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

May 5, 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 [17]. 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 [14] 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 [17]. 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 [18] and sales forecasting [23], 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 works 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, in contrary to computers.

The second approach that might be used is to look at companies and to look in which cities they are located. 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 [11].

The aforementioned methods are clearly 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 [13, 20, 21]. 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.

4.2 Selecting Relevant Data

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. [19], 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.

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 Grouping the Data

After we've stored the indexes of relevant websites we want to group each website according to subject: examples include, but are not limited to economy, politics and migration. There are two possible approaches for this: clustering and classifying. Clustering means grouping without first defining what groups (or classes) should be used. For classifying you need to first define groups and then the websites are sorted in those groups.
A standard approach for grouping is to use only the text from the websites and removing all other data. The big advantage is that it costs much less storage. Information from images may be lost however. Since we have to go over millions of pages, storage is a big issue for us therefor we choose this approach. Text based classification/Cauterization itself can be split on structured and unstructured text. 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.

4.3.1 Pre-processing

Before we can use the unstructured text we need to pre-process it. There are three steps to this.

[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/`

1. Tokenization.
   splitting up the text into words and other symbols called tokens.

2. Removing stop words.
   Removing all common words (the, a, an etc) and symbols ('.', ',', '!', etc).

3. Stemming.
   Reducing derived words to their stem (e.g. fishing -¿ fish).

The programs Snowball [16] and NLTK [6] (which uses the snowball version) have a dutch implementation for this, although it might need to be improved a bit.

4.3.2 Clustering

One method of grouping websites is clustering. For clustering text documents the standard approach is using k-means clustering, which uses the bag of words model. Scikit-Learn [10] has a library for this. In this library you can specify what feature extractors are used (TfidfVectorizer for tf-idf, a method for determining weights of words, and HashingVectorizer which hashes the words). Furthermore this library automaticly places the documents in batches if it would become too big to do it in one time. Initially we will test this method, however as we don't know whether we will get valid results we will keep the other method (classifying) in mind.

4.3.3 Classifying

Classifying is done in two steps: choosing the classes and actual classifying. Classes can be chosen manually, however, you can also apply certain techniques to automatise results. The advantage of this that you will get unbiased results, the disadvantage is that it takes more time to make these results and it may also cause classes to be chosen which the user can't use, for example when using Google Trends [7] results as "AFC Ajax", "Aeroflot" and "Eric Dane" may be found. The same problem occurs when searching for sub-classes.

Defining classes

One problem we need to solve is the problem of choosing classes. This can be done in several ways. The easiest way is to define these ourselves, however we may get better results if we used some algorithm. When using algorithms we may also decide if we want to consistently use the same classes for each pair of cities, or define the classes per pair depending on the importance. For example if Rotterdam and Vlissingen have a huge trade of fish, "fish" we be an important class of the relation between Rotterdam and Vlissingen. However if Leiden does nothing with fish, the class will be absent for the relation Rotterdam-Leiden or Vlissingen-Leiden. One way to do this is to look at the websites for each relation and apply the 'bag of words' model to check which words occur most frequently (after stemming and the removal of stop words). To make sure we don't get similar results (for example the relations 'fish' and 'fishmarket') we could remove all websites containing 'fish' and then applying the same model again. Our plan is to first find some general classes (e.g. economy) from all websites. And then make subclasses (e.g. fish) for each pair. The general classes are used to compare between relations of different cities.

Machine Learning for classification of unstructured text

Text based machine learning for unstructured texts is done using the 'bag of words'. This model counts how often each word is used. There are 3 libraries available which contain most steps needed. There is scikit-learn [10] and TensorFlow [22] for Python and Weka [15] for Java. Since we write our program in python, and TensorFlow is only about neural networks, we choose to use scikit-learn. The machine learning works in 4 steps:

1. **Creating a feature extractor**
   To prepare the features for the machine learning algorithm. We need to give each feature/token a numeric id. Count each of these tokens. And we need to normalise the tokens. For this scikit-learn provides algorithms.

2. **Manually labelling**
   For each of the classes (e.g. business, tourism, art etc) we select a few websites we know fit to that class. Here occurs the problem of defining classes which will be expanded on later. From these websites all the words will be extracted and their occurrence will be counted. Possibly some

normalisation functions are applied to get better values. We call these values the weight for each word for each class. From this we create a two dimensional array with in the rows each of the websites and in the columns all different words and one extra for the class. We fill the fields with the weights or a zero if the words don't occur.

*	Bedrijf	Tourisme	...	Zwerver	Klasse
Website 1	0	4		1	1
Website 2	4	1		2	2
Website 3	1	3		3	1

3. **Generating a classifier**
   The array is fed to a learning algorithms. This will generate a classifier. There are a multitude of classifiers importable from Scikit-learn and TensorFlow. For choosing the classifier we make use of the Microsoft Azure Machine Learning Test Sheet [12]. Several factors should be taken into account when choosing an algorithm. These are:

   (a) Accuracy - How well the algorithm separates the websites.

   (b) Training Time - How long it takes to train the algorithm.

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

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

   (e) 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.

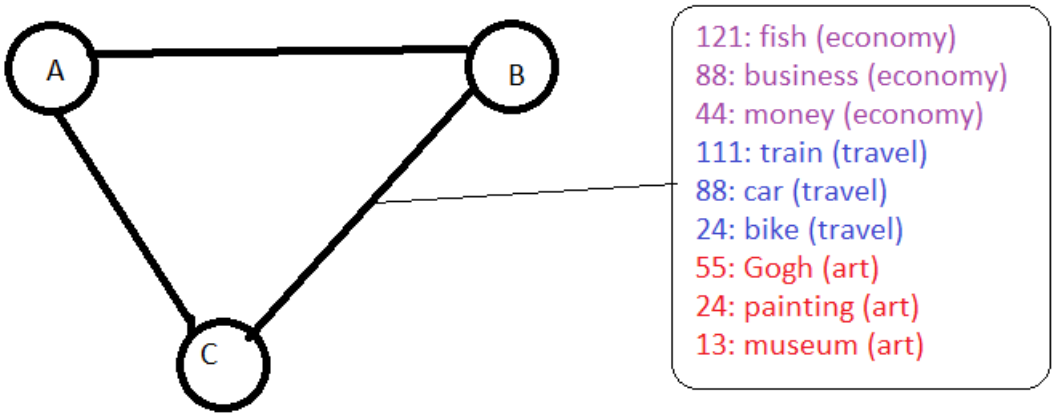
   (f) Special Cases - Some learning algorithms make particular assumptions about the data or the results.

   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.

4. **Entering new examples**
   When a new (unlabelled) example (website) comes - extract the features and feed it to your classifier - it will tell you what it thinks it is (and usually - what is the probability the classifier is correct). Afterwards the classifier can be updated to include new features extracted from the example. This updating probably needs to be done a couple of times because the first few times not all features (possible words in the dutch dictionary) will be included. It is important to choose the same amount of websites for each class.

4.3.4 Conclusion

Our approach will be to first transform the data into the bag of words model using tokenizing, stemming and the removal of stop words. Afterwards we will try to group the websites in 3 tries. First we try to cluster all the websites into groups. We will then whether these clusters are actually viable clusters and whether they can be labelled into useful categories. If this is not the case we will instead try classification. To define classes we will try to look at the most common words. Then removing all associated websites and repeating this until multiple words are chosen as categories. If even this does not work we will define our own categories. After all websites are initially clustered or classified we will use the same method for the websites for a pair of cities. We will try to automatically assign the found relations to one of the main categories as well (e.g. fish to economy, train to travel etc).
After all websites are classified or clustered and possibly sub-classified or sub-clustered we can use that data to show the strengths of the connections between cities by counting for each class how many websites there are that contain information about both cities.
An earlier problem we encountered was differentiating between the cities Utrecht and Groningen and the provinces Utrecht and Groningen. We might be able to use machine learning as well to check if a website is talking about the province or the city, however it could very well be that this does not significantly improve our data.



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 [14], 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].

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

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

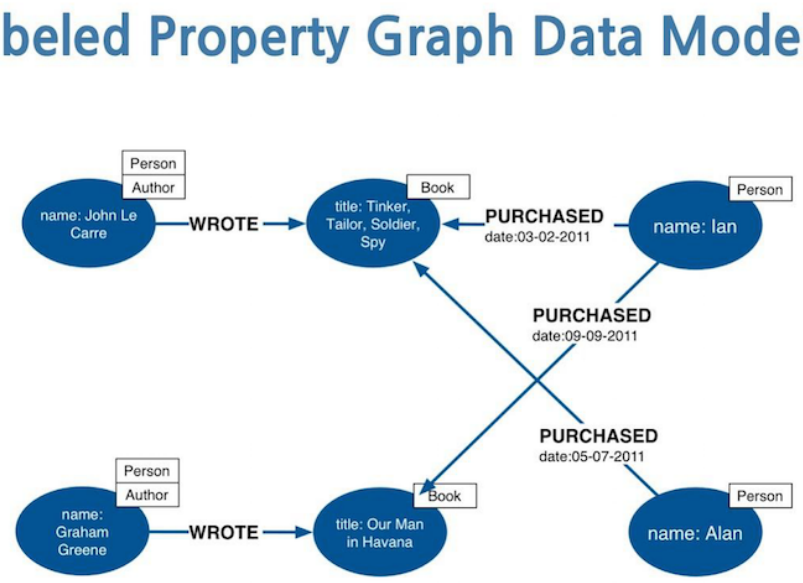


Figure 1: The Neo4j model

In figure 2, an example is shown that queries the "Shopping" relation between Rotterdam and Amsterdam and between Rotterdam and Den Haag.

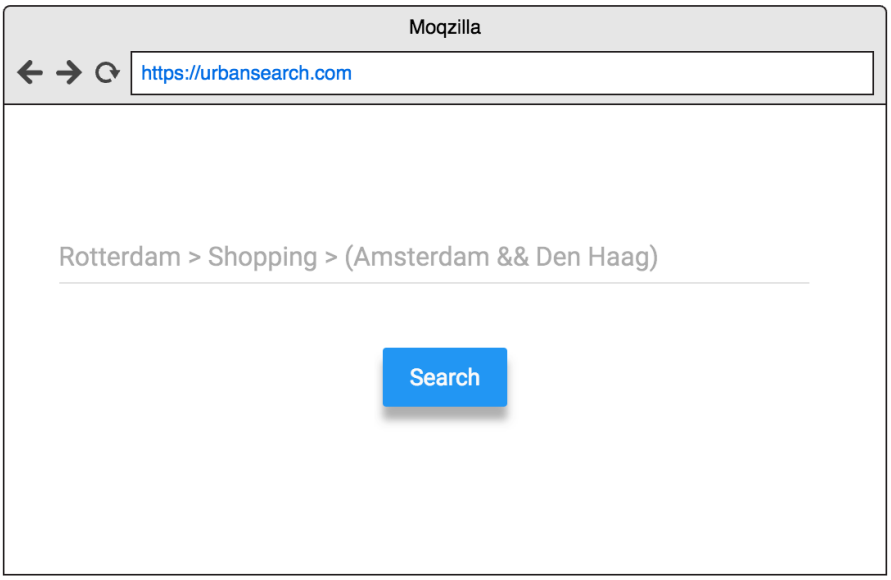


Figure 2: Example interface for the query language

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.

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.

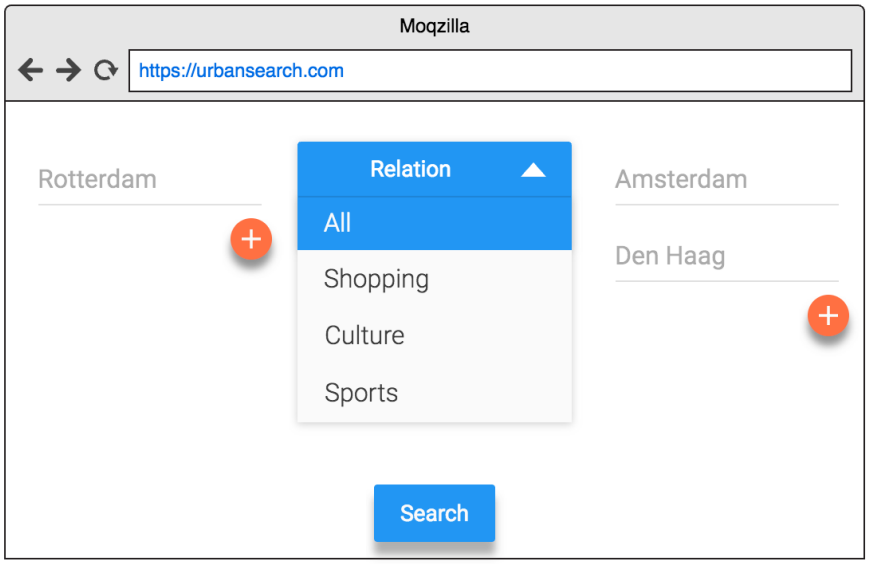


Figure 3: Example interface for the query composer

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 suits the work flow that the client envisioned prior to the project best. 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

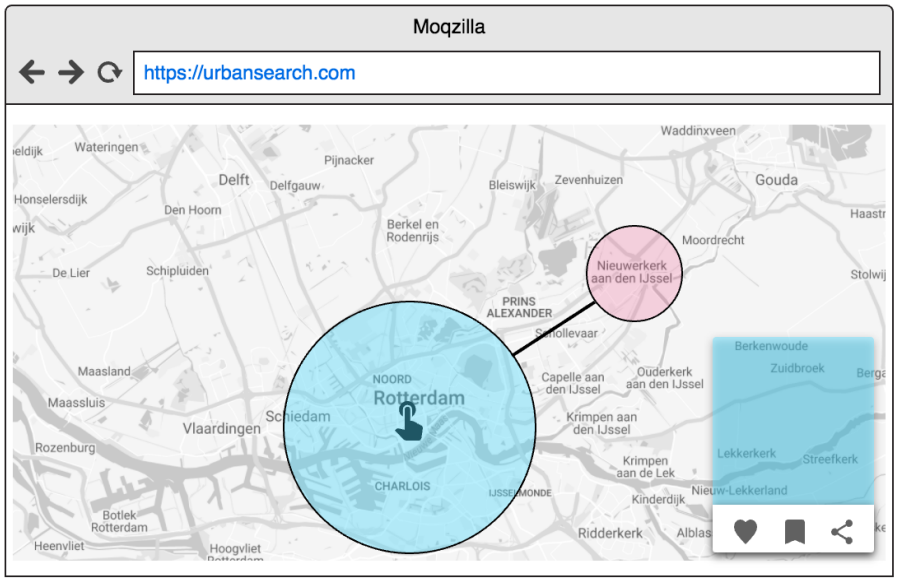


Figure 4: Example of an interactive map

with Google Maps, because of the existing experience of the group members. Also, we feel Google Maps is better supported through documentation and communities, that can help us should we get stuck.

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 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.

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] Google. Google trends. `https://trends.google.com/trends/home/all/NL`, 2017. Accessed: 2017-05-01.

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

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

[12] 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.

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

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

[15] University of Waikato. Weka. `http://www.cs.waikato.ac.nz/ml/weka/`, 2017. Accessed: 2017-04-26.

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

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

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

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

[20] 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.

[21] 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.

[22] Google Brain Team. Tensorflow. `https://www.tensorflow.org/`, 2017. Accessed: 2017-04-28].

[23] 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.