Delft University of Technology

Bachelor Graduation Project

Final Report

# UrbanSearch

*Authors:*
Tom Brunner
Marko Mališ
Gijs Reichert
Piet van Agtmaal

*Supervisor:*
Claudia Hauff
*Clients:*
Evert Meijers
Antoine Peris

June 23, 2017

TU Delft

# Preface and Acknowledgements

blah

# Summary

Instead of abstract, add a single page summary

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 [26]. 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 [24] or document search engines such as ElasticSearch [9] are best suited and describe machine learning algorithms for categorising data based on the co-occurrence of city names. Additionally, we present visualisation techniques to be able to intuitively analyse the results.

# Contents


**1 Introduction** **1**

**2 Related Work** **3**

**3 Problem Definition and Analysis** **5**
3.1 Problem Definition . . . 5
3.2 Problem Analysis . . . 5
3.3 Requirement Analysis . . . 6
3.3.1 User Stories . . . 6
3.3.2 Design Goals . . . 6
3.3.3 Product Requirements . . . 7
3.3.4 Design Decisions . . . 8

**4 Framework and Tools** **11**
4.1 High-level Overview . . . 11
4.2 Gathering the Data . . . 12
4.2.1 Common Crawl . . . 13
4.2.2 Other Data Sources . . . 13
4.3 Filtering Documents . . . 14
4.4 Extracting Relations from Documents . . . 14
4.4.1 Defining Categories . . . 15
4.4.2 Pre-processing . . . 16
4.4.3 Data set . . . 16
4.4.4 Modelling . . . 17
4.4.5 Remarks . . . 18
4.5 Storing and Ingesting the Data . . . 18
4.5.1 Storing Extracted Categories . . . 18
4.5.2 Graph Database or Traditional Database . . . 19
4.5.3 Comparing Graph Databases . . . 19
4.5.4 Using Neo4j for Storage and Ingestion . . . 20
4.6 Interacting with the Data . . . 21
4.6.1 Design for a Query Language . . . 21
4.6.2 Design for a Query Composer Interface . . . 22
4.6.3 Design for Querying Interactively . . . 22
4.6.4 Deciding on the Implementation . . . 23
4.7 Visualising the Data . . . 24
4.7.1 Representing the Data Graphically . . . 24
4.7.2 Using Geographical Maps . . . 24
4.7.3 Handling Map Clutter . . . 24
4.8 Validation and Verification . . . 24
4.8.1 Testing the Application . . . 24
4.8.2 SIG . . . 25
4.8.3 Evaluating the Classification . . . 25
4.8.4 Evaluation of Relation Scores . . . 26

**5 Implementation** **29**
5.1 Downloading and parsing indices . . . 29
5.2 Filtering . . . 30
5.3 Classification . . . 30
5.4 Storing the Data . . . 30

5.5 API . . . 31
5.5.1 General Remarks . . . 31
5.5.2 Classify Route: /classify . . . 31
5.5.3 Data-set Route: /datasets . . . 32
5.5.4 Documents Route: /documents . . . 33
5.5.5 Indices Route: /indices . . . 34
5.5.6 Classify Documents Route: /classify_documents . . . 34
5.6 Front-End . . . 35
5.6.1 Technical Overview . . . 35
5.6.2 Interfaces . . . 35
5.6.3 Recommendations . . . 35

**6 Project Evaluation** **37**
6.1 Fulfilment of Requirements . . . 37
6.2 Process . . . 38
6.2.1 Collaboration Between the Team Members . . . 38
6.2.2 Collaboration Between the Team Members and the TU Delft Coach . . . 38
6.2.3 Collaboration Between the Team Members and the Client . . . 38

**7 Discussion** **39**
7.1 Answering the research question . . . 39
7.2 influence of the answers . . . 39
7.3 issues faced and remaining . . . 39
7.4 Ethics . . . 40

**8 Recommendations** **41**

**9 Conclusion** **43**

**Bibliography** **45**

**A Better Code Hub Guidelines** **49**

**B Sig Feedback** **51**
B.1 week 5 . . . 51
B.2 week 9 . . . 51

**C User Manual** **53**

**D Developers Manual** **55**

**E Used Libararies** **57**

**F Validation and Verification results** **59**
F.1 Testing the Application . . . 59
F.1.1 Unit Tests . . . 59
F.1.2 Integration Tests . . . 59
F.1.3 System Tests . . . 59
F.1.4 Acceptance Tests . . . 59
F.2 SIG . . . 59
F.2.1 week 5 . . . 59
F.3 evaluating the classification . . . 60
F.3.1 Accuracy . . . 60
F.3.2 Confusion Matrix . . . 60
F.3.3 Precision, Recall, F1 and UAC . . . 60
F.4 Evaluation of relation scores . . . 60

# 1

# Introduction

With the development of future cities in mind, the interest in city networks has grown over the years. According to our client, a researcher of the citiesenvironment, do not fu built nction in isolation but are connected forming ”systems of cities”. However, appropriate information on how cities are connected and the strength of these connections is hard to find. A humongous amount of raw, unordered data is available to extract the relations from, however, there is no good way yet to process the data. According to Short et al. comparative statistics are not easily available and common assertions are repeated [32]. Although more research was published since then, for example the work of DeRudder et al. [7], using web data as a proxy for determining intercity relations is still unspoken of.

The huge amount of textual data generated online and the numerous historic archives, such as Delpher[1] and the British Newspaper Archive[2], are great sources of information on social and economic behaviours. The client's hypothesis is that ”semantic association”, the co-occurrence of cities within a single document, of cities can give insight in the connections between cities. These associations can be found using advanced text mining on newspapers and web pages. Similar to research efforts in other domains, such as financial trade [27] and sales forecasting [42], where socio-economic phenomenon are derived using web data, the client's wish is to develop an application that captures urban-urban interactions. These interactions should be retrieved from information corpora through the co-occurrence of geographical names in textual data. An example of how one could try to achieve this using the Google search Engine[3] is `"Rotterdam Amsterdam" OR "Amsterdam Rotterdam"`, which searches for the co-occurrence of Amsterdam and Rotterdam. However, manually processing all results a search engine yields is not feasible, because one would have to read each page to determine which types of relationships the page contains. An application should process all the pages that contain co-occurrences of cities to determine what type of relations, for example transportation or leisure, between cities can be extracted from the document. Thus, we will answer the following question:

Include something with ”design” in RQ

how can open data be leveraged such that a metric for the strength of relationships between cities can be defined and visualised?

Aanpassen nadat content van sections klaar is

First, we discuss related work in section 2. Second, we identify the requirements for a solution to the problem and discuss issues that might arise in section 3. Third, we develop a methodology for a framework that satisfies the requirements and tackles the issues in section 4. In the

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

fifth section we discuss evaluating the system. We conclude in section 6 with the results of our research.

# 2

# Related Work

Since the 1960's, the desire to understand the modernisation of the economy, as seen by the increasing concentration of jobs and the cooperation between remote firms resulted in a surge of work on intercity relationships. [39]. One of the most common methods used is the interlocking network model (INM)[36]. This model assumes cities have a flow of knowledge connection if there are offices of the same company in those cities. The biggest problem with this is that it is very limited. It only includes one relation type and it is disputable whether this is a good measurement for the relation [20] because the question remains how much these offices are used for the exchange of knowledge and what kind of knowledge ares exchanged.

The last ten years there has been a lot of development in the field of data production and processing. Information retrieved from existing technologies which have made the automatic extraction of information and labelling a normality, could have an important role in understanding interurban relationships.

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

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

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

# 3

# Problem Definition and Analysis

In this section first the problem definition will be introduced. Next, the analysis of this problem will be discussed. Last, the requirements following from this analysis and the wishes of the client are presented.

## 3.1 Problem Definition

As discussed in the previous sections the hypthesis the client proposed is if a semantic association of cities can give insight into on the actual relationships and strengths between cities. This hypothesis introduces the problem how software could be used to find and analyse these semantic associations. This lead to the following problem definition:

*How can open data be leveraged such that a metric for the strength of relationships between cities can be defined and visualised?*

## 3.2 Problem Analysis

The problem can be divided into four sub-problems that need to be addressed to solve the problem. These are Filtering, Classification, Storing Data and Data Visualisation & Export.

#### Filtering

The first sub-problem is filtering, which means searching through the available text data to find co-occurrences of cities and discarding text data that does not contain co-occurrences. This should reduce the amount of data and thereby potentially speed up the rest of process.

#### Classification

The sub-problem that arises after filtering is how to determine what relationships can be extracted from the text-data, this will be referred to as the classification of the text-data. This requires a method that reliably and efficiently processes the text-data and can be tuned to the clients wishes, meaning that the classification should output what the client desires.

#### Storing Data

Next, when the classification sub-problem is addressed the need arises to store the data and determine the strength of the relationships.

#### Data Visualisation & Export

When these three sub-problems have been successfully solved the last sub-problem that is left is how to combine the stored data and present it to a user, this means visualising and/or exporting the data in an accessible way.

## 3.3 Requirement Analysis

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

### 3.3.1 User Stories

Together with the client, several user stories are identified for interaction with the system. These are listed below.

As a user:

1. I want to be able to see all the identified relations between all cities, so that I can reason about interesting patterns.
2. I want to be able to access extracted relations 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.
3. I want to be able to see relation strengths, which can be expressed by counting the relations.
4. I want to be able to (de)select cities in the user interface, so that I can create a network of cities connected with relations. A network of cities consists of the cities as nodes and the different types of relationships as edges between them.
5. I want to be able to (de)select relations between cities in the user interface, so that I can inspect only the relations I am interested in. For example, as a user I might only be interested in the Transportation relationship between Amsterdam and Rotterdam.
6. 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. I want to be able to export an image of the map that I composed in the user interface so that I can use it for presentations, papers or educational purposes

### 3.3.2 Design Goals

The high-level design goals for this project have been provided by the client. These serve as a guideline to determine the priority label of the specific requirements as defined in section 3.3.3. The design goals are listed below, ordered by priority.

#### credible

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

#### understandable

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

#### scalable

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

#### plugable

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

#### exportable

Besides making the results available visually, all the relevant numeric data should also be exportable, for example in CSV format, so the client is able to process the data beyond the system.

#### fast development

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

### 3.3.3 Product Requirements

As mentioned in the introduction of section 3.3 we will be using the MoSCoW method prioritisation technique. Four levels of priority are defined: must have, should have, could have and would have (also known as would like). We also differentiate between functional and non-functional requirements.

#### 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 section4.2.1) and stored for further processing/access.
2. There has to be a way to export the relations between cities.
3. A machine learning algorithm should analyse and label the collected data to extract 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.

#### Should Have

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

1. Relations between cities should be accessible hierarchically. This means that there is the possibility to explore a relation and, provided that this relation has sub-types associated with it, the relation can be expanded in the different sub-types of the relation.

2. It should be possible to retrain the machine-learning algorithm on demand by feeding it a set of labelled documents.
3. It should be possible to add large data sets, e.g. with more than 1 million documents, on which the system can perform its data mining routines. This way a data set can be created that contains potentially interesting information for intercity relations.
4. The application should be able to deal with the fact that the same city can have different names in different languages/dialects. It should still be able to extract and group relevant data correctly (e.g. 'The Hague' and 'Den Haag' should be viewed as the same city).

#### 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 (see also section 4.2.2), a collection of over 60 million digitalised newspaper articles, books and magazines in the Netherlands, of age ranging from the seventeenth century to now, to characterise relationships between a region and cities outside that region. For example, the local newspaper of the province Gelderland writing about the city of Alkmaar. These relationships are either simple or complex information flows. A newspaper mentioning a city is considered a simple information flow, whereas multiple cities mentioned in a single document is a complex information flow. Both simple and complex flows reside on the basic properties of the document, such as the publication date. An illustration of this is given in figure 3.1.
2. The relations that are extracted from the data by the machine learning algorithm have to be visualised in a way that makes it easy to compare the different relations for the end user. For example, a split-screen comparison in the user interface or an export of graphs comparing selected relations.

#### 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`.

### 3.3.4 Design Decisions

To be able to have a fast development cycle and leverage our experience we chose to develop the application using Python. We plan to not only test the code we deliver thoroughly, but also to cross-validate the obtained results. The specifics of this validation protocol will be discussed in section 4.8.4.

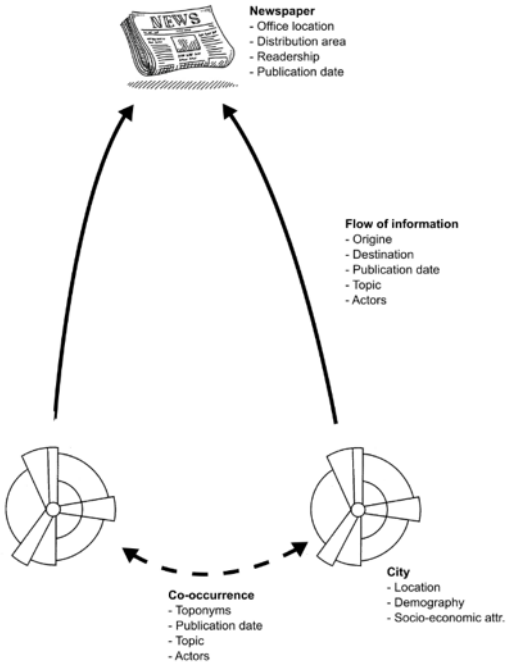


Figure 3.1: Solid lines represent simple information flows, whereas the dashed line is a complex connection of information. We focus on the part depicted by the dashed line.

# 4

# Framework and Tools

split in 04 and 05 and include code quality and validation

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

Marko's stukken aankondigen

## 4.1 High-level Overview

The figure below represents a high-level overview of the system. The most important inputs, outputs and steps in the are displayed. A more in depth explanation of the different steps of the process can be found in the following sections.

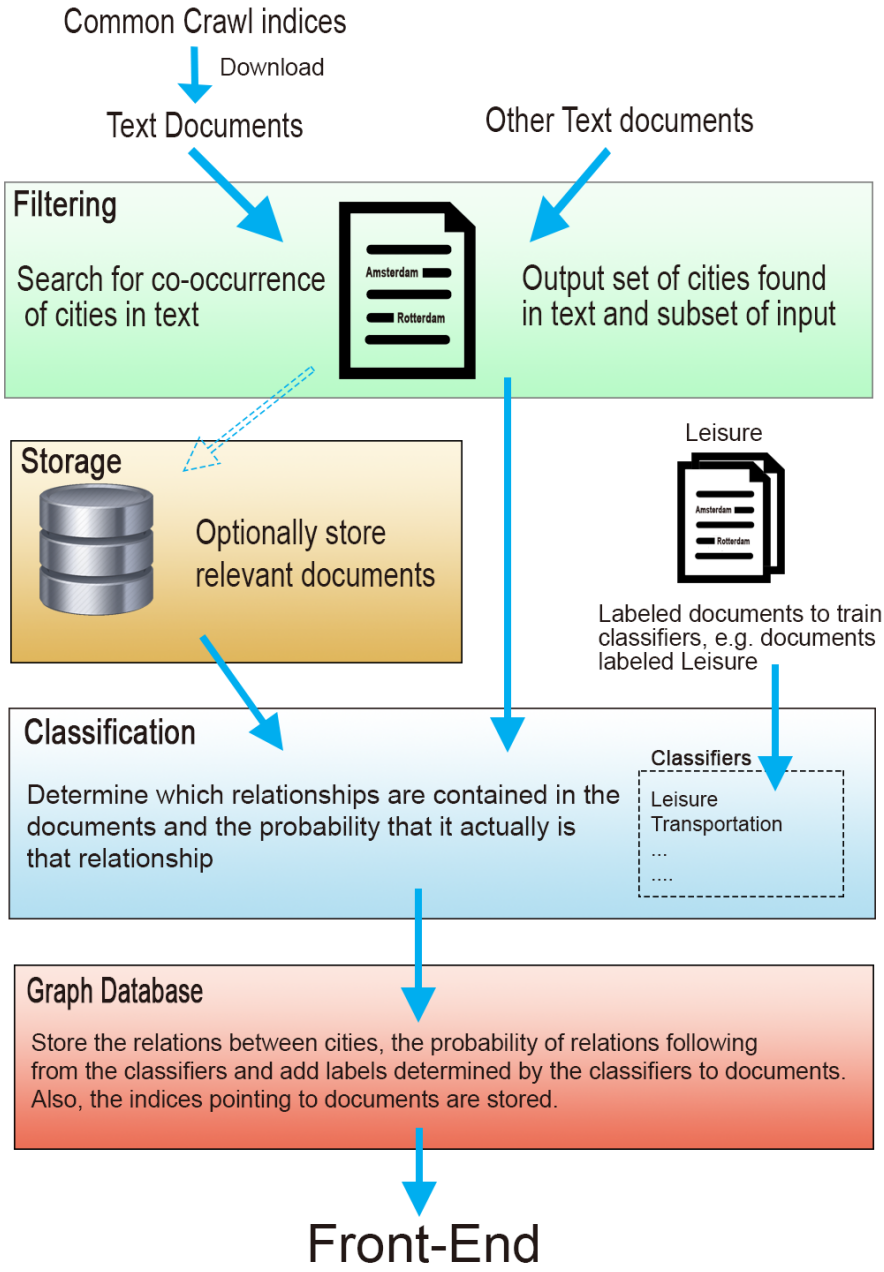


Figure 4.1: High-level overview of the system

## 4.2 Gathering the Data

As explained in section 3.3.2, data sources should be plugable. An initial corpus of documents is needed to base the project, which we will decide on in this section. Nowadays many people have access to the Web, and for a lot of people the Web is probably also their primary source of information. Next to that, the Web also contains vast amounts of documents which could shed some light on relations between cities. Therefore, the decision was made to use web-data as a data source. To avoid duplicate work, which would mean crawling the web, a logical choice is to use Common Crawl as a data source.

### 4.2.1 Common Crawl

Common Crawl [6] is a freely accessible corpus of pages across the web, updated and released on a monthly basis. Many researchers have used the data for various purposes [23, 33, 34]. 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[1]:

**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[2]. 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`[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, there is a problem with this: we are only interested in WET-files, but Common Crawl does not have WET-files indexed. We would therefore have to collect the WARC-files and convert them to WET-files ourselves, requiring us to parse HTML for every document we are interested in.

As mentioned in the design goals section not all available web-data will be used due to limited resources. A simple query `url=*.nl&output=json&showNumPages=true` on the CC-MAIN-2017-13 index using the online interface[4] yields 1676 pages. Pages in this sense are listings of 15000 indices, so there are roughly 25 million entries in total out of the 2.94 billion pages available in Common Crawl. It is very important to note that searching for a top level domain like `.nl` only includes the first page of every matching domain. To get all pages, additional queries for each site with more than one page are to be performed.

Image or description of a Common Crawl Index.

### 4.2.2 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[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, Statistics Netherlands[6] is the governmental organisation collecting statistical data about the Netherlands and comes with

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

| | |
|---|---|
| multi-pattern matching | 0.049831339 |
| plain string matching | 1.870154497 |

Table 4.1: Benchmark of multi-string vs. plain string matching

an API, making most of their data publicly accessible. The NOW Corpus[7] 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.

## 4.3 Filtering Documents

Because not all data from information sources such as Common Crawl is relevant to find relationships between cities, the data needs to be filtered. One way to do this, is to only select the data that mentions at least two different cities. Because the data is plain text, we need a way to scan through the text and determine if the text indeed has a co-occurrence of two different cities. Making use of the comparative analysis of Rasool et al. [29], we chose the Aho-Corasick algorithm [1], which is a multi-pattern exact string matching algorithm and is the driver of widely used tools such as `grep` [19]. The algorithm creates a finite state machine, where strings to match are final states. Since we are looking for the co-occurrence of cities, using a multi-pattern string matching algorithm is preferred over a plain string matching algorithm. This is especially well illustrated by table 4.1 below. The benchmark was performed on a string of 1500 characters, with a million iterations. In the table, the average speed of matching is shown in milliseconds.

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`[8] and is a fast and memory efficient implementation of the Aho-Corasick algorithm.

Using the Aho-Corasick 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. However, an interesting note is that there are pages with lists of cities contained, e.g. to let users select their place of birth. *{ added, please review}* These hardly represent intercity relations, so a maximum of 25 unique occurrences is used to cancel as much of those lists as possible beforehand. The threshold is decided upon by the client, after having analysed figure 4.2 and documents with 20 to 25 unique occurrences.

We make a selection of documents without storing the documents first, because storing all documents is not feasible due to storage constraints. For the .nl web pages only would need about 250GB of storage and to store all available documents around 250TB of storage would be needed. As we do not have access to a fast and large data storage platform, we will not store everything first and then delete documents that were filtered out. However, to test if finding and storing relationships between cities is fast enough when the documents are actually stored on disk a random selection of 1 million documents will be downloaded. Processing the already stored documents could finish within one day[9] whereas downloading all documents will most certainly take multiple days.

## 4.4 Extracting Relations from Documents

Now that a selection of documents has been made, we can make an attempt to identify the relations between cities based on these documents. Since labelling every document by hand is not feasible, an automated approach is desirable. One way to automate this process is by identifying intercity relations using machine learning. Machine learning algorithms can be roughly divided

[7] `http://corpus.byu.edu/now/`
[8] `https://pypi.python.org/pypi/pyahocorasick/`
[9] On a virtual server with 8GB RAM, 4 CPUs and 100GB of HDD storage.

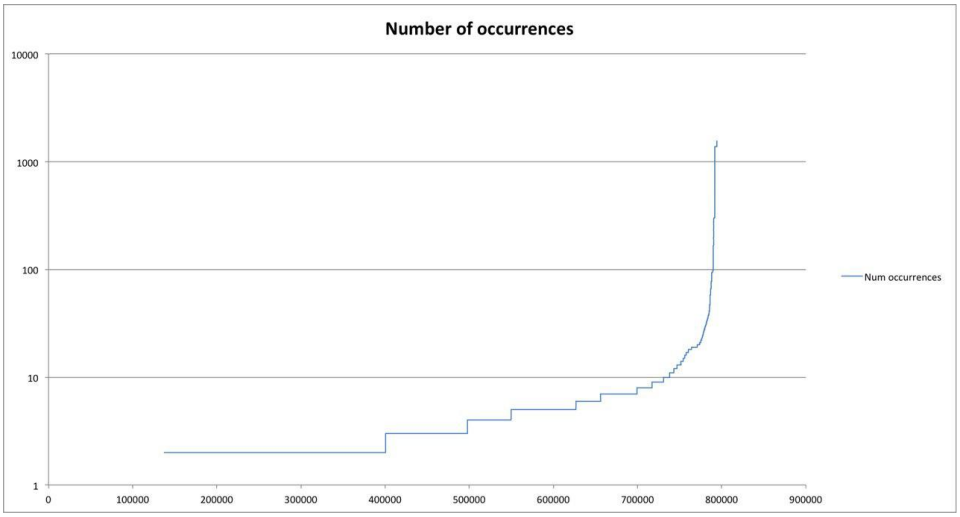


Figure 4.2: Number of documents plotted against the number of unique occurrences contained in these documents.

into two distinct groups: (1) supervised and (2) unsupervised algorithms. Supervised algorithms expect an input set and a corresponding output set, with which a model is trained to predict unseen instances of the problem. Unsupervised algorithms identify clusters of entities, such as documents or pictures, based on similarities in the feature set corresponding to said entity.

We decided to go with the supervised approach, mostly due to the fact that the training and tweaking of supervised algorithms can be done faster compared to unsupervised algorithms. This stems from the fact that we do not need the complete data-set to start training a supervised model, while for the unsupervised case the complete set is needed. However, a good quality training set is required for the supervised algorithm, which might be difficult to find. However, we think finding such a data set is doable within the time constraints.

### 4.4.1 Defining Categories

Our choice of using classification has naturally lead to the need for categories we want to identify within the collected documents. Together with our clients we identified the following categories which are useful to identify from the collected documents. For each category, a fictional article title is given to illustrate what an article in that category would be about.

1. Collaboration - TU Delft builds software for hospital in Leiden
2. Commuting - Most commuters between Almere and Amsterdam
3. Education - University of Amsterdam popular among students from Utrecht
4. Leisure - Blijdorp Zoo welcomes children from asylum seekers' centre Ter Apel
5. Residential mobility - More and more people leaving Maastricht for Den Bosch
6. Shopping - Shops in Breda struggling to compete with mall in Roosendaal
7. Transportation - Diary farms around Leeuwarden export most milk
8. Other

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

### 4.4.2 Pre-processing

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

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

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

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

### 4.4.3 Data set

Before we can start labelling the training data, we need to collect labelled data that can be used as input for the classifier. To collect this data we have considered several options.

The first option is to query for documents from news(paper) sites. Since the documents are categorised by professionals, we may assume these document will be labelled correctly. This method could thus provide us with a reliable training set. Unfortunately, the categories that we identified with our client do not match typical newspaper categories, so this approach was not suitable for us.

Another approach is to use Google Custom Search to obtain results from Google, using the categories the client provided us with as keywords. The main disadvantage of this approach is lack of control over the files that get added into the data set. This way documents that get returned by the query are not analysed on desirable content but are added immediately. An example of a page that is returned for the query "woonwerkverkeer"(commuting) is given below. This page, although it does contain information about commuting, contains more useless information, like the side-menu, than useful information.

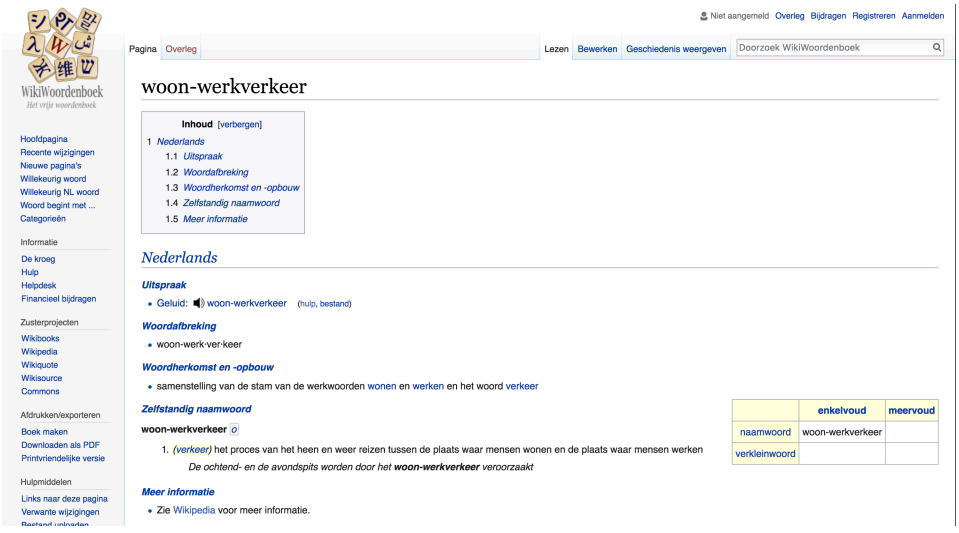


Figure 4.3: Example of an undesired result obtained with Google Custom Search

Finally, we decided to provide our clients with a "labelling interface". This way, we have total control of the documents that are added to the data set. The documents are labelled by experts in

the field of the built environment so we may assume these documents will represent the labelled categories well. The labelling interface provides the user with a document from the set of collected documents. It allows the user to label these documents with zero or more categories, after which the document is saved to the training set(s) corresponding to correct categories. If no category is selected, the document is discarded from the training set.

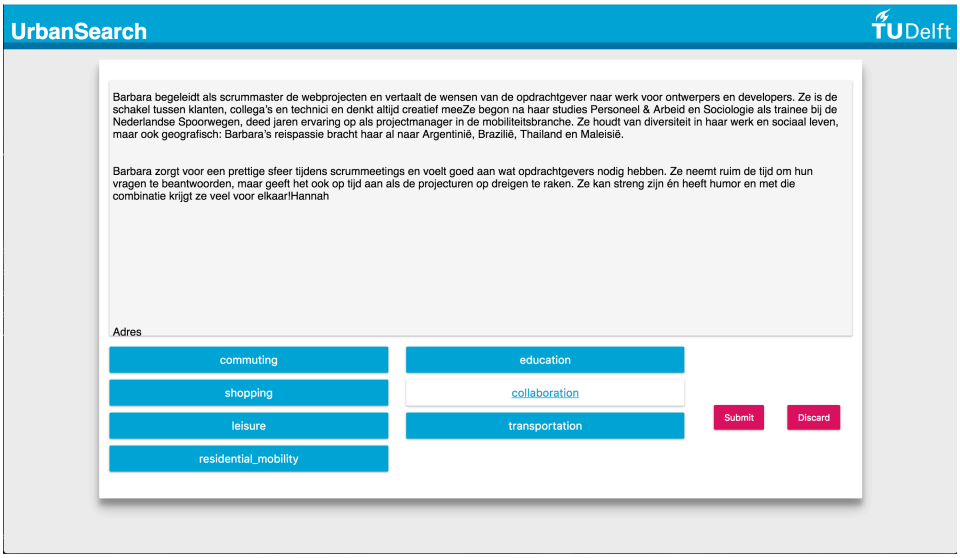


Figure 4.4: Labelling interface

### 4.4.4 Modelling

When considering classification, there are a plethora of algorithms available. When choosing the right algorithm for a problem, several factors should be taken into account[22]. These are:

**Accuracy** How well the algorithm separates the documents.

**Training Time** How long it takes to train the algorithm.

**Linearity** Some problems can be solved by splitting classes using a straight line. For other problems this approach is not feasible.

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

**Number of Features** A large number of features can make some algorithms slow. Extracting features from text-data often results in a huge feature set (65000+ in our case).

**Special Cases** Some learning algorithms make particular assumptions about the data or the results (eg. rank prediction, count prediction). This way we can increase desirable properties like accuracy of the prediction or improved training times.

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

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

set but occur often in a document are assigned a high weight. Words that occur in many documents in the complete document set get awarded a low weight[28]. Using TF-IDF our features become words with weights associated to them.

**Dimensionality Reduction** Since we are working with text documents and our features are words with TF-IDF weights we can assume that our feature set will be very large (65000+). The total number of features determines how fast we can train our model and has implications regarding over-fitting [31]. To reduce the number of features we considered different techniques from [31]. Since we have no time to test all the techniques, we decided to select the top ten percent of our features (based on the TF-IDF weights). In [43] it is stated that a dimensionality reduction with a factor ten using this approach does not lead to a loss in accuracy when classifying text documents. To provide an easy way to add different types of dimensionality reduction techniques later, we will keep the code for defining new Scikit pipelines, which are the basic construct used for creating our classifiers, easily extendable.

**Classification** Even after applying dimensionality reduction which we discussed in the previous section, we are left with a lot of features (6500+). Thus, we need an algorithm that works well with a feature rich problem. From [22] we know Support Vector Machines(SVM) is a algorithm that works well with feature rich problems. Also [31] claims SVM is one of the best techniques when considering text classification. This combined with the fact that Scikit offers an easy to use implementation of SVM has lead us to use SVM as our classification algorithm. The concept of a SVM is that of a hyper-plane that divides two distinct sets, while trying to maximise the margin between these sets [38].

### 4.4.5 Remarks

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

## 4.5 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.

### 4.5.1 Storing Extracted Categories

The categories that are extracted from documents, as described in the previous section, need to be stored. We want to be able to apply different models on the data and we also want access to the raw data.
To keep this flexibility and to maintain scalability, we save the document information in conjunction with the category. The documents that are deemed useful are stored on disk, pointed to by the document information node. Occurrences of cities in a document are stored as a relation of these cities to the document. This means that if a relation ”transportation” is extracted from a document that contains the cities ”Rotterdam” and ”Amsterdam”, we create an document node and create relations from Amsterdam to the node and from Rotterdam to the node. In the end, when all documents have been stored and relations created, the relation between two cities can be computed by counting document in which they both appear, grouping by category.
Considering the fact that relations are bidirectional, meaning a relation of ”Transportation” between ”Rotterdam-Amsterdam” implicates a relation of ”Transportation” between ”Amsterdam-Rotterdam” as well, we only need one relation between two distinct cities.

### 4.5.2 Graph Database or Traditional Database

To store the relationships and documents discussed in the previous subsection, we look into two possibilities: (1) graph databases and (2) traditional relational databases. A database is preferred over for example an in-memory system since the client has asked for both visualisation and export functionality. Databases are designed for this purpose.

Because visualisation of the network of cities as a graph is an important part of the application, and relations between cities play a key role in the system, we need a database that is designed for these features. Relations are the most important in the graph data model, where this is not true for traditional relational databases. Vicknair et al. stated that a graph database such as Neo4j has an easily mutable schema, where a relational database schema is less mutable [41]. Furthermore, the edges between nodes in a graph database can have properties, which is exactly what the envisioned data structure should be for this application. Lastly, if the desire arises to find indirect relationships between cities then a graph database is most appropriate choice. For example, if the client wishes to find out how Alkmaar is connected to Tokyo via other cities then the need for fast graph traversal arises. According to the graph database Neo4j their graph traversal is already 60% faster than a relational database for a depth of just 3[10]. Therefore, we are confident that a graph database is the best choice.

### 4.5.3 Comparing Graph Databases

Next, the type of database needs to be selected. For this, six of the most popular databases according to the solid IT Graph DBMS ranking[11] are compared. This rating is established using multiple parameters, among these parameters is the number of mentions on websites and in job offers. Next to that, the parameters also include the number of searches, relevance in social networks and the general interest in the system[12]. These six most popular Graph Databases are rated on five important aspects. These are, is the graph database open-source, scalable, free, does it support Python and has built-in visualisation. Open-source is important because the application should be as transparent as possible to achieve maximum credibility, therefore it helps that the graph database is open-source. A scalable database is necessary to achieve the design goal ”scalable”. Scalable in this sense means that should the system be extended to many more documents and/or cities, the database should be able to handle such extension. Next to that, a free graph database is preferred so we won't leave the client with costs to keep the application running. The Python and Built-in visualisation aspects are important for fast development, as built-in visualisation allows visualising the data before building a front-end for the application.

| *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

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

know the graph database and functionality. Secondly because it is by far the most popular graph database[11]. Thirdly, since Neo4j is the most popular graph database, the support community and amount of available examples is large.

### 4.5.4 Using Neo4j for Storage and Ingestion

Neo4j is a highly scalable graph database that leverages data relationships as first-class entities [24]. It is the single highly scalable, fast and ACID compliant graph database available. ACID stands for the four properties atomicity, consistency, isolation and durability of transactions in database systems that ensure reliability for query results [12]. The scalability of Neo4j comes from the fact that is easily spread across clusters, which provides a read throughput that scales linearly. Next to that, when spread across clusters Neo4j provides data redundancy and still high write speed [25]. 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[13] use it in their mission-critical systems. Holzschuher and Peinl compared the performance of Neo4j to the more classic and commonly used NoSQL databases and found that the more natural representation of relationships resulted in significant performance increase gains [13]. Jouili et al. concluded that Neo4j has a read-only performance which is comparable to other graph databases [18]. Compared to other databases Neo4j is slower with writing. However, the application will eventually do more reading than writing making writing a less important aspect.

The model of Neo4j is explained by three key concepts. 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 4.5. It consists of nodes, relationships (edges), properties (within the nodes and relations) and labels (coloured blocks above the nodes).

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

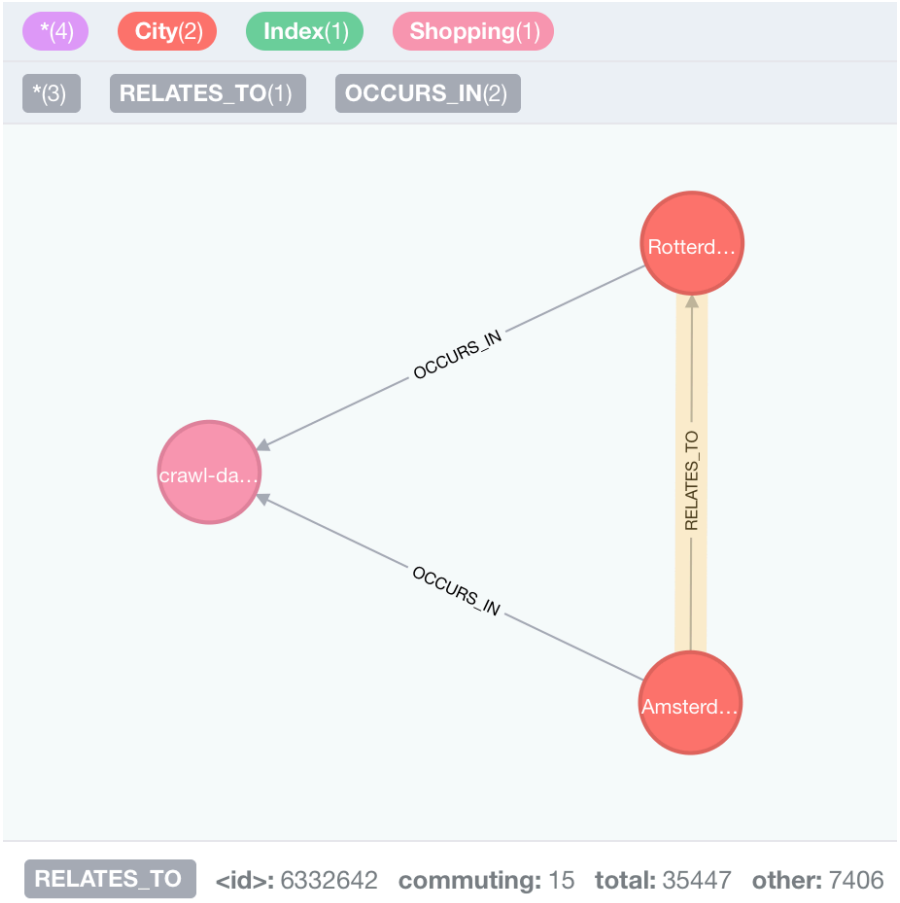


Figure 4.5: The Neo4j model, as it appears in the default user interface

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.

## 4.6 Interacting with the Data

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

### 4.6.1 Design for a Query Language

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

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

In figure 4.6, an example is shown that queries the ”Shopping” relation between Rotterdam and Amsterdam and between Rotterdam and Den Haag.

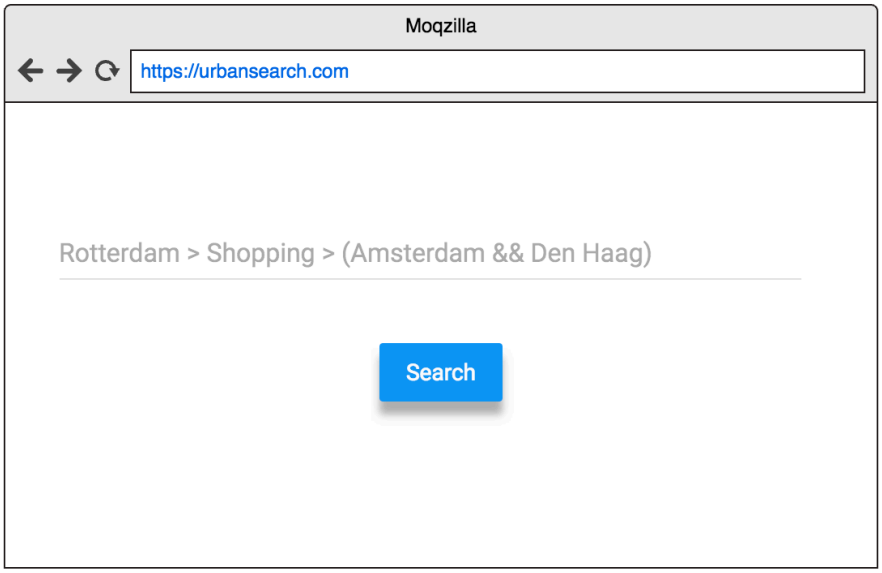


Figure 4.6: Example interface for the query language

### 4.6.2 Design for a 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 4.7.

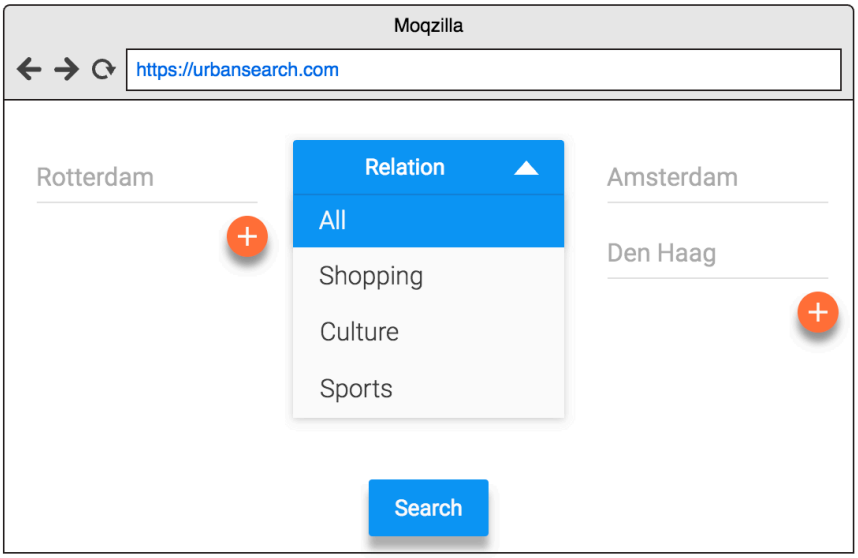


Figure 4.7: Example interface for the query composer

### 4.6.3 Design for Querying Interactively

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 4.8.

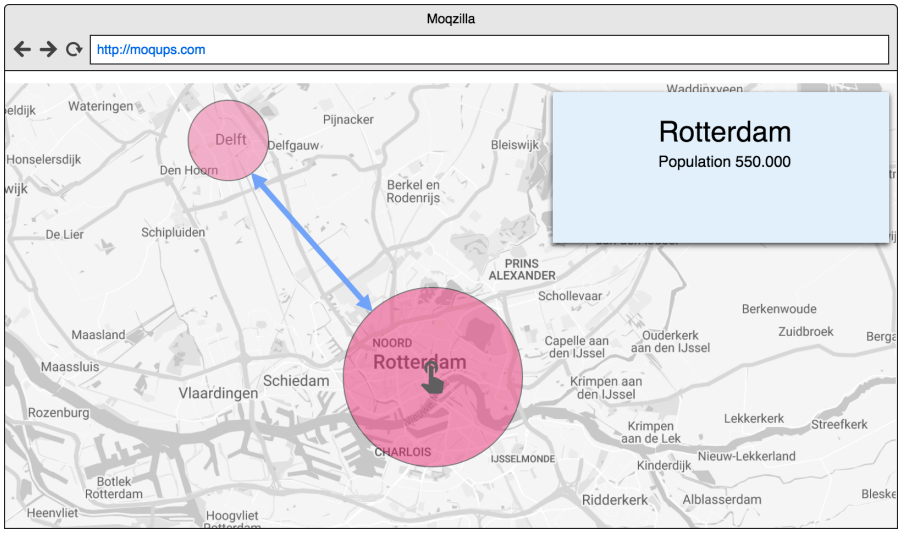


Figure 4.8: Example 1 of an interactive map

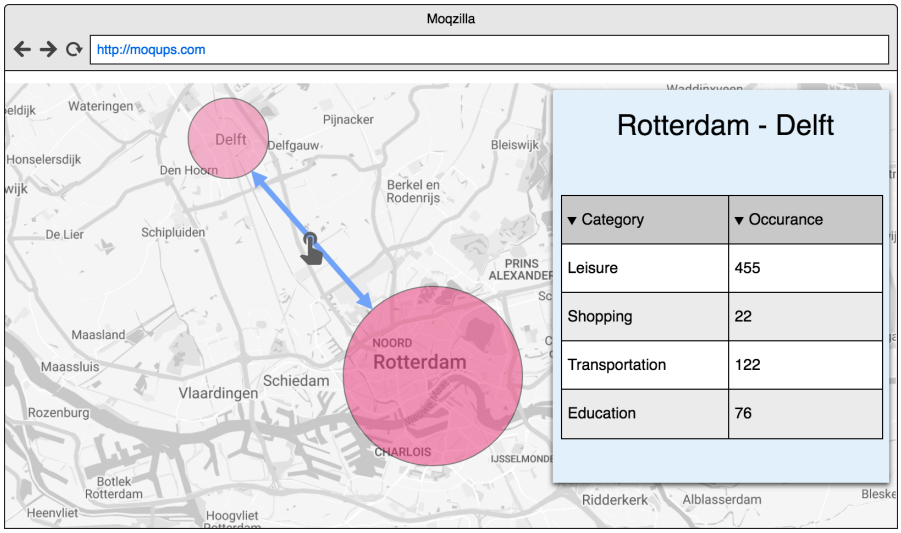


Figure 4.9: Example 2 of an interactive map

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.

### 4.6.4 Deciding on the Implementation

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 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.

## 4.7 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.

### 4.7.1 Representing the Data Graphically

Since we are dealing with strongly related data, it is a natural choice to represent the data as a graph. We chose, 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 believe this will increase the ability of users to interpret the information in a productive manner.

### 4.7.2 Using Geographical Maps

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

### 4.7.3 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.

## 4.8 Validation and Verification

In this section, we first describe how the system is tested and how we verify the quality of the code with SIG [35]. Afterwards we define a protocol with which the results of the system can be evaluated for correctness. We will first evaluate the results of the classification, and then the relation scores. These are related in the sense that a relation score is calculated by the number of occurrences in labelled documents, so the correctness of labelling affects the correctness of relation scores.

### 4.8.1 Testing the Application

We will test the program using four different testing methods. The first is unit testing, which tests the separate components individually. Next comes integration testing, to see how well different components work together. Afterwards we use system testing for testing the different system

components. Lastly, acceptance testing is used for testing how well the clients think the program works.

#### 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.

#### Integration Testing

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.

#### System Testing

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

#### Acceptance Testing

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

### 4.8.2 SIG

SIG [35], short for software improvement development group, is an organisation that analyses the code of projects to give insights in the quality of how the code is written. A high score means the code is highly maintainable and is kept simple. SIG includes Better Code Hub [14] which checks our code according to 10 guidelines as can be seen in appendix A. The great thing about Better Code Hub is that it can be run at anytime. We can check Better Code Hub whenever, whilst for SIG we have to send in our code and wait for feedback.

### 4.8.3 Evaluating the Classification

There are several ways to evaluate machine learning algorithms. We will base our evaluation of the classifier on the guidelines of the Microsoft Azure Machine Learning evaluation model [21]. According to the page binary classification can be evaluated with the following metrics: Accuracy, Precision, Recall, F1 and AUC.

#### Accuracy

Accuracy is the proportion of correctly classified instances. This however a poor indication of how well the classifier works. For instance if you have a test set of 100 websites, of which 90% belongs to Category A. Than if the classifier simply predicts all websites to belong to category A the accuracy would be 90%. It would seem the classifier performs well, but it actually fails to classify the other 10% of the websites correctly.

#### Confusion Matrix

A page can only either belong to class A (positive), or not belong to class A (negative). If a page is predicted by the classifier correctly it is called true positive (TP) or true negative (TN). If the classifier predicts the page incorrectly it results in a false positive (FP) or false negative (FN). This can be seen in the confusion matrix in figure 4.10.

| n=165 | **Predicted: NO** | **Predicted: YES** | |
|---|---|---|---|
| **Actual: NO** | TN = 50 | FP = 10 | 60 |
| **Actual: YES** | FN = 5 | TP = 100 | 105 |
| | 55 | 110 | |

Figure 4.10: confusion matrix [14]

#### Precision, Recall, F1 and UAC

The **precision** of the classifier is the proportion of positives that are classified correctly: $\frac{TP}{TP+FP}$. This is used for questions such as "Out of the pages that were classified as category A, how many were classified correctly?".

the **recall** of the classifier is used to answer the question "What percentage of the pages that fit category A were classified correctly?". In other words: $\frac{TP}{TP+FN}$.

The **F1 Score** uses both precision and consideration. It is computed by using the following formula: $F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$. The F1 score summarised evaluation in a single number, but for evaluation it is better to use recall and precision to understand the behaviour of the classifier.

The **Receiver Operating Characteristic (ROC) curve** and the corresponding **Area Under the Curve (AUC) value** can be used to inspect the true positive rate (Recall) vs. the false positive rate $\frac{FP}{FP+TN}$. To do this, the possibilities pages are correctly classified are needed. For each threshold on these probabilities for the classifier, the true positive rate and the false positive rate are calculated and are plotted in a graph, which results in something like 4.11. The closer the ROC curve is to the upper left corner, the better the classifier's performance is. When close to the diagonal of the plot, the classifier tends to make predictions close to random guessing. The UAC value is the are under the ROC curve.

### 4.8.4 Evaluation of Relation Scores

Evaluating relation scores is done differently. An important factor here is that cities have a natural relation due to their geographical position [37], so one would expect cities that lie close to each other are more related than cities that are on different sides of the country. This natural relation can be represented using the Gravity Model by Reilly [30]. The Gravity Model describes that the expected relation between two cities is based on the population of the two cities and the distance between these cities. A relation between two cities that is extracted from the data should thus expose a similar relative score as they would for the gravity model. Consider for example Amsterdam and Hoofddorp, which are cities that lie close to each other. Amsterdam is a large city, whereas Hoofddorp is much smaller. However, due to their close geographical position, the score that results from the Gravity Model would be high. If they turn out to have a very high score in

[14] https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-evaluate-model-performance
[15] https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-evaluate-model-performance

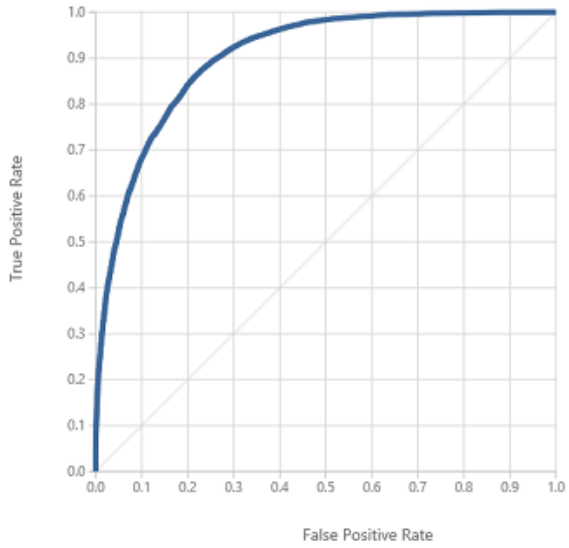


Figure 4.11: ROC / UAC graph [15]

our system, that would imply that the system is correct. Besides the Gravity Model, one can rely on the opinion of an expert in the field of urbanism that can judge whether an extracted relation is close to reality or not. We therefore agreed with the client that they would decide on a small set of relations whether they are correct. Lastly, the relations in the Randstad, a large urban area with the four largest cities of the Netherlands, have been examined before on the basis of firms [40]. These relations can be compared to those extracted by the system.

# 5

# Implementation

## 5.1 Downloading and parsing indices

As can be seen in figure 4.1, the first step of the process is to download data from Common Crawl. This requires functions that will parse the Common Crawl indices and gather the data that corresponds to these indices.

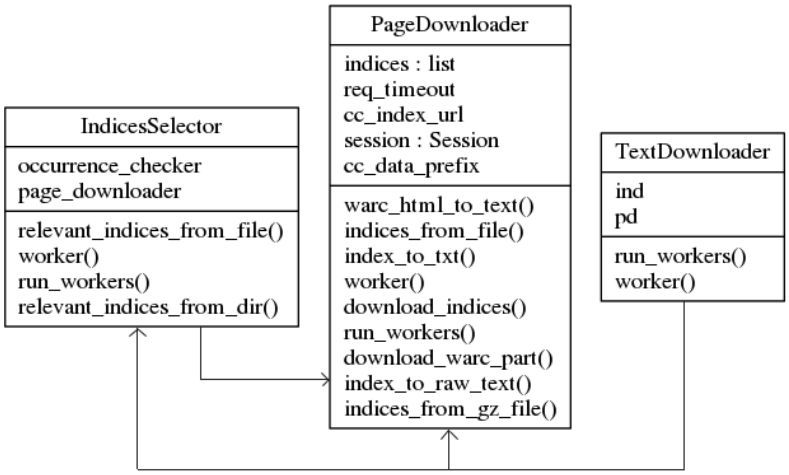


Figure 5.1: Diagram of downloading and parsing classes

The parsing of indices and downloading the data depends on the IndicesSelector and PageDownloader class, methods from these classes are called by the TextDownloader, as can be seen in figure 5.1. The classes contain workers, these workers can be run using the `run_workers()` method which will utilise Python multiprocessing[1] to run workers in parallel. Running these workers in parallel speeds up the downloading of partial WARC files and parsing of Common Crawl indices.

The first step in parsing the Common Crawl indices is filtering out the indices that have a HTTP Status Code [2] other than 200, as only these indices with these HTTP Status Code would be useful.

```
1 def _useful_responsecode(self, index):
2 # Check responsecode of index to determine if it's useful to download
3 # the part. HTTP 200 is useful, other than 200 will be discarded.
4     if index:
```

[1] https://docs.python.org/3.5/library/multiprocessing.html
[2] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

```
        return True if int(index['status']) == 200 else False
    return False

def _clean_indices(self, indices):
    # Removes useless entries with status code other than 200
    for index in indices:
        if not self._useful_responsecode(index):
    indices.remove(index)
```

Listing 5.1: Initial implementation

At first a simple implementation was used as can be seen in listing 5.1. However, a remove operation on a list in Python has a time complexity of O(n), the implementation of `clean_indices()` loops over all indices and removes it if it has status other than 200, which means that this function has a complexity of $O(n^2)$. To improve this, a regular expression to search the string for the status before parsing to JSON is used. This way, the list will never contain any indices with a HTTP Status Code other than 200 this is because the function will be called in a list-comprehension (see listing 5.3). This resulted in a speedup of about 5.6 times compared to the $O(n^2)$ method.

```
def _useful_str_responsecode(string):
    if string:
        return int(re.search('\"status\": \"(\w+)\",', string)
                   .group(1)) == 200
```

Listing 5.2: Regex solution

```
with gzip.GzipFile(filename) as gz_obj:
    # Remove the garbage before {, parse to json and add to list
    indices = [json.loads('{' + x.split('{', 1)[-1]) for x in
               gz_obj.read().decode('utf-8').strip().split('\n')
               if self._useful_str_responsecode(x)]
```

Listing 5.3: List comprehension creating list of indices

While parsing the index the memory footprint of the indices is also reduced, with use of the method from listing 5.4. Parsing every key of the index to JSON means the resulting JSON dict is 480 bytes, where the size of the stripped index is 288 bytes. The size of the objects is determined using the Python built-in `sys.getsizeof()` method.

```
def _remove_keys(json_dict):
# Strip all key-value pairs other than digest, length, offset & name
    return {k: v for k, v in json_dict.items()
            if k in ['digest', 'length', 'offset', 'filename']}
```

Listing 5.4: Reducing memory footprint

## 5.2 Filtering

The next step in the process is filtering documents as explained in section 4.3 and can be seen in figure 4.1. The implementation depends on the pyahocorasick library, which checks the page and tries to match strings within the page. We can supply this class with a list of cities, however, by default the application will retrieve a list of cities from the database.

## 5.3 Classification

## 5.4 Storing the Data

mention multithreading neo4j (embedded)

## 5.5 API

To provide the users an easy way of interacting and controling the system we decided to develop an web API. With this API the different parts that compose the complete UrbanSearch system are easily accessible. During the development of the API we have tried to adhere to best-practices and community standards as described in [**?** ]. The sections below will describe the parts of the system that are controlled by the API in more detail. Finally we will give some recommendations for the API which we feel would be a good addition/improvement of the API.

### 5.5.1 General Remarks

All routes in the API start with the ”/api/v1” prefix. The routes below will be referred to without this prefix to keep the text concise. The API always returns a 200 status code, the response body also contains a status code which indicates if a request was handled successfully.

### 5.5.2 Classify Route: /classify

The classify route is meant as an easy means of labelling a provided document with a category or the probabilities of said document belonging to a set of predefined categories. The available subroutes are specified below.

**/**

**/predict** Predicts the category of the document that is submitted in the body of the request.

**Request:**

| | |
|---|---|
| Method | POST |
| Content-Type | application/json |

**Request data:**

| Property | Required | Description |
|---|---|---|
| document | True | String containing the document that needs to be labelled |

**Response:**

| Property | Description |
|---|---|
| status | Status code for the response |
| category | The category that was predicted for this document |
| error | Boolean indicating if there was an error during the processing of the request |
| message | Message containing extra information about the response |

**/probabilities** Returns the probabilities of the supplied document belonging to each of the predefined categories.

**Request:**

| | |
|---|---|
| Method | POST |
| Content-Type | application/json |

**Request data:**

| Property | Required | Description |
|---|---|---|
| document | True | String containing the document that needs to be labelled |

**Response:**

| Property | Description |
|---|---|
| status | Status code for the response |
| probabilities | The probabilities per category that are predicted for this document |
| error | Boolean indicating if there was an error during the processing of the request |
| message | Message containing extra information about the response |

### 5.5.3 Data-set Route: /datasets

The datasets route is meant for extending and querying information about the data-set which is used to train classifiers.

**/append** Appends a document to the data-set of the category specified in the request.

**Request:**

| | |
|---|---|
| Method | POST |
| Content-Type | application/json |

**Request data:**

| Property | Required | Description |
|---|---|---|
| document | True | String containing the document that needs to be labelled |
| category | True | String specifying the category of the data-set we want to append this document to |

**Response:**

| Property | Description |
|---|---|
| status | Status code for the response |
| error | Boolean indicating if there was an error during the processing of the request |
| message | Message containing extra information about the response |

**/append_all** Appends a document to the data-set of all the categories specified in the request.

**Request:**

| | |
|---|---|
| Method | POST |
| Content-Type | application/json |

**Request data:**

| Property | Required | Description |
|---|---|---|
| document | True | String containing the document that needs to be labelled |
| categories | True | List of strings specifying the categories of the data-sets we want to append this document to |

**Response:**

| Property | Description |
|---|---|
| status | Status code for the response |
| category | The category that was predicted for this document |
| error | Boolean indicating if there was an error during the processing of the request |
| message | Message containing extra information about the response |

**/create** Creates a data-set from all the category specific data-sets.

**Request:**

| | |
|---|---|
| Method | GET |

**Response:**

| Property | Description |
|---|---|
| status | Status code for the response |
| error | Boolean indicating if there was an error during the processing of the request |
| message | Message containing extra information about the response |

**/create/categoryset** Creates a new file for the category specified in the request. In this file we will save the documents that are submitted for this category

**Request:**

| | |
|---|---|
| Method | POST |
| Content-Type | application/json |

**Request data:**

| Property | Required | Description |
|---|---|---|
| category | True | The category for which we want to create a file |

**Response:**

| Property | Description |
|---|---|
| status | Status code for the response |
| error | Boolean indicating if there was an error during the processing of the request |
| message | Message containing extra information about the response |

**/init_categorysets** Appends a document to the data-set of all the categories specified in the request.

**Request:**

| | |
|---|---|
| Method | POST |
| Content-Type | application/json |

**Response:**

| Property | Description |
|---|---|
| status | Status code for the response |
| error | Boolean indicating if there was an error during the processing of the request |
| message | Message containing extra information about the response |

**/lengths** Returns the lengths of the different category-sets

**Request:**

| | |
|---|---|
| Method | GET |

**Response:**

| Property | Description |
|---|---|
| lengths | The lengths of the data-sets per category |
| status | Status code for the response |
| error | Boolean indicating if there was an error during the processing of the request |
| message | Message containing extra information about the response |

### 5.5.4 Documents Route: /documents

The datasets route is meant for extending and querying information about the data-set which is used to train classifiers.

**/** Gets an random document from the downloaded CommonCrawl pages.

**Request:**

| Method | GET |
|---|---|

**Response:**

| Property | Description |
|---|---|
| status | Status code for the response |
| document | String containing the contents of the randomly selected file |

### 5.5.5 Indices Route: /indices

The datasets route is meant for extending and querying information about the data-set which is used to train classifiers.

**/**

**/download** Starts the download of all indices for a given url.

**Request:**

| Method | GET |
|---|---|

**Response:**

| Property | Description |
|---|---|
| indices | String containing a list of indices |
| status | Status code for the response |
| error | Boolean indicating if there was an error during the processing of the request |

### 5.5.6 Classify Documents Route: /classify_documents

Run workers to classify all documents and log only. All the indices from the specified directory will be parsed using the number of workers specified.

**/log_only** Predicts the category of the document that is submitted in the body of the request.

**Request:**

| Method | GET |
|---|---|
| ?pworkers | Number of producing workers, parsing indices and adds to queue |
| ?cworkers | Number of consuming workers, classifying indices from the queue |
| ?directory | Path to directory containing indices |

**Response:**

| Property | Description |
|---|---|
| status | Status code for the response |
| error | Boolean indicating if there was an error during the processing of the request |
| message | Message containing extra information about the response |

**/to_database** Run workers to classify all documents and output to database. Database must be online, all the indices from the specified directory will be parsed using the number of workers specified.

**Request:**

| | |
|---|---|
| Method | GET |
| ?pworkers | Number of producing workers, parsing indices and adds to queue |
| ?cworkers | Number of consuming workers, classifying indices from the queue |
| ?directory | Path to directory containing indices |

**Response:**

| **Property** | **Description** |
|---|---|
| status | Status code for the response |
| error | Boolean indicating if there was an error during the processing of the request |
| message | Message containing extra information about the response |

## 5.6 Front-End

An important part of the UrbanSearch system is the part where the extracted and processed data are visualised and made accessible for the end user. Our goals were to provide the end users with a clear and easy to use interface. Extracted relations should therefore be visualised in such a way that the user can make sense of the information easily. Another desire that was expressed by our client, was the possibility to manipulate the displayed information, in a fast, easy and intuitive way. How we tried to reach these goals is described below.

### 5.6.1 Technical Overview

In this section we will discuss some of the main technical aspects of the UrbanSearch project. We will give an overview of and a motivation for our most important design choices.

#### Modular Design

Dealing with huge amounts of data and displaying this data in a way that makes is easy to understand for users is a challenging task.

#### NodeJS

#### ExpressJS

### 5.6.2 Interfaces

#### Interactive Map

#### Classification Interface

#### Settings Interface

### 5.6.3 Recommendations

# 6

# Project Evaluation

intro

Put results in appendix F

write conclusion about results

## 6.1 Fulfilment of Requirements

In section 3.3.3 we declared the requirements for our program. Table 6.1 shows which of these requirements passed or failed.
The program works as intended so all must haves passed.
There are however two should haves which failed. Finding correct relations proved more time-consuming than expected therefor our algorithm only discerns the top level relations (e.g. trade) and not sub levels (e.g. food trade). Furthermore there are a lot of places with duplicate names, yet no complete lists of these duplicates are available. Therefor it is less easy implementable than first thought.
Since other, more important, tasks took longer to implement than intended we did not make implement functionality to use Delpher to characterise relationships. We did add functionality to visualise the data by using a map.
As expected the would likes did not pass. It is theoretically possible to show all connections of all places on the map at the same time. However, it would result in a completely filled in map because there is a line for each relation so one would not be able to get any useful information from this.

more about not mentioned pass/fail, name reqs?

Kan die tabel niet beter Requirement:Pass/Fail:Uitleg, en dan voor Must, should etc?

Table 6.1: Requirements pass fail

| Must Haves | Pass / Fail | Should haves | Pass / Fail | Could Haves | Pass / Fail | Would Haves | Pass / Fail |
|---|---|---|---|---|---|---|---|
| 1 | Pass | 1 | Fail | 1 | Fail | 1 | Pass? |
| 2 | Pass | 2 | Pass | 2 | Pass | 2 | Fail |
| 3 | Pass | 3 | Pass | | | 3 | |
| 4 | Pass | 4 | Fail | | | | |
| 5 | Pass | | | | | | |

## 6.2 Process

### 6.2.1 Collaboration Between the Team Members

The collaboration 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 divided even over the team members.

### 6.2.2 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.

Claudia absent twice

### 6.2.3 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 product as good as possible to the clients wishes.

# 7 Discussion

This section is divided into 4 parts. First we will answer the research question and comment on the sub-problems defined in the problem definition. Afterwards we will discuss the influence of these answers. Next we will mention issues we faced and which still remain. The last part of this section is dedicated to the ethical questions this project may involve.

## 7.1 Answering the research question

As mentioned in section 3, the problem was the following:

*How can open data be leveraged such that a metric for the strength of relationships between cities can be defined and visualised?*

And the sub-problems were:

- How can we filter the available text data to find co-occurrences of cities and discarding text data that does not contain co-occurrences?

  This should reduce the amount of data and thereby potentially speed up the rest of process.

- The sub-problem that arises after filtering is how to determine what relationships can be extracted from the text-data, this will be referred to as the classification of the text-data.

  This requires a method that reliably and efficiently processes the text-data and can be tuned to the clients wishes, meaning that the classification should output what the client desires.

- The next question is how to store the data and determine the strength of the relationships.

- The last question is how to combine the stored data and present it to a user, this means visualising and/or exporting the data in an accessible way.

## 7.2 influence of the answers

## 7.3 issues faced and remaining

## 7.4 Ethics

When using our program, there are two ethical issues that may arrive. The first is due to the fact that pages may be downloaded and stored, which may result in privacy or copyright issues. The second is about what the results of our program may be used for, and how the world will react to this.

#### Storage

One ethical issue is due to the storing of data. Since we store random web pages we do not know whether or not these pages may contain private or copyrighted data. For instance news articles could be downloaded and stored whilst this could violate the copyright issues. For most free news sites this is not a problem but this especially becomes a problem when using Delpher as a data source. Therefor if this source is added it must also be ensured that the data is stored in a safe way.

#### Influence

Another issue that may occur is the influence this kind of research may have. If there is indeed a correlation between the results of our program and the economic growth of cities this may influence the behaviour of investors, companies and cities. Investors may look at the data and decide to invest in companies from more growing cities. Companies may use this data to decide where to build their new offices. And cities may change their policies based on the results. In future executions of our system it may also occur that data is being manipulated. Involved parties which put extra data online containing the names of cities they want to have a better result for. Whilst these issues may occur, we do not suspect our system to have a large enough impact to cause this. It may rather be a step towards these effects. Over time the effects will become clearer and they should be taken into account when continuing research in this field.

# 8

# Recommendations

For future improvements on this project one might decide upon two ways to do this. The first is to look at the requirements which have not yet been fulfilled and fulfil those. Depending on the needs of the user one might want to implement different things. For extending the application to international places one would want to use data from top-level domains other than .nl. However others might be interested in finding different relations, for which one would need to have other training data. If one has more time it might be interesting to instead of using a classification algorithm, to choose a clustering algorithm.

extend

Make recommendations for future version, for extending the back-end and front-end Try to mention the requirements here

# 9

# Conclusion

Start with ”In the past few months we blah” and shortly mention the chapters Mention the project goal and how well it is met, without duplicating the evaluation chapter

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.

new stuff

# Bibliography


[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975. ISSN 0001-0782. doi: 10.1145/360825.360855. URL http://doi.acm.org/10.1145/360825.360855.

[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. ISSN 1077-2626. doi: 10.1109/2945.468391.

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

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

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

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

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

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

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

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

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

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

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

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

[15] Fermentas Inc. Beautifulsoup: Beautiful soup documentation. URL https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

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

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

[18] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *2013 International Conference on Social Computing*, pages 708–715, Sept 2013. doi: 10.1109/SocialCom.2013.106.

[19] Brian W Kernighan and Rob Pike. *The Unix programming environment*, volume 270. Prentice-Hall Englewood Cliffs, NJ, 1984.

[20] Bart Lambregts. Geographies of knowledge formation in mega-city regions: some evidence from the dutch randstad. *Regional Studies*, 42(8):1173–1186, 2008.

[21] Microsoft. How to evaluate model performance in azure machine learning. https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-evaluate-model-performance, 2017. Accessed: 2017-06-21.

[22] 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.

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

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

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

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

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

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

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

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

[31] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, March 2002. ISSN 0360-0300. doi: 10.1145/505282.505283. URL http://doi.acm.org/10.1145/505282.505283.

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

[33] 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.

[34] 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.

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

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

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

[38] Simon Tong and Daphne Koller. Support vector machine active learning with applications to text classification. *Journal of machine learning research*, 2(Nov):45–66, 2001.

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

[40] Frank van Oort, Martijn Burger, and Otto Raspe. On the economic foundation of the urban network paradigm: Spatial integration, functional integration and economic complementarities within the dutch randstad. *Urban Studies*, 47(4):725–748, 2010.

[41] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0064-3. doi: 10.1145/1900008.1900067. URL `http://doi.acm.org/10.1145/1900008.1900067`.

[42] 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.

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

# A

# Better Code Hub Guidelines

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

1. **Write short units of code**
   Units of code should be no longer than 15 lines.
2. **Write simple units of code**
   Separate units of code should contain no more than 4 branch points (if, for, while, etc)
3. **Write code once**
   Shared code should be extracted, either to a new unit or to a super class
4. **Keep unit interfaces small**
   The number of parameters per unit of code should be no more than four.
5. **Separate concerns in modules**
   Identify and extract responsibilities of large modules to separate modules and hide implementation details behind interfaces.
6. **Couple architecture components loosely**
   minimizing the amount of interface code (e.g. by using 'abstract factory' design pattern)
7. **Keep architecture components balanced**
   Organize code in such a way that the number of components is between 2 and 12, and ensure the components are of approximately equal size (keep component size uniformity less than 0.71).
8. **Keep your codebase small**
   Refactor existing code to achieve the same functionality using less volume, and prefer libraries and frameworks over "homegrown" implementations of standard functionality.
9. **Automate tests**
   Add tests for existing code every time you change it.
10. **Write clean code**
    Remove useless comments, commented code blocks, and dead code. Refactor poorly handled exceptions, magic constants, and poorly named units or variables.

# B

# Sig Feedback

## B.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.

## B.2 week 9

# C

# User Manual

# D

# Developers Manual

# E

# Used Libararies

# F

# Validation and Verification results

**??**

To ensure code quality in our project we used several methods. The results from SIG [35], a tool to ensure code quality and maintainability, are discussed and the testing is discussed.

## F.1 Testing the Application

### F.1.1 Unit Tests

### F.1.2 Integration Tests

### F.1.3 System Tests

### F.1.4 Acceptance Tests

## F.2 SIG

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[14] 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. Since the final report is due to the same date as the second submission for SIG review, the second review will not be included in this report. Instead we will show the final results from BetterCodeHub for week 9. Exact feedback can be found in appendix B.

### F.2.1 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.

fixed this?

**week 9**

## F.3 evaluating the classification

### F.3.1 Accuracy

### F.3.2 Confusion Matrix

### F.3.3 Precision, Recall, F1 and UAC

## F.4 Evaluation of relation scores