Delft University of Technology

Bachelor Graduation Project

Initial Research Report

UrbanSearch

Authors:
Tom Brunner
Marko Mališ
Gijs Reichert
Piet van Agtmaal

Supervisor:
Claudia Hauff
Clients:
Evert Meijers
Antoine Peris

June 20, 2017

TU Delft

Abstract

It is hard if not impossible to measure the strength of relationships between cities using existing technologies. Because of this, it remains uncertain how exactly economic growth is affected by urbanisation. The universally accepted explanation is that only increasing size of cities affects economic growth [23]. In this report, we develop a methodology that allows for determining intercity relationship strengths, using open data. For this, we evaluate whether graph databases like Neo4j [21] or document search engines such as ElasticSearch [9] are best suited and describe machine learning algorithms for categorising data based on the co-occurrence of city names. Additionally, we present visualisation techniques to be able to intuitively analyse the results.

Keywords: urban, relation strength, data mining, graph, classification, filtering

Contents

1 Introduction

With the development of future cities in mind, the interest in city networks has grown over the years. However, appropriate information on how cities are connected and the strength of these connections is hard to find. According to Short et al. comparative statistics are not easily available and common assertions are repeated [29]. Although more research was published since then, for example the work of DeRudder et al. [7], there is still a lot of information missing on relationships between cities. In order to get more insight into these relationships, there is a need for complex data analysis.

The huge amount of textual data generated online and the numerous historic archives, such as Delpher[1] and the British Newspaper Archive[2], are great sources of information on social and economic behaviours. The clients hypothesis is that advanced text mining on newspapers and web pages containing city names would allow for a better understanding of the role of information in shaping urban systems. Similar to research efforts in other domains, such as financial trade [24] and sales forecasting [37], the clients wish is to develop an application that captures urban-urban interactions. These interactions are retrieved from information corpora through the co-occurrence of geographical names in textual data. An example of how one could try to achieve this using the Google search Engine[3] is `"Rotterdam + Amsterdam " OR "Amsterdam + Rotterdam"`, which searches for the co-occurrence of Amsterdam and Rotterdam. However, manually processing all results a search engine yields is not feasible, because one would have to read each page to determine which types of relationships the page contains. An application should process all the pages that contain co-occurrences of cities to determine what type of relations, for example transportation or leisure, between cities can be extracted from the document. Thus, we will answer the following question: how can the strength of relationships between cities be extracted and visualised from open data using software?

First, we discuss related work in section 2. Second, we identify the requirements for a solution to the problem and discuss issues that might arise in section 3. Third, we develop a methodology for a framework that satisfies the requirements and tackles the issues in section 4. In the fifth section we discuss evaluating the system. We conclude in section 6 with the results of our research.

2 Related Work

In the 1960s, progress in the knowledge of urban networks was made feasible by the rapidly increasing availability of useful data [35].
One of the most common methods used is the interlocking network model (INM) [33]. This model assumes cities have a flow of knowledge connection if there are offices of the same company in those cities. The biggest problem with this is that it is very limited. It only includes one relation type and it is disputable whether this is a good measurement for the relation.

Since one decade, we are facing a new revolution in data production and processing. Developing innovative methods to extract relevant information from Big Data could allow important progress in the knowledge of the interurban relationships.

When looking at digital data there are two different approaches: the cyberspace and the cyberplace [8]. The cyberplace measures relations by using the infrastructure of the internet, for example internet hyperlinks, the structure of search engines and email traffic. Most research on this has been done on the 'backbone' of the internet made of cables and routers [4, 11].

The cyberspace method focuses on the virtual communication of people through connected devices. One approach is by registering and mapping the number of pages indexed by search engines for queries containing the names of two cities [8, 16, 17]. In 2010 Brunn et al. evaluated the linkage between two cities by entering those cities into a search query followed by key words such as "global financial crisis" or "climate change" and registering the number of pages indexed [3].

[1] `http://www.delpher.nl`
[2] `http://www.britishnewspaperarchive.co.uk/`
[3] `https://www.google.com`

To improve the textual analysis on websites and search engine queries to find digital links between cities, a more systematic approach is needed. A piece of software designed specifically for this purpose should automatically find predefined relations between cities and their strength, by using all pages available from search engines. In the following chapter we will investigate the requirements for such a program.

3 Requirement Analysis

In this section, we first present user stories that were created together with the client. Next, we define the design goals. Then, we list the requirements which followed from the user stories and which the application should meet. To do so, we use the MoSCoW method [5] as a prioritisation technique. Lastly, we discuss the design decisions that follow from the design goals and the requirements.

3.1 User Stories

Together with the client, several user stories are identified and listed below:

1. As a user, I want to be able to see all the identified relations between all cities, so that I can reason about interesting patterns.
2. As a user, I want to be able to access the processed data in an excel file. I want this to be available per relation type and as a total of all relations, so that I can apply my own models on the data when I want to.
3. As a user, I want to be able to see relation strengths. Which can be expressed by counting the relations
4. As a user, I want to be able to (de)select cities in the user interface, so that I can create a network of cities connected with relations. Where a network of cities will be the cities on a map and the different types of relationships such as Transportation will be the connecting edges between cities.
5. As a user, I want to be able to (de)select relations between cities in the user interface, so that I can inspect only the relations I am interested in at that time. For example, as a user I might only be interested in the Transportation relationship between Amsterdam and Rotterdam.
6. As a user, I want to be able to change the colours associated with the different relation types, so that I can adjust the styling to my own preferences.
7. As a user, I want to be able to export an image of the map that I composed in the user interface so that I can use it for presentations, papers or educational purposes

3.2 Design Goals

The high-level design goals for this project have been provided by the client. These serve as a guideline to determine the priority label of the specific requirements as defined in section 3.3. The design goals are listed below, ordered by priority.

credible The results of the project will be used in research on intercity relations. Therefore, the results must be reliable and verifiable. This means that the application should produce the same results given the same input and it should be possible to manually access the input to verify the output of the application.

understandable The results of the application should be visually understandable, in order to make it easy for the client to deduce conclusions.

scalable During the project a TU Delft server will be used with a limited amount of resources. Therefore only .nl pages will be used as input to limit the amount of data storage and processing power needed. However, allowing for investigating other domains would greatly help the client in a later stadium, which means that the system would have to be scalable where possible. For example, using a dedicated database which can be spread across clusters.

plugable It might be interesting for the user to let the application perform analysis on different data sets without the need of a developer. So if possible within the time constraints the application should be able to use any form of textual input data.

exportable Besides making the results available visually, all the relevant numeric data should also be exportable, for example as a matrix.

fast development Because of the time constraints of the project we need a fast development cycle. As a result of that, choices regarding tools, applications and programming languages are to be made with the time constraint taken into account.

3.3 Product Requirements

As mentioned in the introduction of section 3 we will be using he MoSCoW method prioritisation technique. Four levels of priority are defined: must have, should have, could have and would have (also known as would like). We also differentiate between functional and non-functional requirements.

3.3.1 Must Have

Requirements labelled as "must have" are key to the minimal performance of the application If they are not met, the application can be considered a failure.

1. Data that is of relevance for the UrbanSearch project, should be mined from the Common Crawl web corpus (see section4.1.1) and stored for further processing/access.
2. There has to be a way to extract the collected data based on relations between cities/nodes.
3. A machine learning algorithm should analyse the collected data and label different types of relations that are important for intercity relations.
4. A front-end should be built for the project. This front-end should visualise basic relations and statistics and can be used for presentations and educational purposes.
5. Several statistically important aspects of intercity relations should be extracted from the data set. These statistics should be easily accessible and visualised to the end user. Furthermore, it should be easy to extend or update the list of statistics that are associated with a relation.

3.3.2 Should Have

"Should have" requirements are those that greatly improve system performance and/or usability but might not fit in the available development time.

1. Relations between cities should be accessible hierarchically. This means that there is the possibility to explore a relation and, provided that this relation has sub-types associated with it, the relation can be expanded in the different sub-types of the relation.
2. It should be possible to train the machine-learning algorithm by feeding it labeled data.
3. It should be possible to add data sets with more than 1 million documents on which the system can perform its data mining routines. This way a data set can be created that contains potentially interesting information for intercity relations.
4. The application should be able to deal with the fact that the same city can have different names in different languages/dialects. It should still be able to extract and group relevant data correctly (e.g. 'The Hague' and 'Den Haag' should be viewed as the same city).

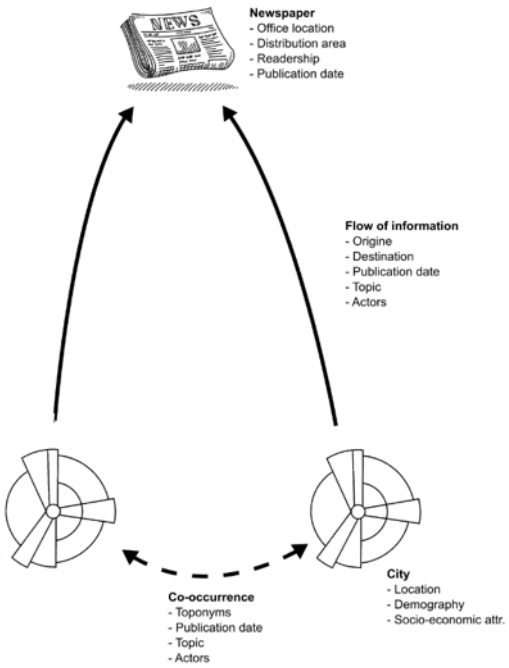

Figure 1: Solid lines represent simple information flows, whereas the dashed line is a complex flow of information.

3.3.3 Could Have

Requirements labelled as "could have" are useful and should be included in the system if time and resources permit.

1. The system should use Delpher to characterise relationships between a region and cities outside that region based on newspapers in those regions. For example, the local newspaper of the province Gelderland writing about the city of Alkmaar. These relationships are either simple or complex information flows. An information flow in this context is shown in figure 1.
2. The relations that are extracted from the data by the machine learning algorithm have to be visualised in a way that makes it easy to compare the different relations for the end user. For example, a split-screen comparison in the user interface or an export of graphs comparing selected relations.

3.3.4 Would Like

"Would like" requirements have been agreed upon to be not important to include within the current time schedule. However, they can be included in future releases.

1. The application would be able to show all connections of all places on the map at the same time.
2. Using data from top-level domains other than `.nl`.

3.4 Design Decisions

To be able to have a fast development cycle and leverage our experience we chose to develop the application using Python. We plan to not only test the code we deliver thoroughly, but also to cross-validate the obtained results. The specifics of this validation protocol will be discussed in section 5.2.1.

4 Framework and Tools

In this section, we gradually develop the framework. First, we decide what data source to use initially and describe how to use the data. Second, we present the method of choice to perform initial data filtering. Third, we agree on which data storage and ingestion to use. Fourth, we provide a methodology to group the data. Fifth, we decide how users will interact with the obtained results. Last, we select the visualisation tools to be used.

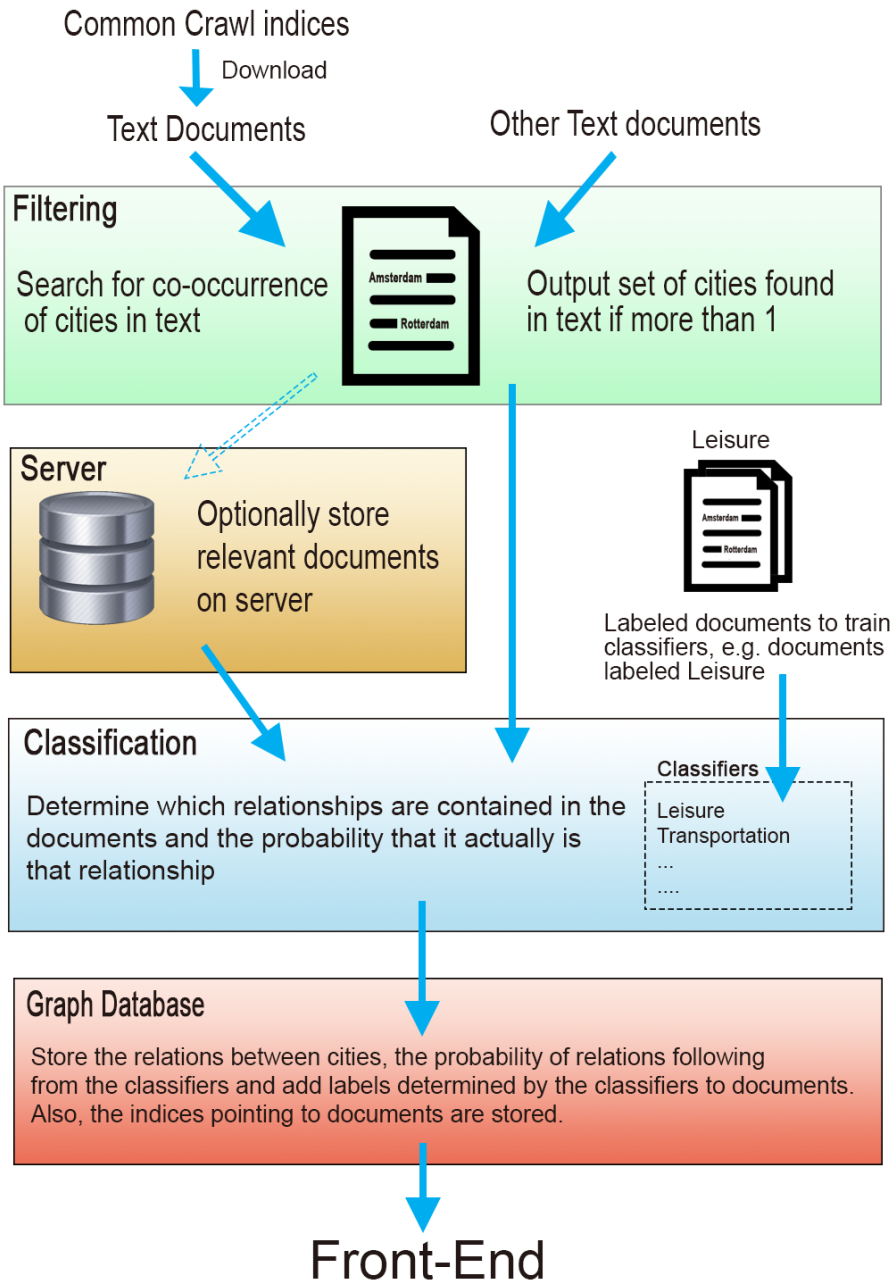

Figure 2: Simplified overview of the system

4.1 Gathering the Data

As explained in section 3.2, data sources should be plugable. An initial corpus of documents is needed to base the project, which we will decide on in this section. Nowadays many people have access to the internet and for a lot of people the internet is probably also their primary source of information. Next

to that, the internet also contains vast amounts of documents which could shed some light on relations between cities. Therefore, the decision was made to use web-data as a data source. To avoid duplicate work, which would mean crawling the web, a logical choice is to use Common Crawl as a data source.

4.1.1 Common Crawl

Common Crawl [6] is a freely accessible corpus of pages across the web, updated and released on a monthly basis. Many researchers have used the data for various purposes [20, 30, 31]. Since the project requires analysis on a very large set of documents, the corpus is a very suitable candidate for us to work with.

The data from Common Crawl comes in three formats[4]:

WARC This is the default and most verbose format. It stores the HTTP-response, information about the request and meta-data on the crawl process itself. The content is stored as HTML-content.

WAT Files of this type contain meta-data, such as link addresses, about the WARC-records. This meta-data is computed for each of the three types of records (meta-data, request, and response). The textual content of the page is not present in this format.

WET This format only contains extracted plain text. No HTML-tags are present in this text. For our purposes, this is the most useful format.

Common Crawl stores these pages in the following way: each archive is split into many segments, with each segment representing a directory. Every directory contains a document listing file and a folder for each file format (WARC, WAT and WET), which in turn contains the compressed pages belonging to the segment. To be able to efficiently get a single page, Common Crawl indexes the segments to directly map URLs to document locations using an offset and length which can be found using the Common Crawl index[5]. Since WAT- and WET-files can be generated from WARC-files, they only provide such indices for WARC-files. If no file index is provided with a data request, an aggregated compressed file of all files of the requested format is returned.

For extracting data from Common Crawl, many open-source libraries are available. Common Crawl's official website refers to `cdx-index-client`[6] as a command line interface to their data indices. It allows for, among others, specifying which data set to use, supports multiple output formats (plain text, gzip or JSON) and can run in parallel. Since this library only retrieves the file indices, we need another way to actually retrieve the pages pointed to. However, there is a problem with this: we are only interested in WET-files, but Common Crawl does not have WET-files indexed. We would therefore have to collect the WARC-files and convert them to WET-files ourselves, requiring us to parse HTML for every document we are interested in.

As mentioned in the design goals section not all available web-data will be used due to limited resources. A simple query `url=*.nl&output=json&showNumPages=true` on the CC-MAIN-2017-13 index using the online interface[7] yields 1676 pages. Pages in this sense are listings of 15000 indices, so there are roughly 25 million entries in total out of the 2.94 billion pages available in Common Crawl. It is very important to note that searching for a top level domain like `.nl` only includes the first page of every matching domain. To get all pages, additional queries for each site with more than one page are to be performed.

4.1.2 Other Data Sources

Besides Common Crawl, there are a plethora of other sources that might contain valuable information. The most notable is the Dutch royal library, Delpher[8]. It contains millions of Dutch digitalised newspapers, books and magazines from the fifteenth century up until about 1995. Because of this, it is a useful resource for historical research. Additionally, Statistics Netherlands[9] is the governmental organisation collecting statistical data about the Netherlands and comes with an API, making most of their data

[4] `https://gist.github.com/Smerity/e750f0ef0ab9aa366558`
[5] `http://index.commoncrawl.org`
[6] `https://github.com/ikreymer/cdx-index-client`
[7] `http://index.commoncrawl.org/CC-MAIN-2017-13-index?url=*.nl\&output=json\&showNumPages=true`
[8] `http://delpher.nl`
[9] `https://www.cbs.nl/en-gb`

publicly accessible. The NOW Corpus[10] collects newspaper and magazine articles through Google News and provides several tools to perform queries on this data. It can also be downloaded.

Due to time and resource constraints, we have chosen to exclude these from the project. Of course, in future versions, other data sources could be included.

4.2 Selecting Relevant Documents

Because not all data from information sources such as Common Crawl is relevant to find relationships between cities, the data needs to be filtered. One way to do this, is to only select the data that mentions at least two different cities. Because the data is plain text, we need a way to scan through the text and determine if the text indeed has a co-occurrence of two different cities. Making use of the comparative analysis of Rasool et al. [26], we chose the Aho-Corasick algorithm [1], which is a multi-pattern exact string matching algorithm. Using this algorithm, a predefined list of cities can be matched against the text of a web page or document. If at least two cities from the list appear in the text, we mark it as a useful document.

The decision to use the Aho-Corasick algorithm is strengthened by the fact that a well documented and stable Python library exists, which implements the aforementioned algorithm. This library is called `pyahocorasick`[11] and is a fast and memory efficient implementation of the Aho-Corasick algorithm.

We make a selection of documents without storing the documents first, because storing and indexing all documents is not feasible due to storage constraints. For the .nl web pages only would need about 250 GB of storage and to store all available documents around 250 TB of storage would be needed. As we do not have access to a fast and large data storage platform, we will not store and index everything first and then delete irrelevant documents. However, to test if finding and storing relationships between cities is fast enough when the documents are actually stored on disk a random selection of 1 million documents will be downloaded. Processing the already stored documents could finish within one day whereas downloading all documents will most certainly take multiple days.

4.3 Extracting Relations from Documents

Now that a selection of relevant documents has been made, we can make an attempt to identify the relations between cities based on these documents. Since labelling every relevant document by hand is not feasible, an automated approach is desirable. One way to automate this process is by identifying intercity relations using machine learning. Machine learning algorithms can be roughly divided into two distinct groups: (1) supervised and (2) unsupervised algorithms. Supervised algorithms expect an input set and a corresponding output set, with which a model is trained to predict unseen instances of the problem. Unsupervised algorithms identify clusters of entities based on similarities in the feature set corresponding to said entity.

Considering the fact that we have a strict time schedule of little over two months to develop the complete system, we decided to go with the supervised approach. Another reason for this choice is that the training and tweaking of supervised algorithms can be done faster compared to unsupervised algorithms. This stems from the fact that we do not need the complete data-set to start training a supervised model, while for the unsupervised case the complete set is needed.

4.3.1 Defining Classes

Our choice of using classification has naturally lead to the need for categories we want to identify within the collected documents. Together with our clients we identified the following categories which are useful to identify from the collected documents:

1. Collaboration
2. Commuting
3. Education
4. Leisure

[10] `http://corpus.byu.edu/now/`
[11] `https://pypi.python.org/pypi/pyahocorasick/`

5. Residential mobility
6. Shopping
7. Transportation
8. Other

These categories represent topics that are of interest for our clients. They relate to research that is being done by them and to relations that were deemed important in previous research on intercity relations. The category *other* is there to make the classification exhaustive, i.e. relevant documents can always be labelled.

4.3.2 Pre-processing

For pre-processing the documents, there are a number of tools available. We used NLTK [10] for removing stopwords and regular expressions for removing unwanted characters. The HTML parsing is done using BeautifulSoup [15].

Stop words Removing all common words (the, a, an etc) and symbols ('.', ',', '!', etc). For removing stopwords, we used a list from NLTK containing Dutch stopwords.

Unwanted characters To strip unwanted characters we have defined a regular expression that identifies unwanted characters (punctuation marks, years, etc.) Matching characters are removed from the document.

HTML Since we are dealing with HTML pages which we are parsing to plain text documents, we need to strip the HTML so that only the plain text remains. Using BeautifulSoup we strip unwanted tags (script, style, link, etc.) and parse the rest of the page to plain text.

4.3.3 Data-set

Before we can start building up and training our model, we need to collect data that can be used as input for the model. To collect this data we have considered several options.

The first option is to query for documents from news(paper) sites. Since the documents are categorised by professionals, we may assume they will provide us with features that represent the categories associated with these documents well. Unfortunately, the categories that we identified with our clients do not match typical newspaper categories, so this approach was not suitable for us.

Another approach is to use Google Custom Search to obtain results from Google, using the categories the client provided us with as keywords. The main disadvantage of this approach is lack of control over the files that get added into the data set. This way documents that get returned by the query are not analysed on desirable content but are added immediately.

Finally, we decided to provide our clients with a "labelling interface". The labelling interface provides the user with a document from the set of collected documents. It allows the user to label these documents with zero or more categories, after which the document is assigned to the correct category. If no category is selected, the document is discarded from the training set. This way, we have total control of the documents that are added to the data set. The documents are labelled by experts in the field of the built environment so we may assume these documents will represent the labelled categories well.

4.3.4 Modelling

When considering classification, there are a plethora of algorithms available. When choosing the right algorithm for a problem, several factors should be taken into account [19]. These are:

Accuracy How well the algorithm separates the websites.

Training Time How long it takes to train the algorithm.

Linearity Linear regression assumes data trends follow a straight line. This is trade-off between accuracy and speed.

Number of Parameters Adjustable parameters increase the flexibility of the algorithms. This is a trade-off between training time and accuracy.

Number of Features A large number of features can make some algorithms slow. Especially text data (what we are using) has a high number of features.

Special Cases Some learning algorithms make particular assumptions about the data or the results.

Keeping all these properties in mind we construct a setup that fits our purposes best. Below we have stated our approach of how we reached the setup we think is best suited for our goals.

Features To get a useful set of inputs (features) for our system we need to decide what describes the properties of our documents best. Since we are dealing with text-documents a natural choice for these inputs are the words contained in these documents. The words alone do not provide us a very useful input to the system. That is why we use TF-IDF to give the words that we encountered a weight. TF-IDF (Term Frequency over Inverse Document Frequency) gives words a weight based on their frequency in a document and on the frequency of the word in the complete document set. This way words that are rare in the complete document set but occur often in a document are assigned a high weight. Words that occur in many documents in the complete document set get awarded a low weight [25]. Using TF-IDF our features become words with weights associated to them.

Dimensionality reduction Since we are working with text documents and our features are words with TF-IDF weights we can assume that our feature set will be very large. The total number of features determines how fast we can train our model and has implications regarding over-fitting [28]. To reduce the number of features we considered different techniques from [28]. Since we have no time to test all the techniques, we decided to select the top ten percent of our features (based on the TF-IDF weights). In [38] it is stated that a dimensionality reduction with a factor ten using this approach does not lead to a loss in accuracy. To provide an easy way to add different types of dimensionality reduction techniques later, we will keep the code for defining new classifiers easily extendable.

Classification Even after applying dimensionality reduction which we discussed in the previous section, we are left with a lot of features. Thus, we need an algorithm that works well with a feature rich problem. From [19] we know SVM is a algorithm that works well with feature rich problems. Also [28] claims SVM is one of the best techniques when considering text classification. This combined with the fact that Scikit offers an easy to use implementation of SVM has lead us to use SVM as our classification algorithm.

4.3.5 Remarks

Scikit offers a lot of useful features to optimise the classifier. For example, using Scikit-"Pipelines" composing a classifier is an relatively easy task. Since we unfortunately do not have the time to benchmark the results of different types of classifiers and to play around with the different optimisation options, we plan on implementing our code in such a way that extending the code to use these optimising functionality and different pipelines will be really easy.

4.4 Storing and Ingesting the Data

In this section we will discuss which data storage solution we are going to use and why. We will compare a few options and select the best. We will then briefly explain how it works and how we plan to use it.

4.4.1 Storing Extracted Relations

The relations that are extracted from documents, as described in the previous section, need to be stored. We want to be able to apply different models on the data and we also want access to the raw data.
To keep this flexibility and to maintain scalability, extracted occurrences of relations are saved as a raw count. This means that if a relation "transportation" is extracted from a document that contains the cities "Rotterdam" and "Amsterdam", we increment the total count of "transportation" on the "Rotterdam-Amsterdam" relation.
Considering the fact that relations are bidirectional, meaning a relation of "transportation" between

"Rotterdam-Amsterdam" implicates a relation of "transportation" between "Amsterdam-Rotterdam" as well, we only need one relation between two distinct cities.

4.4.2 Graph Database, Search Engine or Traditional Database

Now that relationships have been extracted from the documents and web pages, we need to store these. Note that is not necessary to store all the documents, because the relationships which have been extracted are the information that is actually worth storing. Therefore, the original document can be discarded. However, the index pointing to the original document will be stored to be able to view it on demand. To store these relationships and indices there are two possibilities: (1) graph databases and (2) traditional relational databases.

Because visualisation of the network of cities as a graph is an important part of the application, and relations between cities play a key role in the system, we need a database that is designed for these features. Relations are the most important in the graph data model, where this is not true for traditional relational databases. Vicknair et al. stated that a graph database such as Neo4j has an easily mutable schema, where a relational database schema is less mutable [36]. Furthermore, the edges between nodes in a graph database can have properties, which is exactly what the envisioned data structure should be for this application. Lastly, if the desire arises to find indirect relationships between cities then a graph database is most appropriate choice. For example, if the client wishes to find out how Alkmaar is connected to Tokyo via other cities then the need for fast graph traversal arises. According to the graph database Neo4j their graph traversal is already 60% faster than a relational database for a depth of just 3[12]. Therefore, we are confident that a graph database is the best choice.

4.4.3 Comparing Graph Databases

Next, the type of database needs to be selected. For this, six of the most popular databases according to the solid IT Graph DBMS ranking[13]. This rating is established using multiple parameters, among these parameters is the number of mentions on websites and in job offers. Next to that, the parameters also include the number of searches, relevance in social networks and the general interest in the system[14]. These six most popular Graph Databases are rated on five important aspects. These are, is the graph database open-source, scalable and free and does it support Python and have built-in visualisation. TODO: Argumentatie voor deze 5

name	Open-source	Scalable	Python support	Free	Built-in Visualisation
AllegroGraph	✗	✓	✓	✓[a]	✗[b]
ArangoDB	✓	✓	✓	✓	✓
Neo4j	✓	✓	✓	✓[c]	✓
OrientDB	✓	✓	✓	✓	✓
Teradata Aster	✗	✓	✓	✗	✗[d]
Titan	✓	✓	✗	✓	✗[e]

[a] Only free up to 5 million triples
[b] With separate tool called Gruff: `https://allegrograph.com/gruff2/`
[c] Non-commercial use
[d] Using a separate tool Aster AppCenter
[e] Using a separate tool

From this table, it can be deduced that three of these graph databases are viable candidates: ArangoDB, Neo4j and OrientDB. For this project, Neo4j is the best choice because of three reasons. Firstly because we have experience with Neo4j, which means less time will be spent on getting to know the graph database and functionality. Secondly because it is by far the most popular graph database [13]. Thirdly, since Neo4j is the most popular graph database, the support community and amount of available examples is large.

[12] `https://neo4j.com/news/how-much-faster-is-a-graph-database-really/`
[13] `https://db-engines.com/en/ranking/graph+dbms`
[14] `https://db-engines.com/en/ranking_definition`

4.4.4 Using Neo4j for Storage and Ingestion

Neo4j is a highly scalable native graph database that leverages data relationships as first-class entities [21]. It is the single highly scalable, fast and ACID compliant graph database available. ACID stands for the four properties atomicity, consistency, isolation and durability of transactions in database systems that ensure reliability for query results [12]. The scalability of Neo4j comes from the fact that is easily spread across clusters, which provides a read throughput that scales linearly. Next to that, when spread across clusters Neo4j provides data redundancy and still high write speed [22]. Additionally, Neo4j is free to use for non-commercial purposes. To illustrate how scalable Neo4j is, consider that very large companies such as eBay, Cisco, Walmart, HP and LinkedIn[15] use it in their mission-critical systems. Holzschuher and Peinl compared the performance of Neo4j to the more classic and commonly used NoSQL databases and found that the more natural representation of relationships resulted in significant performance increase gains [13]. Jouili et al. concluded that Neo4j has a read-only performance which is comparable to other graph databases [18]. Compared to other databases Neo4j is slower with writing. However, the application will eventually do more reading than writing making writing a less important aspect.

There are some specific aspects of Neo4j that make it a very suitable candidate for this application. These are:

properties Any entity in the Neo4j graph can be given properties (key-value pairs) containing information about the entity. Properties are primarily meant to provide additional information and are less suitable to be queried on. As an example, a city can have a number of inhabitants and districts attached to it as a property.

labels Nodes can be tagged with a label, describing their roles in the network. These annotations are especially useful to filter the data set on one or more categories. For example, a city can be labelled as "capital" to be able to distinguish between regular and capital cities.

relations Nodes can be connected using relationships. These are always directed, typed and named and can have properties. Using these properties, one can control how the graph is traversed. For example, if a path (relationship) is to be avoided unless absolutely necessary, the relation can be given a high cost. To give importance to some relationship, one could also assign a strength score to it. Since relationships are handled efficiently by Neo4j, nodes can have any number of relationships linked to it without compromising performance. For our purpose, a relation could comprise the strength of the relationship between two cities (nodes).

The Neo4j model can be depicted as shown in figure 3. It consists of nodes, relationships (edges), properties (within the nodes) and labels (rectangular blocks above the nodes).

Besides the aforementioned useful properties of Neo4j, the graph can be put to good use for visualising the global urban network. By adding a location property to a city, nodes and relations can be mapped directly to a geographical map. Most importantly, indices of text files can be stored that mention the city as properties of nodes. That way, we are able to generate a subset of files that can be analysed for calculating the strength of the relationship between the nodes.

4.5 Interacting with the Data

After having filtered and classified the data, the framework should provide a means for the client to interact with the resulting data. In this section, several ways to do so are compared, after which we decide which path to take. We selected these three options because they match best with the clients' experience. The system should be intuitive and easy to use. Since the interface should allow the user to update the information displayed on the map (relation and city properties), performance of the interface is also a parameter we need to consider in our choices.

4.5.1 Query Language

One possibility is to let the client query the data. For this, we propose a simple, easy to use query language specific to the domain of research. It has the following syntax:

[15] `https://neo4j.com/customers/`
[16] `https://neo4j.com/developer/graph-database/`

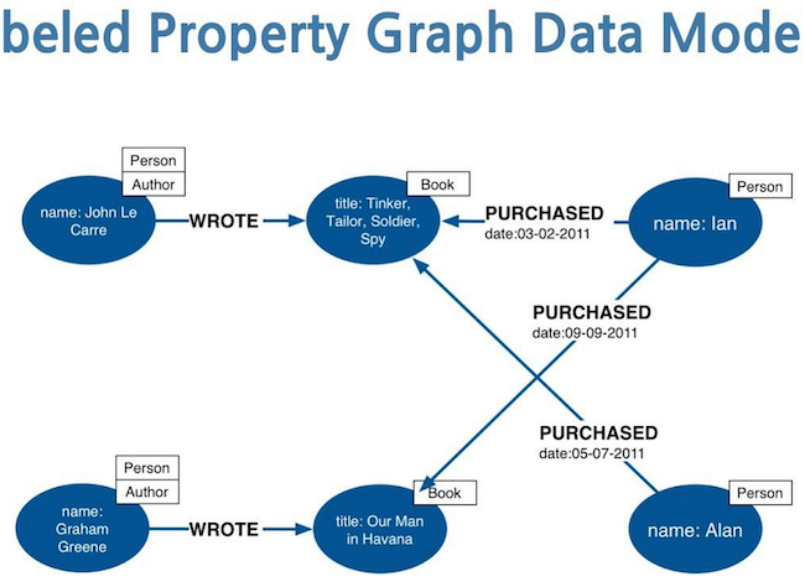

Figure 3: The Neo4j model [16]

!	Logical NOT operation
&	Logical AND operation
|	Logical OR operation
(A & B)	Grouping of clauses
$A > R > B$	Relation R between cities A and B

In figure 4, an example is shown that queries the "Shopping" relation between Rotterdam and Amsterdam and between Rotterdam and Den Haag.

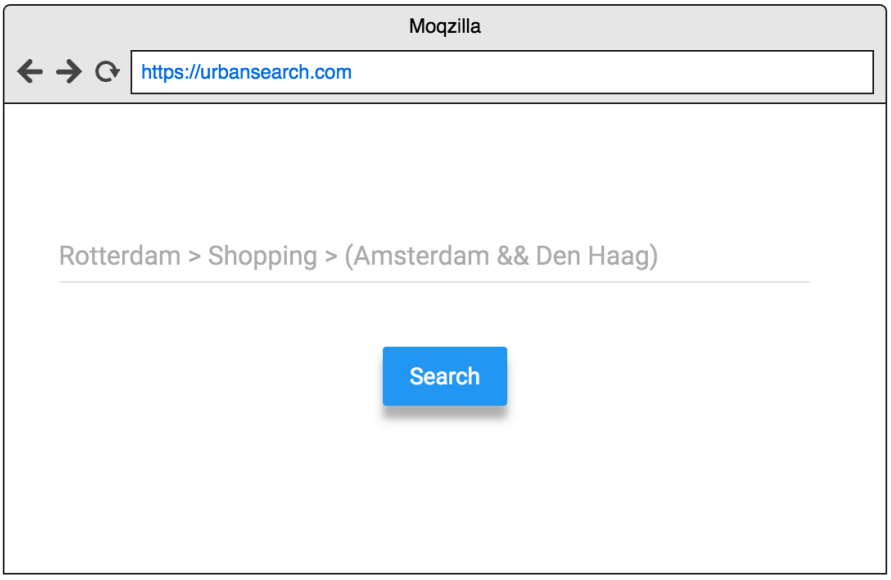

Figure 4: Example interface for the query language

4.5.2 Query Composer Interface

Another possibility is to offer the client a query composition interface. This interface would have the same functionality as the previously mentioned query language, but is more intuitive to use for new users. An example of the interface is given in figure 5.

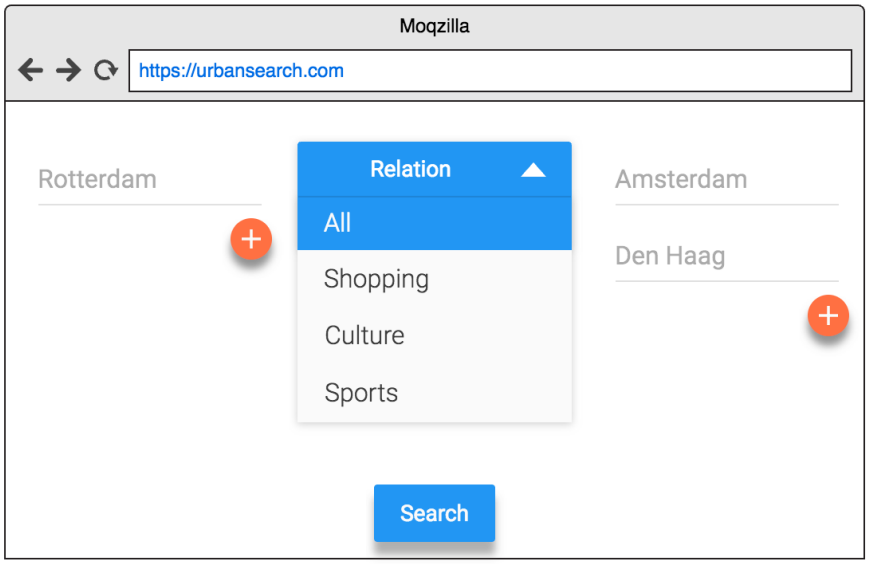

Figure 5: Example interface for the query composer

4.5.3 Interactive Querying

The last option we investigated is an interactive approach to querying data. For this, the client interacts with a map containing relations and cities. A very simple example is given in figure 6.

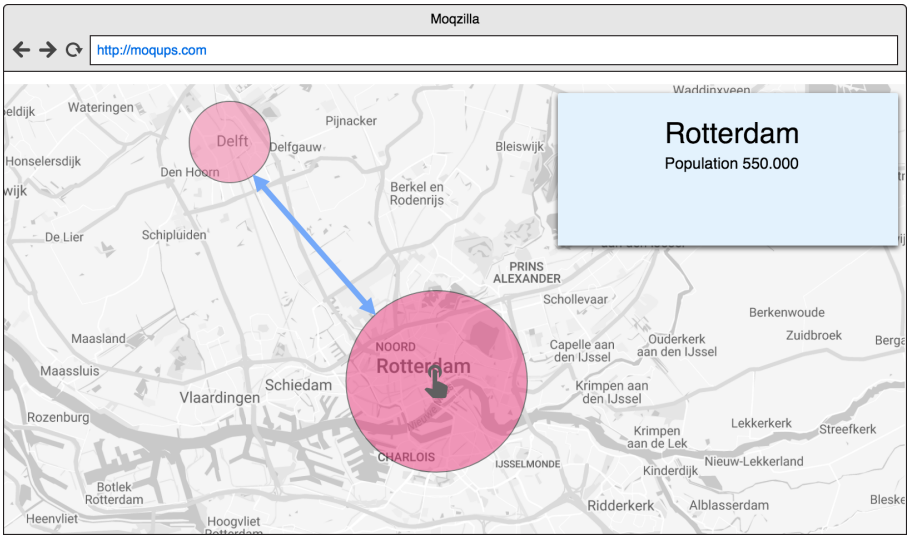

Figure 6: Example 1 of an interactive map

In this setup the user clicks on cities and relations on the map. This event triggers a query on the back-end and the resulting data is visualised on the map. An example of such an event is to show information about the selected city.

4.5.4 Conclusion

In association with the client, we conclude that the best option to go with is the interactive map. This way, the client has easy access to the data and this pattern of interaction best suits the work flow that the client envisioned prior to the project. The creation of complex queries is also made a lot easier. The user does not have to write or compose a complex query in advance but can do it directly on the map. Thus, retrieving a visual representation of several cities, interconnected with multiple relations, only involves selecting cities and relations on the map. Interaction directly with the map also reduces

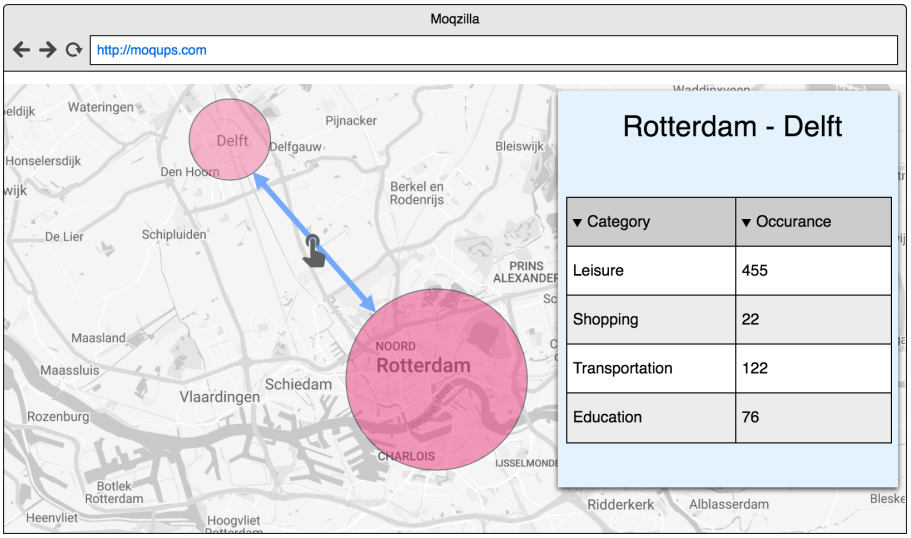

Figure 7: Example 2 of an interactive map

the need to go to a separate page to compose a query. This speeds up the use of the system by reducing page loads and it interrupts the work flow of the user less.

4.6 Visualising the Data

This section focuses on the visual representation of the processed data. This should be done in way that is easy to comprehend for users and helps them to interpret the data. To reach these goals, we identified the clients needs and desires. We have discussed the preferences of the client and have drawn up a global plan, which we present below.

4.6.1 Representing the Data Graphically

Since we are dealing with strongly related data, it is a natural choice to represent the data as a graph. The choice was made, in association with the client, to show the nodes and relations on a geographical map. Visualising cities on a map is intuitive to the user and we think this will increase the ability of users to interpret the information in a productive manner.

4.6.2 Using Geographical Maps

We investigated two map libraries we can use to display our data on a map. The first one is Google Maps, which can be used freely and offers a lot of customisation options. The API is well defined and some of the group members have previously worked with it. The second option we investigated is Leaflet. Leaflet is an open-source JavaScript library that provides responsive maps. It also has fine grained API and lots of plugins available. Both libraries are well suited for our needs. However, we decided to go with Google Maps, because of the existing experience of the group members. Another reason to gi with Google Maps is the amount of community support. This reasoning is best supported by the fact that querying "Google Maps" on StackOverflow.com returns 100.000+ results, while querying "Leaflet" gives us around 13000 results.

4.6.3 Handling Map Clutter

One of the challenges of visualising networks, as stated in [2], is the occurrence of so-called map clutter. Map clutter means the network is displayed as an incomprehensible set of nodes and edges. Several methods to prevent this are given in [2]. We will adopt some of these methods in our application, as explained next.

Users should be able to select what information they want to display. This will be included in the system by allowing the user to select cities and relations, enabling them to filter nodes and edges. The use of different sizes for nodes and edges or other attributes that are displayed can convey extra information to the user. We will use this to represent, for example, city population and exact strengths of relations.

The use of colour is another method mentioned in [2]. We will use colours to represent different types of relations and utilise colour intensity and opacity to represent the strengths of these different types of relations.

5 Validation

In this section, we first define a protocol with which the results of the system can be evaluated for correctness. Then, we describe how the system is tested and how we measure the quality of the code.

5.1 Testing the Application

We will test the program using four different testing methods. The first is unit testing, which tests the separate components individually. Integration testing for testing how well different components work together. system testing for testing the different entire system and Acceptance testing for testing how well the clients think the program works.

5.1.1 Unit Tests

Unit testing is done by writing automatic tests and making sure they pass every time the tests are executed. Unit tests test each method of a function separately, checking that the method does what it is supposed to do. If the method would need information from outside the class that information is mocked. This means that instead of using that other class, a fake object is made which returns a fake value. This ensures the tests will never fail due to changes in other classes.

5.1.2 Integration Tests

Integration testing uses automated tests which test how well different components of the system work together. This is done more or less the same as unit testing, however whilst you would mock methods from other classes in unit testing, with integration testing you do not. It is assumed that the separate modules are unit tested, therefor if an error occurs it is because something is wrong with the interaction between the modules and not with the modules themselves.

5.1.3 System Tests

We are also planning to use system testing. System testing provides a more complete test of the entire system. This means it is useful to detect faults in the overall system, but less easy to determine where these faults may be located. System testing is done manually, which means the tests can not be easily repeated when the system changes whilst with other testing techniques this is possible.

5.2 Validating the Application's Results

For validating the results of the application, we use acceptance testing. This is testing done to see if the software does what the clients are expecting it to do. These tests are therefore also executed by the clients manually. Afterwards they can say what worked, what did not work, what was missing and what could be improved. For this, we set up an evaluation protocol.

5.2.1 Evaluation Protocol

To be able to verify the relations that the system identifies, we need a protocol. Two parts have to be validated: (1) classification of documents and (2) relation scores. These are related in the sense that a relation score is calculated by the number of occurrences in labelled documents, so the correctness of labelling affects the correctness of relation scores.

Classification of documents is to be evaluated as follows. The client, considered a professional, labels a predefined set of documents, D of size d. The result set is called C. This same document set D is fed to the classification algorithm, which results in set A. The result sets C and A are then compared. The accuracy r of the classification algorithm is determined by the following formula: $r = \frac{|C \cap A|}{d}$. For example, if $d = 40$ and $|C \cap A| = 20$, r would be 0.50. The classification algorithm is then determined

to have an accuracy of 50%.

Evaluating relation scores is done differently. An important factor here is that cities have a natural relation due to their geographical position [34], so one would expect cities that lie close to each other are more related than cities that are on different sides of the country. This natural relation can be represented using the Gravity Model by Reilly [27]. The Gravity Model describes that the expected relation between two cities is

5.3 SIG

SIG [32], short for software improvement development group, is an organisation that analyses the code of projects to give insights in the quality of how the code is written. A high score means the code is highly maintainable and is kept simple. SIG includes Better Code Hub [14] which checks our code according to 10 guidelines as can be seen in appendix A. The great thing about Better Code Hub is that it can be run at anytime. We can check Better Code Hub whenever, whilst for SIG we have to send in our code and wait for feedback.

6 Conclusion

First, we discussed related work. We saw that there are currently two methods for analysing the relations between cities. Manually analysing search engine data is very slow and requires a lot of man-hours and looking at the different locations where businesses are located is only interesting for the economic relation and still misses a lot of data.

Second, we identified the requirements for a solution to the problem and discuss issues that might arise. The used the MoSCoW model to describe the importance of the different requirements. The most import must haves we found are being able to input place names, displaying a map with the connection data and being able to extract this data.

Third, we developed a methodology for a framework that satisfies the requirements and tackles the issues. We decided to start by using data from Common Crawl, although we might later extend this to other data sources such as Delpher. After selecting relevant data (data which contains 2 or more city names) we store the data with Neo4j. We then use clustering and classifying machine learning to group the data. First we use this on all data to get the general groups (e.g. economy, health-care, immigration) and then we use this on the data per pair of cities to see what the important connection types for each city are. Then we link these connections to the general groups ('fish' might relate to economy.. etc). To visualise this data we use the graph Neo4j provides.

Last we discussed how to validate the methodology applied. We will test our code with four levels of testing and send our code to SIG for analysis. We will use unit tests to tests individual modules, integration tests to check how different modules that call upon each other work together, system tests to check the entire system ourselves and acceptance testing to see how well the system fits the clients needs. To ensure visibility and maintainability in our code send the code twice to SIG for feedback and frequently use the service of Better Code Hub.

With this setup we should be able to make a well-tested functioning system that meets the requirements of our clients. Furthermore, using this system will enable us to answer the question "how can the strength of relationships between cities be extracted and visualised from open data?"

References

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. Commun. ACM, 18(6):333–340, June 1975.

[2] R. A. Becker, S. G. Eick, and A. R. Wilks. Visualizing network data. IEEE Transactions on Visualization and Computer Graphics, 1(1):16–28, Mar 1995.

[3] Stanley D Brunn, Lomme Devriendt, Andrew Boulton, Ben Derudder, and Frank Witlox. Networks of european cities in worlds of global economic and environmental change. Fennia, 188(1):37–49, 2010.

[4] Junho H Choi, George A Barnett, and BUM-SOO CHON. Comparing world city networks: a network analysis of internet backbone and air transport intercity linkages. Global Networks, 6(1):81–99, 2006.

[5] Dai Clegg and Richard Barker. Case method fast-track: a RAD approach. Addison-Wesley Longman Publishing Co., Inc., 1994.

[6] Common Crawl. Common crawl. `https://commoncrawl.org/`, 2017. Accessed: 2017-04-25.

[7] Ben Derudder and Frank Witlox. An appraisal of the use of airline data in assessing the world city network: a research note on data. Urban Studies, 42(13):2371–2388, 2005.

[8] Lomme Devriendt, Ben Derudder, and Frank Witlox. Cyberplace and cyberspace: two approaches to analyzing digital intercity linkages. Journal of Urban Technology, 15(2):5–32, 2008.

[9] Elasticsearch. Elasticsearch. `https://www.elastic.co/products/elasticsearch`. Accessed: 2017-04-26.

[10] Alex Rudnick et al. Nlkt. `http://www.nltk.org/api/nltk.stem.html`, 2017.

[11] Sean P Gorman and Edward J Malecki. The networks of the internet: an analysis of provider networks in the usa. Telecommunications Policy, 24(2):113–134, 2000.

[12] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. ACM Computing Surveys (CSUR), 15(4):287–317, 1983.

[13] Florian Holzschuher and René Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In Proceedings of the Joint EDBT/ICDT 2013 Workshops, pages 195–204. ACM, 2013.

[14] Software improvement group. better code hub. `https://bettercodehub.com/`, 2017.

[15] Fermentas Inc. Beautifulsoup: Beautiful soup documentation.

[16] Krzysztof Janc. Geography of hyperlinks—spatial dimensions of local government websites. European Planning Studies, 23(5):1019–1037, 2015.

[17] Krzysztof Janc. Visibility and connections among cities in digital space. Journal of Urban Technology, 22(4):3–21, 2015.

[18] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In 2013 International Conference on Social Computing, pages 708–715, Sept 2013.

[19] Microsoft. Microsoft azure machine learning algorithm cheat sheet. `https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-algorithm-choice`, 2017. Accessed: 2017-04-28.

[20] Hannes Mühleisen and Christian Bizer. Web data commons-extracting structured data from two large web corpora. LDOW, 937:133–145, 2012.

[21] Neo4j. Neo4j, the world's leading graph database. `https://www.neo4j.com`. Accessed: 2017-04-26.

[22] Neo4j. Understanding neo4j scalability. `https://neo4j.com/resources/understanding-neo4j-scalability-white-paper/`. Accessed: 2017-06-19.

[23] Michael E Porter. Location, competition, and economic development: Local clusters in a global economy. Economic development quarterly, 14(1):15–34, 2000.

[24] Tobias Preis, Helen Susannah Moat, and H Eugene Stanley. Quantifying trading behavior in financial markets using google trends. Nature: scientific reports, 2013.

[25] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In Proceedings of the first instructional conference on machine learning, 2003.

[26] Akhtar Rasool, Amrita Tiwari, Gunjan Singla, and Nilay Khare. String matching methodologies: A comparative analysis. REM (Text), 234567(11):3, 2012.

[27] William John Reilly. The law of retail gravitation. WJ Reilly, 1931.

[28] Fabrizio Sebastiani. Machine learning in automated text categorization. ACM Comput. Surv., 34(1):1–47, March 2002.

[29] John Rennie Short, Y Kim, Merje Kuus, and Heather Wells. The dirty little secret of world cities research: data problems in comparative analysis. International Journal of Urban and Regional Research, 20(4):697–717, 1996.

[30] Sameer Singh, Amarnag Subramanya, Fernando Pereira, and Andrew McCallum. Wikilinks: A large-scale cross-document coreference corpus labeled via links to wikipedia. University of Massachusetts, Amherst, Tech. Rep. UM-CS-2012-015, 2012.

[31] Jason R Smith, Herve Saint-Amand, Magdalena Plamada, Philipp Koehn, Chris Callison-Burch, and Adam Lopez. Dirt cheap web-scale parallel text from the common crawl. In ACL (1), pages 1374–1383, 2013.

[32] software improvement group. Sig. `https://www.sig.eu/`, 2017.

[33] Peter J Taylor. The interlocking network model. International handbook of globalization and world cities, pages 51–63, 2012.

[34] Waldo R Tobler. A computer movie simulating urban growth in the detroit region. Economic geography, 46(sup1):234–240, 1970.

[35] Gunnar Törnqvist. Flows of information and the location of economic activities. Geografiska Annaler. Series B, Human Geography, 50(1):99–107, 1968.

[36] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In Proceedings of the 48th Annual Southeast Regional Conference, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.

[37] Lynn Wu and Erik Brynjolfsson. The future of prediction: How google searches foreshadow housing prices and sales. In Economic analysis of the digital economy, pages 89–118. University of Chicago Press, 2014.

[38] Yiming Yang and Jan O Pedersen. A comparative study on feature selection in text categorization. In Icml, volume 97, pages 412–420, 1997.

A Better Code Hub Guidelines

Better Code Hub [14] checks our code according to ten guidelines:

1. **Write short units of code**
   Units of code should be no longer than 15 lines.

2. **Write simple units of code**
   Separate units of code should contain no more than 4 branch points (if, for, while, etc)

3. **Write code once**
   Shared code should be extracted, either to a new unit or to a super class

4. **Keep unit interfaces small**
   The number of parameters per unit of code should be no more than four.

5. **Separate concerns in modules**
   Identify and extract responsibilities of large modules to separate modules and hide implementation details behind interfaces.

6. **Couple architecture components loosely**
   minimizing the amount of interface code (e.g. by using 'abstract factory' design pattern)

7. **Keep architecture components balanced**
   Organize code in such a way that the number of components is between 2 and 12, and ensure the components are of approximately equal size (keep component size uniformity less than 0.71).

8. **Keep your codebase small**
   Refactor existing code to achieve the same functionality using less volume, and prefer libraries and frameworks over "homegrown" implementations of standard functionality.

9. **Automate tests**
   Add tests for existing code every time you change it.

10. **Write clean code**
    Remove useless comments, commented code blocks, and dead code. Refactor poorly handled exceptions, magic constants, and poorly named units or variables.