TODO: subsections: 3.1 problem definition 3.2 problem analysis 3.3 requirement analysis (with design goals, requirements and minimal viable product)

0.1. Requirement Analysis

In this section, we first define the design goals. Next, we present user stories that were created together with the client. Then, we list the requirements which followed from the user stories and which the application should meet. To do so, we use the MoSCoW method

0.1.1. Design Goals

The high-level design goals for this project have been provided by the client. These serve as a guideline to determine the priority label of the specific requirements as defined in section 0.1.3. The design goals are listed below, ordered by priority.

credible The results of the project will be used to dispute a widely spread belief. Therefore, the results must be reliable and verifiable.

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

scalable Restricting the project to a set of non-English domains might impair the probability that the results are generally accepted. Allowing for investigating other domains would greatly help the client in a later stadium.

plugable It might be interesting for the user to let the application perform analysis on different data sets without the need of a developer.

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

0.1.2. User Stories

Together with the client, several user stories are identified and listed below:

1. As a user, I want to be able to see all the identified relations between all cities, so that I can reason about interesting patterns.
2. As a user, I want to be able to access the processed data in an excel file. I want this to be available per relation type and as a total of all relations, so that I can apply my own models on the data when I want to.
3. As a user, I want to be able to switch between absolute and relative relation strengths, so that I can interpret the data in a more complete way.
4. As a user, I want to be able to (de)select cities, so that I can create a network of cities connected with relations.
5. As a user, I want to be able to (de)select relations between cities, so that I can inspect only the relations I am interested in at that time.
6. As a user, I want to be able to change the colours associated with the different relation types, so that I can adjust the styling to my own preferences.
7. As a user, I want to be able to export an image of the map that I composed, so that I can use it for presentations, papers or educational purposes

0.1.3. Product Requirements

As mentioned in the intro of section 0.1 we will be using he MoSCoW method prioritisation technique. Four levels of priority are defined: must have, should have, could have and won't have (also known as would like). We also differentiate between functional and non-functional requirements.

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 section0.2.1) and stored for further processing/access.
2. There has to be a way to query the collected data based on relations between cities/nodes.
3. A machine learning algorithm should analyse the collected data and attempt to identify different types of relations that are important for intercity relations.
4. A front-end should be built for the project. This front-end should visualise basic relations and statistics and can be used for presentations and educational purposes.
5. Several statistically important aspects of intercity relations should be extracted from the data set. These statistics should be easily accessible and visualised to the end user. Furthermore, it should be easy to extend or update the list of statistics that are associated with a relation.

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

1. Relations between cities should be accessible hierarchically. This means that there is the possibility to explore a relation and, provided that this relation has sub-types associated with it, the relation can be expanded in the different sub-types of the relation.
2. A machine learning algorithm should be able to group different relationships that are strongly connected to each other.
3. It should be possible to add big data sets on which the system can perform its data mining routines. This way a data set can be created that contains potentially interesting information for intercity relations.
4. The application should understand that the same city can have different names in different languages/dialects. It should still be able to extract and group relevant data correctly (e.g. 'The Hague' and 'Den Haag' should be viewed as the same city).

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

1. The system should use Delpher to characterise relationships between a region and cities outside that region based on newspapers in those regions, aggregated over the past 50 years. These relationships are either simple or complex information flows. Simple information flows consist of a newspaper located in city i publishing about city j, whereas complex information flows are co-occurrences of cities in an article.
2. The relations that are extracted from the data by the machine learning algorithm have to be visualised in a way that makes it easy to compare the different relations for the end user.

Would Like
"Would like" requirements have been agreed upon to be not important to include within the current time schedule. However, they can be included in future releases.

1. The application would be able to show all connections of all places on the map at the same time.
2. Using data from top-level domains other than `.nl`.
3. Evaluating places of residency with less than 750 inhabitants.

0.1.4. Design Decisions

To be able to have a fast development cycle and leverage our experience we chose to develop the application using Python. Since the validity of our results is very important for the results to be considered credible, we need to do thorough validation of the obtained results. We plan to not only test the code we deliver thoroughly, but also to cross-validate the obtained results. TODO: split in 04 and 05 and include code quality and validation

0.2. Framework and Tools

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

0.2.1. Gathering the Data

As explained in section 0.1.1, data sources should be plugable. An initial corpus of documents is needed to base the project, which we will decide on in this section.

Common Crawl
Common Crawl

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.

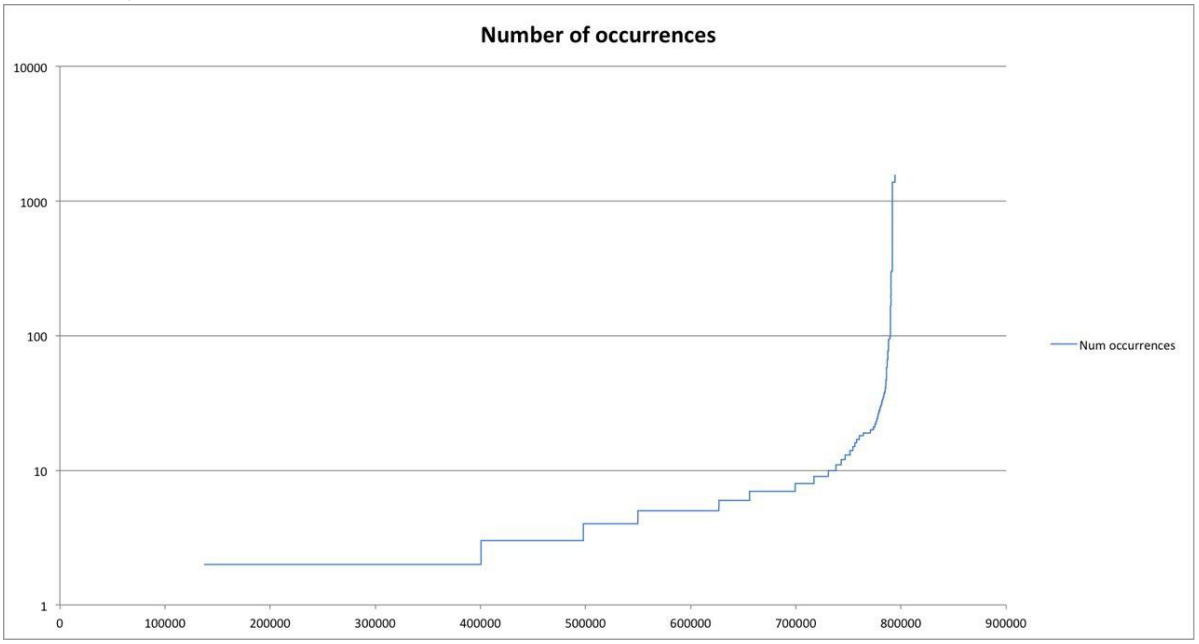
For extracting data from Common Crawl, many open-source libraries are available. Common Crawl's official website refers to `cdx-index-client`[3] as a command line interface to their data indices. It allows for, among others, specifying which data set to use, supports multiple output formats (plain text, 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.

[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`

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.



0.3. framework implementation

0.4. project evaluation

TODO:

0.5. discussion

0.6. recommendation

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

0.7. Conclusion

TODO: 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

0.7.1. related work

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.

[5] http://delpher.nl
[6] https://www.cbs.nl/en-gb
[7] http://corpus.byu.edu/now/

0.7.2. related work

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.

0.7.3. Requirement analysis

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.

0.7.4. Framework-and-tools

.1. Sig Feedback

.1.1. 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.

.1.2. week 9

.2. User manual
.3. dev-manual
.4. Used Libararies

Bibliography