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 4, 2017

TU Delft

Abstract

It is yet to be discovered how the strength of relationships between cities relates to economic growth. Many believe it is merely the increasing size of cities that cause economic growth. 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 [6] 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, document analysis, classification, filtering

Contents

1 Introduction **3**

2 Related Work **3**

3 Requirement Analysis **4**
3.1 Design Goals 4
3.2 Product Requirements 4
3.2.1 Must Have 5
3.2.2 Should Have 5
3.2.3 Could Have 5
3.2.4 Would Like 5

4 Framework and Tools **6**
4.1 Gathering the Data 6
4.1.1 Common Crawl 6
4.1.2 Other Data Sources 7
4.2 Selecting relevant data 7
4.3 Storing and Ingesting the Data 8
4.3.1 Graph database, search engine or traditional database . . 8
4.3.2 Comparing Graph Databases 8
4.3.3 Neo4j 8
4.4 Grouping the Data 9
4.4.1 Pre-processing 10
4.4.2 Clustering 10
4.4.3 Classifying 11
4.4.4 Conclusion 12
4.5 Interacting with the Data 13
4.5.1 Query language 13
4.5.2 Query composer interface 14
4.5.3 Interactive search 14
4.5.4 Conclusion 14
4.6 Visualising the Data 16
4.6.1 Representing the data visually 16
4.6.2 Maps 16
4.6.3 Map clutter 16

5 Conclusion **16**

1 Introduction

Nowadays, many 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. Common belief is the benefits of agglomerations are key to economic growth [18]. However, it is still unknown what primarily causes economic growth. It could either be these advantages or an increase in the contesting position of the agglomeration in the global urban network.

The huge amount of textual data generated online and the numerous historic archives are great sources of information on social and economic behaviours. They constitute the bulk of information flowing among each other. 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. We thus 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. Last, we conclude in section 5 with the results of our research.

2 Related Work

In economics there is the question "What factors play a role in economic growth?". To answer this question you would first need to give a clear definition of economic growth itself. Economic growth can be seen as a positive change in the level of goods and services produced by a city over a certain period of time. An important characteristic is that economic growth is not the same in different sectors. Economic growth can be achieved when the rate of increase in total output is greater than the rate of increase in population of a city.

According to Harvard University three general theories of Economic Growth within cities [9] are those of Marshall-Arrow-Romer (MAR) theory (1986), Porter's theory (1990) and Jacobs theory (1969). These theories focus on knowledge spillovers and claim they are most effective in cities because communication between people is more extensive. The thoeries are based on in one company improves techniquely other companies near it will also benefit. These theories differ on whether monopolies or competition benifit the growth and whether the

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

influence is within the same industry or not (e.g. brassiere and lingerie industry). Other studies focus instead on the growth of countries instead of cities. Economist Alexander Cairncross wrote that the most important factors are investment, technical process, development and trade [2]. Economist Stanley Fischer focusses on the influence of macroeconomics (inflation, large budged deficits, distorted foreign echange markets) [8]. Economists Rudiger Dornbusch and Alejandro Reynoso claim the most important aspects differ per region [5].
In other words, there are many theories and there is much research on what plays an important factor in economic growth. Although MAR's Porter's and Jacob's theory do claim one of the reason for more economic growth in cities is due to more communication between people, much research into the connectivity between cities seem to be missing. One approach that has been taken is to look at where international companies are located. This only gives limited information however. Therefor we would like to see what information can be gathered from the internet by using search engines with input consisting of 2 cities.

3 Requirement Analysis

In this section, we first define the design goals. Then, we list the requirements which the application should meet. We use the MoSCoW method [3] as a prioritisation technique.

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 3.2 and are listed here, ordered by priority.

credible The client wants to dispute a widely spread belief. Therefore, the basis on which he does that must be sound.

understandable The results of the application should be visually understandable, so it is easy for the client to deduce conclusions from them. Additionally, retrievable numeric data enable the client to further investigate the results outside of the scope of the application, should the need arise.

scalable The client has expressed his concerns that restricting the application to a set of non-English domains might impair the probability that his research will be published in an acknowledged journal. Allowing for investigating other domains would greatly help the client in a later stadium.

plugable The client conveyed it might be interesting to let the application perform analysis on different data sets without the need of a developer.

3.2 Product Requirements

The MoSCoW method is a prioritisation technique used in management, business analysis, project management, and software development to reach a com-

mon understanding with stakeholders on the importance they place on the delivery of each requirement - also known as MoSCoW prioritisation or MoSCoW analysis. Four levels of priority are defined: must have, should have, could have and won't have (also known as would like).

3.2.1 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. A user must be able to input place names.
2. The system should display a map with the before mentioned places and the important connection they have to other places.
3. A user must be able to click on a connection between two places and get information about what kind of relations they have.
4. The strength of all relations must be displayed.
5. The user must be able to export the found connections and their strengths between places.

3.2.2 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. The application should be able to use multiple data sets.
2. The application should be able to group the relations (e.g. 'fish-trade' and 'finance' to economy, 'medicine' to health-care etc).
3. The user should be able to 'zoom' on places in order to see more/less connections to other places.

3.2.3 Could Have

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

1. The application could be able to use international names.
2. There could be a front end for the app.

3.2.4 Would Like

"Would like" requirements have been agreed upon as not important to deliver 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.

4 Framework and Tools

In this section, we evaluate what data sources are useful. Additionally, we discuss several methods and tools that can be helpful in storing and ingesting data. Furthermore, we describe numerous methods to filter and classify textual data. Then, we elaborate on different methods to perform queries with. We conclude with an overview of the available visualisation tools that can be used for displaying the results of the analysis.

4.1 Gathering the Data

4.1.1 Common Crawl

Common Crawl [4] is a freely accessible corpus of ~~the~~ pages across the web. Their data ~~are~~is updated and released on a monthly basis. Many researchers have used the data for ~~varying~~various purposes [22] [14] [21]. Since the project requires us to crawl the web (see section FIXME), the corpus is a very suitable candidate for us to work with.

The data of Common Crawl ~~come~~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 important 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 files describing all files present 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*} TODO: Reference the CC index.Since WAT- and WET-files can be generated from WARC-files, they only provide such indices for WARC-files. If no 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`[3] 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,

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

there is a problem with this: we are only interested in WET-files, but Common Crawl does not have WET-files indexed. This means one is not able to directly access WET-files based on the retrieved index and offset. We would therefore have to collect WARC-files and convert them to WET-files ourselves, requiring us to parse HTML for every document we are interested in.

A simple query on the latest index using the online interface[4] yields 1676 pages of 15000 entries each, which are roughly 25 million entries in total. It is very important to note that searching for a top level domain like `.nl` only the first page of every matching domain is included. To get all pages, additional queries for the remaining pages are to be performed.

4.1.2 Other Data Sources

Besides Common Crawl and Delpher, there are a plethora of other sources that might contain valuable information. The most notable is the Dutch royal library, Delpher[5]. 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, the Statistics Netherlands[6] is the governmental organisation collecting statistical data about the Netherlands and comes with an API. The NOW Corpus[7] collects newspaper and magazine articles through Google News and provides several tools to perform queries on this data. It is also available as a download.

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*} CommonCrawl is relevant, we can make a selection of data. We can do this by only selecting the data that mentions at least 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. We chose this because it is a fast exact string algorithm for which a well documented and maintained python library exists. This python library is called pyahocorasick [8] and is a fast and memory efficient python implementation of the Aho-Corasick algorithm. Using this python library we will reduce the amount of documents by disregarding the documents that don't mention at least two different cities. This reduces the amount of data that we need to parse in the next step of the pipeline.

We have chosen this approach because storing and indexing all documents/pages is not feasible as this requires large data storage. Because we don't have access to a fast and large data storage platform we will not store and index everything and then delete irrelevant documents.

[4] `http://index.commoncrawl.org/CC-MAIN-2017-13-index?url=*.nl&output=json&showNumPages=true`

[5] `http://delpher.nl`

[6] `https://www.cbs.nl/en-gb`

[7] `http://corpus.byu.edu/now/`

[8] `https://pypi.python.org/pypi/pyahocorasick/`

4.3 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 one that we think is the best choice. We will then briefly explain how it works and how we plan to use it.

4.3.1 Graph database, search engine or traditional database

To store the relevant data we can can choose from 3 categories that could best suit our needs, these categories are Graph Databases, Search Engines or traditional databases. Because we want to visualise the network of cities as a graph and are interested in relations between cities we want to use a Graph Database. A Search Engine is less useful for this project because as a user you don't want to search through the documents but you want to be able to explore the relations that can be extracted from the data. A Graph Database is a better choice compared to traditional relational databases because relations are the most important in the graph data model where this is not true for traditional relational databases. Therefore, we will be using a Graph Database.

4.3.2 Comparing Graph Databases

Now that we've established that we will be using a Graph Database, we need to choose which Graph Database we're going to use. To do this we made a table in which we compare aspects that are important to us of the available Graph Databases.

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

[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: `http://www.teradata.com/products-and-services/appcenter/`
[e] Using a separate tool

4.3.3 Neo4j

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 (see section **??** for a short explanation) graph database available. Additionally, it is free to use for non-commercial use. To

illustrate how scalable Neo4j is, consider that very large companies such as ebay, Cisco, Walmart, HP and LinkedIn[9] 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 out that the more natural representation of relationships resulted in significant performance increase gains [11].

There are some specific aspects of Neo4j that make it a very suitable candidate for the TODO: project. 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 decreasing performance. For our purposes, 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, we can put the graph to good use for visualising the global urban network. By adding a location property to a city, we can directly map nodes and relations to a geographical map. Most importantly, we can store indices of text files that mention the city as properties of nodes. That way, we are able to generate a subset of files that we can analyse for calculating the strength of the relationship between the nodes.

4.4 Grouping the Data

After the websites are stored in Neo4J we want to group each website according to subject: for example economy, politics and migration. There are two methods that can be taken for this: clustering and classifying. Clustering means grouping without first defining what groups (or classes) should be used while classifying does need classes do be defined first.
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.

[9] `https://neo4j.com/customers/`

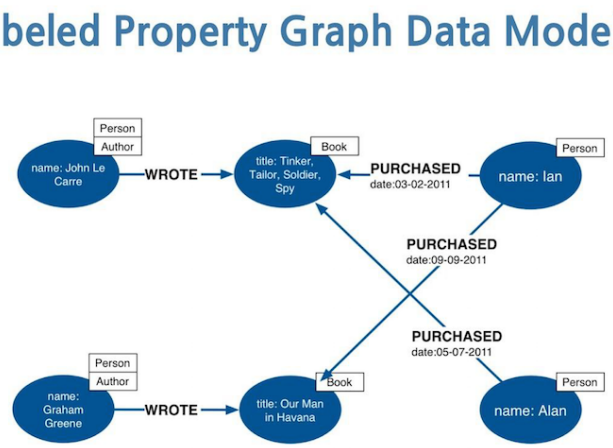


Figure 1: The Neo4j model

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.

4.4.1 Pre-processing

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

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 [17] and NLKT [7] (which uses the snowball version) have a dutch implementation for this, although it might need to be improved a bit.

4.4.2 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 [12] 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.

4.4.3 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 [10] 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 [12] and TensorFlow [23] for Python and Weka [16] 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 [13]. 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.

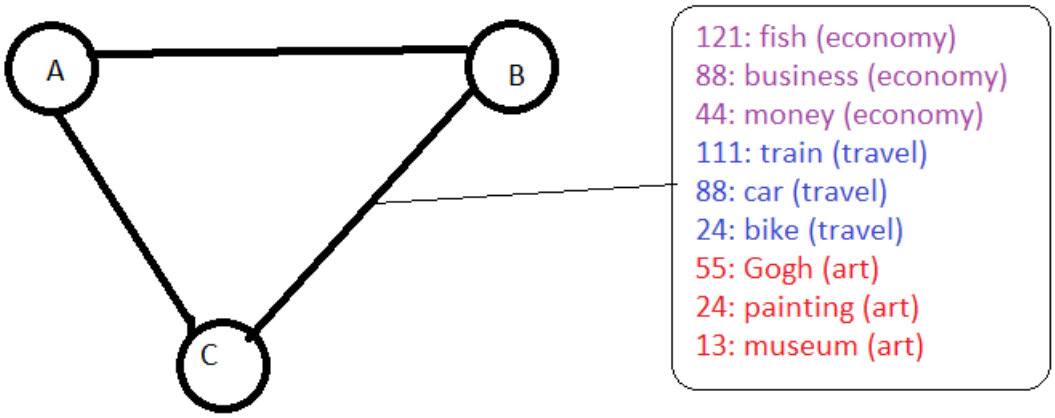
   For textual data especially support vector machines are recommended, so it is most likely we will choose that machine learning algorithm. Depending on whether we have time we might do some tests before making our decision however.

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.

4.4.4 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.



4.5 Interacting with the Data

For the application to be a success the processed data should be easily available to the end user. The data should be easy to query and should be presented in an accessible way. We researched several options to offer the end user this experience.

4.5.1 Query language

The first option was to develop an easy to use/learn query language specific to our domain (intercity relations). For this we designed a simple query language with the following syntax.

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

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

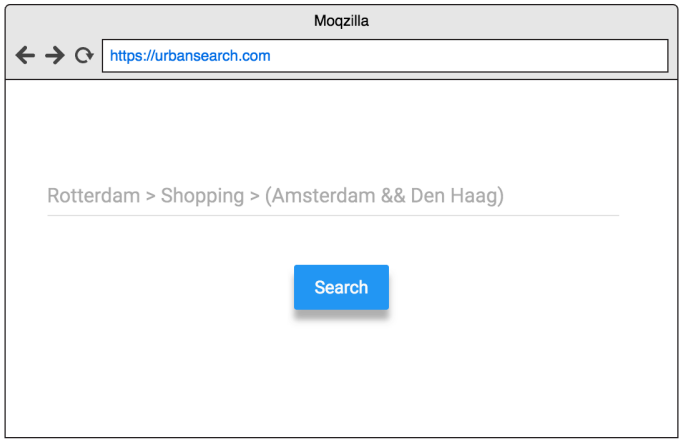


Figure 2: Example interface for the query language

4.5.2 Query composer interface

Another option was to offer the end user a "query composition interface". This interface would have the same possibilities as the former mentioned query language, but should be 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 was an interactive approach to querying data. This means that the user interacts with a map containing relations and cities. A very simple example is given in figure 4.

In this setup the user can click on cities and relations on the map which then triggers a query on the Neo4j back-end. The results can be visualised on the map (eg. showing information about the selected city).

4.5.4 Conclusion

Together with the client we decided that the best option was to go with the interactive map. This would lead to easy access to the data by the client and would fit better with the flow of use the client envisioned prior to the project. Also from the point of view of the researcher using the application, it fits a lot better in his/her work flow. The application is used to analyse and visualise

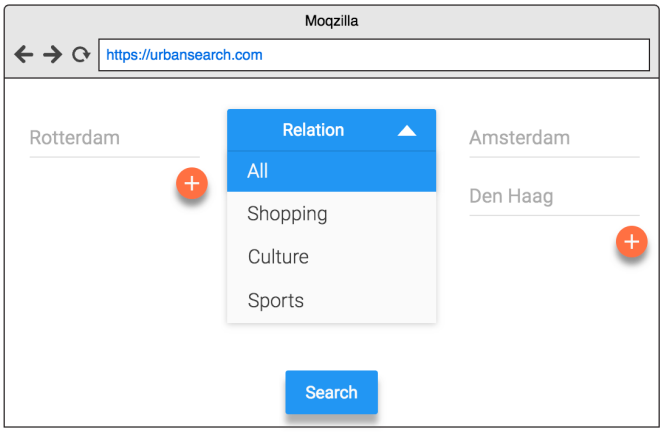


Figure 3: Example of a query composer interface

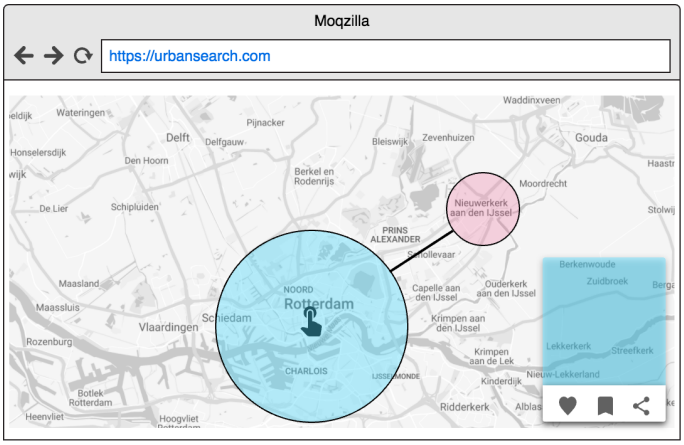


Figure 4: Example of an interactive map

relations between cities, if this can be done directly on the map it speeds up the users use of the system, since it is faster to select cities and relations on the fly. This helps make the application more intuitive since the user is probably more familiar with a map than with a new query language. The creation of complex queries is also made a lot easier. The user doesn't have to write or compose a complex query in advance but can do it directly on the map. So getting a visual representation of several cities, interconnected with multiple relations, doesn't mean writing or composing a very long query but clicking/selecting cities and relations on the map. Interaction directly with the map also reduces the need to go to a separate page to enter/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. A final advantage of using an interactive map over raw queries/query composition is that we don't have users entering invalid queries, this leads to less frustration using the system.

4.6 Visualising the Data

This section focuses on the visual representation of the processed data. Our goals are to present the data and the things we learned during the processing of the data in a way that is easy to comprehend for users and can help ease the interpretation of the data. To reach these goals we focused on the clients needs and desires. We discussed the preferences off the client and drew up a global plan, which we present below. We left space

4.6.1 Representing the data visually

Since we are dealing with strongly related data, which is comprised off cities and the relations between cities, it was a natural choice to represent the data as a graph. The choice was made, together with the client, to show the nodes and relations on a map. This was done because people are used to cities being visualised on a map and we think this will increase the ability of users to interpret the information in a productive manner.

4.6.2 Maps

We investigated two technologies we could use for the map on which we will display our data. The first one is Google Maps (GM). GM can be used freely and offers a lot of customisation options. The API is well defined and some of the group members worked with GM before. The second option we investigated was Leaflet. Leaflet is an open-source javascript library which provides responsive maps. It also has an well defined API and a lot of plugins. Both libraries are well suited for our needs. We decided to go with GM, because of the experience of the group members with using GM. Also we feel that there are more resources available on GM, which would help us if we get stuck with an issue.

4.6.3 Map clutter

5 Conclusion

First, we discussed related work. TODO:

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 TODO:

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] Alexander Kirkland Cairncross. Factors in economic development. Routledge, 2013.

[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] Rudiger Dornbusch and Alejandro Reynoso. Financial factors in economic development, 1989.

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

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

[8] Stanley Fischer. The role of macroeconomic factors in growth. Journal of monetary economics, 32(3):485–512, 1993.

[9] Edward L Glaeser, Hedi D Kallal, Jose A Scheinkman, and Andrei Shleifer. Growth in cities. Journal of political economy, 100(6):1126–1152, 1992.

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

[11] 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.

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

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

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

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

[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.