersonItem, VenueItem, EventItem
from bs4 import BeautifulSoup
import datetime

class ArtistSpider(scrapy.Spider):
    name = "artist" # name of the spider
    domain = 'https://kunstaspekte.art' # this is optional, but makes sure spider
    stays on this website

    def start_requests(self): # initial request method
        page_list = {'0', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
        'm',
        'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'}
        url = 'https://kunstaspekte.art/artists-overview/'
        for char in page_list: # this for loop iterates through all letters and
        yields a new request for that page
            yield scrapy.Request(url=url + char, callback=self.parse_initial)

    def parse_initial(self, response): # parser for the initial start_requests method

        raw = response.body # raw response from the website
        soup = BeautifulSoup(raw, 'html.parser') # initiate bs4 parser object
        url_list = soup.find_all('a') # find all links in the page

        for urls in url_list: # this for loop iterates through all the links and
        yields a request for that page
            yield scrapy.Request(url=self.domain + urls['href'], callback=self.parse)

    def parse(self, response): # main parser that decided what kind of page it is

        raw = response.body # raw response from the website
        soup = BeautifulSoup(raw, 'html.parser') # initiate bs4 parser object

        try: # try to find page type from the main header
            page_type = soup.find(class_='content-heading').find('h3').get_text(' ',
            strip=True)
        except AttributeError:
            page_type = 0
            print("An exception occurred")

        if page_type == 'artist / curator':
            yield from self.person_parse(response) # if page is a person page send
            the data to person_parse parser
            print('person page')
        elif page_type == 'venue':
            yield from self.venue_parse(response) # if page is a venue page send the
            data to venue_parse parser
            print('venue page')
        elif page_type == 'exhibition':
            yield from self.event_parse(response) # if page is a event page send the
            data to event_parse parser
            print('event page')
        else:
            print('non valid page')

    def event_parse(self, response): # event parser that extracts event
    dataprint('event scraper')
    raw = response.body

```

```

soup = BeautifulSoup(raw, 'html.parser')

event = EventItem() # creates an empty event Item

# extract relevant event data

category = 'event'
url = response.url

name = ''
try:
    name = soup.find('h1').get_text(' ', strip=True)
except Exception as ex:
    print(ex)

venue = ''
try:
    venue = self.domain + soup.find(class_='venue-
module').find('h3').find('a')['href']
except Exception as ex:
    print(ex)

start_date = ''
try:
    start_date = soup.find(class_='begins').get_text(' ', strip=True)
except Exception as ex:
    print(ex)

end_date = ''
try:
    end_date = soup.find(class_='ends').get_text(' ', strip=True)
except Exception as ex:
    print(ex)

press_release = ''
try:
    text = soup.find(id='textblock').find_all('p')
    for i in text:
        press_release = press_release + i.get_text('\n', strip=True)
except Exception as ex:
    print(ex)

participant_list = []
try:
    participants = soup.find_all(class_='artist-list')
    for i in participants:
        links = i.find_all('a')
        for j in links:
            participant_list.append(self.domain + j['href'])
except Exception as ex:
    print(ex)

# store the collected data in the event Item
event['category'] = category
event['url'] = url
event['name'] = name
event['venue'] = venue
event['start_date'] = start_date
event['end_date'] = end_date
event['press_release'] = press_release
event['participant_list'] = participant_list
event['scrape_time'] = datetime.datetime.now()

```

```

        yield scrapy.Request(url=venue, callback=self.parse) # yield a request for
the connected venue

    for url in participant_list: # yield a request for all participants
        yield scrapy.Request(url=url, callback=self.parse)

    yield event # send the event Item to the items pipeline

def venue_parse(self, response):
    raw = response.body
    soup = BeautifulSoup(raw, 'html.parser')

    venue = VenueItem() # creates an empty venue Item

    # extract relevant event data
    category = 'venue'
    url = response.url

    name = ''
    try:
        name = soup.find('h1').get_text(' ', strip=True)
    except Exception as ex:
        print(ex)

    connected_venue_list = []
    try:
        dependencies = soup.find(id='texts').find_all('a')
        for i in dependencies:
            connected_venue_list.append(self.domain + i['href'])
    except Exception as ex:
        print(ex)

    event_list = []
    try:
        exhibition = soup.find_all(class_='exhib-title')
        for links in exhibition:
            event_list.append(self.domain + links['href'])
    except Exception as ex:
        print(ex)

    artist_list = []
    try:
        artists = soup.find(class_='artist-list').find_all('a')
        for links in artists:
            artist_list.append(self.domain + links['href'])
    except Exception as ex:
        print(ex)

    city = ''
    coordinates = ['', '']
    street_address = ''
    website = ''
    mail = ''
    try:
        address = soup.find('div', class_='address')
        city = address.find('p').find('a').get_text(' ', strip=True)
        coordinates = address['data-latlon'].split(',')
        street_address = address.find('p').get_text(' ', strip=True)
        website = address.find(class_='website')['href']
        mail = address.find(class_='mail')['href'].split(':')[1]
    except Exception as ex:
        print(ex)

```

```

# store the collected data in the venue Item
venue['category'] = category
venue['url'] = url
venue['name'] = name
venue['connected_venue_list'] = connected_venue_list
venue['event_list'] = event_list
venue['artist_list'] = artist_list
venue['city'] = city
venue['latitude'] = coordinates[0]
venue['longitude'] = coordinates[1]
venue['street_address'] = street_address
venue['website'] = website
venue['email'] = mail
venue['scrape_time'] = datetime.datetime.now()

for url in connected_venue_list: # yield a request for all connected venues
    yield scrapy.Request(url=url, callback=self.parse)

for url in event_list: # yield a request for all events
    yield scrapy.Request(url=url, callback=self.parse)

for url in artist_list: # yield a request for all artists
    yield scrapy.Request(url=url, callback=self.parse)

yield venue # send the venue Item to the items pipeline

def person_parse(self, response):
    raw = response.body
    soup = BeautifulSoup(raw, 'html.parser')

    person = PersonItem() # creates an empty person Item

    # extract relevant event data
    category = 'person'
    url = response.url

    name = ''
    try:
        name = soup.find('h1').get_text(' ', strip=True)
    except Exception as ex:
        print(ex)

    bio = ''
    try:
        for i in soup.find_all('p'):
            bio = bio + ' ' + i.get_text(' ', strip=True)
    except Exception as ex:
        print(ex)

    venue_list = []
    try:
        collections = soup.find('div', class_='collections').find_all('a')
        for collection in collections:
            venue_list.append(self.domain + collection['href'])
    except Exception as ex:
        print(ex)

    try:
        galleries = soup.find('div', class_='galleries').find_all('a')
        for gallery in galleries:
            venue_list.append(self.domain + gallery['href'])
    except Exception as ex:
        print(ex)

```



```

event_list = []
try:
    events = soup.findAll(class_='exhib-title')
    for event in events:
        event_list.append(self.domain + event['href'])
except Exception as ex:
    print(ex)

# store the collected data in the person Item
person['category'] = category
person['url'] = url
person['name'] = name
person['bio'] = bio
person['venue_list'] = venue_list
person['event_list'] = event_list
person['scrape_time'] = datetime.datetime.now()

for url in venue_list: # yield a request for all venues
    yield scrapy.Request(url=url, callback=self.parse)

for url in event_list: # yield a request for all events
    yield scrapy.Request(url=url, callback=self.parse)

yield person # send the person Item to the items pipeline

```

*[Code 2.] The pipelines.py code*

```

from py2neo import Node, Relationship
from py2neo import Graph

class ArtistPipeline(object): # definition of the pipeline class

    person_list = []
    venue_list = []
    event_list = []

    def open_spider(self, spider):
        self.graph = Graph("bolt://localhost:7687", auth=('neo4j', '123456')) #
connect to graph
        self.graph.schema.create_uniqueness_constraint("person", "url")
        self.graph.schema.create_uniqueness_constraint("venue", "url")
        self.graph.schema.create_uniqueness_constraint("event", "url")

    def close_spider(self, spider):
        pass

    def __init__(self):
        pass

    def process_item(self, item, spider): # item process definition

        transaction = self.graph.begin() # create a graph object
        item_type = item.get('category') # get the category of the object

        if item_type == 'person': # if the object is a person
            person = Node(item.get('category'), # fill the fields with appropriate
values

```

```

        url=item.get('url'),
        name=item.get('name'),
        bio=item.get('bio'),
        scrape_time=item.get('scrape_time'))
person.__primarylabel__ = 'person'
person.__primarykey__ = 'url'
sub_graph = person

    for venues in item.get('venue_list'):

        venue = self.graph.evaluate('match (x:venue {url:{param}}) return x',
param=venues)

        if venue is not None and len(venue) > 1:
            pass
        else:
            venue = Node('venue', url=venues)

            venue.__primarylabel__ = 'venue'
            venue.__primarykey__ = 'url'
            sub_graph = sub_graph | venue
            sub_graph = sub_graph | Relationship(person, 'has an exhibition in',
venue)

    for events in item.get('event_list'):
        event = self.graph.evaluate('match (x:event {url:{param}}) return x',
param=events)

        if event is not None and len(event) > 1:
            pass
        else:
            event = Node('event', url=events)

            event.__primarylabel__ = 'event'
            event.__primarykey__ = 'url'
            sub_graph = sub_graph | event
            sub_graph = sub_graph | Relationship(person, 'participated in', event)

    transaction.merge(sub_graph)
    print('person item')

elif item_type == 'event':
    event = Node(item.get('category'),
        url=item.get('url'),
        name=item.get('name'),
        start_date=item.get('start_date'),
        end_date=item.get('end_date'),
        press_release=item.get('press_release'),
        scrape_time=item.get('scrape_time'))

    event.__primarylabel__ = 'event'
    event.__primarykey__ = 'url'
    sub_graph = event
    venue_url = item.get('venue')

    venue = self.graph.evaluate('match (x:venue {url:{param}}) return x',
param=venue_url)

    if venue is not None and len(venue) > 1:
        pass
    else:
        venue = Node('venue', url=item.get('venue'))

        venue.__primarylabel__ = 'venue'
        venue.__primarykey__ = 'url'

    sub_graph = sub_graph | venue

```

```

        sub_graph = sub_graph | Relationship(event, 'took place in', venue)

        for participants in item.get('participant_list'):
            participant = self.graph.evaluate('match (x:person {url:{param}})')
            return x', param=participants)
            if participant is not None and len(participant) > 1:
                pass
            else:
                participant = Node('person', url=participants)

            participant.__primarylabel__ = 'person'
            participant.__primarykey__ = 'url'
            sub_graph = sub_graph | participant
            sub_graph = sub_graph | Relationship(participant, 'participated in',
event)

        transaction.merge(sub_graph)
        print('event item')

    elif item_type == 'venue':
        venue = Node(item.get('category'),
            url=item.get('url'),
            name=item.get('name'),
            city=item.get('city'),
            latitude=item.get('latitude'),
            longitude=item.get('longitude'),
            street_address=item.get('street_address'),
            website=item.get('website'),
            email=item.get('email'),
            scrape_time=item.get('scrape_time'))
        venue.__primarylabel__ = 'venue'
        venue.__primarykey__ = 'url'
        sub_graph = venue

        for venues in item.get('connected_venue_list'):
            connected_venue = self.graph.evaluate('match (x:venue {url:{param}})')
            return x', param=venues)
            if connected_venue is not None and len(connected_venue) > 1:
                pass
            else:
                connected_venue = Node('venue', url=venues)

            connected_venue.__primarylabel__ = 'venue'
            connected_venue.__primarykey__ = 'url'
            sub_graph = sub_graph | connected_venue
            sub_graph = sub_graph | Relationship(connected_venue, 'is cooperating
with', venue)

        for events in item.get('event_list'):
            event = self.graph.evaluate('match (x:event {url:{param}}) return x',
param=events)
            if event is not None and len(event) > 1:
                pass
            else:
                event = Node('event', url=events)

            event.__primarylabel__ = 'event'
            event.__primarykey__ = 'url'
            sub_graph = sub_graph | event
            sub_graph = sub_graph | Relationship(event, 'took place in', venue)

        for artists in item.get('artist_list'):

```

```

        artist = self.graph.evaluate('match (x:person {url:{param}}) return
x', param=artists)
        if artist is not None and len(artist) > 1:
            pass
        else:
            artist = Node('person', url=artists)

        artist.__primarylabel__ = 'person'
        artist.__primarykey__ = 'url'
        sub_graph = sub_graph | artist
        sub_graph = sub_graph | Relationship(artist, 'has an exhibition in',
venue)

        transaction.merge(sub_graph)
        print('venue item')

    else:
        print('invalid item')

    transaction.commit()
    return item

```

[Code 3.] The items.py code

```

import scrapy

class PersonItem(scrapy.Item):
    category = scrapy.Field()
    url = scrapy.Field()
    name = scrapy.Field()
    bio = scrapy.Field()
    venue_list = scrapy.Field()
    event_list = scrapy.Field()
    scrape_time = scrapy.Field()

class EventItem(scrapy.Item):
    category = scrapy.Field()
    url = scrapy.Field()
    name = scrapy.Field()
    venue = scrapy.Field()
    start_date = scrapy.Field()
    end_date = scrapy.Field()
    press_release = scrapy.Field()
    participant_list = scrapy.Field()
    scrape_time = scrapy.Field()

class VenueItem(scrapy.Item):
    category = scrapy.Field()
    url = scrapy.Field()
    name = scrapy.Field()
    connected_venue_list = scrapy.Field()
    event_list = scrapy.Field()
    artist_list = scrapy.Field()
    city = scrapy.Field()
    latitude = scrapy.Field()
    longitude = scrapy.Field()

```

```
street_address = scrapy.Field()
website = scrapy.Field()
email = scrapy.Field()
scrape_time = scrapy.Field()
```