Delft University of Technology

Bachelor Graduation Project

Final Report

UrbanSearch

Authors:
Tom Brunner
Marko Mališ
Gijs Reichert
Piet van Agtmaal

Supervisor:
Claudia Hauff
Clients:
Evert Meijers
Antoine Peris

June 22, 2017

TU Delft

Preface

blah

Summary

Instead of abstract, add a single page summary

It is hard if not impossible to measure the strength of relationships between cities using existing technologies. Because of this, it remains uncertain how exactly economic growth is affected by urbanisation. The universally accepted explanation is that only increasing size of cities affects economic growth [**?**]. In this report, we develop a methodology that allows for determining intercity relationship strengths, using open data. For this, we evaluate whether graph databases like Neo4j [**?**] or document search engines such as ElasticSearch [**?**] are best suited and describe machine learning algorithms for categorising data based on the co-occurrence of city names. Additionally, we present visualisation techniques to be able to intuitively analyse the results.

Contents

1

Introduction

The huge amount of textual data generated online and the numerous historic archives are great sources of information on social and economic behaviours. Advanced text mining on newspapers and web pages containing city names would allow for a better understanding of the role of information in shaping urban systems. Similar to research efforts in other domains, such as financial trade [**?**] and sales forecasting [**?**], the idea is to develop search queries that capture urban-urban interactions. These interactions are retrieved from information corpora through the co-occurrence of geographical names in textual data. An example of such a query on the Google search engine[1] is `"Rotterdam + Amsterdam " OR "Amsterdam + Rotterdam"`, which searches for the co-occurrence of Amsterdam and Rotterdam. However, manually processing all results a search engine yields is not feasible. Thus, we will answer the following question: how can the strength of relationships between cities be extracted and visualised from open data?

Aanpassen nadat content van sections klaar is

First, we discuss related work in section 2. Second, we identify the requirements for a solution to the problem and discuss issues that might arise in section 3. Third, we develop a methodology for a framework that satisfies the requirements and tackles the issues in section 4. We conclude in section 5 with the results of our research.

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

2

Related Work

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

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

When looking at digital data there are two different approaches for determining intercity relationships: the cyberspace and the cyberplace [**?**]. 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 [**? ?**].

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[**? ? ?**]. 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 [**?**]. However, this method is very limited since you would have to manually enter a new query for each pair of cities for each relation.

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

3

Problem Definition and Analysis

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

3.1 Problem Definition

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

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

3.2 Problem Analysis

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

Filtering

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

Classification

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

Storing Data

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

Data Visualisation & Export

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

3.3 Requirement Analysis

In this section, we first present user stories that were created together with the client. Next, we define the design goals. Then, we list the requirements which followed from the user stories and which the application should meet. To do so, we use the MoSCoW method[**?**] 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 **??**. The design goals are listed below, ordered by priority.

credible

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

understandable

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

scalable

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

plugable

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

exportable

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

fast development

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

3.3.3 Product Requirements

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

Must Have

Requirements labelled as must have are key to the minimal performance of the application. If they are not met, the application can be considered a failure.

1. Data that is of relevance for the UrbanSearch project, should be mined from the Common Crawl web corpus (see section**??**) and stored for further processing/access.
2. There has to be a way to export the relations between cities.
3. A machine learning algorithm should analyse and label the collected data to extract different types of relations that are important for intercity relations.
4. A front-end should be built for the project. This front-end should visualise basic relations and statistics and can be used for presentations and educational purposes.
5. Several statistically important aspects of intercity relations should be extracted from the data set. These statistics should be easily accessible and visualised to the end user. Furthermore, it should be easy to extend or update the list of statistics that are associated with a relation.

Should Have

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

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

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

Could Have

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

1. The system should use Delpher (see also section **??**), 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 **??**.
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 **??**.

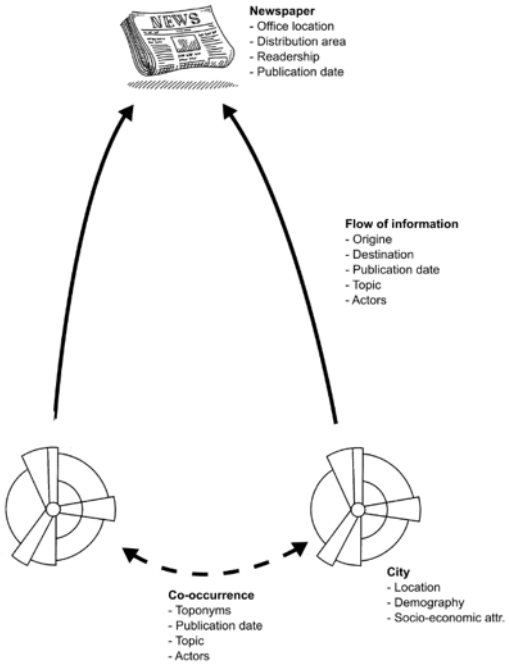


Figure 3.1: Solid lines represent simple information flows, whereas the dashed line is a complex connection of information. We focus on the part depicted by the dashed line.

4

Framework and Tools

split in 04 and 05 and include code quality and validation

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

Marko's stukken aankondigen

4.1 High-level overview

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

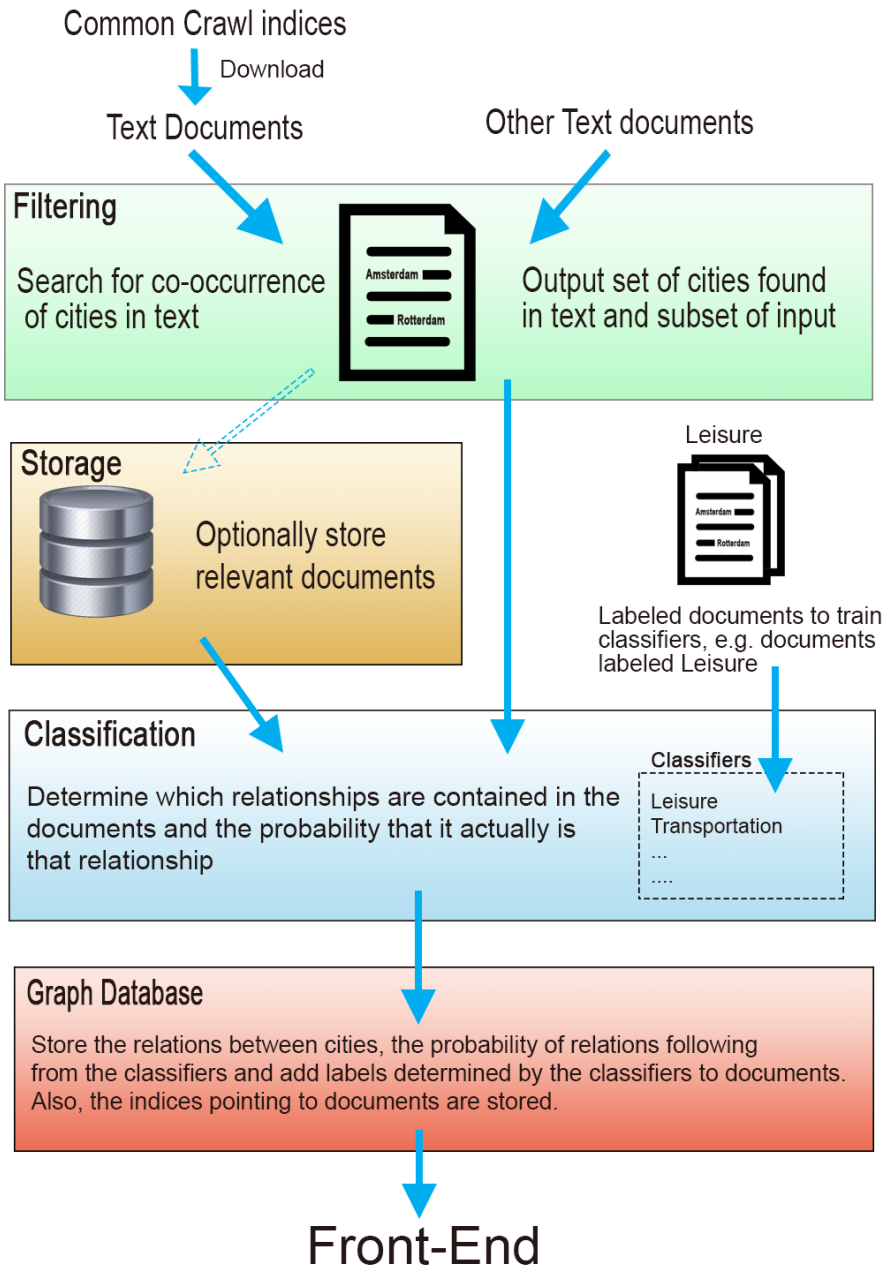


Figure 4.1: High-level overview of the system

4.2 Gathering the Data

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

4.2.1 Common Crawl

Common Crawl [**?**] 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 [**? ? ?**]. Since the project requires analysis on a very large set of documents, the corpus is a very suitable candidate for us to work with.

The data from Common Crawl comes in three formats[1]:

WARC This is the default and most verbose format. It stores the HTTP-response, information about the request and meta-data on the crawl process itself. The content is stored as HTML-content.

WAT Files of this type contain meta-data, such as link addresses, about the WARC-records. This meta-data is computed for each of the three types of records (meta-data, request, and response). The textual content of the page is not present in this format.

WET This format only contains extracted plain text. No HTML-tags are present in this text. For our purposes, this is the most useful format.

Common Crawl stores these pages in the following way: each archive is split into many segments, with each segment representing a directory. Every directory contains a document listing file and a folder for each file format (WARC, WAT and WET), which in turn contains the compressed pages belonging to the segment. To be able to efficiently get a single page, Common Crawl indexes the segments to directly map URLs to document locations using an offset and length which can be found using the Common Crawl index[2]. Since WAT- and WET-files can be generated from WARC-files, they only provide such indices for WARC-files. If no file index is provided with a data request, an aggregated compressed file of all files of the requested format is returned.

For extracting data from Common Crawl, many open-source libraries are available. Common Crawl's official website refers to `cdx-index-client`[3] as a command line interface to their data indices. It allows for, among others, specifying which data set to use, supports multiple output formats (plain text, gzip or JSON) and can run in parallel. Since this library only retrieves the file indices, we need another way to actually retrieve the pages pointed to. However, there is a problem with this: we are only interested in WET-files, but Common Crawl does not have WET-files indexed. We would therefore have to collect the WARC-files and convert them to WET-files ourselves, requiring us to parse HTML for every document we are interested in.

A simple query on the latest 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. It is very important to note that searching for a top level domain like `.nl` only includes the first page of every matching domain. To get all pages, additional queries for each site with more than one page are to be performed.

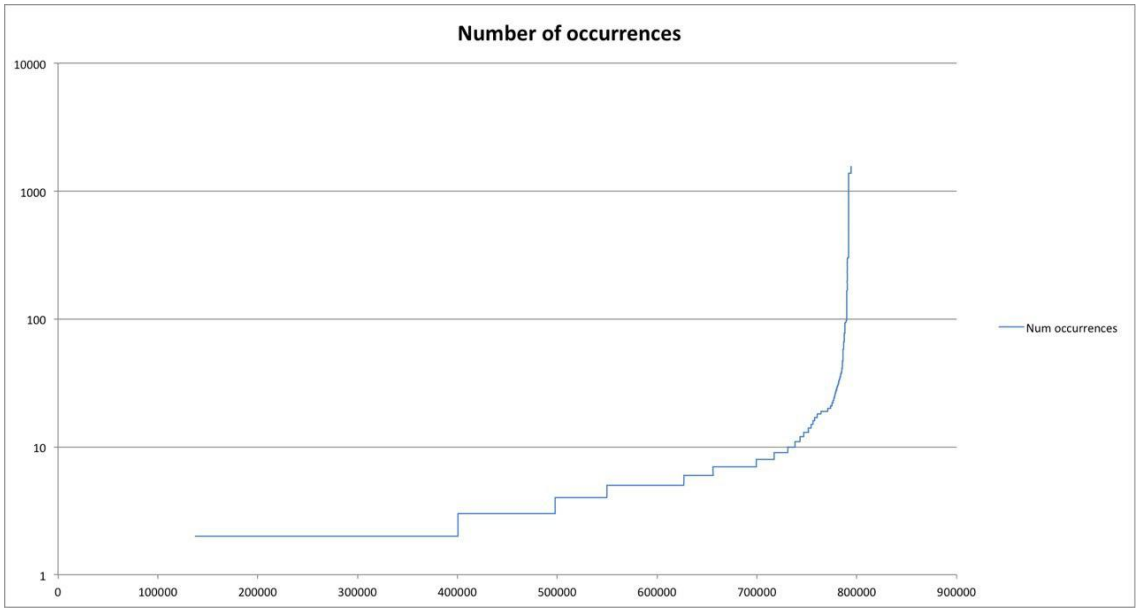
Image or description of a Common Crawl Index.

4.2.2 Other Data Sources

Besides Common Crawl, there are a plethora of other sources that might contain valuable information. The most notable is the Dutch royal library, Delpher[5]. It contains millions of Dutch digitalised newspapers, books and magazines from the fifteenth century up until about 1995. Because of this, it is a useful resource for historical research. Additionally, Statistics Netherlands[6] is the governmental organisation collecting statistical data about the Netherlands and comes with 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.

[1] `https://gist.github.com/Smerity/e750f0ef0ab9aa366558`
[2] `http://index.commoncrawl.org`
[3] `https://github.com/ikreymer/cdx-index-client`
[4] `http://index.commoncrawl.org/CC-MAIN-2017-13-index?url=*.nl&output=json&showNumPages=true`
[5] `http://delpher.nl`
[6] `https://www.cbs.nl/en-gb`
[7] `http://corpus.byu.edu/now/`



4.3 Selecting Relevant Documents

Because not all data from information sources such as Common Crawl is relevant, the data needs to be filtered. One way to do this, is to only select the data that mentions at least two different cities. Because the data is plain text, we need a way to scan through the text and determine if the text indeed has a co-occurrence of two different cities. Making use of the comparative analysis of Rasool et al. [**?**], we chose the Aho-Corasick algorithm [**?**], which is a multi-pattern exact string matching algorithm. Using this algorithm, a predefined list of cities can be matched against the text of a web page or document. If at least two cities from the list appear in the text, we mark it as a useful document.

The decision to use the Aho-Corasick algorithm is strengthened by the fact that a well documented and stable Python library exists, which implements the aforementioned algorithm. This library is called `pyahocorasick`[8] and is a fast and memory efficient implementation of the Aho-Corasick algorithm.

We make a selection of documents without storing the documents first, because storing and indexing all documents is not feasible due to storage constraints. As we do not have access to a fast and large data storage platform, we will not store and index everything first and then delete irrelevant documents.

4.4 Extracting Relations from Documents

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

Clustering, is sorting these websites into groups based on how much they are alike. It does however not assign these groups with a label (or class). this can be a large problem because in one group may be several topics that have similarities with some of the other topics within the

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

group but not with all other topics in the group, which makes it very hard to assign a label to these groups. Although it is possible to label these groups manually, since it is quit tedious and we are on a limited time schedule, it is better to use a classification algorithm.

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

4.4.1 defining classes

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

4.4.2 preprocess

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

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

2. Removing stop words.
   Removing all common words (the, a, an etc) and symbols ('.', ',', '!', etc). For removing stopwords we used a list from NLTK containing dutch stopwords. For removing symbols we made a method that removes all non alphabet values except spaces from text.

3. Stemming.
   Reducing derived words to their stem (e.g. fishing -> fish). We first wanted to use NLTK, which itself implements several stemmers including SnowBallStemmer [**?**] and Porter [**?**]. However research from the Rijks Universiteit Groningen (RUG) by Tanja Gaustad and Gosse Bouma [**?**] evaluated a Bayesian text classification system with either no stemming or the Porter or dictionary based stemmer. Which concludeded stemming does not lead to significant change in classification accuracy.

4. Modelling (TF-IDF).
   Before we can enter this tokenized data into the machine learning algorithm we first need to transform it to the correct model. Gensim provides two ready to use method for this to change tokenized text into the proper input data. The first is token2ID which gives each unique word a numeric ID and counts the amount that those words occur in a text and the second is tfidf which changes the amount that words occur in tf-idf values. tf-idf means term frequence inverse documet frequency. Term frequency is the number of times the word

occurs in a document and idf is the number of documents devided by the number of documents where the term occurs. These two values are then multiplied. The results are normalized to make up for different document lenghts [https://radimrehurek.com/gensim/models/tfidfmodel.htr

4.4.3 training

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

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

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

4.4.4 classifying

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

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

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

4.5 Storing and Ingesting the Data

In this subsection 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 Graph Database, Search Engine or Traditional Database

Now that relationships have been extracted from the documents and web pages, we need to store these. Note that is not necessary to store all the documents, because the relationships which have been extracted are the information that is actually worth storing. Therefore, the original document can be discarded. To store these relationships there are two possibilities: (1) graph databases and (2) traditional relational databases.

Because visualisation of the network of cities as a graph is an essential part of the application, and relations between cities play a key role in the system, we need a database that is designed for these features. Relations are the most important in the graph data model, where this is not true for traditional relational databases. Therefore, we are confident that a graph database is the best choice.

4.5.2 Comparing Graph Databases

Next, the type of database needs to be selected. For this, six of the most popular databases are rated on five important aspects. These are, is the graph database open-source, scalable and free and does it support Python and have built-in visualisation.

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

[a] Only free up to 5 million triples
[b] With separate tool called Gruff: `https://allegrograph.com/gruff2/`
[c] Non-commercial use
[d] Using a separate tool Aster AppCenter
[e] Using a separate tool

From this table, it can be deduced that three of these graph databases are viable candidates: ArangoDB, Neo4j and OrientDB. For this project, Neo4j is the best choice because of three reasons. Firstly because we have experience with Neo4j, which means less time will be spent on getting to know the graph database and functionality. Secondly because is by far the most popular graph database [9]. Thirdly, since Neo4j is the most popular graph database, the support community and amount of available examples is large.

4.5.3 Using Neo4j for Storage and Ingestion

Neo4j is a highly scalable native graph database that leverages data relationships as first-class entities [**?**], enabling enterprises of any size to connect their data and use the relationships to improve their businesses. It is the single highly scalable, fast and ACID compliant graph database available. ACID stands for the four properties atomicity, consistency, isolation and durability of transactions in database systems that ensure reliability for query results [**?**]. 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[10] use it in their mission-critical systems. Holzschuher and Peinl compared the performance of Neo4j to the more classic and commonly used NoSQL databases and found that the more natural representation of relationships resulted in significant performance increase gains [**?**].

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

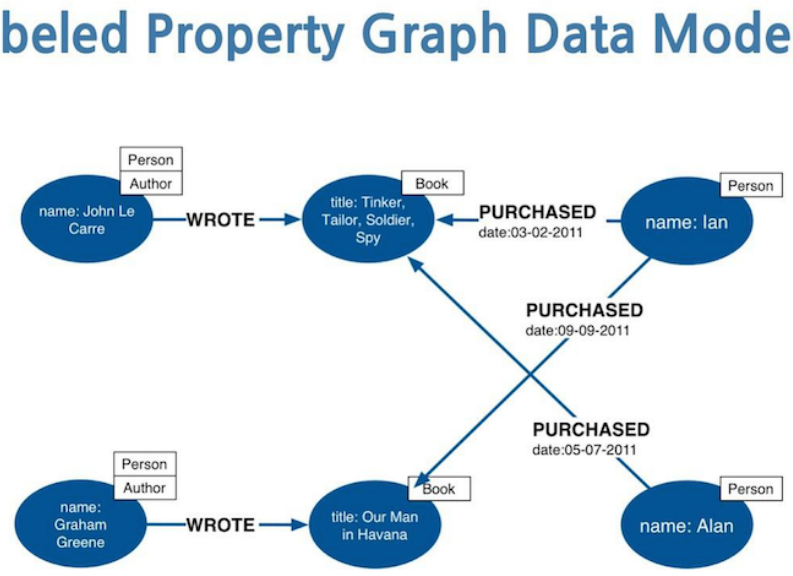


Figure 4.2: The Neo4j model

There are some specific aspects of Neo4j that make it a very suitable candidate for this application. These are:

properties Any entity in the Neo4j graph can be given properties (key-value pairs) containing information about the entity. Properties are primarily meant to provide additional information and are less suitable to be queried on. As an example, a city can have a number of inhabitants and districts attached to it as a property.

labels Nodes can be tagged with a label, describing their roles in the network. These annotations are especially useful to filter the data set on one or more categories. For example, a city can be labelled as ”capital” to be able to distinguish between regular and capital cities.

relations Nodes can be connected using relationships. These are always directed, typed and named and can have properties. Using these properties, one can control how the graph is traversed. For example, if a path (relationship) is to be avoided unless absolutely necessary, the relation can be given a high cost. To give importance to some relationship, one could also assign a strength score to it. Since relationships are handled efficiently by Neo4j, nodes can have any number of relationships linked to it without compromising performance. For our purpose, a relation could comprise the strength of the relationship between two cities (nodes).

The Neo4j model can be depicted as shown in figure **??**. It consists of nodes, relationships (edges), properties (within the nodes) and labels (rectangular blocks above the nodes).

Besides the aforementioned useful properties of Neo4j, the graph can be put to good use for visualising the global urban network. By adding a location property to a city, nodes and relations can be mapped directly to a geographical map. Most importantly, indices of text files can be stored that mention the city as properties of nodes. That way, we are able to generate a subset of files that can be analysed for calculating the strength of the relationship between the nodes.

4.6 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 subsection, several ways to do so are compared, after which we decide which path to take.

4.7 Query Language

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

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

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

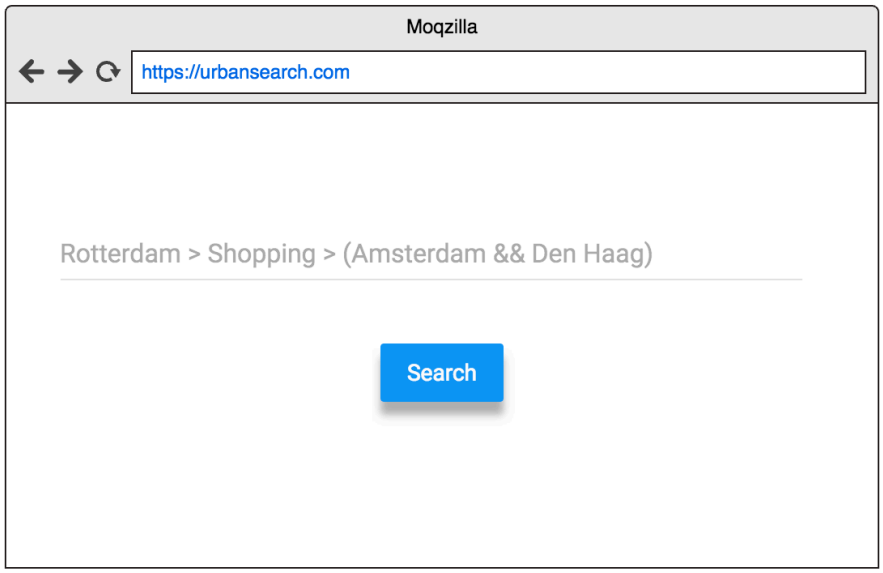


Figure 4.3: Example interface for the query language

4.8 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 **??**.

4.9 Interactive Search

The last option we investigated is an interactive approach to querying data. For this, the client interacts with a map containing relations and cities. A very simple example is given in figure **??**.

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.

4.10 Conclusion

In association with the client, we conclude that the best option to go with is the interactive map. This way, the client has easy access to the data and this pattern of interaction best suits the work

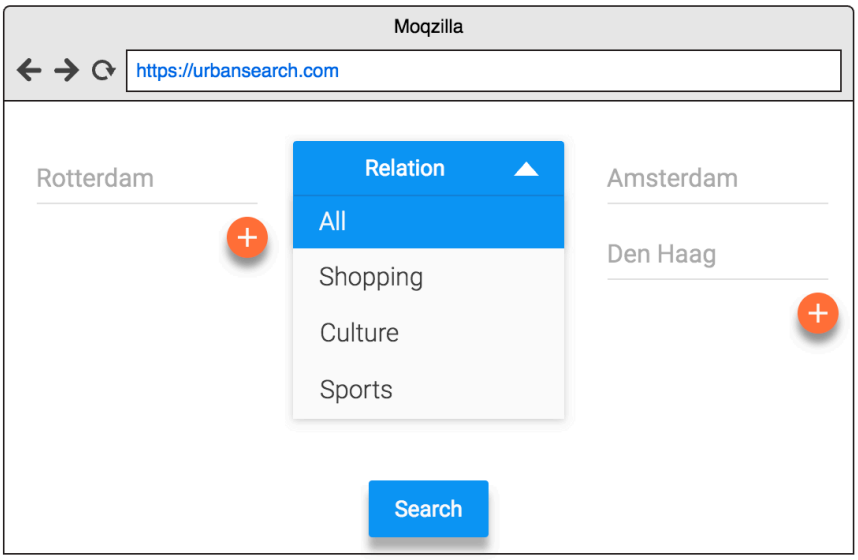


Figure 4.4: Example interface for the query composer

flow that the client envisioned prior to the project. The creation of complex queries is also made a lot easier. The user does not have to write or compose a complex query in advance but can do it directly on the map. Thus, retrieving a visual representation of several cities, interconnected with multiple relations, only involves selecting cities and relations on the map. Interaction directly with the map also reduces the need to go to a separate page to compose a query. This speeds up the use of the system by reducing page loads and it interrupts the work flow of the user less.

4.10.1 Visualising the Data

This subsection focuses on the visual representation of the processed data. This should be done in way that is easy to comprehend for users and helps them to interpret the data. To reach these goals, we identified the clients needs and desires. We have discussed the preferences of the client and have drawn up a global plan, which we present below.

Representing the Data Graphically

Since we are dealing with strongly related data, it is a natural choice to represent the data as a graph. The choice was made, in association with the client, to show the nodes and relations on a geographical map. Visualising cities on a map is intuitive to the user and we think this will increase the ability of users to interpret the information in a productive manner.

Using Geographical Maps

We investigated two map libraries we can use to display our data on a map. The first one is Google Maps, which can be used freely and offers a lot of customisation options. The API is well defined and some of the group members have previously worked with it. The second option we investigated is Leaflet. Leaflet is an open-source JavaScript library that provides responsive maps. It also has fine grained API and lots of plugins available. Both libraries are well suited for our needs. However, we decided to go with Google Maps, because of the existing experience of the group members. Also, we feel Google Maps is better supported through documentation and communities.

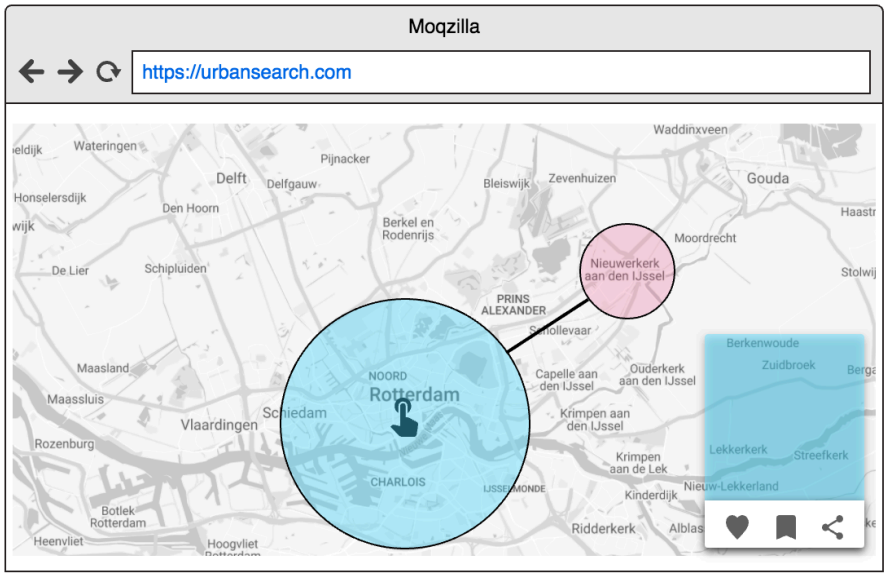


Figure 4.5: Example of an interactive map

Handling Map Clutter

One of the challenges of visualising networks, as stated in [**?**], is the occurrence of so-called map clutter. Map clutter means the network is displayed as an incomprehensible set of nodes and edges. Several methods to prevent this are given in [**?**]. We will adopt some of these methods in our application, as explained next.

Users should be able to select what information they want to display. This will be included in the system by allowing the user to select cities and relations, enabling them to filter nodes and edges. The use of different sizes for nodes and edges or other attributes that are displayed can convey extra information to the user. We will use this to represent, for example, city population and exact strengths of relations. The use of colour is another method mentioned in [**?**]. We will use colours to represent different types of relations and utilise colour intensity and opacity to represent the strengths of these different types of relations.

4.11 API

Verplaatsen naar 5?

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

4.11.1 General Remarks

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

4.11.2 Classify Route: /classify

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

subroutes are specified below.

/

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

Request:

Method	POST
Content-Type	application/json

Request data:

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

Response:

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

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

Request:

Method	POST
Content-Type	application/json

Request data:

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

Response:

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

4.11.3 Data-set Route: /datasets

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

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

Request:

Method	POST
Content-Type	application/json

Request data:

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

Response:

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

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

Request:

Method	POST
Content-Type	application/json

Request data:

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

Response:

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

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

Request:

Method	GET

Response:

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

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

Request:

Method	POST
Content-Type	application/json

Request data:

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

Response:

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

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

Request:

Method	POST
Content-Type	application/json

Response:

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

/lengths Returns the lengths of the different category-sets

Request:

Method	GET

Response:

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

4.11.4 Documents Route: /documents

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

/ Gets an random document from the downloaded CommonCrawl pages.

Request:

Method	GET

Response:

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

4.11.5 Indices Route: /indices

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

/

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

Request:

Method	GET

Response:

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

4.11.6 Classify Documents Route: /classify$_{d}ocuments$

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

4.11.7 Front-End

Verplaatsen naar 5.5?

5

Implementation

5.1 Downloading and parsing indices

As can be seen in figure **??**, the first step of the process is to download data from Common Crawl. This required functions that would parse the Common Crawl indices and gather the data that corresponds to these indices. The first step in parsing the Common Crawl indices and filtering out the indices that have a HTTP Status Code [1] other than 200, as only these indices with these HTTP Status Code would be useful.

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

Listing 5.1: Regular expression solution

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

.5

5.2 Filtering

5.3 Classification

5.4 Storing the Data

mention multithreading neo4j (embedded)

5.5 Front-End

[1] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

6

Project Evaluation

Put results in appendix F

write conclusion about results

6.1 Validation and Verification

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

6.1.1 Testing the Application

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

Unit Tests

Unit testing is done by writing automatic tests and making sure they pass every time the tests are executed. Unit tests test each method of a function separately, checking that the method does what it is supposed to do. If the method would need information from outside the class that information is mocked. This means that instead of using that other class, a fake object is made which returns a fake value. This ensures the tests will never fail due to changes in other classes.

Integration Tests

Integration testing uses automated tests which test how well different components of the system work together. This is done more or less the same as unit testing, however whilst you would mock methods from other classes in unit testing, with integration testing you do not. It is assumed that the separate modules are unit tested, therefor if an error occurs it is because something is wrong with the interaction between the modules and not with the modules themselves.

System Tests

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

Acceptance Tests

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

6.1.2 SIG

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

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 [**?**]. According to the page binary classification can be evaluated with the following metrics: Accuracy, Precision, Recall, F1 and AUC.

Accuracy

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

Confusion Matrix

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

Precision, Recall, F1 and UAC

The **precision** of the classifier is the proportion of positives that are classified correctly: 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: TP/(TP+FN).

The **F1 Score** uses both precision and consideration. It is computed by using the following formula: F1 = 2 (precision x recall) / (precision + recall). The F1 score summarised evaluation in

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

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

Figure 6.1: confusion matrix [1]

a single number, but for evaluation it is better to use recall and precision to understand the behaviour of the classifier.

The **Receiver Operating Characteristic (ROC) curve** and the corresponding **Area Under the Curve (AUC) value** can be used to inspect the true positive rate (Recall) vs. the false positive rate (FP / (FP + TN). To do this, the possibilities pages are correctly classified are needed. For each threshold on these probabilities for the classifier, the true positive rate and the false positive rate are calculated and are plotted in a graph, which results in something like **??**. 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.

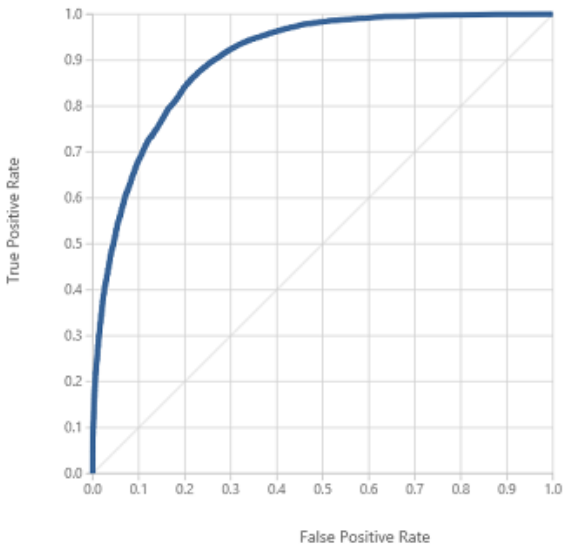


Figure 6.2: ROC / UAC graph [2]

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 [**?**], so one would expect cities that lie close to each other are more related than cities that are on different sides of the country. This natural relation can be represented using the Gravity Model by Reilly [**?**]. The Gravity Model describes that the expected relation between two cities is based on the population of the two cities and the distance

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

between these cities. A relation between two cities that is extracted from the data should thus expose a similar relative score as they would for the gravity model. Consider for example Amsterdam and Hoofddorp, which are cities that lie close to each other. Amsterdam is a large city, whereas Hoofddorp is much smaller. However, due to their close geographical position, the score that results from the Gravity Model would be high. If they turn out to have a very high score in our system, that would imply that the system is correct. Besides the Gravity Model, one can rely on the opinion of an expert in the field of urbanism that can judge whether an extracted relation is close to reality or not. We therefore agreed with the client that they would decide on a small set of relations whether they are correct. Lastly, the relations in the Randstad, a large urban area with the four largest cities of the Netherlands, have been examined before on the basis of firms [**?**]. These relations can be compared to those extracted by the system.

6.2 Fulfilment of Requirements

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

more about not mentioned pass/fail, name reqs?

Table 6.1: Requirements pass fail

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

6.3 Evaluation of Design Goals

6.4 Process

6.4.1 Collaboration between the team members

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

6.4.2 Collaboration between the team members and the TU Delft Coach

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

Claudia absent twice

6.4.3 Collaboration between the team members and the client

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

6.5 ethics

7

Discussion

8

Recommendations

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

extend

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

9

Conclusion

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

First, we discussed related work. We saw that there are currently two methods for analysing the relations between cities. Manually analysing search engine data is very slow and requires a lot of man-hours and looking at the different locations where businesses are located is only interesting for the economic relation and still misses a lot of data.

Second, we identified the requirements for a solution to the problem and discuss issues that might arise. The used the MoSCoW model to describe the importance of the different requirements. The most import must haves we found are being able to input place names, displaying a map with the connection data and being able to extract this data.

Third, we developed a methodology for a framework that satisfies the requirements and tackles the issues. We decided to start by using data from Common Crawl, although we might later extend this to other data sources such as Delpher. After selecting relevant data (data which contains 2 or more city names) we store the data with Neo4j. We then use clustering and classifying machine learning to group the data. First we use this on all data to get the general groups (e.g. economy, health-care, immigration) and then we use this on the data per pair of cities to see what the important connection types for each city are. Then we link these connections to the general groups (’fish’ might relate to economy.. etc). To visualise this data we use the graph Neo4j provides.

new stuff

Bibliography

[] 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.

[] 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.

[] 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.

[] 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.

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

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

[] 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.

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

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

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

[] 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.

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

[] 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.

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

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

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

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

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

[] 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.

[] 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.

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

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

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

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

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

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

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

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

[] 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.

[] 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.

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

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

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

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

[] 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.

[] 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.

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

A

Better Code Hub Guidelines

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

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

B

Sig Feedback

B.1 week 5

[Analyse]

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

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

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

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

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

B.2 week 9

C

User Manual

D

Developers Manual

E

Used Libararies