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 19, 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 [18]. 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 [15] or document search engines such as ElasticSearch [5] 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

Nowadays, many people live in or around an agglomeration, to reap the benefits it comes with. An agglomeration in this sense is an extended town area consisting of the built-up area of a central place and any suburbs linked by continuous urban area. A universally accepted idea is that benefits of agglomerations are key to economic growth [18]. In order to get more insight in the relationships between cities, there is a need for complex data analysis.

The huge amount of textual data generated online and the numerous historic archives are great sources of information on social and economic behaviours. 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 [19] and sales forecasting [24], the idea is to develop search queries that capture urban-urban interactions. These interactions are retrieved from information corpora through the co-occurrence of geographical names in textual data. An example of such a query on the Google search engine[1] 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. Thus, we will answer the following question: how can the strength of relationships between cities be extracted and visualised from open data?

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. We conclude in section 5 with the results of our research.

# 2 Related Work

In this section, we discuss existing methods to measure relations and their strength between cities. Currently there are only two methods to be considered.

The first method involves manually processing data from search engines. One queries a search engine (e.g. Google) checks how many websites are found. A small subset of the resulting pages is then classified manually. There are a few problems that arise with this approach. A lot of work needs to be done by hand, which takes a lot of man-hours. Furthermore, humans are biased and inconsistent when it comes to classifying documents, contrary to computers.

Another approach that is currently used is looking at office locations of companies in different cities. Although this might be a good indicator of the economic relations between cities, it completely ignores other relations cities might have with each other. According to Meijers, there might be many more factors [12].

The aforementioned methods are incomplete measures of relationship strengths. They fail to look at the bulk of the factors that play a role in forming relationships and scale badly. The framework described in this report aims to be a more robust solution for calculating intercity relationship strengths.

# 3 Requirement Analysis

In this section, we first define the design goals. Next, we present user stories that were created together with the client. 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 [3] as a prioritisation technique. Lastly, we discuss the design decisions that follow from the design goals and the requirements.

[1] `https://www.google.com`

## 3.1 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 to dispute a widely spread belief. Therefore, the results must be reliable and verifiable.

**understandable** The results of the application should be visually understandable, in order to make it easy for the client to deduce conclusions. Additionally, retrievable numeric data enable the client to further investigate the results outside of the scope of the application, should the need arise.

**scalable** Restricting the project to a set of non-English domains might impair the probability that the results are generally accepted. Allowing for investigating other domains would greatly help the client in a later stadium.

**plugable** It might be interesting for the user to let the application perform analysis on different data sets without the need of a developer.

**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.

## 3.2 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 switch between absolute and relative relation strengths, so that I can interpret the data in a more complete way.
4. As a user, I want to be able to (de)select cities, so that I can create a network of cities connected with relations.
5. As a user, I want to be able to (de)select relations between cities, so that I can inspect only the relations I am interested in at that time.
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, so that I can use it for presentations, papers or educational purposes

## 3.3 Product Requirements

As mentioned in the intro 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 won't 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 query the collected data based on relations between cities/nodes.
3. A machine learning algorithm should analyse the collected data and attempt to identify 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. A machine learning algorithm should be able to group different relationships that are strongly connected to each other.
3. It should be possible to add big data sets 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 understand 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).

### 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, aggregated over the past 50 years. These relationships are either simple or complex information flows. Simple information flows consist of a newspaper located in city $i$ publishing about city $j$, whereas complex information flows are co-occurrences of cities in an article.
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.

### 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. Evaluating places of residency with less than 750 inhabitants.

### 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. Since the validity of our results is very important for the results to be considered credible, we need to do thorough validation of the obtained results. We plan to not only test the code we deliver thoroughly, but also to cross-validate the obtained results.

## 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.

### 4.1 Gathering the Data

As explained in section 3.1, data sources should be plugable. An initial corpus of documents is needed to base the project, which we will decide on in this section.

#### 4.1.1 Common Crawl

Common Crawl [4] 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 [14, 21, 22]. 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[2]:

**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[3]. 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`[4] 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.

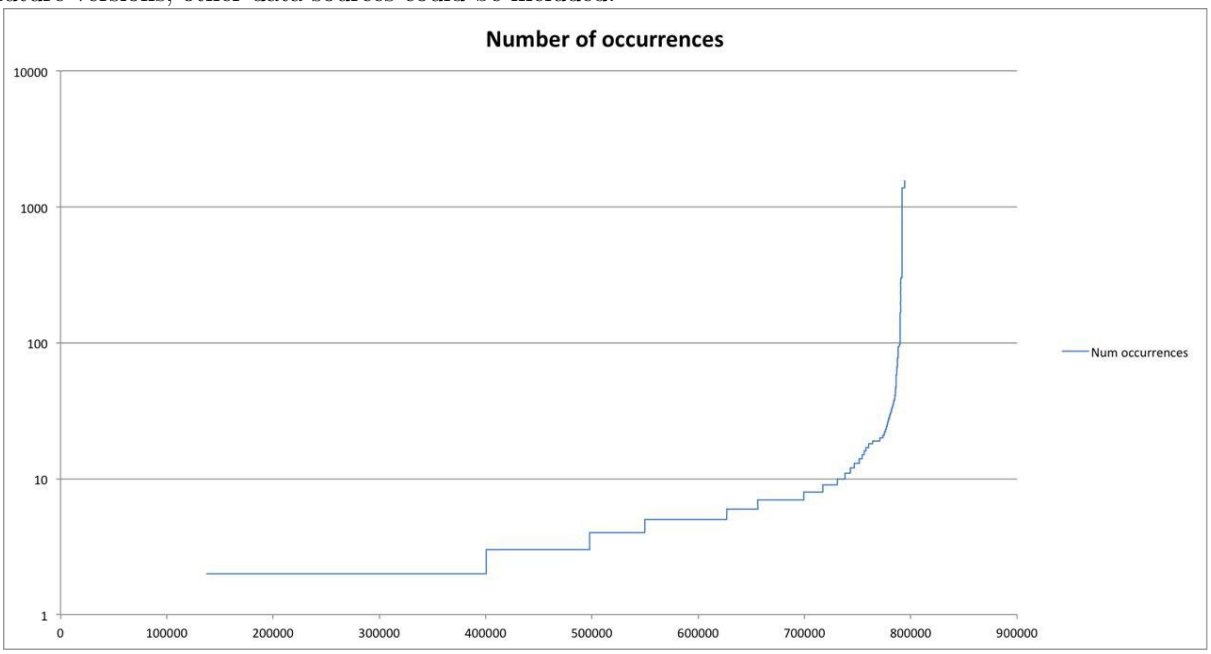
[2] `https://gist.github.com/Smerity/e750f0ef0ab9aa366558`
[3] `http://index.commoncrawl.org`
[4] `https://github.com/ikreymer/cdx-index-client`

A simple query on the latest index using the online interface[5] yields 1676 pages. Pages in this sense are listings of 15000 indices, so there are roughly 25 million entries in total. 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[6]. 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[7] is the governmental organisation collecting statistical data about the Netherlands and comes with an API, making most of their data publicly accessible. The NOW Corpus[8] 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.




TODO: add info:

Piet Bep, [16.06.17 11:55] num 20 773665 num 30 7510 num 40 4020 num 50 979

Piet Bep, [16.06.17 11:55] bovenstaande is aantal documenten met minder dan 20 occurrences, tussen 20 en 30, tussen 30 en 40, en tussen 40 en 50

## 4.2 Selecting Relevant Documents

Because not all data from information sources such as Common Crawl is relevant, 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. [20], 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`[9] and is a fast and memory efficient implementation of the Aho-Corasick algorithm.

[5] `http://index.commoncrawl.org/CC-MAIN-2017-13-index?url=*.nl&output=json&showNumPages=true`
[6] `http://delpher.nl`
[7] `https://www.cbs.nl/en-gb`
[8] `http://corpus.byu.edu/now/`
[9] `https://pypi.python.org/pypi/pyahocorasick/`

We make a selection of documents without storing the documents first, because storing and indexing all documents is not feasible due to storage constraints. 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.

## 4.3 Extracting Relations from Documents

Now that a selection of relevant documents has been made, we can identify the relations between cities based on these documents. Text documents can be split into two types: structured and unstructured. Text is structured when sentences are used, meaning grammar is used. In the case of structured texts a word may say something about the next word in the sentence, therefor other techniques can be used (for example n-grams) then in case of unstructured texts.Because the text from websites can be structured as well as unstructured, we will most likely use unstructured approaches. One way to do this is by using machine learning. The two common approaches are supervised classification and unsupervised clustering.

Clustering, is sorting these websites into groups based on how much they are alike. It does however not assign these groups with a label (or class). this can be a large problem because in one group may be several topics that have similarities with some of the other topics within the group but not with all other topics in the group, which makes it very hard to assign a label to these groups. Although it is possible to label these groups manually, since it is quit tedious and we are on a limited time schedule, it is better to use a classification algorithm.

With classification you need to decide upon the labels (classes) you want to use beforehand, and then for each class find a small subset as training data. Afterwards the classifier can automatically label a new document to one of the labels. Before we can enter the training data or classify a text document we need to preprocess the document into valid input and define which classes are important.

### 4.3.1 defining classes

Since we decided on using classification as our supervised machine learning algorithm we need to define the classes beforehand. Since we are not familiar with the most useful categories for measuring relationships between cities, we asked our clients Evert Meijers and Antoine Peris for their help. They choose the following topics for us to investigate:
- Commuting ('woon-werk')
- Shopping or servicetrips ('winkelen / diensten')
- Leisure trips ('vrije tijd / recreatie / toerisme')
- Residential mobility ('verhuizingen')
- Business trips ('zakelijk verkeer')
- Education ('Onderwijs, School')
- Collaboration ('Samenwerken')
- Goods Transportation ('Goederenvervoer')
- Other ('overig / restcategorie')

### 4.3.2 preprocess

For text preprocessing there are a number of tools available. We used NLTK [6] for tokenizing, removing stopwords and stemming and Gensim [25] for Modelling.

1. Tokenization.
   splitting up the text into words and other symbols called tokens. For this we implemented the tokenizer NLTK provides. Testing showed that it does what it is supposed to do.

2. Removing 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. For removing symbols we made a method that removes all non alphabet values except spaces from text.

3. Stemming.
   Reducing derived words to their stem (e.g. fishing -¿ fish). We first wanted to use NLTK, which

itself implements several stemmers including SnowBallStemmer [16] and Porter [17]. However research from the Rijks Universiteit Groningen (RUG) by Tanja Gaustad and Gosse Bouma [7] evaluated a Bayesian text classification system with either no stemming or the Porter or dictionary based stemmer. Which concludeded stemming does not lead to significant change in classification accuracy.

4. Modelling (TF-IDF).
   Before we can enter this tokenized data into the machine learning algorithm we first need to transform it to the correct model. Gensim provides two ready to use method for this to change tokenized text into the proper input data. The first is token2ID which gives each unique word a numeric ID and counts the amount that those words occur in a text and the second is tfidf which changes the amount that words occur in tf-idf values. tf-idf means term frequence inverse documet frequency. Term frequency is the number of times the word occurs in a document and idf is the number of documents devided by the number of documents where the term occurs. These two values are then multiplied. The results are normalized to make up for different document lenghts [https://radimrehurek.com/gensim/models/tfidfmodel.html].

#### 4.3.3 training

In order to use supervised machine learning, we first need to find training data for each class. We thought about two ways to do this. In order to get a good training sample for these categories, we discussed two different methods.

The first option was to query for results from news(paper) sites. However, this approach did not give us the desired results. The reason for this is that the categories we find in the newspapers do not match the categories Antoine and Evert provided us with. We also tried using the search engine from those news sites, however using those we found completely unrelated articles to the search queries (A search query about 'verhuizen' resulted in an article about parrots and minecraft).

The second option we are considering is to use Google Custom Search to obtain results from Google using the categories/keywords Antoine and Evert provided us with. A quick test with this last method provided us with quite good results (although we still see some noise in the results this is less than with the search engines from news sites).

#### 4.3.4 classifying

For classifying there are a multitude of algorithms available. For choosing the classifier we make use of the Microsoft Azure Machine Learning Test Sheet [13]. Several factors should be taken into account when choosing an algorithm. These are:

1. Accuracy - How well the algorithm separates the websites.
2. Training Time - How long it takes to train the algorithm.
3. Linearity - Linear regression assumes data trends follow a straight line. This is trade-off between accuracy and speed.
4. Number of Parameters - Adjustable parameters increase the flexibility of the algorithms. This is a trade-off between training time and accuracy.
5. Number of Features - A large number of features can make some algorithms unfeasibly long. Especially text data (what we are using!) has a large number features. Support Vector Machines are especially well suited in this case.
6. Special Cases - Some learning algorithms make particular assumptions about the data or the results.

For textual data especially support vector machines are recommended, so it is most likely we will choose that machine learning algorithm. Depending on whether we have time we might do some tests before making our decision however. The sci-kit package [11] provides an easy to implement module to use this.

## 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 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. To store these relationships there are two possibilities: (1) graph databases and (2) traditional relational databases.

Because visualisation of the network of cities as a graph is an essential 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. Therefore, we are confident that a graph database is the best choice.

### 4.4.2 Comparing Graph Databases

Next, the type of database needs to be selected. For this, six of the most popular 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.

| 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 is by far the most popular graph database [10]. Thirdly, since Neo4j is the most popular graph database, the support community and amount of available examples is large.

### 4.4.3 Using Neo4j for Storage and Ingestion

Neo4j is a highly scalable native graph database that leverages data relationships as first-class entities [15], enabling enterprises of any size to connect their data and use the relationships to improve their businesses. 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 [8]. 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[11] 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 [9].

[10] `https://db-engines.com/en/ranking/graph+dbms`
[11] `https://neo4j.com/customers/`

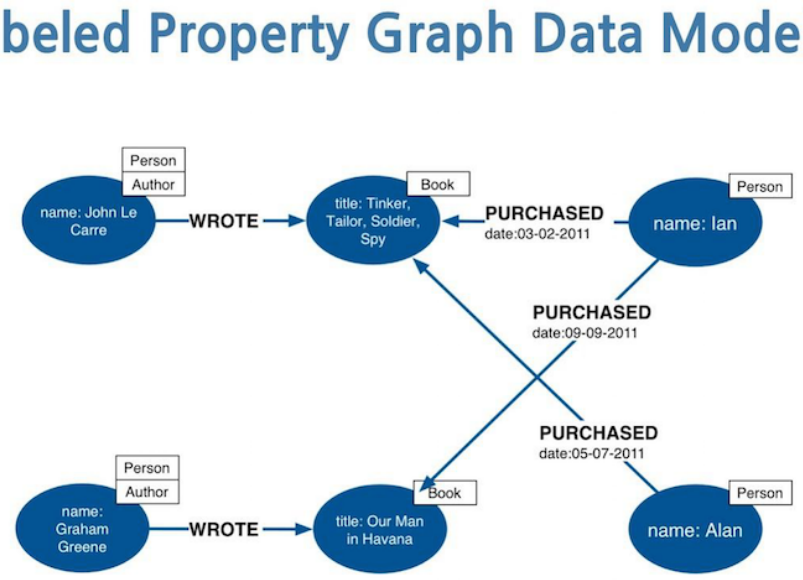



Figure 1: The Neo4j model

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 1. 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.

### 4.5.1 Query Language

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

| | |
|---|---|
| ! | 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 2, an example is shown that queries the "Shopping" relation between Rotterdam and Amsterdam and between Rotterdam and Den Haag.

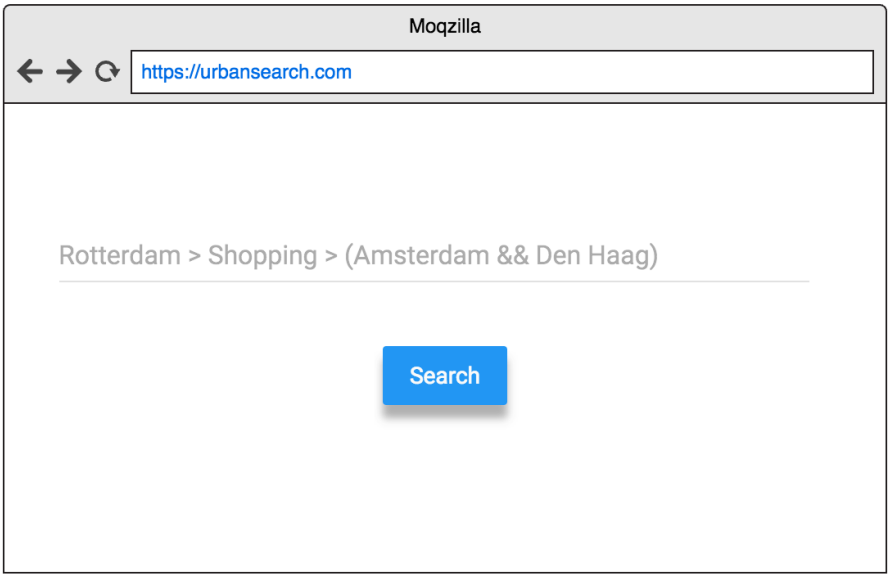



Figure 2: Example interface for the query language

### 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 3.

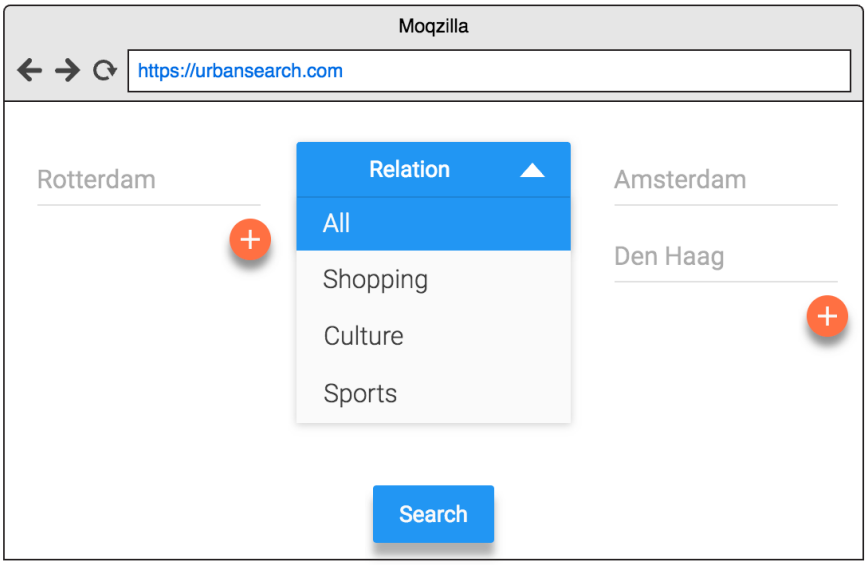



Figure 3: Example interface for the query composer

### 4.5.3 Interactive Search

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 4.

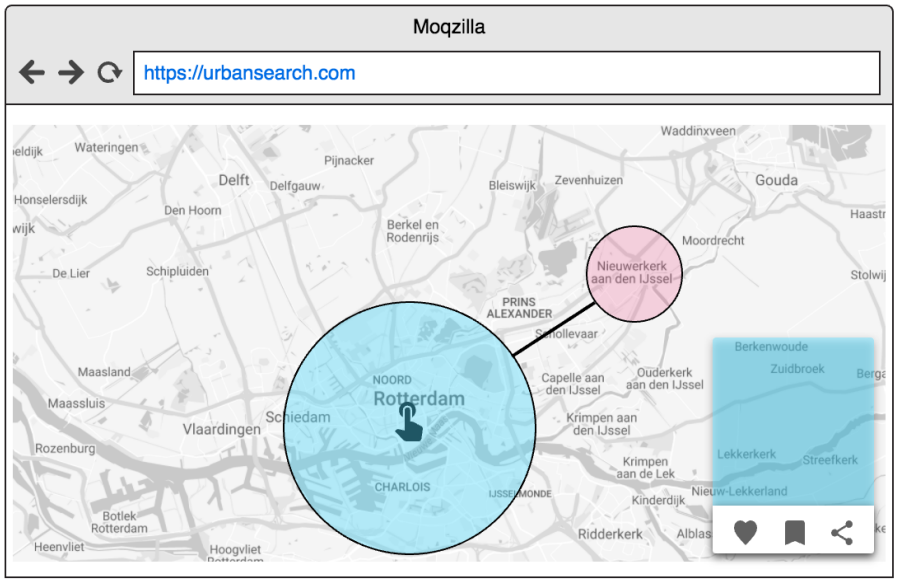



Figure 4: Example of an interactive map

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 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. Also, we feel Google Maps is better supported through documentation and communities.

### 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.

## 4.7 server

# 5 Code Quality

To ensure code quality in our project we used several methods. The results from SIG [23], a tool to ensure code quality and maintainability, are discussed and the testing is discussed.

## 5.1 SIG & BetterCodeHub

SIG, Software Improvement Group, gives detailed insight needed to achieve better code quality and maintainability. SIG rates the code on a five star scale based on nine different values concerning code quality. Before submitting code to SIG we used BetterCodeHub [10] to check for possible faults in our code. BetterCodeHub does partly what SIG also does, but it is done online instead and can be done on every moment. Code was submitted to SIG on week 5 and week 9 of the project. Feedback can be found in appendix A.

### 5.1.1 week 5

The first feedback from SIG was in the fifth week of development. Before uploading on BetterCodeHub our code passed all checks. For SIG it had a score from four out of five stars which means our code is above average maintainable. The last star was missed because the code is above average complex. This means that some of the functionality of some methods should be split into separate methods. TODO: fixed this?

### 5.1.2 week 9

## 5.2 Testing

We tested the program using three different testing methods. The first is unit testing, which tests the separate components individually. Selenium testing for testing the front-end of the app. And system testing for testing the different components together.

### 5.2.1 Unit Testing

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. In total these test cover TODO: % of the code.

**Selenium** Selenium tests are automated tests that are run to test the front-end of systems. Just as normal unit testing, selenium testing is useful for regression testing.

### 5.2.2 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.

### 5.2.3 System Testing

Last we are also using 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.

#### 5.2.4 Acceptance Testing

# 6 Validation

## 6.1 machine learning

To test the machine learning we will manually search for a small set of 50 pages per class. We will then manually decide to which class we would expect those pages to be sorted. Then we will run the algorithm and check whether or not a page was correctly sorted. There are 4 outcomes per page per class: true/false positives and true/false negatives. Positives are the pages that should be sorted to that class; true means they were sorted to that class, false meaning they were not sorted to that class. Negatives are pages that should not be sorted to that class and in this case true means they were indeed not sorted to that class and false means they were. So the true values are the ones that should be high. From this we can say how accurate our classifier is.

## 6.2 unit tests

## 6.3 selenium tests

## 6.4 SIG

SIG [23], short for software improvement development group, is a tool to give deep code visibility we require so we can remedy current problems and prevent future vulnerabilities. A part of sig TODO: (?) also includes betterCodeHub [10] which checks our code in 10 steps:

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.

# 7 Process

## 7.1 Collaboration between the team members

The collebaration between the team members went well. The team members worked in a room in the faculty of architecture from 9-5 each day. Three of the four team members knew each other already. The work was devided even over the team members.

## 7.2 Collaboration between the team members and the TU Delft Coach

Each week 9:30 on monday the team members had a meeting with the TU Delft coach. In the beginning there were some communication issues between the team and the coach but as the process went on communication became better.

## 7.3 Collaboration between the team members and the client

The collaboration between the team members and the client was good as well. Weekly meetings helped the team members making the productas good as possible to the clients wishes.

# 8 Implementation

TODO: To do? To fake? To leave out? You decide!

## 8.1 Scrum

## 8.2 Implementation per week

### 8.2.1 week 1

### 8.2.2 week 2

### 8.2.3 week 3

### 8.2.4 week 4

### 8.2.5 week 5

### 8.2.6 week 6

### 8.2.7 week 7

### 8.2.8 week 8

### 8.2.9 week 9

# 9 Conclusion

## 9.1 related work

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.

## 9.2 related work

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.

## 9.3 Requirement analysis

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.

## 9.4 Framework-and-tools

# A Sig Feedback

## A.1 week 5

[Analyse]

De code van het systeem scoort 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Complexity.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt.

Omdat jullie qua score al vrij hoog zitten gaat het hier voornamelijk om kleine refactorings. Methodes als IndicesSelector.run_workers en CoOccurrenceChecker._calculate_occurrences zou je nog iets verder kunnen opsplitsen in functionele gebieden.

De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.

## A.2 week 9

# B User manual

# C Used Libararies

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

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

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

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

[7] Tanja Gaustad and Gosse Bouma. Accurate stemming of dutch for text classification. Language and Computers, 45(1):104–117, 2002.

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

[9] 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.

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

[11] Scikit learn developers. Scikit-learn. `http://scikit-learn.org/stable/index.html`, 2017. Accessed: 2017-04-26].

[12] Evert J Meijers. Synergy in polycentric urban regions: complementarity, organising capacity and critical mass, volume 13. IOS Press, 2007.

[13] 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.

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

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

[16] Richard Boulton Olly Betts. Snowball. `http://snowball.tartarus.org/algorithms/dutch/stemmer.html`, 2017.

[17] M.F. Porter. Porter. `https://tartarus.org/martin/PorterStemmer/`, 1980.

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

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

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

[21] 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.

[22] 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.

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

[24] 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.

[25] Radim Řehůřek. Gensim. `https://radimrehurek.com/gensim/models/tfidfmodel.html`, 2017. Accessed: 2017-05-17].