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

July 3, 2017

TU Delft

Preface and Acknowledgements

As part of the TU Delft Computer Science Bachelor Project, we present our final report on the development of the UrbanSearch project. The client during this project was the OTB group, which is part of the Faculty of Architecture and the Built Environment. The OTB group ”seeks to make a visible contribution to society by addressing societal challenges in the field of the built environment”. This report describes the research and development of the project over the course of ten weeks.

Acknowledgements

First of all, we would like to thank Evert Meijers and Antoine Peris for the opportunity to work on this project and for their continuous input and guidance. We also express our thanks to Claudia Hauff for her continuous support and useful reviews of reports during the project.

Summary

The intention of the project is to design and implement a system that is able to quantify and visualise intercity relations, using open data.

According to their web site[1], "OTB seeks to make a visible contribution to society by addressing societal challenges in the field of the built environment". However, with the growing amount of information available online, they have the need for IT solutions to support their research. This project was intended to design and implement a system that is able to quantify and visualise intercity relations, using the information from open data. Since there is a lot of data freely available, the challenge was to accurately delimit the boundaries of the project and then convert the information from within these boundaries to something the researchers could work with. Moreover, we had to focus on both a full-fledged back-end and front-end.

The back-end consists of four main parts: (1) data gathering and processing, (2) data filtering, (3) document classifying and (4) data storage. Data is gathered from the freely available corpus Common Crawl. The cleaned textual data is then passed to the filtering component, which checks the text for co-occurring cities. If either too few or too many co-occurrences are found, the document is filtered out. Next, the document is passed to the classifier, which determines to which of a predefined set of categories the document belongs to. Finally, the document is stored in a graph database, along with the resulting category and probabilities retrieved from the classifier. After this, relations are drawn between cities and documents they occur in. Finally, the relation between two cities are calculated from documents in which they both occur, grouped by category.

The front-end is a Web application to allow the client to interactively browse the results and to be able to see patterns in visualised data, which are otherwise hard to extract from raw numbers. The application pulls relations from the database and visualises them on a geographical map, to place cities on their natural locations. Lines are drawn between cities to represent relations, and every item is clickable in order to see more information. In addition, there are sliders that allow the client to control what the application shows. Besides the visualisation of data, we created an interface in which one can manually classify documents, in order to further train the classifier. The application focuses on efficiency, as it has to deal with large amounts of data.

The client was happy to see that we were able to deliver a viable product. However, some work has to be done before the product can fully satisfy the client's needs. For this, recommendations have been made, in order to provide future developers with pointers to continue on the application.

[1] `https://otb.tudelft.nl`

Contents

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 built environment, cities do not function 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. An enormous amount of raw, unordered data is available to extract the relations from, there is, however, no good way yet to process the data. According to Short et al. (2017) comparative statistics are not easily available and common assertions are repeated [35]. Similar assertions are made by DeRudder et al. [9]. Using web data as a proxy for determining intercity relations is still in its infancy.

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 [30] and sales forecasting [46], where socio-economic phenomenon are derived using web data, the client's wish is to develop an application that captures urban-urban interactions. An example of how one could try to achieve this using the Google search Engine[3] is `"Rotterdam Amsterdam" OR "Amsterdam Rotterdam"`, which searches for the co-occurrence of Amsterdam and Rotterdam. However, manually processing all results a search engine yields is not feasible, because one would have to read each page to determine which types of relationships the page contains. An application should process all the pages that contain co-occurrences of cities to determine what type of relations, for example transportation or leisure, between cities can be extracted from the document. Thus, we will answer the following question: how can a framework be designed for leveraging open data such that a metric for the strength of relationships between cities can be quantified and visualised?

First, we discuss related work in chapter 2. Second, we explore the problem domain, identify the requirements for a solution to the problem and discuss issues that might arise in chapter 3. Third, we develop a methodology for a framework that satisfies the requirements and tackles the issues in chapter 4. In chapter 5, we explain how the framework was realised and discuss the most important issues we were able to solve. Then, in chapter 6, we evaluate the product, showing how well it meets its requirements and review the development process. Next, we discuss open issues as well as some possible ethical issues that may come with the system. In chapter 8 we provide some recommendations for the system for future versions. We conclude in section 9.

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

2

Related Work

Since the 1960's the economy modernised rapidly. Concentration of jobs increased and the amount of cooperation between remote firms grew. The desire to understand this led to an increase of work on intercity relationships[43].

One of the most common methods used is the interlocking network model (INM)[39]. 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 the INM 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 because there is no precise measurement [23].

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 [10]. 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 [6, 12]. 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[10, 18, 19]. 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 [5]. However, this method is very limited since you would have to manually enter a new query for each pair of cities and for each relation.

Improving the textual analysis on websites to find digital links between cities requires a more systematic approach. 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. After that, the requirements following from this analysis and the wishes of the client are presented. Last, a few design decisions are presented that followed from the problem analysis and requirements.

3.1 Problem Definition

As discussed in the previous sections the hypothesis the client proposed is if a semantic association of cities can give insight in 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. These are Filtering, Classification, Storing Data and Data Visualisation & Export.

Filtering

Filtering for this problem 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, which 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.

Storing Data

Next, the need arises to store the data and determine the strength of the relationships. This means choosing an organised data storage solution to store the results, for example a relational database.

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[7] 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 will 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 prioritisation technique. Four levels of priority are defined: must have, should have, could have and would like. We also differentiate between functional and non-functional requirements.

Must Have

Requirements labelled as must have are key to the minimal viable product. 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 and/or 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 one 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 and/or 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 6.1.4.

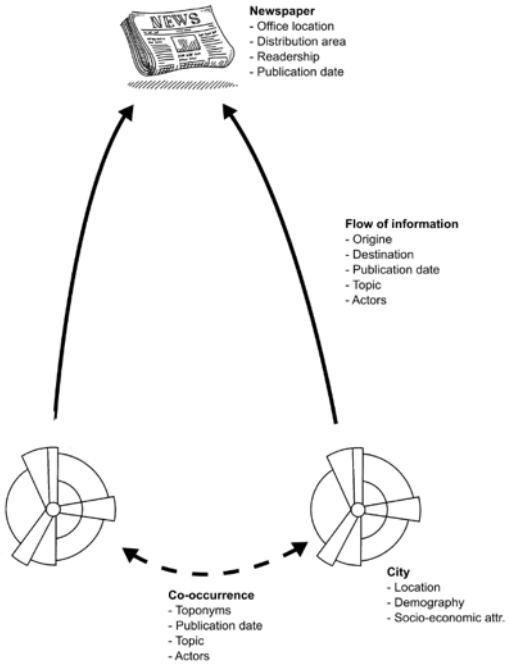


Figure 3.1: Solid lines represent simple information flows, whereas the dashed line is a complex connection of information. We focus on the part depicted by the dashed line.

4

Framework and Tools

In this section, we gradually design 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 explain what kind of classification will be used. Fifth, we agree on which data storage and ingestion to use. Sixth, we decide how users will interact with the obtained results. Last, we select the visualisation tools to be used.

4.1 High-level Overview

The figure 4.1 represents a high-level overview of the system. The most important inputs, outputs and steps in system the are displayed. A more in-depth explanation of the different stages of the process can be found in the following sections.

4.2 Gathering the Data

As explained in section 3.3.2, data sources should be plugable. An initial corpus of documents is needed for the project, which we will choose 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 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 could occur by randomly searching the web, an obvious choice is to use Common Crawl as a data source.

4.2.1 Common Crawl

Common Crawl [8] 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 [26, 36, 37]. 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.

[1] `https://gist.github.com/Smerity/e750f0ef0ab9aa366558`

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]. A single index is a combination of multiple key-value pairs, for an example of a single index see listing 4.1. An index contains important information such as for example the name of a WARC file, and the index and offset to find the correct data within that WARC file. 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.

```
{"urlkey": "nl,tudelft)/", "timestamp": "20170323161043", "status": "200", "url": "http://
    www.tudelft.nl/", "filename": "crawl-data/CC-MAIN-2017-13/segments/1490218187144.60/
    warc/CC-MAIN-20170322212947-00594-ip-10-233-31-227.ec2.internal.warc.gz", "length": "
    6837", "mime": "text/html", "offset": "727926652", "digest": "
    WPTH3FM5VR7UGLA5PZS5L5YI22TNIKXG"}
```

Listing 4.1: Common Crawl index example

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.

4.2.2 Other Data Sources

Besides Common Crawl, there are a number 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 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.

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

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. [32], 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` [21]. 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.

multi-pattern matching	0.049831339
plain string matching	1.870154497

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

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 in C with Python bindings.

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. 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.

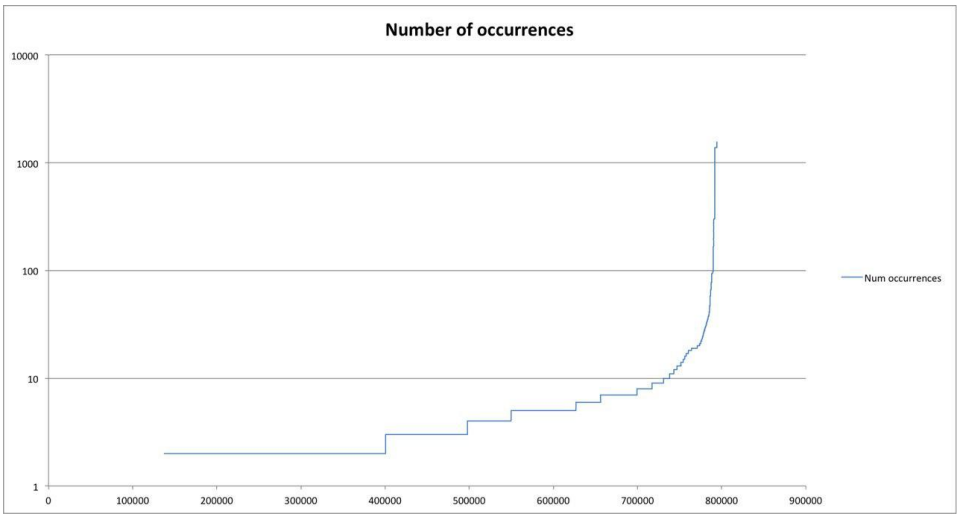


Figure 4.2: Number of documents plotted against the number of unique occurrences contained in these documents.

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 we would only 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

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

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 one 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 into two distinct groups: supervised and unsupervised algorithms. Supervised algorithms expect an input set and a corresponding expected 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. 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 led to the need for categories to be identified within the collected documents. Together with our client, we identified the following categories which are useful to search for in 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.

[9] On a virtual server with 8GB RAM, 4 CPUs and 100GB of HDD storage.

4.4.2 Pre-processing

For pre-processing the documents, there are a number of tools available. We used NLTK [11] for removing stop words and we use regular expressions for removing unwanted characters. The HTML parsing is done using BeautifulSoup[17].

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

Unwanted characters To strip unwanted characters we have defined a regular expression that identifies unwanted characters that are not removed using the list of stop words and symbols. Matching characters are removed from the document.

HTML Since we are dealing with HTML pages that are parsed 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 collected document, labelled data has to be collected that can be used as training data for the classifier. To collect this data we considered several options.

The first option is to query for documents from news(paper) sites. Since the documents are categorised by professionals, we assume these document are 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 than useful information.

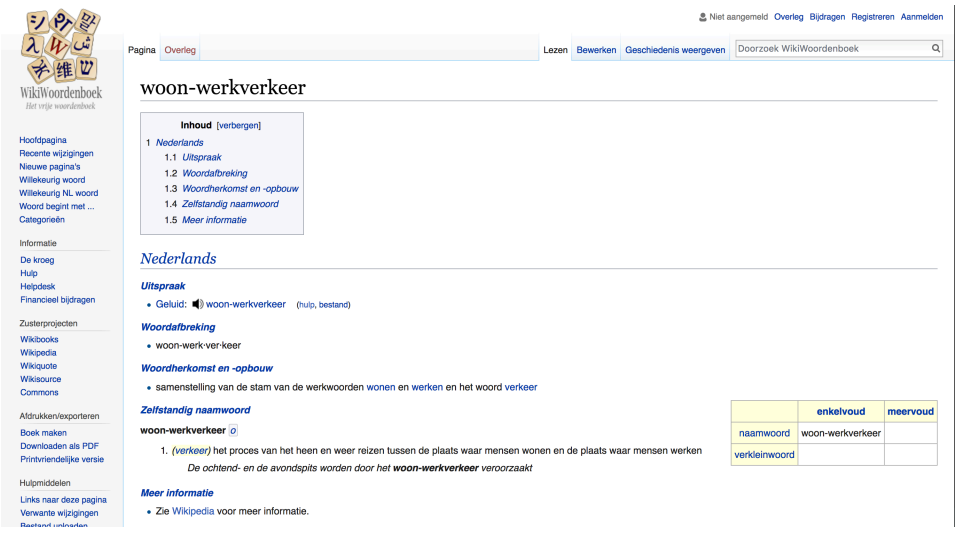


Figure 4.3: Example of an undesired result obtained with Google Custom Search

We found the open data repository of the "Centraal Bureau voor de Statistiek" (CBS) to be a source of quality documents that could be used as training data for the classifier. The articles in the repository can be queried using taxonomy tags with which the articles are tagged. Since

these articles are tagged by professionals we may assume that these tags are valid. The only downside of the CBS data is that it is a limited source of documents, which means we will have to find additional sources for training data in the future.

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[25]. 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 along a straight line in two. 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 (e.g. 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.

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[31]. 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 [34]. To reduce the number of features we considered different techniques from Sebastiani (2002)[34]. 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 Yang (1997) [47] 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.

Classification Even after applying dimensionality reduction we are left with a lot of features (6500+). Thus, we need an algorithm that works well with a feature rich problem. From the Microsoft machine learning cheatsheet[25] we know Support Vector Machines(SVM) is an algorithm that works well with feature rich problems. Also Sebastiani (2002)[34] 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 an SVM is that of a hyper-plane that divides two distinct sets, while trying to maximise the margin between these sets [42].

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 (e.g. dimensionality reduction transforms) is a 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 this optimising functionality and these different pipelines will be easy to do.

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 Data

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 a 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 documents 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: graph databases and 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 [45]. 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 three[10]. Therefore, we are confident that a graph database is the best choice.

[10] https://neo4j.com/news/how-much-faster-is-a-graph-database-really/

4.5.3 Comparing Graph Databases

Next, which graph database we are going to use 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 five million triples
[b] With separate tool called Gruff: `https://allegrograph.com/gruff2/`
[c] Non-commercial use
[d] Using a separate tool called 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 it is by far the most popular graph database[11]. Secondly, since Neo4j is the most popular graph database, the support community and amount of available examples is large. Lastly because we have experience with Neo4j, which means less time will be spent on getting to know the graph database and functionality.

4.5.4 Using Neo4j for Storage and Ingestion

Neo4j is a highly scalable graph database that leverages data relationships as first-class entities [28]. 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 [13]. 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 [29]. 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 (2013) 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 gains [15]. Jouili et al. concluded that Neo4j has a read-only performance which is comparable to other graph databases

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

[20]. 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.

Querying a Neo4j graph is done using the query language Cypher. 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 4.4. It consists of nodes, relationships (edges), properties (within the nodes and relations) and labels (coloured blocks above the nodes).

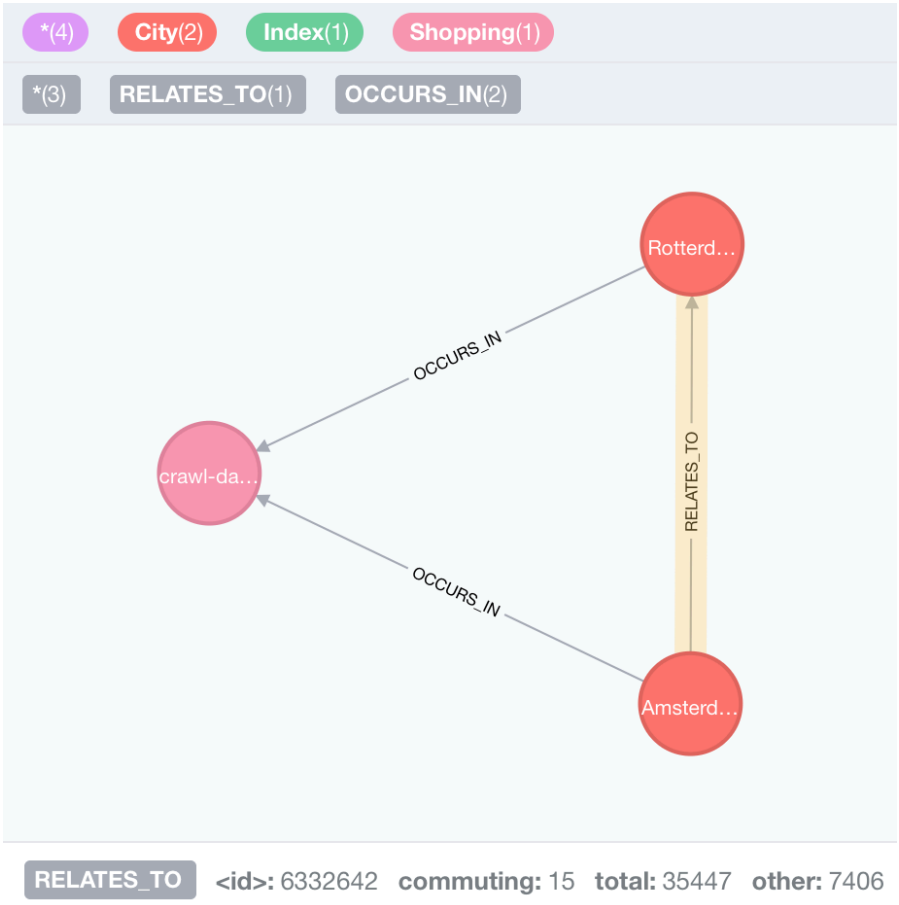


Figure 4.4: The Neo4j model, as it appears in the default user interface

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.
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. We considered the following three options because they match best with the client's experience and preferences:

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

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 4.6.

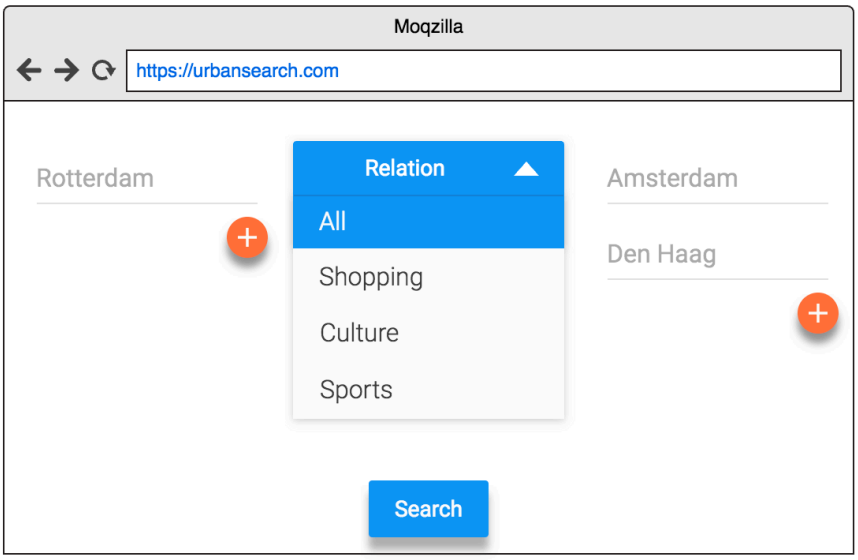


Figure 4.6: Example interface for the query composer

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 4.7 and 4.8.

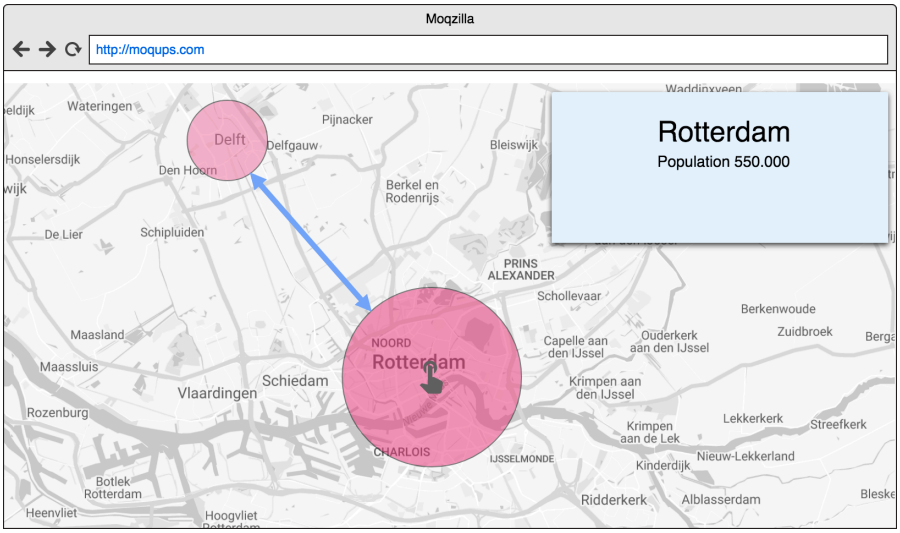


Figure 4.7: Example 1 of an interactive map

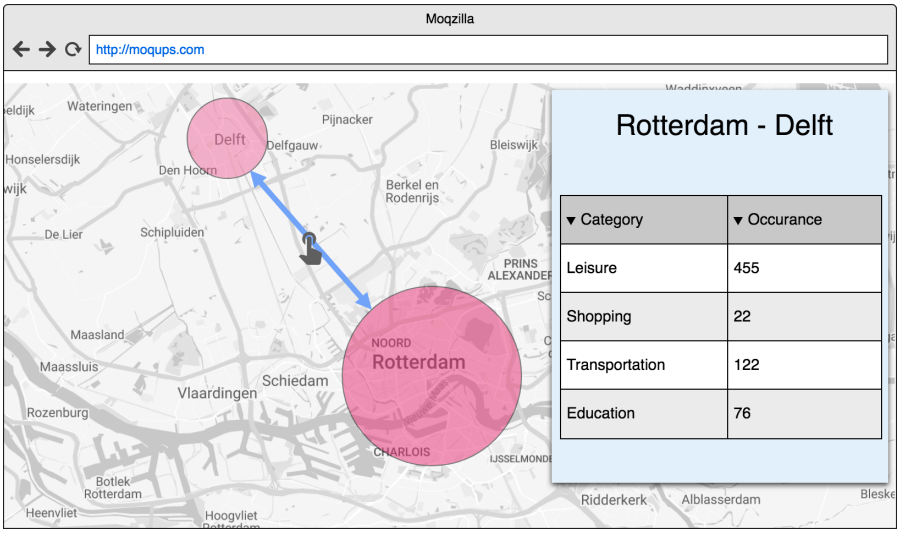


Figure 4.8: Example 2 of an interactive map

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

Together 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 client's needs and desires. We have discussed the preferences of the client and have drawn up a 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, together 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 experience of the group members. Another reason to go 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.

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. 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.

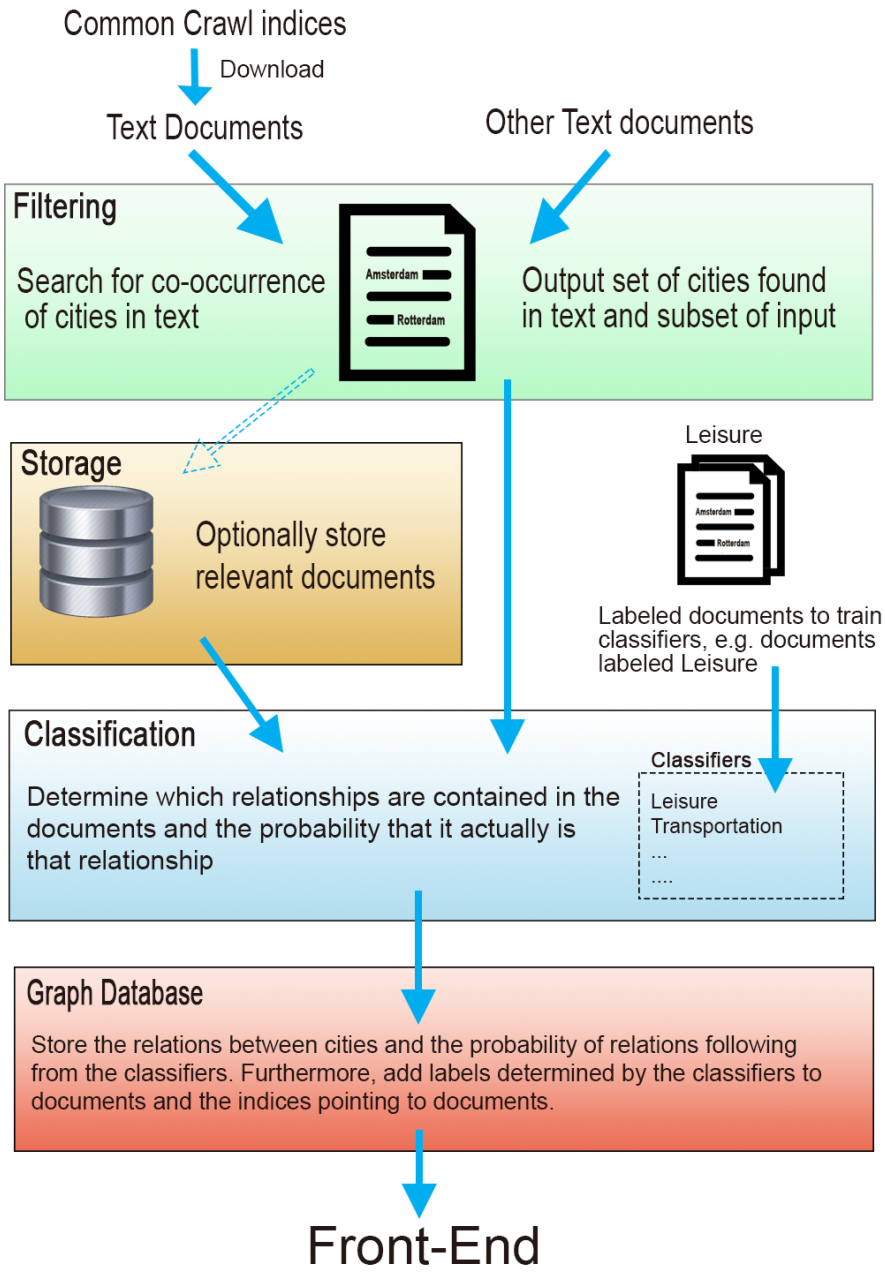


Figure 4.1: High-level overview of the system

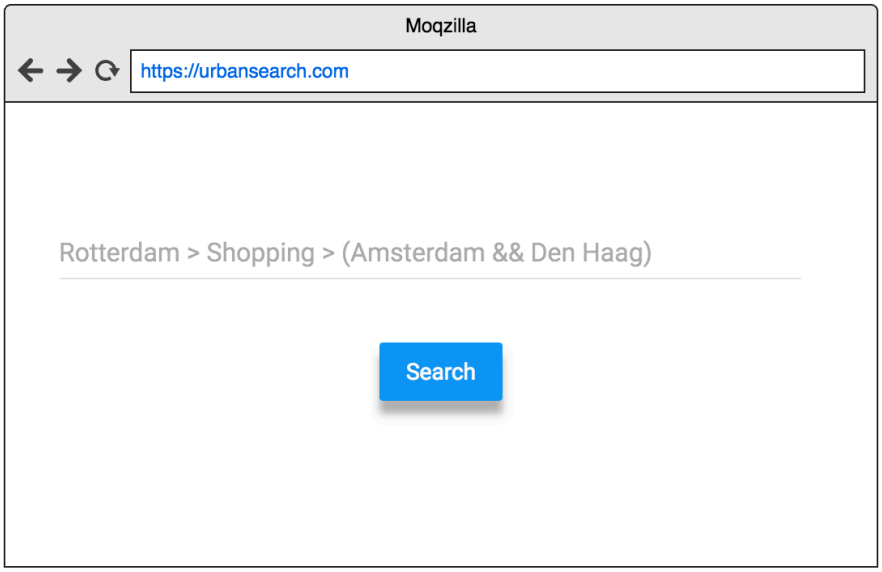


Figure 4.5: Example interface for the query language

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 correspond to these indices.

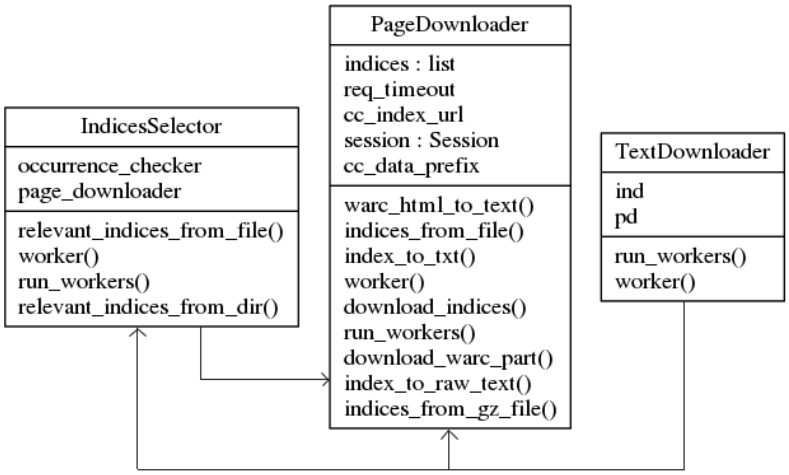


Figure 5.1: UML Diagram of downloading and parsing classes

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

The first step in parsing the Common Crawl indices is to filter out the indices that have a HTTP Status Code [2] other than 200, as only indices with this HTTP Status Code are 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`

```
5         return True if int(index['status']) == 200 else False
6     return False
7
8 def _clean_indices(self, indices):
9     # Removes useless entries with status code other than 200
10    for index in indices:
11        if not self._useful_responsecode(index):
12            indices.remove(index)
```

Listing 5.1: Initial implementation

At first, a simple implementation was used as can be seen in listing 5.1. However, removing an element from a Python list has a time complexity of $O(n)$. Since the implementation of `clean_indices()` loops over all indices and removes it if it has status code other than 200, this function has a complexity of $O(n^2)$. To improve on 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 adjustment resulted in a speedup of about 5.6 times compared to the $O(n^2)$ method.

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

Listing 5.2: Regex solution

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

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

Listing 5.4: Reducing memory footprint

5.2 Filtering the Data

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 which the Aho-Corasick algorithm will try to match. By default the application will retrieve a list of cities from the database and use this list to find matches in the text.

The implementation seemed straightforward at first, however, it became apparent that the algorithm matched sub-strings. For example, if a text contains "Leidende Amsterdammers" the Aho-Corasick algorithm would match because of the sub-strings "Amsterdam" and "Leiden". However, this should not be a match because "Leidende" is a verb. To solve this we added an additional check in the co-occurrence filtering which can be seen in listing 5.5. However, this might result in discarding documents that do contain interesting relations. This was the best way to get rid of a lot of false positives that we found within the short time span of the project.

```
1 for end, name in names:
2     # Skip words that contain city names (e.g. Amsterdammers)
3     if page[prev_end + 1] in 'abcdefghijklmnopqrstuvwxyz':
4         prev_end, prev_name = end, name
```

Listing 5.5: Additional check to prevent substring matches

Another problem that surfaced during the implementation of the co-occurrence check was the fact that Aho-Corasick is designed to be multi-matching. This means that it matches every occurrence of a city in the text, where we were interested in which cities were found in the text but not if it was found multiple times. Therefore, the result of the Aho-Corasick algorithm was put into a set to solve this problem.

5.3 Classification

This section describes how we developed our interface for classifying documents. First, we will first give a short overview of the classification part of the application. Next, we discuss how classifiers are defined. Then we will discuss how classifiers are created or loaded in the application. Finally we will discuss the interface that accepts documents and returns a prediction of the category or categories associated with the supplied document.

5.3.1 Overview

In figure 5.2 we give a concise overview of the classification subsystem. The UrbanSearch API is used to classify documents by loading an instance of the `ClassifyText` class on startup. Furthermore the API offers the possibility to create new (default) classifiers or to modify existing classifiers.
The `ClassifyText` object uses an implementation of the `ModelManager` class to predict categories of documents.

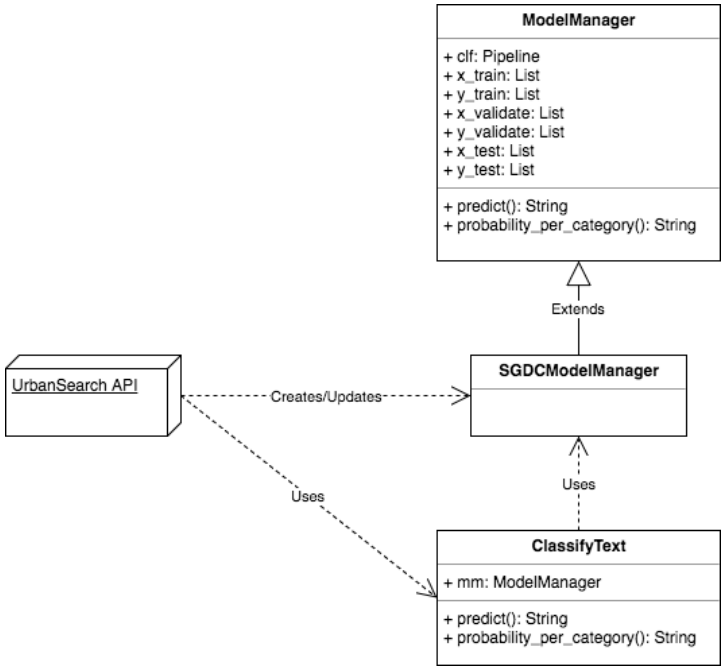


Figure 5.2: UML Diagram depicting the interaction between the API and the classification subsystems

5.3.2 Scikit Pipelines

As explained in section 4.4 we decided to use the Scikit-Learn library for all our classification functionalities. A key concept of Scikit is the so called `Pipeline`. A `Pipeline` in Scikit is an assembly

of intermediate transform steps, combined with a final estimator [3]. The intermediate transforms the input data to get the best perfomance out of the final estimator. In listing 5.6 we show an example of a `Pipeline` that we use in the system.

```
sgdc = Pipeline([
            ('tfidf', TfidfVectorizer(stop_words=sw.words('dutch'))),
            ('select', SelectPercentile(f_classif)),
            ('clf', SGDClassifier(alpha=0.0001,
                                  average=False,
                                  class_weight=None,
                                  epsilon=0.1,
                                  eta0=0.0,
                                  fit_intercept=True,
                                  l1_ratio=0.15,
                                  learning_rate='optimal',
                                  loss='log',
                                  n_iter=5,
                                  n_jobs=1,
                                  penalty='l2',
                                  power_t=0.5,
                                  random_state=None,
                                  shuffle=True,
                                  verbose=0,
                                  warm_start=False))
        ])
```

Listing 5.6: SGDC Pipeline

This `Pipeline` consists of three parts. The "tfidf"-part transforms a text into a matrix of words with corresponding TF-IDF scores (which are calculated first using the training set). The "select"-part selects the top ten percent of features which were returned by the previous transform, in our case the `TfidfVectorizer` transform. For this particular `Pipeline` this means that 10 percent of the features with the highest TF-IDF score are returned. Finally, the "clf"-part is the final estimator. For this `Pipeline` it is an SVM (see 4.4) that uses stochastic gradient descent (SGD) training [4]. Gradient descent tries to find the minimum of a cost function by traversing the function in the opposite way of the descent and does this step by step. SGD takes random (stochastic) steps which works more efficient when dealing with large data sets. The fully defined pipeline can now be used to train the classifier. This is done by inputting a set of data with corresponding expected outputs.

5.3.3 ModelManagers

To provide an easy way to work with Scikit `Pipeline`s, we implemented a utility class called `ModelManager`. The `ModelManager` is a super class that should be used to implement algorithm specific `Pipeline`s, while providing easy to use interfaces for loading, saving, training and predicting. Listing 5.7 shows a snippet of the `ModelManager` base class.

```
class ModelManager(object):
    """
    ModelManager base class.
    Should only be used to load saved models from disk.
    If a file name is passed this file will be used to load a pickled
    classifier from that location on disk.
    """

    def __init__(self, filename=None):
        super(ModelManager, self).__init__()
        self.x_train = []
        self.y_train = []
        self.x_validate = []
        self.y_validate = []
        self.x_test = []
        self.y_test = []
```

[3] http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

```
17
18          self.clf = self.load(filename) if filename else None
19
20      def load(self, filename):
21          """
22          Load the classifier from the supplied file
23
24          :param filename: the file containing the pickled classifier instance
25          :return: a Scikit classifier object
26          """
27          with open(os.path.join(MODELS_DIRECTORY, filename), 'rb') as f:
28              return pickle.load(f)
```

Listing 5.7: ModelManager base class

The base class can be used to implement specific `ModelManagers` which define a `Pipeline` with a final estimator of choice, like the `MultinomialNB` (Multinomial Naive Bayes) estimator used in listing 5.8.

```
1  class MNBModelManager(ModelManager):
2      """
3      An implementation of the ModelManager base class which uses a Multinomial
4      Naive Bayes classifier as its default classifier.
5      """
6
7      def __init__(self, filename=None):
8          super().__init__(filename)
9
10         if not filename:
11             self.clf = Pipeline([
12                 ('tfidf', TfidfVectorizer(stop_words=sw.words('dutch'))),
13                 ('anova', SelectPercentile(f_classif)),
14                 ('clf', MultinomialNB())
15             ])
```

Listing 5.8: ModelManager using the Multinomial Naive Bayes estimator

The `MNBModelManager` inherits all the load, save, predict and train functionality of the base class. The base class can be used to load saved classifiers from disk. This is done by providing the `ModelManager` class with a file name on initialisation. If the file is found, the classifier is loaded from disk and ready to be used.

5.3.4 ClassifyText Interface

Most of the time we do not want to be busy creating and training classifiers; we want to classify documents. To provide an interface that can be used easily to input documents and get back predictions of which category a document belongs to, we implemented the `ClassifyText` class. The class loads a default classifier that should be available in the provided directory. After the class is initialised, the `predict` and `probability_per_category` methods can be used to predict categories for given documents.

The predict method, which is shown in listing 5.9, takes a document as input and returns a prediction of the category which the inputted document best matches.

```
1  def predict(self, text, pre_processor=None):
2      """
3      Predict the class of the supplied text
4
5      :param :text the text that needs to be classified
6      :return: a prediction of the category for the passed text
7      """
8      if pre_processor:
9          text = pre_processor(text)
10
```

```
11      return self.mm.predict([text])
```

Listing 5.9: Predict method of the ClassifyText class

```
1  def probability_per_category(self, text, pre_processor=None):
2      """
3      Predict the class of the supplied text
4
5      :param :text the text that needs to be classified
6      :return: a prediction of the category for the passed text
7      """
8      if pre_processor:
9          text = pre_processor(text)
10
11     return dict(zip(self.mm.clf.classes_,
12                     self.mm.probabilities([text])[0]))
```

Listing 5.10: probability_per_category method of the ClassifyText class

5.4 Storing the Data

In this section, we discuss how the filtered documents are stored and how Neo4j was used for storing extracted relations, following the design described in section 4.5. We discuss the storage and graph database parts from the overview (see figure 4.1).

5.4.1 Storing Filtered Documents

The documents that pass the filtering stage can be stored for several reasons. For example, the classifier can be retrained and might thus label documents differently. To avoid having to download and process all the pages again, it is useful to store the documents on disk. If the disk is small, it is wise to compress the documents. However, compression is a slow process, so if enough disk space is available, storing the documents uncompressed is more feasible.

In the `TextDownloader` class, that was already shortly discussed in section 5.1, storage to disk is done without compression in all cases. We did this since this project only involves a relatively small data set, e.g. one that can be stored without the need for compression.

5.4.2 Storing Extracted Data

To be able to interact with the results of the application, it is required to store extracted relations. The implementation of the storage follows the design of section section 4.5.1, using the graph database Neo4j.

Neo4j Model

The model used for the graph structure follows the concepts described in section 4.5.4. It consists of nodes, labels and properties. To distinguish between and to efficiently query for specific (types of) nodes, at least one label is assigned to the node. Below, the labels we used are listed and per label, a description of the nodes they are attached to is given:

:City Nodes labelled as `:City` represent the cities the application uses. These nodes contain multiple properties: name, population, longitude and latitude. Population is used in the visualisation (see section 5.6) for scaling. Longitude and latitude are used to place the cities on a map and were retrieved through the Google Maps Geocoding API[4]. However, this did not work out quite well for duplicate city names. Google picks the coordinates for the city it considers the most important. This was fixed manually. Indeed, this is not a feasible solution, should there be many duplicate city names.

[4] `https://developers.google.com/maps/documentation/geocoding/start?hl=en_US`

:Index Nodes with the `:Index` label represent documents that are found useful. Every document has this label, in addition to the label representing the category they are classified as. The following basic properties belong to these nodes: file name, offset and length. They point to the exact file location where the page can be downloaded from CommonCrawl. Additionally, the nodes contain a probability property per category. These probabilities come from the classifier and are there for validation purposes.

Categories For each category, a label exists to separate a category from the bulk of documents. This way, documents of a specific category can be matched against. This is particularly useful to count, for example, the documents about "Leisure" (and thus labelled `:Leisure`), that two cities have in common. Category labels are only applied to nodes that also have the `:Index` label and thus share the same properties.

The nodes are connected using relations. Cities (`:City` labelled nodes) occurring in documents (`:Index` labelled nodes) are connected with a `:OCCURS_IN` relation. It is mainly used to find documents in which a pair of cities occurs. For example, the query below matches the documents Rotterdam and Amsterdam have in common and returns the file names:

```
1 MATCH (:City { name: 'Amsterdam' })-[:OCCURS_IN]->
2     (i:Index)
3         <-[:OCCURS_IN]-(b:City { name: 'Rotterdam'})
4 RETURN i.filename
```

Listing 5.11: Querying documents containing two cities

Intercity relations are the relations between distinct `:City` labelled nodes and are called `:RELATES_TO`. They represent what the client is actually interested in and have a property for every category, containing the score. Additionally, the sum of the individual category scores is kept in a "total" property. The relations are used for exporting, visualisation and interaction. When populating the relations, the count for every category is needed for all city pairs. Cypher, however, has no easy way to count all labels per type. Therefore, the query to achieve the desired counts is slightly more complex than necessary for example in a SQL based query language:

```
1 MATCH (:City { name: 'Amsterdam' })-[:OCCURS_IN]->
2             (i:Index)
3       <-[:OCCURS_IN]-(b:City { name: 'Rotterdam')
4 WITH DISTINCT LABELS(i) as labels, COUNT(i) AS labelCount
5 UNWIND labels AS category
6 RETURN category, SUM(labelCount) AS score
```

Listing 5.12: Counting distinct labels

The query in listing 5.12 starts with matching all common `:Index` nodes between Amsterdam and Rotterdam. Then, it collects the labels of the found nodes and counts them. However, the issue with this is that `LABELS(node)` returns a list of labels. The counts (`labelCount`) therefore do not represent individual labels but groups of labels. The lists are therefore expanded using `UNWIND` and all counts are collected.

The full implemented model is given in figure 5.3

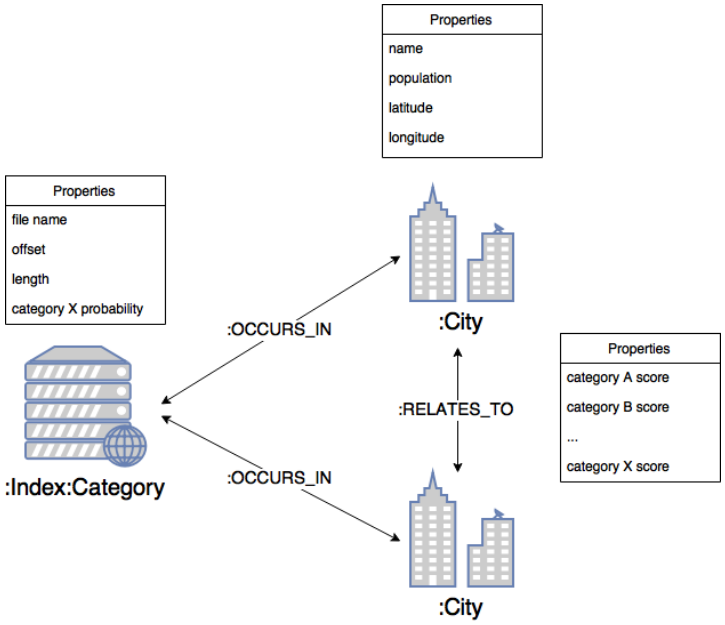


Figure 5.3: The Neo4j model as implemented

During the first tests of the database functions, it appeared that matching and creating was very slow. Checking the query plans (by means of prepending the very useful `PROFILE` to the queries) gave no hints however. It turned out that we did not use parameterised queries, where we should have. Because of this, Cypher had to recompile, plan and optimise the queries over and over again, which is clearly unnecessary if only the query parameters change. To illustrate the effect of this change, consider the list of cities we used, containing over 1600 cities. Matching all of these in a single transaction by name, without parameterising the query, comes down to repeating the following query for every city:

```
1 MATCH (c:City { name: 'Amsterdam' }) RETURN c.name
```

Listing 5.13: Matching cities without parameterising

The same approach can be used for a parameterised query, where `{name}` is the parameter:

```
1 MATCH (c:City { name: {name} }) RETURN c.name
```

Listing 5.14: Matching cities without parameterising

Even for these 1600 simple matches, the improvements are already significant, as can be seen in table 5.1.

non-parameterised	2.84s
parameterised	0.85s

Table 5.1: Parameterised versus non-parameterised total execution times

Another issue that was encountered during initial testing, is that executing only one query at a time creates and commits a transaction for every query. This is an expensive process. Using the parameterised query from listing 5.14 in a single transaction is a 15% performance gain, as shown in table 5.2. Moreover, it seemed as if performance was fluctuating with a transaction per query. However, we have not benchmarked this.

single transaction	1.07s
parameterised	0.85s

Table 5.2: Parameterised versus non-parameterised total execution times

As will be explained later in section 5.7, we use multiprocessing to glue the system together efficiently. However, it turned out that Neo4j did not behave correctly when nodes and relations where matched, updated or created from multiple processes. The number of queries performed did not match the number of results returned. In fact, the numbers diverged about 15%. However, we currently suspect it is not a platform-wide issue. The server the system runs on successfully inserted 154000 `:Index` labelled nodes and created all required relations, with the use of multiprocessing. We currently suspect (but cannot verify) the problem is related to the version of Neo4j designed for macOS (which half of the group uses). However, we only discovered this close to the deadline of the project. We therefore decided not to include multiprocessing at all for database communication.

5.5 System API

To provide an easy way of interacting with the back-end of the system we decided to develop a web API. With this API the different parts that compose the complete UrbanSearch system are made easily accessible. During the development of the API we have tried to adhere to best-practices and community standards of Apigee [27]. Appendix D contains a detailed description of the available API routes and their functionality.

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 if available, the response body also contains a status code which indicates if a request was handled successfully.

5.5.2 Flask

The UrbanSearch back-end is written completely in the Python programming language. To be able to accept HTTP requests that leverage the system we needed a framework that works efficient with Python code. We chose Flask[5], a Python microframework that lets us handle incoming requests and use the functionalities of the UrbanSearch system.
The easy setup and good documentation, combined with the small size and efficiency made us decide to use Flask as our web framework.

5.5.3 Blueprints

To keep our API modular we use the concept of `Blueprints`[6]. `Blueprints` in flask are modules that implement functionality which can be registered to an application. This way we can keep parts of the API separated and relatively simple. After initialising a `Blueprint`, routes can be assigned to it. An example of `Blueprint` is shown in listing 5.15.

```
1 predict_api = Blueprint('predict_api', __name__)
2 ct = ClassifyText()
3
4
5 @predict_api.route('/', methods=['POST'], strict_slashes=False)
6 @predict_api.route('/predict', methods=['POST'], strict_slashes=False)
7 @is_json
8 def predict():
9     """
10    API route for predicting the category of the supplied text.
11    The request should have type set to application/json and the provided JSON
12    should have a text attribute containing the text for which we want to
13    predict the category.
14    """
```

[5] http://flask.pocoo.org/
[6] http://flask.pocoo.org/docs/0.12/blueprints/

```
15      try:
16          prediction = ct.predict(request.json['document'])
17
18          return jsonify(category=str(prediction[0]),
19                         status=200)
20      except Exception as e:
21          return jsonify(error=True,
22                         status=500,
23                         message='Getting the prediction failed')
```

Listing 5.15: Example code of a Blueprint used in the UrbanSearch system

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 to 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 a way that is easily understandable for an end user. Another desire that was expressed by our client, was the possibility to manipulate the displayed information in a fast, easy and intuitive way. To keep the front-end flexible we decided to separate the front-end from the API. This way we can easily make changes to both systems while not having to worry about breaking either system.

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 motivation for our most important design choices.

Modular Design

Dealing with huge amounts of data and displaying this data in a way that makes this easy to understand for users is a challenging task. The complexity of handling the data and making it easy to manipulate by the end user means an increase in the complexity of our code. The growing desire of our client to be able to view and manipulate the data lead us to a modular implementation of the front-end. Besides the fact that this approach increases readability, maintainability and extensibility it is also a best practice in the front-end realm[7].
Modular development means writing self-contained elements of a web page, consisting of HTML, CSS and JavaScript. The components can be reused easily throughout the entire page and can be initialised with different sets of data to alter their appearance or functionality.
We also used the concept of container and presentational components[8]. The idea behind this is that container components are concerned with the application logic. Presentational components on the other hand, are concerned with how elements looks, e.g. the styling and appearance of elements.

NodeJS

We chose NodeJS as the backbone of our front-end server. The fact that NodeJS is easy to setup and has a lot of modules that are quickly accessible through NPM[9] was one of the main reasons for selecting NodeJS. Having a server running in a matter of minutes is a big advantage for a short project.
Another advantage of using NodeJS is the availability of tools like Webpack and Gulp. Webpack[10] is a module bundler for NodeJS that provides us with the ideal infrastructure for modular development. It allows us to bundle JavaScript code in different files, containing only the required code

[7] https://developers.google.com/web/fundamentals/
[8] https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0
[9] https://www.npmjs.com
[10] https://webpack.js.org/concepts/

for a specific page. Gulp[11] is a task runner for NodeJS that allows for tasks such as compiling HTML templates that can later be used in the front-end. Another example of a Gulp task is bundling the CSS of all the modules to one file.

ExpressJS

To handle the routing of incoming requests in the front end we use ExpressJS[12]. ExpressJS is a minimal NodeJS web framework that can be set up and extended easily. With over ten million downloads per month it can be considered a community standard. It allows for easy and clean routing using the `Routing` module, and next to that, custom middelware can be added easily per route or for the whole app.

5.6.2 User Interfaces

In this section we will provide an overview of the different interfaces that we implemented. We will start with the main interface, the interactive map that shows the extracted relations. Next, we will discuss the document classification interface. Finally we will present the system settings interface that we implemented.

Interactive Map

The main part of the front-end is the interactive map. The extracted data is visualised on a (n almost) full-screen Google Maps map. On the map we place markers which represent cities and we draw poly-lines which represent relations between the cities. Only cities that have more than 500 documents in total associated with them are displayed on the map. This way we keep the front end performance up while preventing unnecessary map clutter.

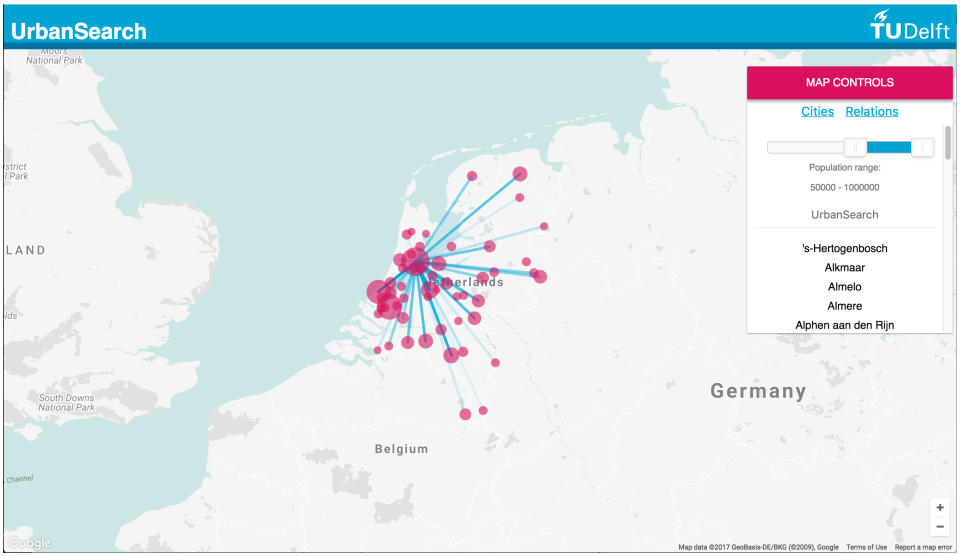


Figure 5.4: View on page-load of the interactive map

On the right side of the interface a ”Map Controls” card is always shown. This card makes it possible for users to manipulate the data shown on the map. Since we have two main entities of interest (cities and relations) that are shown on the map, the controls offer an intuitive way to control these.

[11] `http://gulpjs.com/`
[12] `https://expressjs.com/`

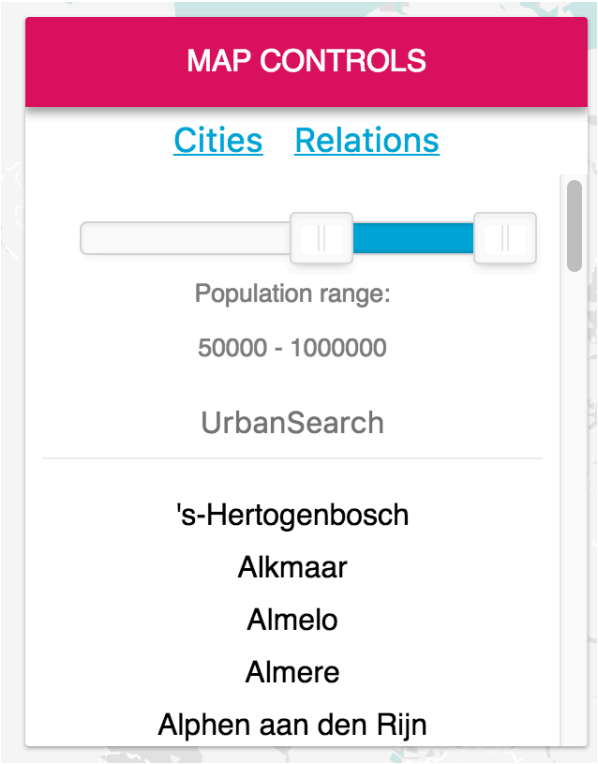


Figure 5.5: Map Controls for the interactive map - Cities

Cities are indicated by the circles on the map. The circles have a diameter that is calculated based on the population size of the city. Figure 5.5 shows the controls for the city entities. The slider shown in the upper part of the card allows filtering of cities based on population. All cities that are within the range of the slider are shown, all cities not within this range will be invisible. Besides displaying and hiding cities based on population size, users can also toggle visibility of cities by clicking on the cities displayed in the list below the slider.
Relations are shown as poly-lines on the map. The total strength of the relation (which will be discussed below) is used to set an opacity for the relation, as shown in figure 5.6. A low opacity indicates a strong total strength of the relation and vice versa.

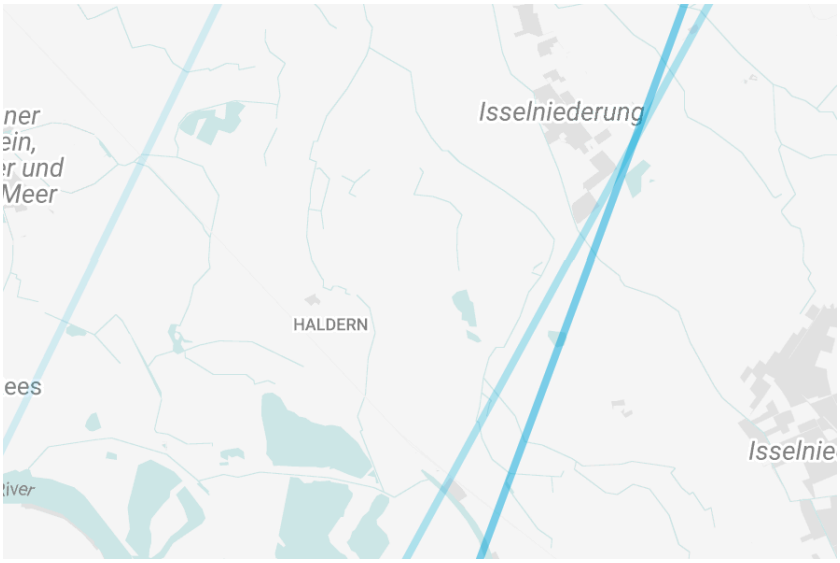


Figure 5.6: Different relation strengths are shown by adjusting opacity

The relations controls are shown in figure 5.7. Here the user can select which relations are considered when calculating the total relation strengths. This is done by clicking the check-boxes next to the name of the relations. The sliders provide a way to filter the relations based on the total relation strength of that particular relation.

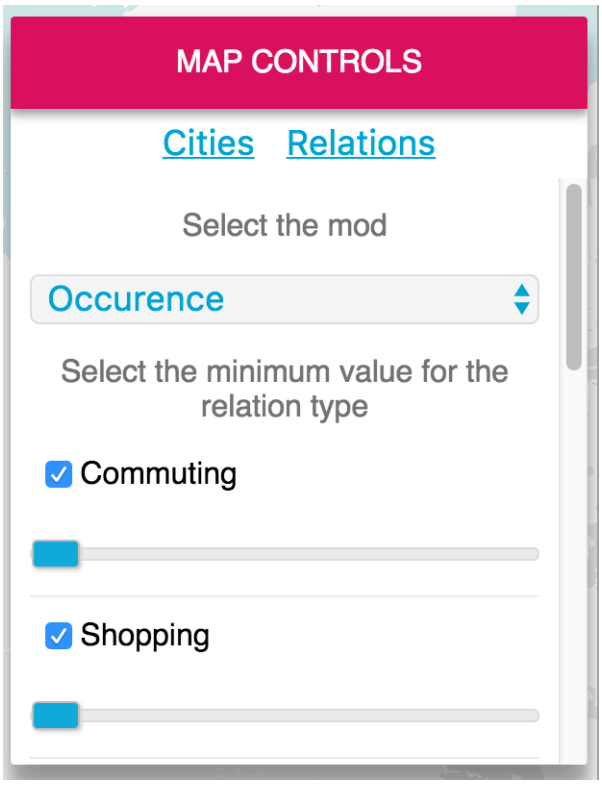


Figure 5.7: Map Controls for the interactive map - Relations

The drop-down shown in the relations control interface provides a way to scale the relation totals. For example, if we do not want the absolute count of occurrences as a measure for total relation strength but we want it to be scaled relative to the population size of two cities. This can be done by selecting the right option in the drop-down.
All of these features are meant to provide the user with means for constructing a visualisation that uncovers patterns which are not (easily) visible when looking at exported numerical data.

Document Classification

The document classification interface is designed to provide an easy way of extending our dataset which is used to train and test our classifier.
As stated in 4.4.3 we looked at several options of collecting training data.
We decided to provide our clients with a document classification interface, because we found this was the best way to extend the training data set. This way, we have total control of the documents that are added to the data set. The documents are labelled by experts in the field of the built environment so we may assume these documents will represent the labelled categories well.
The interface loads documents that we have deemed relevant while analysing the available Common Crawl pages. The user can select one or multiple categories which the user feels best relate to the document, after which the document will be saved to the training set(s) corresponding to the selected category or categories. If no category is selected, the document is discarded from the training set. If the user finds the document to be irrelevant the document can also be discarded immediately. An example of the interface with a document belonging in the ”collaboration” category is shown in figure 5.8).

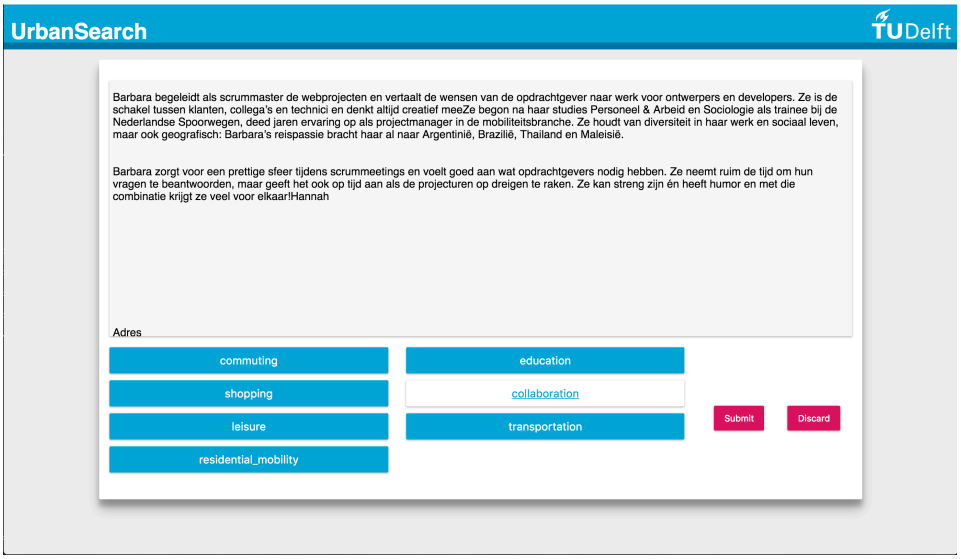


Figure 5.8: The labelling interface

System Settings

Both the back-end and the front-end of the UrbanSearch system share and depend on some default settings. To keep the system configurable, even for a user that is not a developer, we have implemented a system settings interface (figure 5.9). The interface can be extended with more settings easily if the need arises. This is achieved by the loose coupling of the front-end and the API that was mentioned before. Open issues regarding the settings interface will be discussed in 7.1.5.

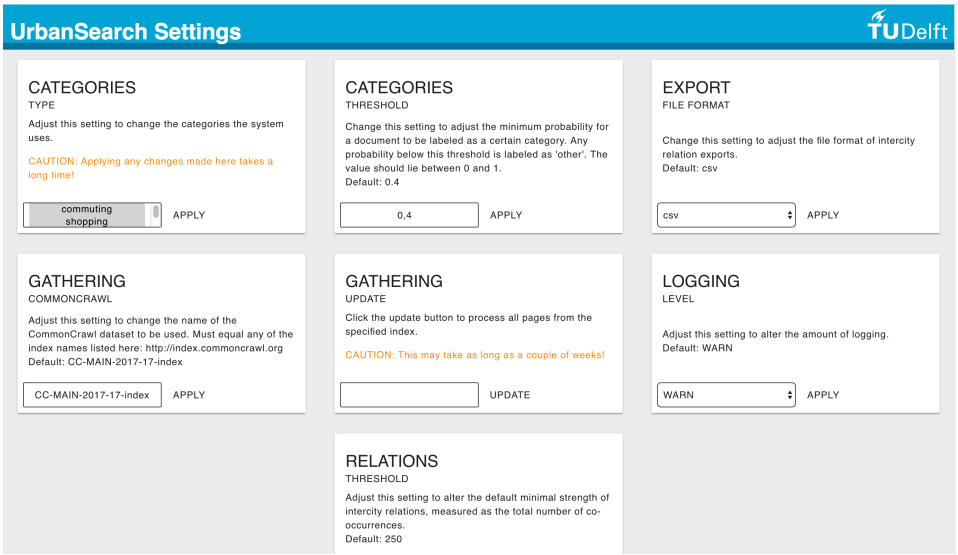


Figure 5.9: The systems settings interface

5.7 Main Application

To couple all functions of the system, we created a main module. This file contains functions that call all the appropriate functions of the subsystems, thereby providing the functionality that is desired. For example, the `classify_textfiles_to_db` function (see listing 5.16), which combines filtering, classification and storing data while making use of multiprocessing to divide and speed up the execution.

```
def classify_textfiles_to_db(num_cworkers, directory, threshold, to_db=False):
    """ Run workers to classify all documents and output to database.
    Database must be online, all the textfiles from the specified directory
    will be parsed using the number of workers specified.
    :num_cworkers: Number of consuming workers, classifying indices from the
    queue.
    :directory: Path to directory containing textfiles
    :to_db: Output results to database specified in config, True or False
    """
```

Listing 5.16: Header of a function in main.py

The most important functions in main are listed in table 5.3. These functions can be called using the API or by adjusting the function that is called in the `__name__ == "__main__"`[13] part of this file.

Name	**Description**
`download_indices_for_url(url)`	Download all indices for a given url and return as string.
`classify_documents_from_indices(pworkers, cworkers, directory, threshold)`	Run workers to classify all documents and only log results, no output to database.
`classify_indices_to_db(pworkers, cworkers, directory, threshold)`	Run workers to classify all documents from a file/directory with indices and output to database.
`classify_textfiles_to_db(num_cworkers, directory, threshold, to_db=True)`	Run workers to classify all textfiles and output to database.
`create_ic_relations_to_db(num_workers, to_db=True)`	Create intercity relations and store them in the database if desired.

Table 5.3: Functions available in main

[13] `https://docs.python.org/3.5/library/__main__.html`

6

Project Evaluation

In this chapter we will first discuss the testing of the system, evaluation of the classifier and evaluation of the final results. After this we will evaluate the fulfilment of the requirements we set up with our client. Next we will evaluate the fulfilment of the design goals. Finally, we will give the product evaluation and a short evaluation of the development process.

6.1 Validation and Verification

In this section, we first describe how we tested the system by means of automated and manual testing. Next we discuss how we tried to ensure code quality by following guidelines set up by SIG[38]. Finally we will evaluate classification results and the validity of the relation scores that were assigned to relations between cities.

6.1.1 Testing the Application

We will test the program using four different testing methods. The first is unit testing, which tests the individual functions in our code. Next we consider integration testing. This tells us if different system components interact like we expect them to. Finally acceptance testing is used to verify the clients requirements. The testing framework that we used is pytest[22], which is part of Python.

Unit Testing

Unit testing is the process of testing individual units of code for correctness. If a unit of code depends on other parts of the system, mocks can be used to mimic this dependency. This way we can test the functionality of all the individual components of the system, without having to rely on correctness, or even the presence, of units of code the tested unit depends on.
In figure 6.1 a coverage report can be found, which summarises the percentage of the code that is covered by the implemented unit tests.

Integration Testing

Integration testing uses automated tests which test how well different components of the system work together. Do to time constraints we did not implement any integration tests.

System Testing

We also used 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.

Name	Stmts	Miss	Cover
urbansearch/__init__.py	0	0	100%
urbansearch/clustering/__init__.py	0	0	100%
urbansearch/clustering/classifytext.py	33	2	94%
urbansearch/clustering/link2doc.py	20	1	95%
urbansearch/clustering/mnb_modelmanager.py	11	0	100%
urbansearch/clustering/modelmanager.py	57	8	86%
urbansearch/clustering/sgdc_modelmanager.py	11	0	100%
urbansearch/clustering/text_preprocessor.py	20	0	100%
urbansearch/filtering/__init__.py	0	0	100%
urbansearch/filtering/cooccurrence.py	36	0	100%
urbansearch/gathering/__init__.py	0	0	100%
urbansearch/gathering/gathering.py	125	12	90%
urbansearch/gathering/indices_selector.py	63	8	87%
urbansearch/gathering/text_downloader.py	48	8	83%
urbansearch/main.py	103	12	88%
urbansearch/server/__init__.py	0	0	100%
urbansearch/server/classifier.py	63	2	97%
urbansearch/server/classify_documents.py	66	0	100%
urbansearch/server/dataset.py	59	8	86%
urbansearch/server/decorators.py	8	0	100%
urbansearch/server/documents.py	22	0	100%
urbansearch/server/indices.py	19	2	89%
urbansearch/server/main.py	25	1	96%
urbansearch/server/predict.py	21	0	100%
urbansearch/utils/__init__.py	0	0	100%
urbansearch/utils/dataset_p_utils.py	99	23	77%
urbansearch/utils/db_utils.py	202	6	97%
urbansearch/utils/p_utils.py	14	0	100%
urbansearch/utils/process_utils.py	20	3	85%
urbansearch/utils/score_utils.py	6	0	100%
urbansearch/workers.py	208	22	89%
TOTAL	1359	118	91%

Figure 6.1: Report of test coverage using coverage.py

Acceptance Testing

Finally we used acceptance testing to verify if the software does what the client expects it to do. The tests are therefore executed by the client manually. These tests are derived from the system requirements which can be found in 3.3.3.
An example is the testing of the interactive map which visualises the relations between cities.

6.1.2 SIG

SIG [38], short for Software Improvement Group, is an organisation that analyses code of software projects. SIG tries to give insights about the quality of the analysed code. SIG has an online tool, Better Code Hub, which rates code based on the criteria that can be found in appendix A. It is a convenient tool, which we used during the project to verify the quality of our code was up to par.

week 5

The first feedback from SIG was in the fifth week of development. While we scored 10 out of 10 on Better Code Hub, we scored 4 out of 5 in the review we got back from SIG. The main issue with our code was that some functions were too complex (too long, too many branches or too many input parameters). Still they rated our code as above average maintainable.

week 9

Since the deadline of the final upload to SIG is the same as for this report, we unfortunately can not present these results. Instead we will present the results we got from Better Code Hub, ran on the due date of this report. This time there were two issues which resulted in a score of 8 out of 10. The issue was that we did not keeping unit interfaces small. Meaning we had methods that take too many input parameters. The other issue was with coupling architecture components loosely, which is the result of our workers being used in multiple parts of the application.

6.1.3 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 [24]. These guidelines state that binary classification can be evaluated with the following metrics: Accuracy, Precision, Recall, F1 and AUC. Note that we will use a one vs the rest strategy, meaning the result will be evaluated per category.

Accuracy

Accuracy is the proportion of correctly classified instances. This however is a poor indication of how well a 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 will be 90%. It would seem the classifier performs well, but it actually fails to classify the other 10% of the websites correctly.
To calculate the mean accuracy of our classifier we implemented an API route that splits our complete data set in random training and test samples. The training sample contains 75% of the complete data set and the test sample contains the remaining 25%. After issuing multiple calls to the API we calculated a mean accuracy of 79.64% for our current classifier.

Confusion Matrix

A page can belong to class A (positive), or not belong to class A (negative). If a page is labelled by the classifier with the correct class it is called a true positive (TP) or a true negative (TN). If the classifier labels the page incorrectly it results in a false positive (FP) or false negative (FN). This concept is visually depicted in figure 6.2.

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

Figure 6.2: confusion matrix [1]

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 recall. It is computed by using the following formula: $F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$. The F1 score summarises evaluation in a single number. But to get an better understanding of the behaviour of the classifier considering recall and precision on their own is recommended. The reason for this is that the F1 score gives no information on what is the cause for the loss of accuracy, while the scores for precision and recall give us a better insight.

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

	precision	recall	f1-score
collaboration	0.83	0.56	0.67
commuting	1.00	0.88	0.93
education	1.00	0.78	0.88
leisure	0.57	1.00	0.72
residential_mobility	1.00	0.89	0.94
shopping	0.94	1.00	0.97
transportation	0.93	0.87	0.90
avg / total	0.90	0.86	0.87

Figure 6.3: Precision Recall and F1 table

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, we need the probabilities that pages are classified correctly. For each possible threshold, the true positive rate and the false positive rate are calculated. These are plotted in a graph, which results in something like 6.4. 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. The ROC curves for our categories can be seen in appendix C. The low smoothness is due to the low sample amount, nevertheless they show good results as the curves are near the upper left corner.

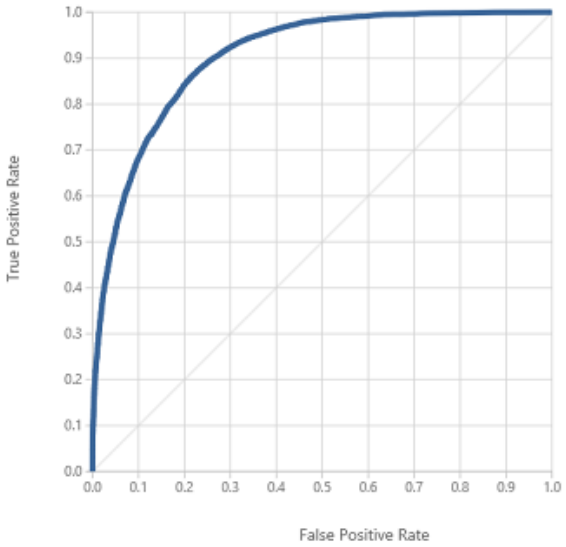


Figure 6.4: ROC / UAC graph [2]

6.1.4 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 [41], so one would expect cities that lie close to each other to be more related than cities that are located on opposite sides of the country. This natural relation can be represented using the Gravity Model by Reilly [33]. 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

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

thus expose a similar relative score as it would in the gravity model. Consider for example Amsterdam and Hoofddorp, two 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 high score in our system, this would imply that the system scored this relation correct. Besides the Gravity Model, one can rely on the knowledge of an expert in the field of urbanism that can judge whether an extracted relation resembles reality. We therefore agreed with the client, which in fact is an expert in the field of urbanism, that they would verify on a subset of relations if these resemble reality. Lastly, the relations in the Randstad, a large urban area with the four largest cities of the Netherlands, have been examined before in [44]. These relations can be compared to the results extracted by the UrbanSearch system.

6.2 Evaluation 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 and why. Failed requirements are discussed in section 7.1. As can be seen most of the requirements passed, but unfortunately some failed. Most of the failed requirements are acceptable. We plan to implement the failed must have in the last week of coding. Since this is after the due date of this report, this fix can not be included in the report.

Table 6.1: Requirements fulfilment

Must Haves	Pass / Fail	Comment
1 Mining from Common Crawl	Pass	Data is successfully gathered from Common Crawl.
2 Exporting relations	Fail	Due to many delays in other parts, exporting the data was not achievable within the time set for the product.
3 Extracting relations	Pass	Relations are successfully extracted from documents for about 80-85 %.
4 Visualisation	Pass	A front-end which shows the data on a map is ready for use.
5 Present statistics	Pass	The front-end presents statistics with the shown data.
Should Haves	Pass / Fail	Comment
1 Hierarchical relations	Fail	Since extracting relations from text documents was more challenging than first thought, hierarchical relations were not included.
2 Machine learning retrainable	Pass	It is possible to retrain the machine learning by feeding it a set of labelled documents.
3 Add large data sets	Pass	It is possible to add large data sets, however since it does cause an increase in time needed to run the algorithm, we did not use a large data set for our demo version.
4 Duplicate city names	Fail	The algorithm does not take cities with duplicate names, or names fitting to multiple cities into account, due to time constraints.
Could haves	Pass / Fail	Comment
1 Use Delpher	Fail	It is not possible to use data from Delpher unless it is already downloaded and stored.
2 Visualisation for comparing	Pass	The front-end includes visualisation for comparing cities and relationships to each other.
Would likes	Pass / Fail	Comment
1 Show all connections	Fail	Theoretically it is possible to do this, but it would result in a map on which relationships can not be differentiated from each other due to the large amount the system was not build for (see section 4.7.3).
2 Other than .nl data	Fail	The classifier is only trained on Dutch domains.

6.3 Evaluation of Design Goals

In section 3.3.2 we discussed the design goals for this project. We came up with seven design goals on which we will reflect.

6.3.1 Credible

As stated in section 6.1.3 the classification seems to be pretty accurate except for the recall score for collaboration and the precision value for leisure. This is most likely due to the limited amount of training data. These positive results can also be seen in appendix C.

6.3.2 Understandable

Together with the client we concluded the visualisation as presented in 5.6 meets their requirements.
Although their still remain issues with the visualisation, which are discussed in 8, especially when visualising many relations.

6.3.3 Scalable

The main goal concerning scalability was to allow the system to process pages from domains other than ".nl". Since we used Neo4J for storing relationships, which is highly scalable, storing the relationships for domains other than ".nl" should not be a problem. The main issue here is that of processing other languages than Dutch.

6.3.4 Plugable Data

The design goal plugable the goal was to make the application able to perform analysis on different data sets without the need of a developer. Whilst this is possible, the documents from other data sets would have to be in the correct format before they can be processed. For small data sets this might not be that much of a problem, but for larger data sets it would become very time-consuming without the help of a developer.

6.3.5 Exportable

At this moment we have not met the design goal exportable, which was to ensure the numeric data could be exported, for example in CSV format. Since this is not a vital part of the application, this has not yet been included. We are however working on this and expect it to be done in the final week of development.

6.3.6 Fast Development

Another goal was to have a fast development cycle because of the time constraints. To do this we choose tools, applications and programming languages with which members of our team were already familiar. Even so, some setbacks occurred which caused the cycle to slow down.

6.4 Product Evaluation

To conclude the two previous sections; we have a functioning program that consists of most vital components. Whilst the results from the application are not exportable (yet), the other design goals were met. The program provides credible and understandable data which can be used to analyse relationships between cities in the Netherlands.

6.5 Process Evaluation

In this section, we evaluate the development process, explain what methods were used and if they were used correctly. Additionally, we discuss the collaboration with the client, the coach and within the group.

6.5.1 Development Process Evaluation

In order to have a smooth development cycle, we made several agreements in the beginning of the project. All code changes had to be submitted through a pull request and needed to be reviewed before they could be merged into the main code base. This was enforced using the project settings in GitHub, where branch access can be regulated. We believe that this approach has helped us to write quality code and to make sure everyone knew what was going on.

Furthermore, we used Travis CI[3] for continuous integration. Building was automatically triggered by both pull requests and normal pushes. GitHub also provides an option to require status checks to pass before being able to merge. However, we disabled this due to some testing stability issues in the first few weeks of development. We did however agree to only merge when all status checks passed. In all but a few hasty merges we managed to adhere to this agreement. Because of the continuous integration of Travis CI, we hardly ever had to deal with unforeseen integration problems of new features.

Travis was also configured to submit coverage reports to Coveralls[4], so we could easily monitor how test coverage was affected by code changes. Rule of thumb was that all newly added files should have at least 80% code coverage. Through Coveralls, we could quickly verify this.

The product development was managed using the agile development methodology Scrum. Each week was a single sprint. We kept track of the current sprint and the product backlog in Trello[5], which helped us to have a clear overview of the product's status. We did however notice that weekly sprints were a bit too short. Usually, a sprint was too full and in the end we noticed that we got somewhat careless about sprints. Moreover, at times, unexpected time consuming issues lead to not finishing the sprint at all. It might therefore be beneficial to extend sprint duration to two weeks to allow for unexpected problems.

The system was initially run on a relatively small virtual private server (4GB RAM, 2 CPUs, 150GB SSD) of one of the group members. With increasing database size, we noticed that we would require more resources, especially RAM. We therefore asked the client to request a server of the TU Delft that we could use for the application. After a few weeks of inefficient (mis)communication, we eventually got access to a 8GB RAM, 4CPUs, 100GB HDD virtual machine. This server meets the minimal requirements for the system, but does not provide enough resources if the data set is extended to more than a million documents. Moreover, the virtualisation is not ideal for the many disk IO the application requires. Therefore, it would have been better to have a physical device at hand.

6.5.2 Communication Evaluation

Communication with the client went very well throughout the entire project. We could always walk into the office with questions or remarks, or email the client if he was not present. He also complimented us whenever he liked something we achieved, but remained critical in his feedback.

The coach and the group had some teething problems but managed to improve quite satisfactory on this. We received useful feedback on both the reports and design. Moreover, she kept forcing us to keep feasibility of our solutions in mind, to make sure we kept aligned with the time schedule.

The group communicated well over the course of the project. We agreed to be present every day of the week between 09:00 and 17:00 to make sure everyone was involved.

[3] `https://travis-ci.org/`
[4] `https://coveralls.io`
[5] `https://trello.com`

7 Discussion

This section is divided into two parts. First we will discuss the issues that remain. Since the classifier requires some more attention, it is discussed in a separate section. The last part of this section is dedicated to discuss possible ethical issues this project may cause.

7.1 Open Issues

Although we managed to handle most of the issues that arose during development, some remain unsolved. However, we believe that with more time, we could have found a solution to most issues. This is especially true for the classifier. Open issues with the classifier are therefore discussed separately, in section 7.2.

7.1.1 Downloading and Parsing Indices

The downloading part of the system is arguably the easiest of all. Indeed, the issues that remain are more related to the resources available, than to the implementation. Downloading speed is dependent on the connection to Common Crawl. Since their data is hosted at Amazon, it might be a lot faster to use a virtual private server from Amazon to host the system on, at least for data collection and storage. One is then able to use the Simple Storage Server (S3)[1] to retrieve data faster from Common Crawl.

Another significant improvement is to use SSD instead of HDD storage, to speed up both reading and writing of files.

A final issue is that Web developers can choose which character set they use for the page content. We were struggling to find a fast and correct way to determine the encoding of the page and then convert it accordingly to a general encoding. Eventually, we decided to stick to UTF-8 for every document and ignore characters that cannot be encoded in UTF-8. However, ISO 8859-1 (also known as latin-1) is widely used in the Netherlands for special characters.

7.1.2 Filtering the Data

Filtering the downloaded documents went quite well overall, as explained in section 5.2. However, we did leave out some important aspects. Most importantly, we had no means of checking on city aliases (like 's-Gravenhage is for Den Haag and Domstad for Utrecht). A possible solution to this is to keep track of a list of aliases pointing to the actual name. However, since this also requires changes further on in the system, we decided to leave this out. Additionally, we did not check for complex occurrences of cities, such as "Amsterdammers", people from Amsterdam. We decided to do this due to some large cities that are identical to or contained in commonly used words. An example of this is the city Leiden, which translates directly to the verb "to lead". However, filtering

[1] `https://aws.amazon.com/s3/`

every complex occurrence out is a too aggressive kind of filtering, resulting in discarding otherwise useful documents.

7.1.3 Storing the Data

Storing the data efficiently turned out to be slightly harder than expected due to the many issues we faced. The issues that were overcome are discussed in section 5.4. There are some that have been left open. The most structural issue is that after all, a graph database is a bit overkill for our purposes. We traverse nodes with only a maximum depth of two (for documents in which both city A and city B occur). Such traversals are not complex and might even perform better in a SQL database with foreign keys.

Besides, we did not manage to get the database speed that we expected to reach. This could have to do with the fact that we use the HTTP transactional endpoint instead of the internal API, which is only available for the Java programming language. Perhaps it would have been better to write the database code in Java instead of in Python to use this internal API. To some extent this also has to do with the fact that we do not have a physical server with a modern SSD at hand to run the database on. This would increase reading and writing speeds significantly.

7.1.4 System API

The most prominent issue with the system API is the lack of authentication and authorisation. This can lead to serious security issues, since anyone can send requests to the API and make the system execute operations. User authentication and authorisation can be added easily[2] but was left out unfortunately due to time constraints.

Another issue that is left for the API is that it does not provide all the functionality that may be wanted in the (near) future. Again this is because of time constraints, which forced us to implement only the routes that were needed for the minimum viable product.

7.1.5 Front-End

The main issue in the front end is that settings interface is not connected yet to the API. This means, although we do have the setup for the interface, that submitting a setting from the interface does not trigger a call to the API.

Furthermore, we have small styling issues. For example, relations that have a really small total score are almost invisible when visualised on the map. Fixing this would be done by adjusting the scaling function, which takes the total relations score and scales it to an number between 0 and 1 relative to the maximum total relation score. Scaling now is done by taking the square root of $total\_score/max\_total\_score$.

One last issue is the styling of the loaded document in the classification interface. To display the document we load it into a HTML `<pre>` tag. The `<pre>` tag displays text like it was formatted, other HTML tags skip white space and line breaks. Since we parse HTML pages to plain text, we end up with documents that have a lot of white space and line breaks. This leads to barely readable text, an example is shown in figure 7.1. This can be fixed by pre-processing the documents before they get loaded into the `<pre>` tag. This will not harm the validity of the data set since we do not alter any meaningful content.

[2] `https://realpython.com/blog/python/token-based-authentication-with-flask/`

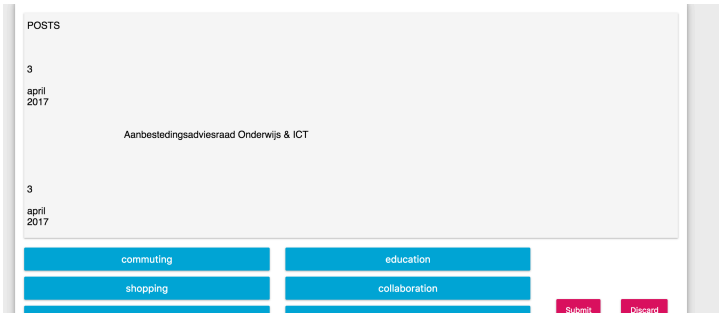


Figure 7.1: Example of barely readable text in a <pre> tag

7.1.6 Main Application

Combining all components of the system into a full-fledged main application went fine overall. However, a few features were left out that in fact should have been present. For example, there is no decent way for invoking the application via the command line. The code has to be adjusted depending on what part of the system should be started. Moreover, not every part of the system is configurable and we have no progress indication if the system is running. However, we believe these features can be easily added in future versions. It was therefore not our top priority to include them.

7.2 Classification

In this section we will discuss some of the issues that are prevalent in the classification subsystem. First we will talk about the modest size of our data set. After that we will discuss how selecting a threshold proved to be a more difficult task then envisioned at the start of the project. Finally we will explain how this threshold lead to difficulties with assigning categories to documents.

7.2.1 Data Set

One of the main issues, if not the main issue, of the classification subsystem was the lack of quality documents in our data set. Finding documents that describe our categories well, proved more challenging than envisioned at first. This was one of the reasons why we implemented the classification interface in section 5.6. The disadvantage of this method is that it takes an individual to manually assign categories to the documents that get loaded into the interface. Since we were busy developing most of the time we did not have a lot of time to categorise documents. Also our client had a busy schedule which lead to the modest size of our data set.

Another issue concerning the data set is the fact that, to construct a fair classifier, we need data sets of equal size for all the categories. While this is easy to implement, it does mean that we are forced to use only the number of documents of the category data set of minimal size for every category. So if for example ”shopping” has the least documents, lets say 50, we can use only 50 documents of every category data set.

7.2.2 Threshold

We use our trained classifier to estimate the probabilities of a document belonging to each available category. From the probabilities calculated by the classifier, we can decide which categories are assigned to the document in question. We do this by using a threshold which tells us what the minimum probability of a document belonging to a category is, that leads to assigning that category to the document.
To decide the optimal threshold, we would need to manually examine results of multiple runs to decide on the ideal value for the threshold. Unfortunately we did not have the time to do this, which means we had to decide on the threshold rather arbitrarily.

7.2.3 Ambiguity of Document Subject

An issue flowing from the issue with deciding on a solid threshold is that of it leads to documents being classified ambiguously. The fact that we do not have a solid threshold which we can rely on means documents may get added to categories they do not belong to, or that documents are assigned to one category, for example ”shopping”, while they also should be tagged with the category of ”transportation” for example.

7.3 Ethics

In this section some of the ethical issues with respect to the developed product are discussed. First, possible issues with storing web data are discussed. Next, we discuss the potential consequences of extracted relations.

7.3.1 Storage of Data

One of the ethical issues is the storage of web pages. Although these pages are accessible to anyone at the time of downloading, this might change in the future. The owner of the original web page may have good reason to delete the original page, however, this does not mean it is deleted from the local storage of our application. Another issue with storing the web pages locally is a potential violation of copyright. As Thelwall et al. stated, ”web crawlers ostensibly do something illegal: They make permanent copies of copyright material (Web pages) without the owner’s permission.”[40] Because we store copies of the web data that has been crawled and stored by Common Crawl, the same applies to our application.

7.3.2 Consequences of Extracted Relations

Another issue is that it is unknown how the results of the application will be used. It was designed for research purposes, but there is no way of knowing what the results will be used for. For example, the extracted relationships show which cities are the most important in a network of cities. This information can be used by terrorists to decide to strike in the most important city to maximise the impact.

The results may also result in some cities becoming more popular, which means they would grow in size. This might have a negative impact on for example the health and living conditions of the people in these cities.

8

Recommendations

In this chapter, we will give some pointers for both the back-end and the front-end as to how the system can be extended and improved.

8.1 Extending the Back-End

In this section, we present several recommendations to improve the back-end.

8.1.1 Extending the Data Set

The input data is now limited to Dutch documents only. This limits the amount of available data and also limits the network of cities that can be extracted from this data. In a future version, it will be interesting to be able to parse other languages as well, starting with English. This would require some investigation regarding stop words in English and what primarily causes false positives for semantic association of cities in English documents. Next to that, in a future version of the application it would be interesting to be able to use other data sources than Common Crawl. This would require a few modifications to the data storage functions, because a unique identifier per stored document is needed. These modifications should lead to an application that is able to use a lot more data to find relations between cities, which in turn should lead to more reliable and credible results.

8.1.2 Improving Document Filtering

Other than the required co-occurrence of cities, there is little noise cancelling in documents. Some documents contain lists of cities and are cancelled out as explained in section 4.3. However, there might be more intuitive ways for better document filtering. One could for example discard the contents of specific HTML elements, such as forms and input fields. Moreover, some kind of domain blacklist could be constructed to filter out entire (sub)domains that are known to contain mostly false-positives. For example, the client mentioned that many pages of Airbnb were reviews from people of all over the country, but represent no relation between these locations. Another way to extend the document filtering is to have some way to allow for city aliases. By doing so, more documents will pass the filtering stage so more relations can be extracted.

8.1.3 Constructing More Advanced Classifiers

While our classifier achieves a reasonable mean accuracy of 75% - 80% (see 6.1.3), these results can be improved upon to reach an even higher mean accuracy. Below we will explain what we see as potential improvements to the classifier.

Data Set

Our first recommendation is to keep expanding the data set. This will lead to a more accurate classifier. The classification interface can be used for this task. This way the data set can keep getting expanded over time.

Parameter Optimisation

The estimator algorithm that we use in our Scikit `Pipeline` takes a number of parameters that effect the outcomes we get back from the classifier. These parameters can be tuned as described in [1]. These parameters, so called hyper-parameters, influence the workings of the algorithm which they are associated with. Each hyper-parameter can have a lot or almost no influence on the returned results [3]. That is why it can be of great value to run a (random) search on these hyper-parameters after a `Pipeline` is defined.

Empirical Tests of Classifiers

Besides these two options it would be a good idea to implement more algorithm specific `ModelManagers` and to compare the results of these different classifiers on our data set. This way we could make sure we always select the most accurate classifier when running the classification over our downloaded documents.
Another improvement concerning the classifier would be to try out binary classifiers instead of a multi-value classifier which we currently use. A binary classifier is a classifier that takes positive inputs and negative inputs and can decide if an unseen input is in that specific class or not. This means we would generate a binary classifier for every category that we have and run a document through every binary classifier after which we would select the category or categories to which this document belongs. A multi-value classifier on the contrary can label an input with multiple classes, or categories in our case[14].

8.1.4 Upgrading the Server

As discussed in section 7.1, multiple open issues are due to the server we used, which is not as powerful as we would have liked. For future versions, the server should be upgraded to at least 16GB RAM and preferably at least 8 CPUs, to allow for more efficient multiprocessing and to be able to keep the database in memory for much faster reading. Additionally, an SSD instead of HDD would greatly improve the database speed for both reading and writing. To speed up database access even more, it would help to have a dedicated physical server that only runs the database.

8.1.5 Building a Command Line Interface

The delivered application provides an API and a visual interface, however, there is no command-line interface yet. If a user knows how to program Python it would be possible to call functions using the Python interpreter, but this is far from ideal. A future version could have a CLI (Command-Line Interface) that makes calling parts of the system fairly straightforward. For example, this would enable a user to call just the filtering subsystem or just the classification functionality on one or more documents. Next to that, the user could also change and run a separate configuration locally to experiment with parts of the application without changing the configuration of the application running on a server.

8.2 Extending the Front-End

An improvement in the front end that could add great value to the system, would be an interface that lets users manipulate classifiers and data sets. The interface should allow user first of all to select which classifier is used as the default classifier for the system. Besides that allowing a user

[1] `http://scikit-learn.org/stable/modules/grid_search.html`

to adjust parameters of the classifier in the interface and subsequently running accuracy tests that get displayed clearly, would add a lot of flexibility and value to the system.

Furthermore, viewing and editing the data set associated to an classifier and/or category would allow for validation by human users of the system and for quick adjustments if for example a document would be added to a wrong category by accident.

Finally, we would recommend to look for better solutions of displaying relations on the map. Now, if we show many cities and relations, the map basically becomes a incomprehensible clutter of lines, as show in figure 8.1.

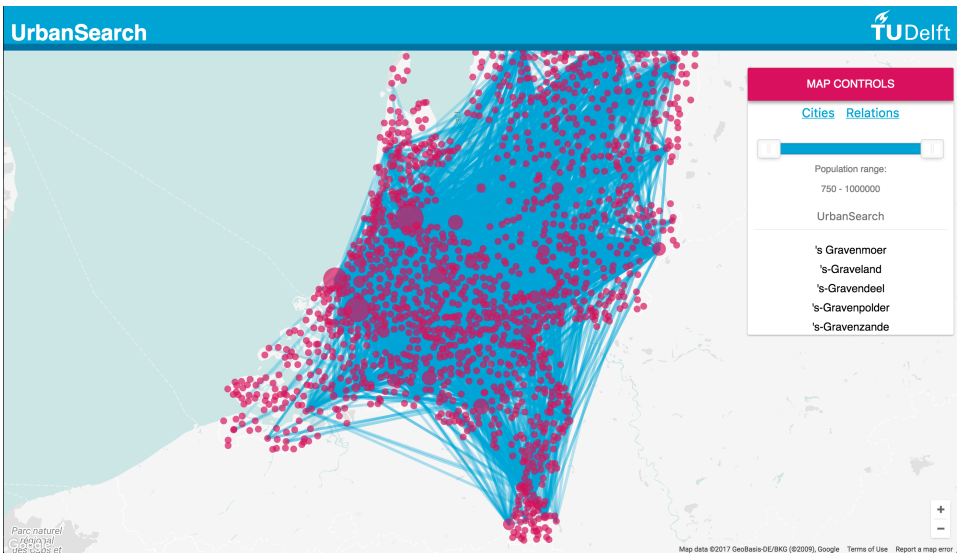


Figure 8.1: Example of map clutter

9

Conclusion

In the past few months we worked towards building an application to find relationships between cities by using data from online sources.

First, in section 2 we discovered that the related work currently present is either very limited or questionable. Next, in section 3 we identified the requirements for a solution to the problem and discuss issues that might arise. Afterwards in sections 4 and 5 we described a framework that satisfies the requirements and tackles the issues and the implementation of this framework. In section 6 we discussed the fulfilment of the requirements and the design goals, and evaluated the process. Next, in section 7 we discussed the issues we are still facing, the results of the classification, and the ethical issues our project might induce. Last, in section 8 we made recommendations for future projects on this subject.

As explained in section 6 all of the design goals, except the exporting of extracted relations, were met. While there are still some issues, the most important requirements were also successfully implemented.

We can now reflect on the problem definition from 3.1: how can open data be leveraged such that a metric for the strength of relationships between cities can be defined and visualised?

One way open data can be leveraged such that a metric for the strength of relationships between cities can be defined and visualise is as follows. First, text data is collected from a corpus (in our case documents from Common Crawl). Each document is then checked for the occurrence of two or more city names by using the multi-pattern string matching and is discarded if it does not meet this check. This selection of documents is classified according to predefined relationship types using the SVM machine learning algorithm. The documents are stored on disk and the relations are stored in the graph database Neo4J. The strength of relationships between two cities is then found by counting the number of all documents for each relationship that contain the two city names. This is visualised using a Web application.

With this, we believe we have proven the application can be an asset to the research on intercity relations and we hope this will give our clients a tool to help with their further research.

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] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[4] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] Andreas Heß, Philipp Dopichaj, and Christian Maaß. Multi-value classification of very short texts. *KI 2008: Advances in Artificial Intelligence*, pages 70–77, 2008.

[15] 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.

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

[17] Fermentas Inc. Beautifulsoup: Beautiful soup documentation, 2017. URL https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

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

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

[20] 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.

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

[22] Holger Krekel et al. pytest. `https://docs.pytest.org/en/latest/`, 2004-2017. Accessed: 2017-06-25.

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

[24] 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.

[25] 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.

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

[27] Brian Mulloy. Web api design-crafting interfaces that developers love, 2012.

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

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

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

[31] 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.

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

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

[34] 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`.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] Mike Thelwall and David Stuart. Web crawling ethics revisited: Cost, privacy, and denial of service. *Journal of the American Society for Information Science and Technology*, 57(13):1771–1779, 2006.

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

[42] 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.

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

[44] 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.

[45] 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`.

[46] 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.

[47] 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 [16] 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 Intermediary Feedback

[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.

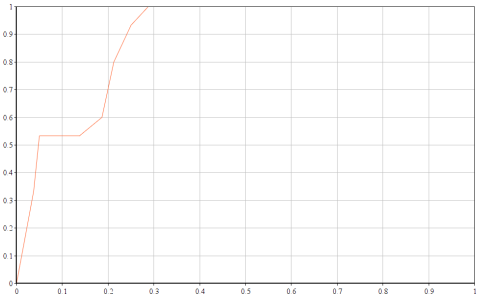
Omdat jullie qua score al vrij hoog zitten gaat het hier voornamelijk om kleine refactorings. Methodes als IndicesSelector.run_workers en CoOccurrenceChecker._calculate_occurrences zou je nog iets verder kunnen opsplitsen in functionele gebieden.

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.

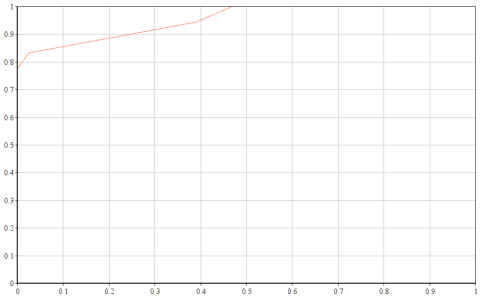
Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.

C

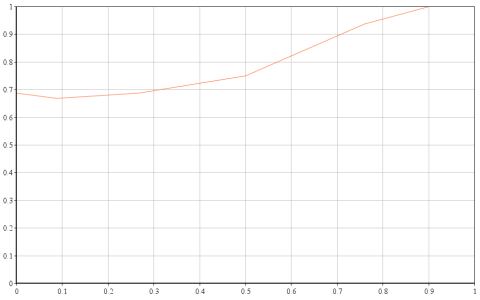
ROC



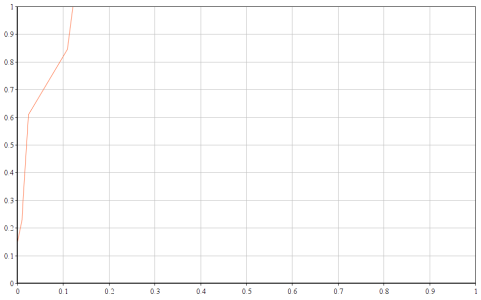
Collaboration



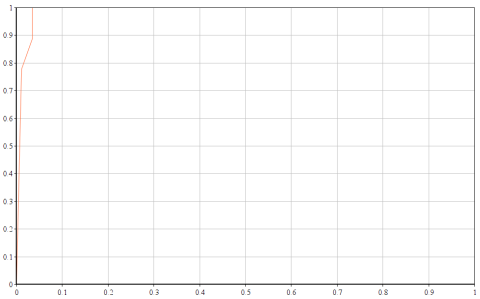
Commuting



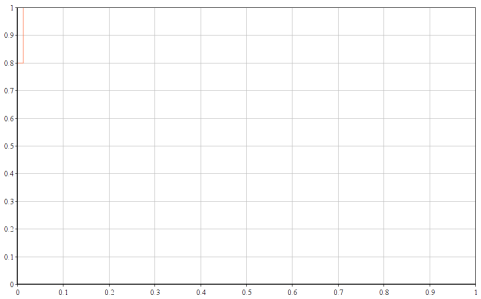
Education



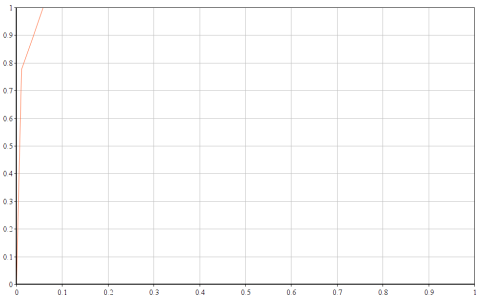
Leisure



Residential Mobility



Shopping



Transportation

Table C.1: ROC curves per class

D

API

D.1 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

D.2 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

D.3 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

D.4 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

D.5 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

E

Used Libraries

For the project, we used quite some open-source libraries. Some of them are Python libraries, others server utilities. The libraries have been split in back-end and front-end and are listed below.

E.1 Libraries Used in the Back-End

For the back-end, we used the libraries listed below. They can all be installed via `pip`. The links they can be found at are constructed by substituting "<package>" in the following URL: `https://pypi.python.org/pypi/<package>`

beautifulsoup4 Used for HTML processing

flask Used for the API

neo4j-driver Used to connect to the Neo4j database via Python

nltk Used to process textual documents

pyahocorasick Used to find co-occurrences within text

pytest Used for testing the application

pytest-cov Used to be able to see lines that have been covered in tests

PyYAML Used for system configuration

requests Used to connect to Common Crawl

scikit-learn Used for the classifier

testfixtures Used for (complex) test setup/teardown

E.2 Libraries Used in the Front-End

For the front-end, we used the libraries listed below, which can all be installed through `npm`. The links they can be found at are constructed by substituting "<package>" in the following URL: `https://www.npmjs.com/package/<package>`.

gulp Used for creating tasks

gulp-concat Used to concat JavaScript, CSS and HTML files

gulp-nunjucks Used for pre-compiling HTML templates

gulp-rename Used for renaming concatenated files

mocha Used for testing the application

nodemon Used as a development server

rimraf Used for cleaning up before a new build

uglifyjs-webpack-plugin Used to make scripts unreadable by end users

webpack Bundles JavaScript files into a single file

express Provides routing functionality for NodeJS applications

nouislider Provides JavaScript and styling for sliders

nunjucks Provides a HTML templating system for dynamic HTML

watch-object Used for observer-like functionality for controlling visibility of map objects