



universität  
wien

# MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Geospatial Information Retrieval for POIs with the use of  
a Data Mining System“

verfasst von / submitted by

Alexander Czech BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of  
Master of Science (MSc)

Wien, 2015 / Vienna 2015

Studienkennzahl lt. Studienblatt /  
degree programme code as it appears on  
the student record sheet:

A 066 856

Studienrichtung lt. Studienblatt /  
degree programme as it appears on  
the student record sheet:

Masterstudium  
Kartographie und Geoinformation UG2002

Betreut von / Supervisor:

Ass.-Prof. Mag. Dr. Andreas Riedl

Mitbetreut von / Co-Supervisor:

-

## Table of Contents

Table of Contents.....	ii
Table of Figures.....	vi
Table of Maps.....	viii
Table of Tables .....	ix
List of Abbreviations .....	x
Abstract.....	xi
Kurzfassung .....	xi
1. Introduction .....	1
1.1. Aim of the Thesis.....	1
1.2. Research question.....	2
1.3. Structure of Research design .....	3
2. Introduction to Data Mining and Big Data.....	4
2.1. Data Mining.....	4
2.2. Big Data .....	6
3. The Common Crawl Dataset .....	8
3.1. What is Web Crawling.....	8
3.2 The Common Crawl Dataset .....	9
3.3. The Common Crawl Dataset Index.....	11
3.4 Downloading the Common Crawl Dataset.....	14
4. OpenStreetMap Geocoder.....	19
4.1. OpenStreetMap .....	19
4.2. Data Structure and Source .....	19
4.3. Extracting Addresses.....	21
4.4. Write Addresses to a Database.....	27
5. Common Crawl Database Transfer .....	31

5.1. Folder and file structure .....	31
5.2. Subdivide files into individual HTML files .....	33
6. HTML Tag Stripper .....	37
6.1. Find Vienna .....	37
6.2. Remove HTML Tags.....	38
7. Geotagging .....	42
7.1. Creating an index in PostgreSQL.....	42
7.1.1. Converting a text to a list of stemmed tokens.....	42
7.1.2. Creating a Token Index .....	45
7.2. Create a unique set of Addresses and prepare them for Search Queries .....	46
7.3. Preparing the SQL Statement for Geotagging .....	48
7.4. Joining Addresses with HTML Documents.....	52
7.5. Discussion.....	55
8. Finding Links.....	57
8.1. Preparation .....	57
8.1. Link Extraction.....	58
8.2. Geotagging the found linked websites .....	61
8.3. Discussion.....	62
9. The Vector Space Model .....	65
9.1. The Document Vector .....	65
9.2. Term frequency Inverse Document Frequency .....	66
10. Categories for classification .....	69
10.1. Daseinsgrundfunktionen.....	69
10.2. Classes for addresses .....	70
11. Co-occurrence Groups .....	74
11.1. Introduction Natural Language Processing.....	74
11.2. Part-of-speech Tagging .....	74
11.3. POS tagging Wikipedia .....	77

11.4. Co-occurrence .....	80
11.5. Generating Co-occurrence query expansion groups from Wikipedia .....	83
12. Address Classification .....	87
12.1. Creating the Vector Space .....	87
12.1.1. Creating a unique set of HTML documents .....	87
12.1.2. Creating the HTML documents Vector Space .....	89
12.1.3. Creating the Wikipedia Vector Space .....	90
12.1.4. Combined Vector Space .....	92
12.2. Calculating the idf-tf vectors .....	93
12.2.1. Calculating Inverse Document Frequency per Term .....	93
12.2.2. Calculating the Term Frequency-Inverse Document Frequency Vector for HTML Files ...	94
12.2.3. Calculating the tf-idf Vector for Wikipedia Co-Occurrences groups and search terms ...	97
12.2.4. Cosine Similarity Calculations .....	101
12.3. Address Classification .....	103
13. Mapping .....	121
13.1. Selecting and Mapping a Control Group .....	121
13.2. Comparing the Control Group to Vector Classification .....	122
14. Conclusion .....	129
15. References .....	132
15.1. Scientific References .....	132
15.2. Programming Library References and Technical Documentations .....	135
15.3. Websites .....	137
Annex .....	xiii
Source Code .....	xiii
Threading Example .....	xiii
DBconnector .....	xiv
OSM Parser .....	xv
Disassemble HTML .....	xviii



SetVienna .....	xx
Tag Stripper .....	xxii
Geo Tagging .....	xxiv
Find Links.....	xxvii
Wikipedia POS Tagging .....	xxix
Co-occurrence Group Generation.....	xxx
Inverse Document Frequency .....	xxxi
Wikipedia Vector Space .....	xxxii
HTML Vector Space .....	xxxiii
Combined Vector Space .....	xxxiv
HTML Tokenization .....	xxxv
TFIDF Vector HTML Documents .....	xxxvi
TFIDF Vector for Search Term and Co-ccurence Groups .....	xxxvii
Cosine Similarity.....	xxxviii
Address Classification .....	xl
Database Schema.....	xl ii
Mapping Results.....	xl iii
Lebenslauf .....	liii

## Table of Figures

<b>FIGURE 3.1</b> ARC FILE EXAMPLE (ARCHIVE.ORG) .....	10
<b>FIGURE 3.2</b> EXAMPLE BINARY TREE (MANNING ET AL, 2009) .....	12
<b>FIGURE 3.3</b> EXAMPLE B-TREE (MANNING ET AL, 2009).....	13
<b>FIGURE 3.4</b> INDEX CHECK EXAMPLE DERSTANDARD.AT .....	14
<b>FIGURE 3.5</b> INDEX CHECK EXAMPLE .AT TLD .....	14
<b>FIGURE 3.6</b> CODE EXAMPLE URL STUMP CREATION .....	15
<b>FIGURE 3.7</b> THREADING CODE .....	16
<b>FIGURE 3.8</b> STARTING AND CONTROLLING THREADS .....	17
<b>FIGURE 4.1</b> OSM DATA EXAMPLE (OPENSTREETMAP WIKI; OSM XML) .....	20
<b>FIGURE 4.2</b> EXAMPLE ADDRESS TAG (OPENSTREETMAP.ORG).....	21
<b>FIGURE 4.3</b> OPENSTREETMAP XML PARSER START ELEMENT PART ONE.....	22
<b>FIGURE 4.4</b> OPENSTREETMAP XML PARSER START ELEMENT PART TWO .....	23
<b>FIGURE 4.5</b> OPENSTREETMAP XML PARSER START ELEMENT PART THREE .....	24
<b>FIGURE 4.6</b> OPENSTREETMAP XML PARSER END ELEMENT PART ONE .....	24
<b>FIGURE 4.7</b> OPENSTREETMAP XML PARSER END ELEMENT PART TWO .....	25
<b>FIGURE 4.8</b> OPENSTREETMAP XML PARSER END ELEMENT PART THREE.....	26
<b>FIGURE 4.9</b> DATABASE IMPORT AND CALLING DATABASE IMPORT FROM THE PARSER .....	27
<b>FIGURE 5.1</b> SCHEMATIC EXAMPLE FILE STRUCTURE.....	31
<b>FIGURE 5.2</b> DATABASE TRANSFER SCRIPT PART ONE.....	32
<b>FIGURE 5.3</b> DATABASE TRANSFER SCRIPT OPEN_PATHS() METHOD .....	33
<b>FIGURE 5.4</b> TRANSFER SCRIPT DATABASE_EXPORT() METHOD PART ONE.....	33
<b>FIGURE 5.5</b> TRANSFER SCRIPT DATABASE_EXPORT() METHOD PART TWO .....	34
<b>FIGURE 5.6</b> TRANSFER SCRIPT DATABASE_EXPORT() METHOD PART THREE .....	35
<b>FIGURE 5.7</b> TRANSFER SCRIPT DATABASE_EXPORT() METHOD PART FOUR .....	35
<b>FIGURE 5.8</b> WITEMANYTOTABLE() METHOD .....	36
<b>FIGURE 6.1</b> SET VIENNA .....	37
<b>FIGURE 6.2</b> FETCHING HTML DOCUMENTS TO STRIP TAGS .....	39
<b>FIGURE 6.3</b> REGULAR EXPRESSION TAG STRIPPER .....	40
<b>FIGURE 6.4</b> WRITING STRIPPED HTML DOCUMENTS TO THE DATABASE .....	41
<b>FIGURE 7.1</b> INDEX CREATION.....	46
<b>FIGURE 7.2</b> POPULATE TABLE ADDRESSESUNIQUE.....	47
<b>FIGURE 7.3</b> PREPARING ADDRESSES FOR SEARCH QUERIES .....	47
<b>FIGURE 7.4</b> SQL STATEMENT CONSTRUCTION PART ONE .....	49
<b>FIGURE 7.5</b> SQL STATEMENT CONSTRUCTION PART TWO.....	51
<b>FIGURE 7.6</b> PERFORM THE JOIN OF ADDRESSES WITH HTML DOCUMENTS.....	52

<b>FIGURE 8.1</b> CREATING THE URL DICTIONARY AND THE HTML JOINED TO ADDRESSES DICTIONARY.....	57
<b>FIGURE 8.2</b> FINDING LINKS AND CONVERTING THEM.....	59
<b>FIGURE 8.3</b> MALFORMED LINK EXAMPLES.....	60
<b>FIGURE 8.4</b> GEOTAG LINKED WEBSITES .....	61
<b>FIGURE 9.1</b> COSINE SIMILARITY EXAMPLE (MANNING ET AL.; 2009;) .....	66
<b>FIGURE 9.2</b> STOP WORD LIST OF 25 WORDS THAT ARE COMMON IN THE REUTERS CORPUS VOLUME 1 (MANNING ET AL.; 2009) .....	67
<b>FIGURE 11.1</b> CREATING THE FILE LIST .....	78
<b>FIGURE 11.2</b> EXECUTING THE POS TAGGER.....	79
<b>FIGURE 11.3</b> MULTIDIMENSIONAL SCALING OF CO-OCCURRENCE VECTORS. (SOURCE: LUND AND BURGESS 1996) .....	83
<b>FIGURE 11.4</b> CO-OCCURRENCE GENERATION FROM WIKIPEDIA .....	84
<b>FIGURE 12.1</b> HTML DOCUMENT VECTOR SPACE CODE .....	89
<b>FIGURE 12.2</b> WIKIPEDIA VECTOR SPACE CODE.....	91
<b>FIGURE 12.3</b> COMBINE WIKIPEDIA AND HTML FILE VECTOR SPACE .....	92
<b>FIGURE 12.4</b> CODE FOR INVERSE DOCUMENT FREQUENCY CALCULATION .....	93
<b>FIGURE 12.5</b> HTML DOCUMENT TOKENIZATION .....	95
<b>FIGURE 12.6</b> TF-IDF VECTOR FOR DOCUMENTS.....	96
<b>FIGURE 12.7</b> NORMALIZED TF-IDF VECTOR FOR CO-OCCURRENCE GROUPS AND SEARCH TERMS.....	98
<b>FIGURE 12.8</b> COSINE SIMILARITY CALCULATION .....	102
<b>FIGURE 12.9</b> BREAKS CODE EXAMPLE .....	105
<b>FIGURE 12.10 A)-I)</b> VALUE DISTRIBUTION FOR ALL CLASSES AT THE ADDRESSES FOR CO-OCCURRENCE GROUPS AND SEARCH TERMS (N= 6284).....	109
<b>FIGURE 13.1 A)-I)</b> COMPARING CONTROL GROUP TO VECTOR CLASSIFICATION.....	126
<b>FIGURE 13.2 A),B)</b> CORRELATION BETWEEN ORIGINAL COUNT, COMPLETE COUNT AND VECTOR CLASSIFICATION .....	127
<b>FIGURE 14.1</b> CORRECTLY CLASSIFIED HOTEL ADDRESSES COMPARED TO CLASSIFICATION VALUE RANGE .....	131

## Table of Maps

<b>MAP 1.1</b> SEGREGATED ACTIVITY SPACES FOR TWITTER USERS OF THE WEST AND EAST END IN LOUISVILLE KENTUCKY (SHELTON ET AL.; 2015; P.9) .....	2
<b>MAP 4.1</b> EXTRACTED ADDRESSES .....	29
<b>MAP 7.1</b> DISTRIBUTION OF ADDRESSES JOINED TO HTML DOCUMENTS .....	54
<b>MAP 8.1</b> RESULT OF JOINING LINKED HTML DOCUMENTS TO ADDRESSES .....	63
<b>MAP 10.1 A),B)</b> FUNCTIONAL SUBDIVISION OF THE VIENNESE INNER CITY A) GERMAN B) ENGLISH TRANSLATION (FASSMAN AND HATZ; 2002; P. 37) .....	71
<b>MAP 12.1 A) - R)</b> CLASSIFICATION RESULTS FOR EVERY INDIVIDUAL CATEGORY AND METHOD .....	119
<b>MAP 13.1</b> RANDOMLY SELECTED CONTROL GROUP .....	121

## Table of Tables

<b>TABLE 3.1 A), B)</b> REPRESENTATIVENESS OF TLDs IN THE COMMON CRAWL CORPUS 2012 (SPIELGER; 2013) .....	11
<b>TABLE 7.1</b> WEBSITE MATCH FREQUENCY PER ADDRESS.....	55
<b>TABLE 7.2</b> TOP TEN MATCHED ADDRESSES .....	55
<b>TABLE 8.1</b> URLPARSE.URLJOIN() EXAMPLES .....	60
<b>TABLE 8.2</b> ASSOCIATED WEBSITE FREQUENCIES PER ADDRESS .....	64
<b>TABLE 8.3</b> ASSOCIATED WEBSITE FREQUENCIES PER ADDRESS .....	64
<b>TABLE 9.1</b> COLLECTION FREQUENCY (CF) AND DOCUMENT FREQUENCY (DF) DIFFERENT BEHAVIOR (MANNING ET AL.; 2009).....	67
<b>TABLE 9.2</b> EXAMPLES FOR IDF VALUES BASED ON THE REUTERS COLLECTION CONTAINING 806,791 DOCUMENTS (MANNING ET AL.; 2009) .....	68
<b>TABLE 10.1</b> CLASSES AND THEIR CORRESPONDING QUERIES .....	73
<b>TABLE 11.1</b> THE PEEN TREEBANK II POS TAG SET (SANTORINI 1990).....	75
<b>TABLE 11.2</b> FIVE NEAREST NEIGHBORS FOR TARGET WORDS (SOURCE: LUND AND BURGESS 1996) .....	82
<b>TABLE 11.3</b> CO-OCCURRENCE GROUPS WITH TOP 5 TERMS AND THE NUMBER OF THEIR OCCURRENCES .....	86
<b>TABLE 12.1</b> TOKENIZED CO-OCCURRENCE GROUPS WITH TOP 5 TERMS AND THE NUMBER OF THEIR OCCURRENCES .....	99
<b>TABLE 12.2</b> SIMILARITY MATRIX CO-OCCURRENCE GROUPS.....	100
<b>TABLE 12.3</b> SIMILARITY MATRIX SEARCH TERMS .....	101
<b>TABLE 12.4</b> CLASSIFICATION PROBLEM NUMBER ONE .....	104
<b>TABLE 13.1</b> SEARCH TERM VECTOR CO-OCCURRENCE GROUP VECTOR FITNESS COMPARISON .....	128

## List of Abbreviations

<b>CF</b>	-	Corpus Frequency
<b>DF</b>	-	Document Frequency
<b>HTML</b>	-	HyperText Markup Language
<b>IDF</b>	-	Inverse Document Frequency
<b>IDF -TF</b>	-	Inverse Document Frequency - Term Frequency
<b>NLP</b>	-	Natural Language Processing
<b>NLTK</b>	-	Natural Language Tool Kit
<b>OSM</b>	-	OpenStreetMap
<b>POS</b>	-	Part of Speech
<b>SLD</b>	-	Second Level Domain
<b>SQL</b>	-	Structured Query Language
<b>TF</b>	-	Term Frequency
<b>TLD</b>	-	Top Level Domain
<b>URL</b>	-	Uniform Resource Locator
<b>UTF-8</b>	-	(Universal Coded Character Set + Transformation Format—8-bit)
<b>VGI</b>	-	volunteer geographic information
<b>XML</b>	-	Extensible Markup Language

## Abstract

Up to now, most works about “Neogeography” and “Big Geo-Data” focus on using geotagged social media information for analysis. But this thesis argues that also non-geotagged websites have descriptive capabilities that are of interest. For this, a set of 8 million HTML crawled documents is processed. The crawled data is made manageable and transferred into a PostgreSQL database. To geotag the HTML documents, an address dataset is created from OpenStreetMap data. Multiple variations of each address are then searched for within the HTML documents. Documents containing one or more addresses are geotagged with the coordinates of those addresses. Lastly, websites linking to geotagged websites are also associated with those geotags. To limit the scope of the data that needs to be processed, the HTML documents all have a URL that belongs to the .at top-level domain and the addresses stem from the 1<sup>st</sup> to 9<sup>th</sup> and 20<sup>th</sup> districts of Vienna. This processing creates an information landscape.

The second part of the thesis is to explore the analytic capabilities of this information landscape. A classification attempt based on the information is made. For this, the HTML documents are transformed into a vector in the vector space model. For 9 classes, 18 classification vectors are created and compared with cosine similarity to the HTML document vectors. The results are then associated and summarized on an address basis. These summarized results are sorted on an address level in two steps: once into relevant and irrelevant data and a second time based on whether or not they belong to a class. The results of this classification attempt are mixed. While they only achieve about 19 to 25% correct classifications, they clearly prove that the data has an underlying structure referring to the point of interest they are attached to.

## Kurzfassung

Bisher lag der Fokus der Arbeitsfelder “Neogeography” und “Big Geo-Data” auf der Verwendung von geotagged Informationen aus sozialen Medien. Diese Arbeit versucht zu zeigen, dass auch Webseiten, die keinen geotag im bisherigen Sinne besitzen, den Raum beschreibende Eigenschaften besitzen können. Dafür wurden etwa 8 Millionen gecrawlte HTML-Dokumente verarbeitet. Diese rohen gecrawlten Daten sind für Analysen handhabbar gemacht worden und in eine PostgreSQL-Datenbank überführt worden. Um sie mit geotags zu versehen ist ein Adressdatensatz aus OpenStreetMap-Daten erstellt worden. Die HTML-Dokumente sind nach verschiedenen Schreibweisen derselben Adressen aus diesem Datensatz durchsucht worden. Dokumente, die so

einer Adresse oder mehreren Adressen zugeordnet werden konnten, sind mit den Koordinaten dieser Adresse oder Adressen geotagged worden. Um den Umfang der zu verarbeitenden Daten zu begrenzen sind die HTML-Dokumente auf diejenigen beschränkt worden, die eine URL besitzen, die zu dem Top-level Domain-Bereich von .at gehören und die Adressen sind beschränkt auf den 1. bis 9. sowie 20. Gemeindebezirk Wiens. Dies erzeugt eine Informationslandschaft.

Im zweiten Teil der Arbeit geht es darum, die analytischen Möglichkeiten dieser Informationslandschaft zu untersuchen. Dafür sind die HTML-Dokumente in einen Dokumenten-Vektor im Vektor-Raum-Model überführt worden. Für 9 Klassen werden 18 Klassifizierungsvektoren erzeugt und mit Hilfe der Kosinus-Ähnlichkeit werden diese mit den HTML-Dokument-Vektoren verglichen. Die Ergebnisse werden dann den Adressen zugeordnet und zusammengefasst. Die so zusammen gefassten Ergebnisse werden auf Adressenebene in zwei Schritten sortiert. Erstens werden die Daten für jede Klasse und jede Adresse in relevante und nicht relevante Daten unterschieden und ein weiteres Mal nach Zugehörigkeit zu einer Klasse oder nicht. Die Ergebnisse dieser Klassifizierungsmethode sind durchwachsen. Sie erreichen nur zwischen 19 und 25% korrekte Klassifikationen, aber es ist möglich nachzuweisen, dass es eine den Daten zugrunde liegende Struktur gibt, die in Verbindung zu den Adressen steht.



# 1. Introduction

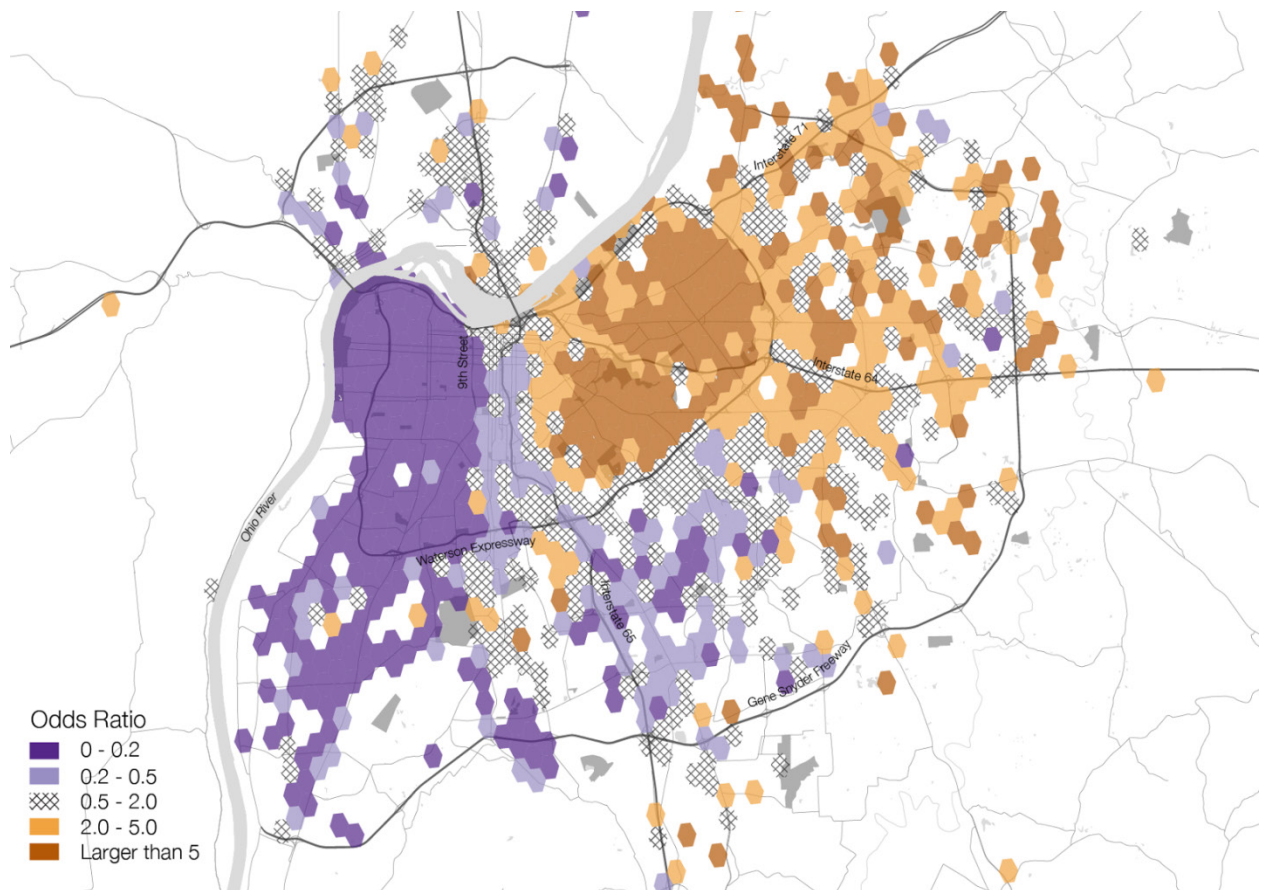
The way people decide where they go when they want to do A or B is increasingly based on information found on the Internet. This can be inquiries like finding a grocery store that is still open or a scenic hiking route. The Internet is a huge body of information and communication that describes all kind of things, but also a space in a geographical sense. This thesis is an attempt to utilize parts of the available information.

## 1.1. Aim of the Thesis

In the last couple of years, there have been research papers that use geotagged information from social media sites like Twitter and Flickr. A good example of how useful information from Flickr can be is the creation of tourist density attractiveness maps. Spatial photography patterns of users that are not residents of a city or area are cumulated, thus creating a tourist attractiveness hot spot map. Also, temporal spatial patterns can be used to show in which order attractions are typically visited (PLADINO ET AL.; 2015; pp. 1-17).

Another study uses the geotagged Twitter data related to the University of Kentucky riots after the 2012 NCAA Championships to criticize the often perceived notion of letting “big” (geo-)data speak for itself. This is because social media is often outlier-driven and the user demographics are often skewed. Another point the paper raises is that just because information is geotagged does not mean that the information is about the place where it is geotagged. Also, the data might include information about other places not referenced by the geotag. The study argues for using implicit geotags, to integrate temporality and, enhance the big data with non-user generated information like census- and social data (CRAMPTON ET AL.; 2012; pp. 1-25).

The last example for the use of social media sites is a paper that looks at the segregation and mobility in Louisville, Kentucky. For this twitter users are identified that live either in the East or West End of the city. The daily activity space of those users is analyzed. One of the results of this work is **Map 1.1**. The odds ratio of the map shows when values approach 1 a relative parity for the chance that users of West or East End are tweeting within this area. A value smaller than 1 shows a higher chance of people from the west end of the city tweeting in this area. In Areas with a value above 1 the chances of twitter users from the East end of the city tweeting is higher.



**Map 1.1** Segregated Activity Spaces for Twitter Users of the West and East End in Louisville Kentucky (SHELTON ET AL.; 2015; p.9)

The analysis shows that there is a divide but that Users of the predominantly poorer west end neighborhood are much more mobile while the users of for the wealthier east end are much more confined to their neighborhood (SHELTON ET AL.; 2015; p.1-17).

While the paper from Crampton et al. (2012) proposes to use other information about places not included in geotags there are so far no works on using standard webpages and HTML documents to map and analyze space.

## 1.2. Research question

The aim of this thesis is to bring together the techniques used by information retrieval systems. These systems are used as discussed before by humans to gain knowledge about space. The information retrieval systems techniques are used on HTML documents that share one or more geotags. For this the HTML documents also need to be geotagged.

From this three research questions can be formulated.

1. How can unstructured information be retrieved and made usable?
2. How can this information be linked to places?
3. How can context be derived from this now structured and geotagged information?

### **1.3. Structure of Research design**

None of this is do able by out of the box software solutions most tools of this thesis are created by translating concepts of the fields data mining, Natural language processing, and information retrieval into code.

The first part of the work is dedicated to created usable data from two big datasets. One consists of raw crawled data and the other OpenStreetMap data. From the OpenStreetMap data a dataset for geotagging a selected part of the crawled data is created. This creates a landscape of spatially distributed information. The geotagged information is then treated with natural language and information retrieval methods. Finally an attempt of classification is made for addresses that could be matched with HTML documents. The machine classification will be evaluated by a hand classification. The scope of the thesis will be the 1<sup>st</sup> to 9<sup>th</sup> and 20<sup>th</sup> district of Vienna, Austria. To further limit the scope the crawled dataset will be limited to URLs of the .at top-level domain (TLD).

## 2. Introduction to Data Mining and Big Data

The terms data mining and Big Data both describe the underlying methods and theories in this thesis. This chapter is meant to give an introduction and overview about what data mining and Big Data is and how it relates to this thesis.

### 2.1. Data Mining

Data mining as a whole is a very broad field that spans many disciplines, for example statistics, database systems, pattern recognition and math, some of which the thesis touches upon. The goal that brings all these fields together is to try to discover patterns that are interesting or novel in a large amount of data. Data mining analysis can roughly be divided into data exploration, frequent pattern mining, classification and clustering. These are the parts of what can be seen as classical data mining. Which is a math and statistics heavy approach and assumes the data is already available in a mathematical usable way. But data mining is only part of a bigger knowledge discovery process. In this process there is a pre and post processing of the data. Examples for pre-processing are data extraction, data cleaning, data reduction and feature construction. Pattern and model interpretation are typical post-processing steps as well as hypothesis confirmation and generation. Both the processing steps and data mining are highly iterative and work interdependent (ZAKI AND MEIRA; 2014; pp. 25-26).

Exploratory data analysis utilizes key statistical values to explore features of a data set. These values show the centrality, dispersion and shape of the data. Discarding the assumption of independent and identically distributed variables or objects, data as a graph approach is a useful tool in the exploratory data analysis. Kernel methods to calculate pairwise similarity with the dot product can be utilized here. Another part of exploratory data analysis is to reduce the data just to the relevant parts. This can either be done by feature selection or by reducing the dimensions. Example methods would be principal component analysis and data sampling methods (ZAKI AND MEIRA; 2014; pp. 26-27).

Extracting useful or interesting patterns from data is the field of frequent pattern mining. Patterns can be co-occurring values or sequences of values. The task is to look for those co-occurrences or sequences that differ from the normal value distribution. Relationships between points can be either explicit positional, temporal, or arbitrary (ZAKI AND MEIRA; 2014; p. 27).

Clustering is the task to find objects that are “naturally” similar and grouping them together into groups or clusters. The goal is to have a cluster of objects that are most similar to each other and as dissimilar as possible to all other objects. There are a couple of different clustering methods for example hierarchy clustering, centroid based clustering, and density based clustering. Every clustering method also has different ways to be implemented (ZAKI AND MEIRA; 2014; pp. 28-29).

Different from clustering, classification is not about finding naturally similar groups, but rather about creating a blueprint of one or more groups and labelling the data points according to those groups. For this a classifier is needed that uses the blueprint to decide if a data point is part of one of the classes. The blueprint for the classes can be either learned or created. In order to learn a classification, a group of data points already needs to be correctly classified; these points are called the training set. The classifier then can “learn” from the training set and create a blueprint. Examples for machine learning algorithms are decision trees, probabilistic classifiers and support vector machines. The other option is to create a blueprint for the classifier by hand (ZAKI AND MEIRA; 2014; pp. 29-30).

A sub-group that falls within the data mining field is mining text based information from structured or unstructured documents. The field can again be divided up in many different groups, but the relevant ones for this thesis are text mining, eXtensible Markup Language (XML) mining and web mining. What applies to all of them is that they need much pre-processing, as can be also seen in this thesis. The increased need for pre-processing is manifold. The main obstacles are the following. The data structure has to be analyzed and understood to make the document useable and extract information. The text data needs to be transformed in such a way that it becomes mathematical and statistically useful. Raw data amount can be very big, text data is unstructured and the meaning can be fuzzy (TAN; 1999; pp. 65-71), (COOLEY ET AL.; 1997; pp. 558-567), (NAYAK ET AL.; 2002; pp. 660-666).

The outline of text mining can be split into two parts. The first part is to transform the text in an intermediate format. The kind of intermediate format depends on what analysis is planned in the second part. A group of documents can be transformed into a graph that shows how they relate to each other or each document can be transferred into an intermediate format. The second part is to perform a form of knowledge distillation on the intermediate format, for example to sort the documents depending on their content (TAN; 1999; pp. 65-68).

The term web mining can mean two different things. The first meaning is in the sense of mining content from the Internet. The second meaning is web usage mining that analyzes the access patterns of web users. For this thesis only the first one is of interest. Slightly different from text mining web content mining can exploited the more structured nature of html documents, for

example the relationship between documents can be mapped via hyperlinks. But many techniques of text mining also apply to web content mining (COOLEY ET AL.; 1997; pp. 558-560).

XML documents are the most structured of the three discussed sources. XML documents are tree like structured documents that can contain different kind of data and information. Examples range from quasi HTML like documents to complex 2D or 3D shapes and models. XML mining can be separated into content and structure mining. Whereas structure mining refers to analyzing and extracting the shape of the XML tree and content mining is about information extraction. Because of the rigid structure of XML documents it is possible to extract only specific information from specific parts of the XML tree, a feature that will be exploited later in the thesis (NAYAK ET AL.; 2002; pp. 660-666).

## **2.2. Big Data**

The term Big Data is describing a trend rather than it is a scientific term or a specific x amount of data. This trend is driven by the fact that computation has become ubiquitous. Computers are now found in smartphones, laptops, TVs, cars, fridges, personal sensors and so on. All these computers create a flood of information that can be analyzed with clusters of computers and sophisticated software tools. This duality of data creation and analysis on a big scale creates the knowledge infrastructure also called Big Data (BOLLIER AND FIRESTONE; 2010; pp. 1-10).

Examples for the use of this infrastructure can be found with companies like Google, which is using search engine queries to predict flu outbreaks and unemployment trends long before government statistics can show these information. It is also used by credit card companies to create heuristics that detect credit card fraud and identify consumer purchasing patterns. This is done by cross-examine large amounts of financial, personal and census data (BOLLIER AND FIRESTONE; 2010; pp. 1-9).

Big data techniques are used in medicine to compare health records on a large scale to find valuable correlation between prescribed treatments and outcomes. Social-networking sites data mine the information of their users for consumption preferences to create better advertisement or sell the information to marketing companies. Also geo-location data can play an influential role, by tracking the length of time consumers are willing to travel to a shopping center, it is possible to measure the consumer demand in an economy (BOLLIER AND FIRESTONE; 2010; pp. 1-9).

This knowledge infrastructure can provide valuable and interesting insights into society that were not possible before. But it also poses significant threats. Most of the players are large corporations and nation states that can use the techniques for surveillance or manipulate persons into buying products. This endangers personal privacy, civil liberties and freedom. Also most of the data

collection happens without informed consent of the individuals that cannot assess the impact it is going to have on their lives (BOLLIER AND FIRESTONE; 2010; pp. 1-9).

Big Geo-Data is a sub field of Big Data that also uses the spatiality of data in analysis. Most of the research done so far focuses on geotagged social media data. Crampton et al. 2015 argues that the approach of this field is often limited by two shortcomings. One is that many analyses do not account for the limitations of the Big Data. Limitation can for example be that social media is outlier driven and generated by a small were skewed fraction of the population. The second shortcoming is that many studies attach to much meaning to the geotag. They propose to compensate for those short comings in different ways. Spatiality should go beyond the here and now and discover how the geotagged data interlocks with other information in information networks. Also bolstering and comparing geotagged social media against other data like news reports or census data can help order and make sense of the information (CRAMPTON ET AL.; 2012; pp. 1-25).

### 3. The Common Crawl Dataset

This chapter is supposed to give an overview of the Common Crawl dataset that is used for the thesis, on how Common Crawl creates the dataset by crawling the Internet. This is followed by a discussion about the representativeness of the dataset. This will be done by comparing some basic metrics of the dataset with metrics from other sources about the Internet. The following section describes the structure of the available files, how they were indexed and how this index is used to only download a selected subset of the Common Crawl dataset.

#### 3.1. What is Web Crawling

Web crawling is the process by which webpages are gathered from the Internet. The program doing the crawling is either referred to as a crawler, spider or web-spider. These crawlers have certain features they must be equipped with and some others that they should be equipped with (MANNING ET AL; 2009; p.443).

**Robustness** is a must feature, because many web servers contain traps for crawlers, either on purpose or by accident. These traps get a crawler stuck in an infinite loop. A crawler must therefore be designed to be resistant to such traps.

**Politeness** is the second must feature of a crawler. There are certain implicit and explicit policies regulating if and how often a website is crawled. Probably best known is the robot.txt which specifies if and which directories the crawler is allowed to crawl.

Features a crawler should provide are:

**Distributed**, which means that the crawler can be run parallel across multiple machines.

**Scalability**, the crawler scales up its performance as linear as possible when more machines are added to the crawling process.

**Performance and efficiency**, the crawling system should use the system resources as efficient as possible.

**Quality**, because a lot of web pages are of poor quality and contain little useful information for the user, the crawler should focus first on “useful” webpages.

**Freshness**, a crawler should be designed in a way that it crawls a site at the same rate that the content on the site changes.

**Extensible**, a crawler should be developed in a modular way, so that it can cope with fetch protocols and new standards (MANNING ET AL; 2009; pp.443-445 ).



The operation of a crawler is fairly simple. The crawler begins with one or more provided seed URLs, the seed set. It fetches the page for one of the uniform resource locators (URL) and then parses it. On the page, the crawler is looking for new URLs, and adding the parsed text to the search index. The newly found links are added to the not yet fetched URLs, also called the URL frontier. Then the next URL from the frontier is fetched and so on (MANNING ET AL; 2009; pp.445-448).

This seemingly simple recursive task is rather complicated because of the heterogeneous nature of the web. Many of the problems only occur when the crawler starts crawling real data. For example the first Google crawler produced error messages in the middle of a web game, but the error only came up after tens of millions of page downloads. To fetch only a small amount of the static web, for example one billion webpages within a month, the crawler must still be able to download several hundred sites per second (BRIN AND PAGE; 1998; p.10).

### 3.2 The Common Crawl Dataset

As described in the previous section crawling a large part of the Internet is a complex and resource demanding and therefore costly task. Common Crawl is a nonprofit project whose goals are to allow access to crawled information to everyone without the costs and complexities that come with crawling the Internet independently. The data is accessible through the Amazon Web Service (COMMON CRAWL).

Crawled information is stored in the ARC File Format. This format meets certain defined requirements. The file must be self-contained. This means that there is no need for an Index file to identify and unpack the archive file. The format is extensible in a way that it can be adapted to be transferred with different network protocols. It is possible to concatenate multiple archives into one data stream. The file is viable and there is no need of an in-file index to guarantee the files integrity. A typical arc file is shown in **Figure 3.1** (COMMON CRAWL).

```

filedesc://IA-001102.arc 0.0.0.0 19960923142103 text/plain 200 - - 0
IA-001102.arc 122
2 0 Alexa Internet
URL IP-address Archive-date Content-type Result-code Checksum
Location Offset Filename Archive-length

http://www.dryswamp.edu:80/index.html 127.10.100.2 19961104142103
text/html 200 fac069150613fe55599cc7fa88aa089d - 209 IA-001102.arc 202
HTTP/1.0 200 Document follows
Date: Mon, 04 Nov 1996 14:21:06 GMT
Server: NCSA/1.4.1
Content-type: text/html Last-modified: Sat,10 Aug 1996 22:33:11 GMT
Content-length: 30
<HTML>
Hello World!!!
</HTML>

```

**Figure 3.1** ARC File example (Archive.org)

There are two main parts to the ARC File: Header and content. The header contains a variety of Meta information about what was crawled, when and, by whom. The Content Part contains the actual content of what was crawled and begins in the line following the content-length information. Common Crawl compresses the ARC files with a compression algorithm and stores them in an Amazon S3 bucket (ARCHIVE.ORG), (COMMON CRAWL).

The first available Common Crawl Corpus in October 2011 contained 5 billion individual pages which equaled more than 40 TB of data. To put this in perspective Google had downloaded around a trillion pages by 2008. But according to Google, most of this was junk information. Common Crawl employs a page ranking algorithm to fetch only relevant webpages from the Internet (COMMON CRAWL BLOG; Community questions).

The corpus used for this thesis was released in July 2012. Even though the thesis was written in 2014 and 2015, it is dependent on the Common Crawl URL index, which was at this time only available for the 2012 corpus (COMMON CRAWL ATLISSIAN), (COMMON CRAWL BLOG; URL index).

Spiegler created detailed statistics for the 2012 Corpus. According to the study, the corpus consists of 210 TB of data, 3.83 billion individual documents and 41.4 million unique second-level domains (SLD). The goal of this paper is to determine how representative the Common Crawl Corpus 2012 is for the whole web. For this the frequency of the 75 most common top-level domains (TLD) of the Common Crawl Corpus 2012 has been compared to the figures of the web technology survey provided by W3Techs. This comparison with a spearman rank correlation coefficient gave a value of 0.84 for  $\rho$  which indicates a high statistical dependency between both datasets. In **Table 3.1a** the 10 most over-represented and in **Table 3.2b** the 10 most under represented TLDs can be found. The

value was calculated by dividing the common Crawl frequency with the web technology survey frequency (SPIEGLER, 2013, p. 1-6).

TLD	Rel. freq. W3 survey	Rel. freq. CC corpus	Ratio
.gov	0.001	0.0026	2.6
.nz	0.001	0.0022	2.2
.edu	0.003	0.0061	2.0
.uk	0.019	0.0346	1.8
.nl	0.008	0.0143	1.8
.se	0.003	0.0053	1.8
.ca	0.004	0.0069	1.7
.ch	0.003	0.0052	1.7
.cz	0.005	0.0076	1.5
.org	0.041	0.0603	1.5

a) Over represented TLDs

TLD	Rel. freq. W3 survey	Rel. freq. CC corpus	Ratio
.in	0.009	0.0021	0.2
.tk	0.001	0.0002	0.2
.th	0.001	0.0002	0.2
.kz	0.001	0.0002	0.2
.co	0.003	0.0004	0.1
.az	0.001	0.0001	0.1
.asia	0.001	0.0001	0.1
.pk	0.001	0.0001	0.1
.ve	0.001	0.0001	0.1
.ir	0.006	0.0005	0.1

b) Under represented TLDs

**Table 3.1 a), b)** Representativeness of TLDs in the Common Crawl Corpus 2012 (SPIELGER; 2013)

Because of the huge amount of data and the limited resources available, this work will only draw on data from URLs with an .at TLD. This includes all the .at special case SLDs .ac.at, .gv.at, .co.at, .or.at and .priv.at. The Common Crawl Corpus 2012 includes 317,578 unique URLs that have an .at TLD. This results in a relative corpus frequency of 0.0076. In comparison according to the figures of the web technology survey .at TLDs account for 0.3% of all TLDs. The .at TLD is therefore over represented by 2.5 in the Common Crawl Corpus 2012 (SPIEGLER, 2013, p. 3), (W<sup>3</sup>TECHS).

According to nic.at in 2012 there have been 1,121,235 domains registered in the .at TLD Zone; this means that the 2012 corpus contains about 28% of all registered .at domains (NIC.AT; .at Report 2012).

### 3.3. The Common Crawl Dataset Index

As mentioned before, to use the complete corpus for this thesis is unfeasible because only limited resources are available. So the data needs to be narrowed down to a manageable size that has the highest probability of containing Viennese street addresses. The decision was made to only include URLs that have an .at TLD. But since URLs within the Common Crawl Corpus are unsorted to access only specific URLs an Index of the corpus is needed. Robertson did create such an index for the Common Crawl Corpus 2012. In the creation of the index, there were a couple of challenges to overcome (ROBERTSON; 2013).

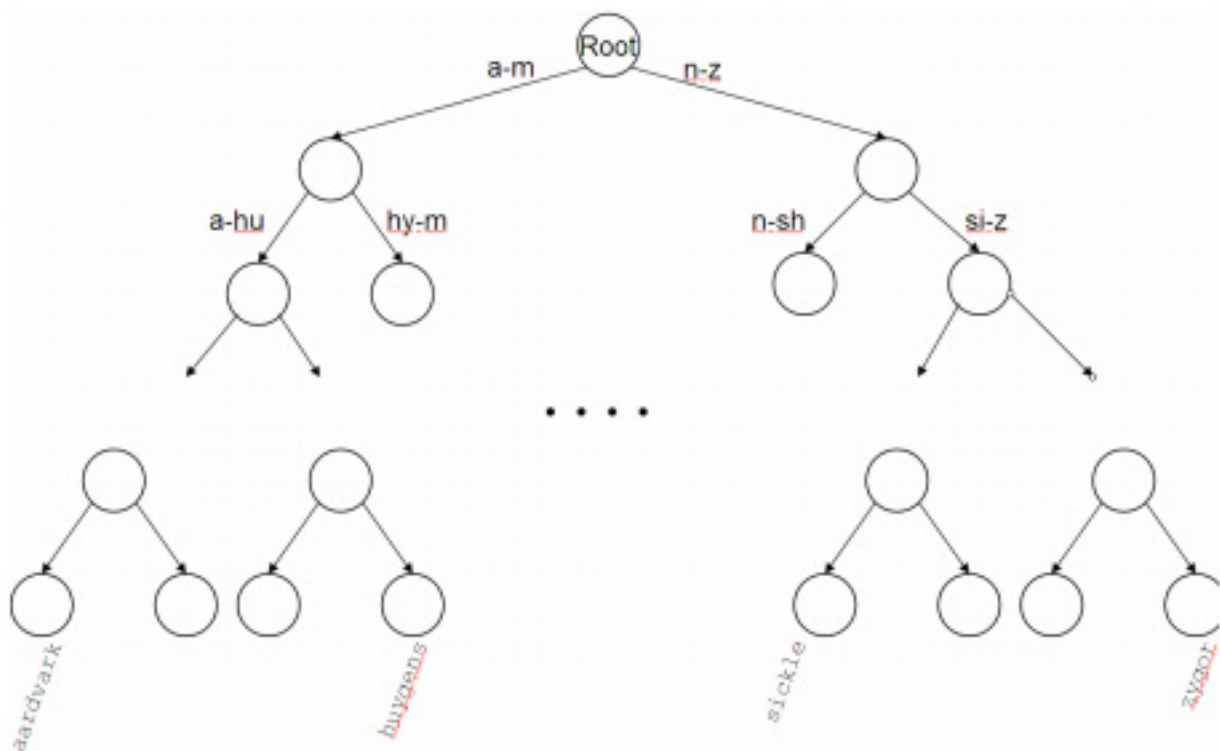
The index needs to be huge because the corpus contains 3.83 Billion URLs. The average URL size is 66 bytes and an additional pointer to the file segment needs another 28 bytes. In order to merely information, a file larger than 360 GB is needed. Because of this large amount of data needed, it is

not possible to create the index in random access memory, but it still needs to be fast (ROBERTSON; 2013).

The index should be accessible by a wide variety of tools and people. But there is also the need to keep hosting and processing costs down, because Common Crawl is a nonprofit organization.

To meet all these demands the index is also hosted, like the Common Crawl Corpus, in an Amazon S3 bucket. It is not necessary to download the full index to work with it. The index can be queried and searched without a local copy (ROBERTSON; 2013).

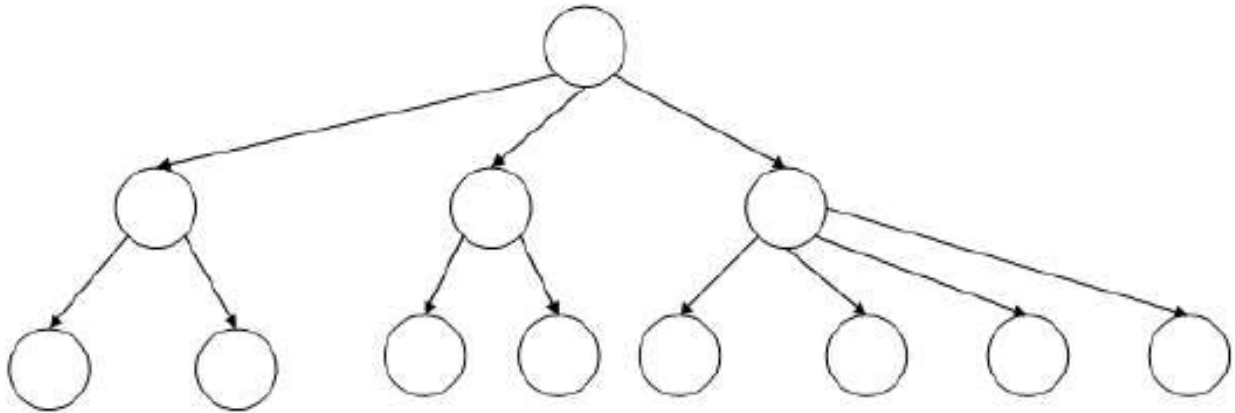
The file format of the index is based on a Prefix B-Tree. These search trees are one of the two broad classes for key lookup operations. In the case of this index the keys are the URLs and corresponding to them the file pointers to the Common Crawl Corpus 2012 (BAYER AND UNTERAUER; 1977; pp. 11-26).



**Figure 3.2** Example Binary Tree (MANNING ET AL, 2009)

**Figure 3.2** depicts a binary tree format. The binary tree owes its name to its structure because every node has two branches. The search for a key begins at the root of the tree. If the first letter is within the range of A to M the algorithm takes the corresponding branch; if not, it takes the other. This process is repeated at every node until it arrives at the final node, which contains the key. An issue with binary trees is that they need to be balanced to be effective. The number of keys beneath each

subtree on every level must be equal. If a key is added the whole tree needs to be rebalanced. To mitigate this problem the number of subtrees of a node in a B-tree (a generalization of a Binary Tree) is not fixed to two but can vary within a defined interval. An example can be seen in **Figure 3.3** (MANNING ET AL; 2009; pp.49-53).



**Figure 3.3** Example B-Tree (MANNING ET AL, 2009)

Prefix B-Trees, like the one used for the Common Crawl index, are a combination of B-Trees and key compression techniques to save space (BAYER AND UNTERAUER; 1977; pp. 11-26).

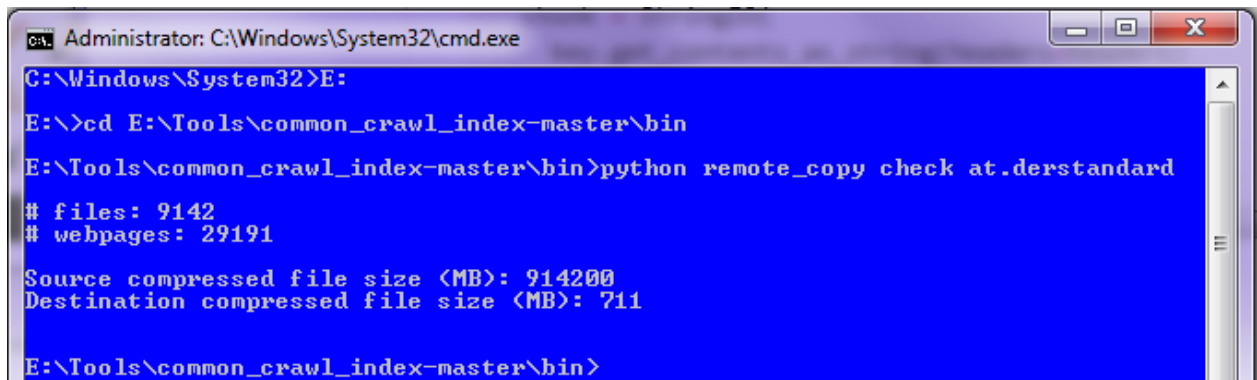
B-Trees are capable of wildcard queries. For example, a query like “tru\*” is a trailing wildcard query because the wildcard symbol is located at the end of the search string. The structure of a B-Tree allows it to handle this query conveniently. The algorithm follows the symbols t then r and then u down the tree at which point it encounters a node beneath all keys begin with “tru” (MANNING ET AL; 2009; pp.49-53).

For wildcards leading a search string like in, for example, “\*paper” an inverted B-Tree is used. That means a key in the inverted B-Tree corresponds to a path written backwards. So the term newspaper is represented as the path root-r-e-p-a-p-s-w-e-n (MANNING ET AL; 2009; pp.49-53).

With a combination of a B-tree and an inversed B-tree, queries like “Me\*ro” become possible. The search string is split at the wildcard symbol. “Me\*” is used on the B-Tree and “\*ro” on the inversion. From both results an intersecting set is created containing all the keys beginning with “Me” and ending in “ro” (MANNING ET AL; 2009; pp.49-53).

Applied to the Common Crawl index, the result looks like the example in **Figure 3.4**. The index is queried for all URLs that point to the SLD derstandard.at. Because the check command is given, the index returns just the number of webpages associated with the search term and the compressed size of them. The order of URLs in the B-Tree of the Common Crawl index is inversed. A TLD is flowed by

an SLD and so on. The script automatically assumes a wild card at the end of the queried URL. So in the example the search term could also be read as “at.derstandard\*” (ROBERTSON; 2013).



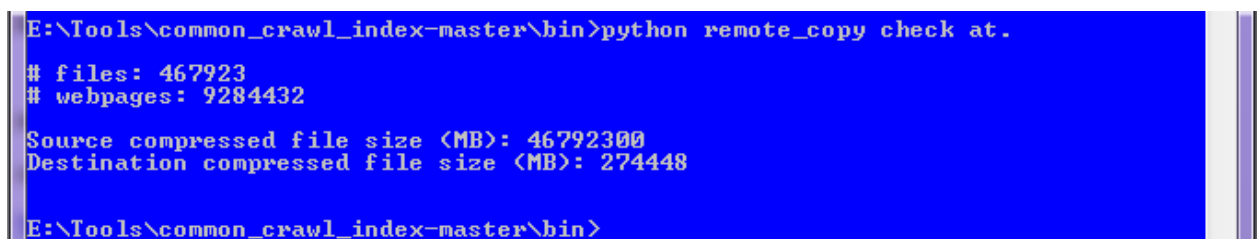
```

Administrator: C:\Windows\System32\cmd.exe
C:\Windows\System32>E:
E:\>cd E:\Tools\common_crawl_index-master\bin
E:\Tools\common_crawl_index-master\bin>python remote_copy check at.derstandard
# files: 9142
# webpages: 29191
Source compressed file size <MB>: 914200
Destination compressed file size <MB>: 711
E:\Tools\common_crawl_index-master\bin>

```

**Figure 3.4** Index check example derstandard.at

To query the index for the whole .at TLD is therefore simple. The index returns all webpages whose URLs end in .at. This includes all the special cases .at SLDs .ac.at, .gv.at, .co.at, .or.at and .priv.at. Results can be seen in **Figure 3.5**.



```

E:\Tools\common_crawl_index-master\bin>python remote_copy check at.
# files: 467923
# webpages: 9284432
Source compressed file size <MB>: 46792300
Destination compressed file size <MB>: 274448
E:\Tools\common_crawl_index-master\bin>

```

**Figure 3.5** Index check example .at TLD

The statistics that the tool generates show, on the one hand, the overall number and size of the arc files the data is distributed over and, on the other, the number of webpages and size when only the relevant data is extracted from all ARC files (ROBERTSON; 2013).

### 3.4 Downloading the Common Crawl Dataset

The naïve solution to download the part of the Common Crawl dataset that is needed would be the command “remote\_copy copy at. --bucket target-bucket-name”. There is the possibility to tweak this command by appending “--parallel x”, which defines in how many parallel instances the script should

download the data. But the script isn't error proof and the .at name space is so big errors always occurred, leaving the script hung up at some point (ROBERTSON; 2013).

To circumvent this problem a second script was written that a) split the .at name space into smaller pieces b) parallelized the original script further and c) added some error handling. For this a new method was created within the remote\_copy script called external. This method allowed handing over arguments to remote\_copy from another script without calling it over the command line like the ones above.

To split the .at name space into smaller pieces, a list of three letter URL stumps was created with the code in **Figure 3.6**.

```

035 def create_urllist():
036
037     list1 = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q',
'r','s','t','u','v','w','x','y','z','0','1','2','3','4','5','6','7','8','9']
038     list2 = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q',
'r','s','t','u','v','w','x','y','z','0','1','2','3','4','5','6','7','8','9','-']
039     list3 = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q',
'r','s','t','u','v','w','x','y','z','0','1','2','3','4','5','6','7','8','9','-','.']
040     completelist = []
041     for first in list1:
042         for second in list2:
043             for third in list3:
044                 completelist.append('at.'+first+second+third)
045     return completelist

```

**Figure 3.6** Code example URL stump creation

The code in Figure 2.X conforms to all rules set by nic.at for domains in the .at TLD space. They must contain at least 3 symbols and an hyphen isn't allowed as a first symbol. The code above results in a list of strings that looks like this: `[at.aaa, at.aab, at.aac, ...]` (NIC.AT; Registration Guidelines).

The next step is to initiate every URL stump as a discrete download process. The easiest way would be to process every element on the list sequentially. But this is a rather slow process and it is necessary to transfer data related to a couple of URL stumps in parallel. For this though there needs to be a threading environment that controls which URL stumps have already been processed and that keeps the number of parallel threads to a certain limit. This is what the following code examples do.

```

008 class myThread (threading.Thread):
009     def __init__(self, urlstump, threadid):
010         threading.Thread.__init__(self)
011         self.urlstump = urlstump
012         self.id = threadid
013     def run(self):
014         threadLimiter.acquire()
015         print 'checking for ' + str(self.urlstump)
016         current_urllist.append(self.urlstump)
017         remote_copy_external.external('AWS-PUBLIC-KEY',
018         'AWS-PRIVATE-KEY', 'tldat', 'Data2/'+str(self.urlstump),
019         self.urlstump, parallelconnections, True)
020
021         print "Exit Thread: %d of %d" %(self.id, NummberIDs)
022         urllist.remove(self.urlstump)
023         current_urllist.remove(self.urlstump)
024         threadLimiter.release()

```

**Figure 3.7** Threading Code

**Figure 3.7** shows the construction code for a new thread object. This creates a parallel python process. The lines 9 to 12 define the variables of this thread, which are only the URL stumps that are to be downloaded and a thread ID used for identification. Lines 13 to 23 state what the thread does as soon as it is started. Lines 15 and 21 just print some console output. Line 16 appends the active URL stump of this thread to a list of active URL stumps. This list is later used to check if one of the threads is hanging. Line 17 and 18 finally execute the `remote_copy` script that was modified to be started from another python script. If the download was successful the URL stump of this thread is deleted from a control list in line 22 and from the `current_urllist` in line 23 (PYTHON 2.7.10 LIBRARY; threading).



```

047 while running:
048     print ''
049     print 'running threads %s'%(len(threading.enumerate()))
050     urllist_pickle = list(urllist)
051     ipickle = iterate_ipickle(ipickle)
052     picklelist(urllist_pickle, 'obj_%s.pickle'%(ipickle))
053
054     if duds+threadnumber > len(threading.enumerate()):
055         if threadlist:
056             element = threadlist[0]
057             threadlist.remove(element)
058             myThread(element, ID).start()
059             ID += 1
060             continue
061         if not threadlist:
062             pass
063
064     elif threadlist:
065         ipickle = iterate_ipickle(ipickle)
066         picklelist(urllist_pickle, 'obj_%s.pickle'%(ipickle))
067         time.sleep(1)
068         print current_urllist
069         continue
070
071     elif not threadlist:
072         ipickle = iterate_ipickle(ipickle)
073         picklelist(urllist_pickle, 'obj_%s.pickle'%(ipickle))
074         pass
075
076     while duds < len(threading.enumerate()):
077         time.sleep(10)
078
079         print ''
080         print 'Current Passnumber: %d'%(Passnumber)
081         print current_urllist
082         urllist_pickle = list(urllist)
083         ipickle = iterate_ipickle(ipickle)
084         picklelist(urllist_pickle, 'obj_%s.pickle'%(ipickle))
085         pass
086
087     if not urllist:
088         running = False
089         print ''
090         print 'Script did run Passnumber %d'%(Passnumber)
091         pass
092     else:
093         threadlist = list(urllist)
094         urllist_pickle = list(urllist)
095         ipickle = iterate_ipickle(ipickle)
096         picklelist(urllist_pickle, 'obj_%s.pickle'%(ipickle))
097         print ''
098         print 'Script did run Passnumber %d'%(Passnumber)
099         Passnumber += 1

```

**Figure 3.8** Starting and controlling threads

The threads need a controlling mechanism. The `while running` loop in **Figure 3.8** is responsible for this function. This loop is executed as long as `running == True`. The lines 50 to 52 are responsible for saving the current state of the URL stump list. This is necessary, because the whole script runs over the length of a couple of days and in the event that it should crash for an unforeseen

reason the last version of the list can be reloaded. Otherwise the script would need to start from the beginning again. The `if` statement in line 54 is responsible for limiting the number of active threads. Before the `while running` loop is started, the number of active threads is saved to the variable `duds`. The test in line 54 checks that the number of active threads does not exceed the number of active threads before the loop was started plus the number of parallel download threads intended. If this is the case, the subsequent `if` statement in line 48 is called upon (PYTHON 2.7.10 LIBRARY; Built-in types).

Line 55 checks if there are still URL stumps in the list `threadlist`, which are not yet downloaded. If this is true, one of the URL stumps is removed from the `threadlist` and a new thread is started.

Lines 64 to 69 and 71 to 74 also save the URL stump list in different phases of the script. The main difference is that lines 71 to 74 are only invoked if the `threadlist` is empty and `pass` in line 74 prompts the script to execute the loop further, while the `continue` in line 69 makes the loop jump back to the beginning in line 47 (PYTHON 2.7.10 LIBRARY; Built-in types), (PYTHON 2.7.10 LIBRARY; pickle).

Line 76 checks if there are still active download threads running even though the `threadlist` is empty. It does that by testing if the number of threads is still bigger than before the `while running` loop was started. If that is the case, the loop waits for 10 seconds, then saves the current version of `urllist` and performs the active thread check again (PYTHON 2.7.10 LIBRARY; Built-in types).

Line 87 checks if there are still URL stumps in the `urllist`. If not, `running` is set to `False`, thus ending the main while loop. If there are still stumps in the `urllist`, those are copied again to the `threadlist` in Line 93, the current `urllist` is saved and the while loop begins again at the top (PYTHON 2.7.10 LIBRARY; Built-in types).

The whole code works with two main lists the `threadlist` and the `urllist`. In the beginning, the `threadlist` is a copy of the `urllist`. During the threading process, one URL stump at a time is removed from the `threadlist` and a new thread is started with this stump. If a thread runs successfully to its end, this stump is also removed from `urllist` (see **Figure 3.7** line 23). In the end, a check is performed to see if all elements of the `urllist` have been processed. If not, the remaining elements are again copied to the `threadlist`. This construction was necessary because single download threads tended to crash (PYTHON 2.7.10 LIBRARY; Built-in types).

All data was copied this way from the Common Crawl Corpus 2012 to the tldat bucket from where it was downloaded to a local machine for further processing.

## 4. OpenStreetMap Geocoder

This chapter looks at the extraction of addresses with spatial information, also known as geocoding, from an OpenStreetMap (OSM) dataset that is limited to Vienna.

First, there is a short of overview what OSM is and how data from OSM is structured. The sections that follow look at the code used to do the extraction and how the addresses are transferred to the database.

### 4.1. OpenStreetMap

OpenStreetMap is a free volunteer based worldwide geodata set. OSM works similarly to Wikipedia but is about geospatial information. The information is freely available in the sense of no attached costs. Another difference to other web map providers like Google Maps or Bing Maps is in the sense of free as in free speech. The information is not only useable in the form of rendered map tiles but the underlying geodata itself is available to every user.

This kind of data is called volunteer geographic information (VGI). The OSM Project is managed by the OSM Foundation. This is a UK based not-for-profit organization that acts as a legal entity for the project. The foundation is the custodian of server hardware necessary to host OSM. It also organizes fundraisers for the project, organizes an annual conference and supports communication with several project workgroups. OSM was founded in 2004 and grew to a base of 640,000 supporters by 2012 with a wide variety of geospatial information entered into the database (NEIS AND ZIPF, 2012, pp. 146-163).

### 4.2. Data Structure and Source

OSM uses Extensible Markup Language (XML) for data exchange. XML is a meta format that provides human readable data exchange. OSM building upon this kind of data exchange has a couple of advantages. First, it is a system-independent format. Already existing XML parsers can be easily modified to parse OSM data. Second, it is human readable because it has a clear tree structure and files have a good compression ratio. The downside of OSM XML (.osm) is that the files are large. Therefore it might necessary to decompressing them first and parsing can take a lot of time. An example of an OSM XML file can be seen in **Figure 4.1** below (NEILS AND ZIPF, 2012, pp. 146-163).

```

<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
  <bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900" maxlon="12.2524800"/>
  <node id="298884269" lat="54.0901746" lon="12.2482632" user="SvenHRO" uid="46882" visible="true"
    version="1" changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
  <node id="261728686" lat="54.0906309" lon="12.2441924" user="PikoWinter" uid="36744"
    visible="true" version="1" changeset="323878" timestamp="2008-05-03T13:39:23Z"/>
  <node id="1831881213" version="1" changeset="12370172" lat="54.0900666" lon="12.2539381"
    user="lafkor" uid="75625" visible="true" timestamp="2012-07-20T09:43:19Z">
    <tag k="name" v="Neu Broderstorf"/>
    <tag k="traffic_sign" v="city_limit"/>
  </node>
  <way id="26659127" user="Masch" uid="55988" visible="true" version="5" changeset="4142606"
    timestamp="2010-03-16T11:47:08Z">
    <nd ref="292403538"/>
    <nd ref="298884289"/>
    ...
    <nd ref="261728686"/>
    <tag k="highway" v="unclassified"/>
    <tag k="name" v="Pastower Stra" />
  </way>
  <relation id="56688" user="kmvar" uid="56190" visible="true" version="28" changeset="6947637"
    timestamp="2011-01-12T14:23:49Z">
    <member type="node" ref="294942404" role=""/>
    ...
    <member type="node" ref="364933006" role=""/>
    <member type="way" ref="4579143" role=""/>
    ...
    <member type="node" ref="249673494" role=""/>
    <tag k="name" v="Kstenbus Linie 123"/>
    <tag k="network" v="VWV"/>
    <tag k="operator" v="Regionalverkehr Kste"/>
    <tag k="ref" v="123"/>
    <tag k="route" v="bus"/>
    <tag k="type" v="route"/>
  </relation>
  ...
</osm>

```

**Figure 4.1** OSM Data Example (OPENSTREETMAP WIKI; OSM XML)

OSM Data is always structured the same way. First, there is an XML suffix declaring that the character set of the file is UTF-8 encoded. This is followed by the `<osm>` element that contains the version of the API with which it was created and the generator tool. The Extent of the data is described by a `<bounds>` block. Next is the nodes block: it contains all nodes displayed within the bounds. All nodes have an ID and coordinates that are expressed in the WGS84 reference system. Nodes may contain nested tags. The next block contains all the ways. Ways are a list of ordered nodes. When the way is a closed way, that means the starting and ending node are the same node, nodes act as vertices of a polygon. If the way is not closed, the way acts as a line feature, again with the nodes used as vertices of the feature. Apart from references to the nodes, a way normally contains a couple of other tags helping to describe the object depicted by it. Lastly, there are relations. These features consist of a group of nodes, ways and other relations, all of them referred to as a member of the relation. A typical example for a relation is a feature that has an inner and outer edge. For this, two closed ways are combined into a relation: one describes the outer, and the

other, the inner edge of the feature. Like a way or a node, a relation can have tags that describe the feature depicted (OPENSTREETMAP WIKI; OSM XML).

### 4.3. Extracting Addresses

Addresses are stored as a group of tags in a node, a way or a relation. An example of the address tag is shown in **Figure 3.2**.

```
<node id="566992041" visible="true" version="1" changeset="3158538" timestamp="2009-11-19T10:44:24Z" user="andreas_k" uid="39877" lat="48.2136818" lon="16.3604013">
  <tag k="addr:city" v="Wien"/>
  <tag k="addr:country" v="AT"/>
  <tag k="addr:housenumber" v="1"/>
  <tag k="addr:postcode" v="1010"/>
  <tag k="addr:street" v="Universitätsstraße"/>
</node>
```

**Figure 4.2** Example address tag (Openstreetmap.org)

An OSM XML file covering the area of Vienna was downloaded from Cloudmade. This file was then parsed for addresses with the following code. For parsing the OSM data, the python sax parser is used. Different from more complex parsers available, this parser does not try to recreate the XML document as a tree-like object in memory. This is of importance because the whole file containing Vienna is about 0.7 GB big in XML and a tree object for this amount of data always exceeds the available 8 GB of random access memory (RAM). The sax parser reads the XML file line by line. Using the sax parser results in more complex code, but allows it to run on much less RAM (PYTHON 2.7.10 LIBRARY; xml.sax), (CLOUDMADE).

**Figure 3.3** shows the first part of this code; the parser is initiated as the class `startendfinder()`, with the content that should be parsed handed over. The class itself consists of a range of methods and a range of variables defined and initiated in the lines 13 to 29. Without getting into each one of them now, the most important ones are the `self.address` object defined in line 13 and the three dictionary lines 15 to 17. The method `startElement()` in line 31 is executed when the parser detects that a line is the start of a new XML element. XML elements are opened with `<ElementName>` and closed with the same tag name preceded by a slash, for example, `</ElementName>`. A special case is a tag that is opened and closed in the same line expressed with a slash trailing the name, for example, `<ElementName/>` (PYTHON 2.7.10 LIBRARY; xml.sax).

The method contains an `if/elif` statement that checks the name of the element. If the element is a node, the information of this node is saved in a couple of variables. First, the `self.address` object is given the coordinate of the node in line 33, even though, at this point it is unknown if this node contains an address. The coordinates are already put into a format `('POINT(%s %s)' % (attrs.get('lon'), attrs.get('lat')))` with which they can be easily written into a PostgreSQL table with a PostGIS field (PYTHON 2.7.10 LIBRARY; xml.sax), (POSTGIS 2.1.3; documentation), (POSTGRESQL 9.3.9; documentation).

The spatial information of the node is also copied into the `self.nodedict` dictionary in line 34 with the ID of the node as the key and the latitude and longitude as values. This is necessary because all other objects (ways and relations) only refer to nodes and do not contain spatial information themselves. But with this dictionary, it is simple to look up this spatial information. Lastly, the `self.nodemode` is set to `True`. The two other `elif` statements that are triggered in the event that the element is not a node simply get the ID of the element and set either `self.waymode` or `self.relationmode` to `True` (PYTHON 2.7.10 LIBRARY; xml.sax).

```

011 class startendfinder(handler.ContentHandler):
012     def __init__(self):
013         self.address = ['lat_lon', 'pcode', 'street', 'number']
014
015         self.nodedict = {}
016         self.waydict = {}
017         self.relationdict = {}
018         self.plz = False
019         self.street = False
020         self.number = False
021         self.nodemode = False
022         self.waymode = False
023         self.relationmode = False
024         self.wayid = False
025         self.relationid = False
026         self.ndlist = []
027         self.memberlist = []
028         self.latlist = []
029         self.lonlist = []
030
031     def startElement(self, name, attrs):
032         if name in ('node'):
033             self.address[0] = 'POINT(%s %s)' % (attrs.get('lon'),
034                                                attrs.get('lat'))
035
036             self.nodedict[int(attrs.get('id'))] = (float(attrs.get('lat')),
037                                                  float(attrs.get('lon')))
038
039             self.nodemode = True
040
041         elif name in ('way'):
042             self.wayid = int(attrs.get('id'))
043             self.waymode = True
044
045         elif name in ('relation'):
046             self.relationid = int(attrs.get('id'))
047             self.relationmode = True

```

Figure 4.3 OpenStreetMap XML parser start element part one

The next part of code seen in **Figure 4.4** is still part of the `startElement()` method. The following lines 96 to 117 are invoked if `self.nodemode` is `True` and the line the parser is parsing is the start of an element. Both of these things happen when there is still an open node element, because of line 36 in **Figure 4.2**, and this element contains nested elements (PYTHON 2.7.10 LIBRARY; xml.sax).

```

096         if self.nodemode == True:
097             if name == 'tag':
098                 k, v = (attrs.get('k'), attrs.get('v'))
099
100                 if k == 'addr:street':
101                     self.address[1] = unicode(v)
102                     self.street = True
103
104                 if k == 'addr:housenumber':
105                     self.address[2] = unicode(v)
106                     self.number = True
107
108                 if k == 'addr:postcode':
109                     try:
110                         if int(v) <= 1099:
111                             self.address[3] = int(v)
112                             self.plz = True
113                         elif int(v) >= 1200 and int(v) <= 1209:
114                             self.address[3] = int(v)
115                             self.plz = True
116                     except:
117                         pass

```

**Figure 4.4** OpenStreetMap XML parser start element part two

Those nested XML elements are then again checked in line 97 for their names. If the name is tag, then the attributes of the element `'k'` and `'v'` are saved to variables with the same name. The lines 100, 104 and 108 test `k` if the tag is part of an address. If so, the `v` is written to the part of the address object defined as the street name, house number or postcode. Also, the corresponding control variables `self.plz`, `self.number` and `self.street` are set to `True` indicating that when all are `True` a complete address was obtained from the node element (PYTHON 2.7.10 LIBRARY; xml.sax), (OPENSTREETMAP, Wiki Addresses).

The postcode is a special case because it has to be put in a `try/except` statement and it filters all addresses that are not within the 1<sup>st</sup> to 9<sup>th</sup>, or 20<sup>th</sup> district. To implement this filter, the `v` variable is converted to an integer value and tested to be within a certain value range as can be seen in lines 110 and 113. This conversion to an integer value is also the reason for the `try/except` statement. Because of errors within the dataset, not all `v` attributes that correspond to a `k` attribute of `'addr:postcode'` can be converted to integer (PYTHON 2.7.10 LIBRARY; xml.sax), (OPENSTREETMAP, Wiki Addresses).

```

046         if self.relationmode == True:
047             if name == 'member':
048                 self.memberlist.append((int(attrs.get('ref')), attrs.get('type')))
049             if name == 'tag':
050                 k, v = (attrs.get('k'), attrs.get('v'))
051
052                 [...]
053
071         if self.waymode == True:
072             if name == 'nd':
073                 self.ndlist.append(int(attrs.get('ref')))
074             if name == 'tag':
075                 k, v = (attrs.get('k'), attrs.get('v'))
076
077                 [...]

```

**Figure 4.5** OpenStreetMap XML parser start element part three

The `self.relationmode` and the `self.waymode` in **Figure 4.5** that are called when the top level element is a relation or a way work analogously to the way `self.nodemode` in acquiring an address. But there is one key difference: relations of all members of the relation are collected within the `self.memberlist` in line 48 and ways where all nodes are part of the way are collected in a `self.ndlist` in line 73. For this, the nested elements are tested for their name and, if they match either `'nd'` or `'member'`, they are appended to the corresponding lists. The `self.ndlist` only contains the references to the nodes because ways can only consist of nodes, while the `self.memberlist` also contains the information about what kind of object (node, way or relation) the element refers to (PYTHON 2.7.10 LIBRARY; xml.sax), (OPENSTREETMAP WIKI; OSM XML).

```

119     def endElement(self, name):
120         if name in ('node'):
121             if self.plz is True and self.street is True and self.number is True:
122                 addresslist.append(tuple(self.address))
123
124             self.nodemode = False
125             self.plz = False
126             self.street = False
127             self.number = False
128             self.address = ['lat_lon', 'pcode', 'street', 'number']

```

**Figure 4.6** OpenStreetMap XML parser end element part one

Following `startElement()` is the `endElement()` method. As the name implies, this method is executed when the end of an element is reached. The arguments passed to the method are `self` and `name`. Again, what the method does is dependent on the type of element that is closed. The code that will be executed if the element is a node can be seen in example **Figure 4.6**. The `if`



statement in line 121 is executed when an address was successfully extracted for this node (compare with **Figure 4.4** lines 100 to 115). The `self.address` object is appended to an `addresslist` which in turn is written to a PostgreSQL/PostGIS Database as soon as the whole OSM XML file is parsed. In lines 124 to 128, all switch variables are returned to their default values. When the node is at an end, the information is no longer relevant for the rest of the process and the default values are needed in order for the process to work correctly (PYTHON 2.7.10 LIBRARY; xml.sax).

The code for relation and way elements is a bit more complex because, as mentioned before, both of those objects only contain references to objects with spatial information and no spatial information themselves.

```

130         if name in ('way'):
131             for nd in self.ndlist:
132                 self.latlon = self.nodedict[nd]
133                 self.latlist.append(self.latlon[0])
134                 self.lonlist.append(self.latlon[1])
135
136                 self.waydict[self.wayid] = (numpy.mean(self.latlist),
                                                numpy.mean(self.lonlist))
137
138             if self.plz is True and self.street is True and self.number is True:
139                 self.address[0] = 'POINT(%s %s)' % (numpy.mean(self.lonlist),
                                                         numpy.mean(self.latlist))
140                 addresslist.append(tuple(self.address))
141
142             self.latlist = []
143             self.lonlist = []
144             self.waymode = False
145             self.plz = False
146             self.street = False
147             self.number = False
148             self.address = ['lat_lon', 'pcode', 'street', 'number']
149             self.ndlist = []

```

**Figure 4.7** OpenStreetMap XML parser end element part two

The code in **Figure 4.7** is still part of the `endElement()` method. It depicts what is executed when the end of a way element is reached. With lines 131 to 134, the collected `self.ndlist` of this way element is used on the `self.nodedict` (compare **Figure 4.5** line 72/73 and **Figure 4.3** line 34) resulting in a `self.latlist` containing all latitude values and a `self.lonlist` with all longitudes associated with this way element (PYTHON 2.7.10 LIBRARY; xml.sax).

The mean value of both latitude and longitude lists is saved to the `self.waydict` with the ID of the way as the key in line 136. If all control variables `self.plz`, `self.number` and `self.street` are `True`, the mean value of the `self.latlist` and `self.lonlist` are assumed to be the coordinates of the address and added to the `self.address` object in line 139. Then the object is appended to the

`addresslist` in line 140. After that, all the variables are reset to the default in the lines 142 to 149 (PYTHON 2.7.10 LIBRARY; xml.sax).

```

151         if name in ('relation'):
152             for member in self.memberlist:
153
154                 if member[1] == 'node':
155                     self.latlist.append(self.nodedict[member[0]][0])
156                     self.lonlist.append(self.nodedict[member[0]][1])
157                 elif member[1] == 'way':
158                     self.latlist.append(self.waydict[member[0]][0])
159                     self.lonlist.append(self.waydict[member[0]][1])
160                 elif member[1] == 'relation':
161                     self.latlist.append(self.relationdict[member[0]][0])
162                     self.lonlist.append(self.relationdict[member[0]][1])
163
164                 self.relationdict[self.relationid] = (numpy.mean(self.latlist),
                                                         numpy.mean(self.lonlist))
165
166                 if self.plz is True and self.street is True and self.number is
167                     True:
168                     self.address[0] = 'POINT(%s %s)' % (numpy.mean(self.lonlist),
169                                                         numpy.mean(self.latlist))
170                     addresslist.append(tuple(self.address))
171
172                 self.latlist = []
173                 self.lonlist = []
174                 self.waymode = False
175                 self.plz = False
176                 self.street = False
177                 self.number = False
178                 self.address = ['lat_lon', 'pcode', 'street', 'number']
179                 self.memberlist = []

```

**Figure 4.8** OpenStreetMap XML parser end element part three

The code at the end of a relation element is even more extensive than for a way. **Figure 4.8** shows the example code for this. In Line 152, every `member` of the `self.memberlist` is called. Afterwards, depending on what type of object the particular `member` references to, the corresponding dictionary is called and the latitude and longitude values appended to the respective lists are added. Because relations can be members of other relations in line 164, the mean value of the coordinates of this relation are added to the `self.relationdict` with the ID as the key. Relations in the XML file are ordered in a way that relations which are part of other relations always come before those relations which they are a part of. The rest of the code from lines 166 to 177 works similarly to the previously described code for nodes and ways (PYTHON 2.7.10 LIBRARY; xml.sax).

## 4.4. Write Addresses to a Database

All addresses are written to the PostgreSQL database with the python psycopg2 library. For this, a separate python script is written containing all the structured query language (SQL) handling parts of the code. It can be seen in **Figure 4.9**.

Saxparser file:

```
180 if __name__ == '__main__':
181     parser = make_parser()
182     parser.setContentHandler(startendfinder())
183     parser.parse('./vienna.osm')
184
185
186     DBconnector.CreateTable()
187     DBconnector.WriteToTableMany(addresslist)
```

-----  
DBconnector file:

```
001 import psycopg2
002
003
004 def DBConnect():
005
006     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres\
                                password=#####")
007
008     cur = conn.cursor()
009     return conn,cur
010
011 def CreateTable():
012
013     conn,cur = DBConnect()
014     cur.execute("CREATE TABLE IF NOT EXISTS Addresses (id serial PRIMARY KEY,\
                    geom geometry, street text, Street_number text, pcode integer,\
                    AddDate integer);")
015     conn.commit()
016     conn.close()
017
018
019 def WriteToTableMany(Values):
020
021     conn,cur = DBConnect()
022     cur.executemany("INSERT INTO Addresses (geom, street, Street_number, pcode,\
                    AddDate) VALUES (%s, %s, %s, %s, 24022015)", (Values))
023     conn.commit()
024     conn.close()
```

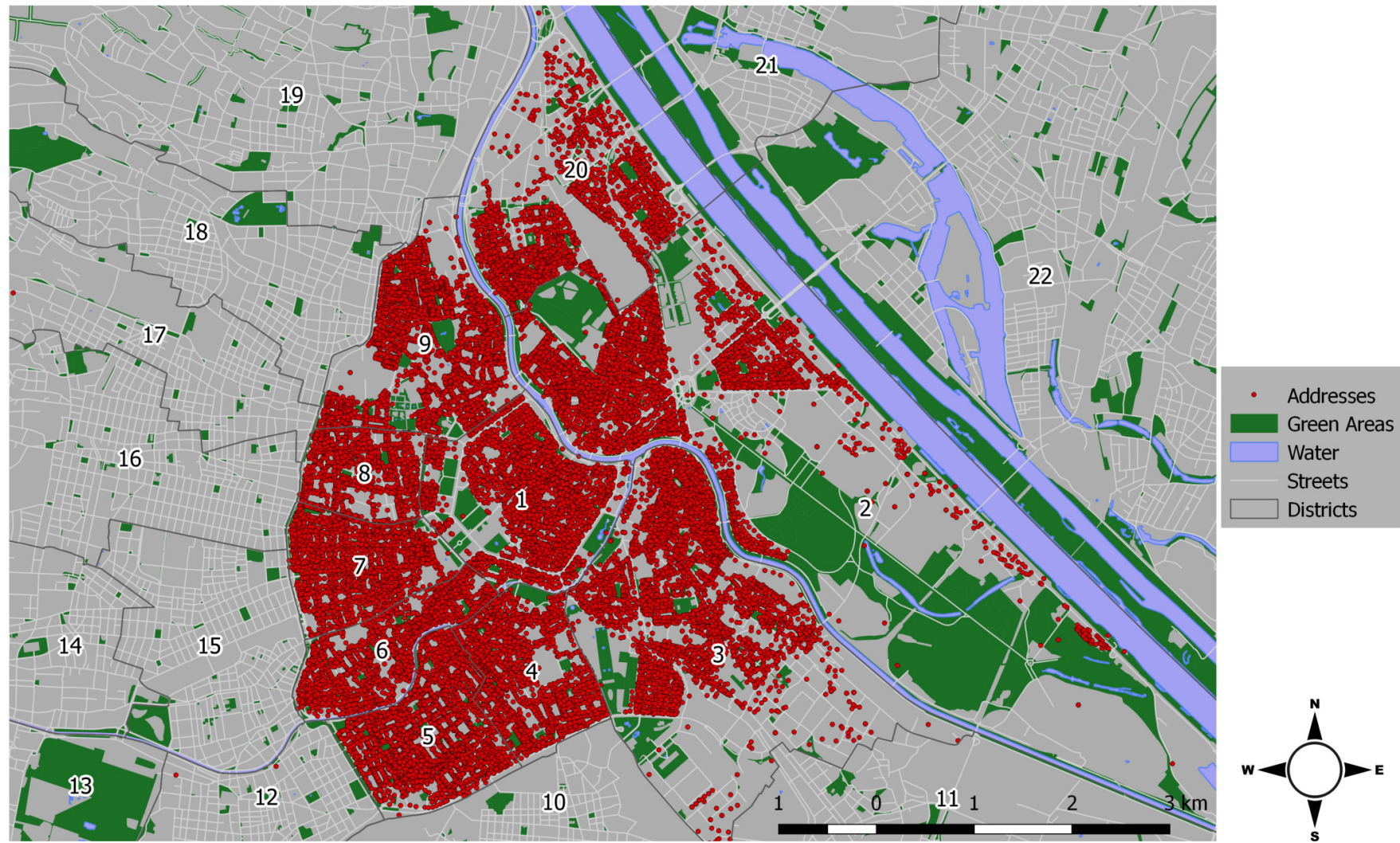
**Figure 4.9** Database Import and calling database import from the parser

The lines at the top of **Figure 4.9** are the initial ones that are executed when the parser is started. In lines 180 to 183 is all the code called discussed previously in this chapter. After these lines call this code, the `addresslist` is populated with addresses and their spatial information. What follows is the transfer of this python list to the PostgreSQL database. First with line 186, the `CreateTable()`

method shown in line 11 is called. This method in turn calls the `DBConnect()` method that creates a database connection object `conn` in line 6 and derives out of this connection object a cursor object `cur` in line 7. Both the cursor object and the connection object are returned to the `CreateTable()` method. With the cursor object, SQL can be executed on the database. So in line 14, the SQL command for the table creation is executed. Notable is the PostGIS field geometry that later holds the spatial information object for each address. The SQL command is committed with `conn.commit()` in line 15 and the connection closed in line 16 (PSYCOPG 2.5.3 LIBRARY).

`WriteToTableMany()` is then called in line 184 and the `addresslist` is passed to it. Again, a connection and cursor object are created, but this time the `cur.executemany()` function is used. This function allows an iterable python object to be passed to the `psycopg2` library. The preceding SQL-like string, with `%s` as placeholders, works as a blue print for every value in the iterable python object. After all items in the `addresslist` are inserted into the database, the changes are committed in line 15 and the connection closed in line 16. Over-all, 23,830 addresses are written to the database (PSYCOPG 2.5.3 LIBRARY).

## From Openstreetmap Extracted Addresses



Date: ETRS89 Projection: UTM 33N Sources: openstreetmap.org; open.wien.gv.at By: Alexander Czech / alexander.czech(at)gmail.com (CC-BY 4.0) 09.06.2015

**Map 4.1** Extracted Addresses

Those 23,830 addresses are visualized on **Map 4.1**. Because the addresses are derived from a Wikipedia-like source, they are most probably not complete and may contain some errors. For example, there are likely some addresses missing in the north western corner of the 20<sup>th</sup> district. When the address points are overlaid over aerial photography, there seem to be a couple of buildings without an address point even though they should have one. Even more easily, one can assume that address points outside the designated districts are mistakes. There are four in the 12<sup>th</sup> district, one in the 13<sup>th</sup> (this one cannot be seen on the map) and one in the 17<sup>th</sup>. All of them have been mistakenly added to the dataset because their post-code are incorrect.

Literature about the correctness and completeness of VGI and OSM based address dataset is sparse. There is a work by Haklay in 2010, on the overall positional accuracy and completeness that compared the Ordnance Survey meridian 2 dataset to OSM data. It shows that relatively wealthy and densely populated places are better mapped in OSM. Another study by Teske compared different geocoders, but this work is focused on how good given geocoder parses a string for an address (HAKLAY; 2010; pp. 682-703), (TESKE; 2014; pp. 161-174).

But, overall and subjectively judging the errors seem to be sparse. For this thesis a 100% complete and error-free dataset is not necessary.



## 5. Common Crawl Database Transfer

The focus of this chapter is how the now downloaded and archived files are transferred to the PostgreSQL Database. For this, the folder and file structure is examined how the ARC files are split up into individual files and how they are added to the database.

### 5.1. Folder and file structure

After downloading the Common Crawl Data to an Amazon S3 bucket the data is transferred to a local machine. The files are grouped into subfolders according to the parts of the .at TLD space they contain. A schematic example of this structure can be seen in **Figure 5.1**.

```
Data/
|
|--at.001/
| |
| | |--1.gz
| | |--2.gz
|
|--at.002/
| |
| | |--1.gz
| | |--2.gz
|
[...]
```

```
--at.zzz/
|
|--1.gz
|
|--2.gz
```

**Figure 5.1** Schematic example file structure

The <name>.gz compressed files contain an ARC File whose structure is described in Chapter 3.2 and **Figure 3.1**. Those ARC Files contain multiple files, separated only by the header of each document. So each ARC file needs to be split up again into single documents. This is what the code **Figure 5.2** is part of. It shows the first section of what is executed when the script is started. The execution starts at line 140, 141 calls a simple test which determines with which database the connection will be established. This is important because, at this point the amount of time a script takes to successfully pass is very long when all available data is used. For this reason, a smaller database containing only

about 1% of all data was created to test and develop scripts before executing them on the actual database (PSYCOPG 2.5.3 LIBRARY).

Line 144 calls the method creating the target table to which all the data will be transferred. This table has three columns, a unique ID for each HTML document, the URL of this document, and the HTML document itself saved as a string (PSYCOPG 2.5.3 LIBRARY).

```

018 def CreateTable():
019     conn, cur = DBConnect()
020     cur.execute("CREATE TABLE IF NOT EXISTS html (id serial PRIMARY KEY,
                                                         url TEXT, html_file text);")
021     conn.commit()
022     conn.close()

132 def current_database():
133     conn, cur = DBConnect()
134     cur.execute('SELECT current_database()')
135     DB_name = cur.fetchone()
136     print('#####')
137     print('Connecting to %s' % DB_name)
138     print('#####')

140 if __name__ == '__main__':
141     current_database()
142     raw_input('Please Press the anykey')
143
144     CreateTable()
145     startall = time.time()
146
147     PATH = './Data'
148     open_Paths(PATH)
149
150     print('+++++#####')
151     print('complete Operation took %s Minutes' % ((time.time() - startall) / 60))
152     print('+++++#####')
153
154     lines = ReadFromTable()
155     print(len(lines))
156     for line in lines:
157         print (line[2])
158     raw_input('Please Press the anykey')

```

**Figure 5.2** Database transfer script part one

Line 147 calls the `open_Paths()` method with the path to all data files as a string. **Figure 5.3** shows this method. Line 119 calls the `os.walk()` method with which all paths to all files in a specific directory (in this case `'./Data'`) can be created and this is what is done in line 122. The `fullpath` to a file is then handed over to the `gzip.open()` method that unpacks the file and returns the unpacked file to the variable `file`. `file` is then passed on to `Database_export()`. This method will divide the file into individual HTML documents that can then be passed to the database (PSYCOPG 2.5.3 LIBRARY), (Python 2.7.10 library; operating system), (Python 2.7.10 library; gzip).



```

118 def open_Paths(PATH):
119     for path, dirs, files in os.walk(PATH):
120         for filename in files:
121             try:
122                 fullpath = os.path.join(path, filename)
123                 print('#####')
124                 print('%s Size: %s MB' % (fullpath,
                                         os.path.getsize(fullpath) / 1048576))
125                 file = gzip.open(fullpath, 'rb')
126                 Database_export(file)
127                 file.close()
128             except:
129                 pass

```

**Figure 5.3** Database transfer Script open\_Paths() method

## 5.2. Subdivide files into individual HTML files

The `Database_export()` function is quite complex. The reason for this is the file's internal structure. Every file is comprised of either one or a few hundred or thousands of arc files in text format. Thus, it is basically a very long text file, which has to be processed line for line. These lines are analyzed to determine if they belong to the current file or the following one. Parts of the code for this process can be seen in **Figure 5.4**.

```

041 def Database_export(file):
042     start1 = time.time()
043     i = ''
044     tup_i = ()
045     url = ''
046     html_written = 0
047     mode = False
048     for line in file.readlines():
049         splitline = line.split(' ')
050         try:
051             if splitline[0][:7] == 'http://' and splitline[3] == 'text/html'
052                                                         and len(splitline) == 5:
053
054                 if len(i) > 0:
055                     tup_i = tup_i + ((url, i),)
056                     url = ''
057                     i = ''
058
059                 url = splitline[0]
060                 mode = True
061
062                 if len(tup_i) >= 100:
063                     print('empty tup_i')
064                     try:
065                         WriteManyToTable(tup_i)
066                     except:
067                         pass
068                     tup_i = ()
069                     print("html files written to DB %s" % html_written)
070
071                 html_written += 1

```

**Figure 5.4** Transfer Script Database\_export() method part one

Lines 42 to 47 define several controlling variables. Most important are `i` and `tup_i`. The variable `i` will contain all lines of the current HTML document, while `tup_i` holds the already parsed HTML documents. These are subdivided into tuples of the specific URL for this document and the content of the document itself as a string.

The file that was passed on from the `open_Paths()` method is then split into individual lines in line 48. Every line is again split at every space in line 49. This split line contained in the variable `splitline` is then tested in line 51 if it is the first line of a new document. If so, the current document along with the URL of this document is added to `tup_i` as a tuple in 54. If there is a document currently contained within `i` the `if` statement in line 53 is executed. The `if` statement tests if there is a document contained within `i` by checking the length of `i`. The document and its corresponding URL are added to `tup_i` in line 54 and `url` and `i` reset (PYTHON 2.7.10 LIBRARY; string). The newly current URL is saved to `url` and `mode` is set to `True`. What follows in line 61 is a test if the amount of already parsed HTML documents contained in `tup_i` has crossed a certain threshold. If so the HTML documents are passed on to the `WriteManyToTable()` method which will be discussed later in detail and `tup_i` is set to an empty tuple.

```

092         if mode == True:
093             try:
094                 i += unicode(line, "utf-8")
095             except:
096                 #i += UnicodeDammit(line).unicode_markup
097                 pass

```

**Figure 5.5** Transfer Script Database\_export() method part two

**Figure 5.5** skips a couple of lines to **Figure 5.4**, which will be discussed later. If `mode` was set to `True` in line 59, then all of the following lines until the next document is encountered will be appended to `i` in line 94. More precisely, a UTF-8 (**U**niversal **C**oded **C**haracter **S**et + **T**ransformation **F**ormat—**8**-bit) version of the `line` is appended to `i`. Text characters can be encoded in several different codecs. UTF-8 is one that strives to make it possible to encode all possible know characters. This step is necessary because all documents are encoded in a variety of codecs, but the database expects only UTF-8 encoded strings (PYTHON 2.7.10 LIBRARY; Unicode), (BEAUTIFUL SOUP 4.3.2 LIBRARY;).

Because certain characters still cannot be encoded into UTF-8, the code sometimes fails to convert a line. This is why the code needs to be in a `try except` block. The original intent was also to convert failed lines to UTF-8 with line 96, but this simply takes up too much calculation time, thereby

stretching the length it takes to process all files from hours to weeks. Thus, all lines that cannot be converted are not included and ignored (UNICODE 7.0.0).

```

072         elif splitline[0][:7] == 'http://' and \
073             splitline[3] != 'text/html' and len(splitline) == 5:
074             if len(i) > 0:
075                 tup_i = tup_i + ((url, i),)
076                 url = ''
077                 i = ''
078
079             mode = False
080
081             if len(tup_i) >= 100:
082                 print('empty tup_i')
083                 try:
084                     WriteManyToTable(tup_i)
085                 except:
086                     pass
087                 tup_i = ()
088                 print ("html files written to DB %s" % html_written)

```

**Figure 5.6** Transfer Script Database\_export() method part three

In **Figure 5.6** we see the case where the script detects the beginning of a new document that is not an HTML text file, but something else, for example a picture, a PDF or a Microsoft Word file. When this is the case in the ARC file, the binary data of such files is encoded to text. Even though it would theoretically be possible to read out a PDF or a Microsoft Word file or other type of text file with suitable python libraries it is too time consuming to do so.

Whenever the parser meets a line of a new document that is not an HTML text file the code in **Figure 5.6** is executed and ignores this file. Again there is the test to determine if a current document exists in line 74, and if so, the document, with its URL, is appended to `tup_i` and the variables `url` and `i` are reset. The `mode` is set to `False`, which has the effect that all of the following lines will not be saved to the now empty variable `i`. And if `tup_i` crosses the threshold of 100 collected HTML documents, those are passed to the `WriteManyToTable()` method and `tup_i` is set to an empty tuple (PYTHON 2.7.10 LIBRARY; Unicode), (PYTHON 2.7.10 LIBRARY; string).

```

099         if len(i) > 0:
100             tup_i = tup_i + ((url, i),)
101
102         try:
103             WriteManyToTable(tup_i)
104         except:
105             pass

```

**Figure 5.7** Transfer Script Database\_export() method part four

Part four shown in **Figure 5.7** concludes what is left. Because the end of the ARC file no longer contains a new document header, `i` and `url` are appended to `tup_i` and `tup_i` one last time, no matter how many documents it contains is passed to the `WriteManyToTable()` method.

#### 4.3. Transfer to Database

The `WriteManyToTable()` method can be seen in **Figure 5.8**. The function makes use of the `.mogrify()` method of the cursor object in line 27, a function that works similarly to the `.execute()` method of the cursor object but without executing the SQL statement on the database. Instead, it just forms the SQL statement with the given parameters. What line 27 now does is iterate through the `Values` variable, which contains all the URL and HTML document tuples, which were formerly known as `tup_i`, and creates one long SQL statement with all 100 URLs and HTML documents contained in it. This statement is then merged in Line 28 with the front part, forming a complete statement that is executed on the database inserting all 100 URLs and HTML files into it. The transaction is committed in line 29 and the connection is closed in line 30. Overall 8,406,507 HTML documents are written to the database.

```
025 def WriteManyToTable(Values):
026     conn, cur = DBConnect()
027     args_str = ', '.join(cur.mogrify("(%s,%s)", x) for x in Values)
028     cur.execute("INSERT INTO html (url, html_file) VALUES " + args_str)
029     conn.commit()
030     conn.close()
```

**Figure 5.8** WriteManyToTable() method

## 6. HTML Tag Stripper

This chapter is about creating a more refined and relevant subset of the 8,406,507 HTML documents and how to strip this subset of all HTML tags and other irregularities and write it back to the database.

### 6.1. Find Vienna

Since not all of the 8.4 million documents are going to be geotagged, it is prudent to only geotag those documents which are most likely going to be geotagged and exclude those which will never be geotagged to an address in Vienna. A simple way of doing this is to mark all documents in which the term 'Wien' appears at least once. That is what the code in **Figure 6.1** demonstrates.

```

009 def addvienna():
010     conn, cur = DBConnect()
011     cur.execute("ALTER TABLE html ADD COLUMN Vienna BOOLEAN;")
012     conn.commit()
013     conn.close()
014     return
015
016
017 def setvienna(lower, upper):
018     conn, cur = DBConnect()
019     cur.execute("""UPDATE html SET Vienna = TRUE WHERE html_file LIKE
                    '%%' || ' %s ' || '%%' AND ID >= %s AND ID < %s RETURNING ID;""" %
                    ('Wien', lower, upper))
020
021
022     conn.commit()
023     conn.close()
024     return
025
026
027 addvienna()
028 lower = 0
029 increment = 10000
030 starttime = time.time()
031 while lower <= 8406507:
032     setvienna(lower, lower+increment)
033     delta_time = time.time() - starttime
034     print lower+increment
035     print "time till now %.2f Minutes" % (delta_time / 60)
036     print "time till end %.2f Minutes" %
037         (((delta_time/60)/(lower+increment))*(8406507-(lower+increment)))
037     lower += increment

```

**Figure 6.1** Set Vienna

First, there needs to be a column in the html table that can tell us if the string 'Wien' occurs in the document. This is done by the `addvienna()` method called in line 27. Next, the code iterates through all IDs and thus documents in chunks of 10,000. For this, an initial lower end is set in line 28 and the increment size in line 29. The `while` loop in 31 is executed as long as `lower` is smaller than 8,406,507, the number of HTML documents contained in the HTML table. `lower` and `lower` plus `increment` of 10,000 are passed to the `setvienna()` method. The purpose of `setvienna()` is to execute an SQL statement that looks through a range of documents and tests these documents with the `LIKE` operator. The `LIKE` operator is a string matching operator and the string it tries to match is 'Wien'. In this case, the string is preceded and followed by a wildcard. All documents where the operator matches the column `Vienna` are set to `TRUE`. In total, 698,524 documents are matched and set to `TRUE` (PSYCOPG 2.5.3 LIBRARY), (POSTGRESQL 9.3.9).

## 6.2. Remove HTML Tags

After all documents are marked, the next step is to remove all HTML tags. For this, regular expressions are used. Regular expressions are sequences of characters that match a certain pattern in a string. The HTML table also needs a column to accommodate documents without HTML tags. In line 69 of **Figure 6.2**, the method `createColumn()` is called and creates such a column. The `DROP COLUMN` SQL statement is in there because, like all the code, also this part is developed by trial and error. It turns out to be much faster to drop a column and then recreate it than to overwrite an old, incorrect column with correct values. `strippedlist` created in line 71 will contain the processed documents before they are written to the database. Then in line 74, the variable `htmls` is populated with the first documents. For this, the `ReadFromHTML()` function is called (PSYCOPG 2.5.3 LIBRARY). Because now only those documents where the column `Vienna` is set to true are of interest, instead of a range of IDs, the limit and offset operators are used in the SQL statement. The table is ordered by id and then the function of offset is to ignore the first n rows defined by the variable `offset`. Limit defines how many rows are returned in total. When offset and limit are used in combination, like in line 16, and offset is iterated higher and higher (see line 104), the database returns the first thousand rows, then the next thousand rows and so on. Consistency of order is guaranteed because the table is always ordered the same way, by id. With all this set up, the code enters the `while` loop. This loop gets executed as long as `htmls` is true and `htmls` is true as long as the database returns documents with the just described SQL statement (see line 103 and 104). The database ceases to do so as soon as offset is higher than the amount of documents where the column `Vienna` is true (PSYCOPG 2.5.3 LIBRARY), (POSTGRESQL 9.3.9).

```

014 def ReadFromHTML(offset):
015     conn, cur = DBConnect()
016     cur.execute("SELECT id,html_file FROM html WHERE vienna = TRUE ORDER BY id
                                limit 1000 offset %s ;" % offset)
017     data = cur.fetchall()
018     cur.close()
019     conn.close()
020
021     return data

032 def createColumn():
033     conn, cur = DBConnect()
034     cur.execute("ALTER TABLE html DROP COLUMN IF EXISTS stripped_html;")
035     conn.commit()
036     cur.execute("ALTER TABLE html ADD COLUMN stripped_html TEXT;")
037     conn.commit()
038     cur.close()
039     conn.close()

069 createColumn()
070 Starttime = time.time()
071 strippedlist = []
072 offset = 0
073 Starttime2 = time.time()
074 htmls = ReadFromHTML(offset)
075 Numberofrows = 8406507
076
077
078
079 while htmls:
080
081     timeregex = time.time()
082     print("starting Regex")
083     for row in htmls:
084         id = row[0]
085         stripped_html = remove_tags(row[1])
086         strippedlist.append((stripped_html, id))
087     print('Regex took %.2f Minutes' % ((time.time() - timeregex) / 60))

    [...]

103     offset += 1000
104     htmls = ReadFromHTML(offset)

```

**Figure 6.2** Fetching HTML documents to strip tags

The `ReadFromHTML()` function returns a list of tuples containing the ID of the row and the content of the `html_file` column. The `for` loop in line 83 iterates through this list and passes one HTML document after another to the `remove_tags()` method shown in **Figure 6.3** (PSYCOPG 2.5.3 LIBRARY).

```

042 def remove_tags(text):
043     text = TAG_RE.sub('', text)
044     text = Short.sub('', text)
045     text = eszt.sub('ß', text)
046     text = ae.sub('ä', text)
047     text = AE.sub('Ä', text)
048     text = oe.sub('ö', text)
049     text = OE.sub('Ö', text)
050     text = ue.sub('ü', text)
051     text = UE.sub('Ü', text)
052     return text
053
054
055
056
057 TAG_RE = re.compile(r'<[^\>]+>')
058 Short = re.compile(r'\S{68,}')
059 eszt = re.compile(r'&szlig;')
060 ae = re.compile(r'&auml;')
061 AE = re.compile(r'&Auml;')
062 oe = re.compile(r'&ouml;')
063 OE = re.compile(r'&Ouml;')
064 ue = re.compile(r'&uuml;')
065 UE = re.compile(r'&Uuml;')

```

**Figure 6.3** Regular expression tag stripper

Regular expressions are created to match certain string patterns. This ability can be used to find all HTML tags in a string and then replace them. The pattern of HTML tags is that they open with a “<” and close with a “>” and have a variable amount of characters in between. The regular expression defined in line 57 matches this pattern exactly. The `r'` in front of the string means that special characters and character combinations, like for example `\t` for tab stop, are ignored and are interpreted as `\t` and don’t need to be escaped. Then follows the first character of the pattern `<`. The `+` character indicates that a variable amount characters follow the “<”. But “>” is excluded from this with the part of the pattern `[^\>]`. Finally, the pattern ends with the `>`. So this regular expression matches every part of a string that begins with a “<” ends with a “>” and has more than one character in between that is not a “>”. All of this is compiled into the `TAG_RE` variable for later use (PYTHON 2.7.10 LIBRARY; regular expression operations).

The regular expression compiled in 58 to the variable `Short` matches every string that is 68 characters or longer because there are a lot of nonsensical strings in the documents. So `\s` matches every non-whitespace character and `{68,}` defines that the strings can be 68 characters or longer. The longest German word excluding numerals is, according to Duden, “Grundstücksverkehrsgenehmigungs-zuständigkeitsübertragungsverordnung” which is 67 characters long (DUDENKORPUS), (PYTHON 2.7.10 LIBRARY; regular expression operations).



Because many HTML documents are encoded in American standard codec two (ASCII) and there are no provisions in it for German special characters, HTML uses character entity names for those special characters. Now that all documents have been transferred to UTF-8, there is no longer a need for this provision and all character entity names can be changed to the correct characters. So the regular expressions in line 59 to 65 match the corresponding character entity names so that they can be replaced with the character (BRAY ET AL, 2008), (PYTHON 2.7.10 LIBRARY; regular expression operations). All those regular expressions are called up one by one in the `remove_tags()` method. The method gets the document passed on to it in line 85 **Figure 6.2** as one continuous string. Everything that matches within the string is replaced by a defined other string. So `TAG_RE.sub('', text)` replaces everything in `text` that matches with the regular expressions saved to `TAG_RE` with an empty string. Similar to that `ue.sub('ü', text)` everything matching the regular expression contained in `ue` is replaced by "ü" (PYTHON 2.7.10 LIBRARY; regular expression operations).

The cleaned string is returned by the `remove_tags()` method and appended along with `id` to `strippedlist` lines 85 and 86 **Figure 6.2**. When all HTML documents within `htmls` have been processed, `strippedlist` is passed on to `UpdateHTMLwithStrippedHTML()` (PSYCOPG 2.5.3 LIBRARY).

```
024 def UpdateHTMLwithStrippedHTML(Values):
025     conn, cur = DBConnect()
026     cur.executemany("UPDATE html SET stripped_html = %s WHERE vienna = TRUE AND id
                                in (%s)", Values)
027     conn.commit()
028     cur.close()
029     conn.close()
```

**Figure 6.4** Writing stripped HTML documents to the database

The Method shown in **Figure 6.4** just contains an `cur.executemany()` where the `id` contained in the `strippedlist` defines in which row the `stripped_html` column is updated. With all of this information, the HTML documents containing the string `'Wien'` are stripped of their html tags, character entity names, other irregularities, and written back to the database (PSYCOPG 2.5.3 LIBRARY).

## 7. Geotagging

The focus of this chapter is on how to geotag all those websites and how to do it in a reasonable time span. For this, there is a brief introduction into how indexing large datasets in PostgreSQL works, followed by how all of this is applied to geotag websites in the thesis. The last part of this chapter is a brief interpretation of the first map produced with this method.

### 7.1. Creating an index in PostgreSQL

At this point, the amount of data is down to around 700,000 HTML documents. This is still too much information to pattern match 24,000 addresses against those 700,000 documents because, even if it only took on average 1 ms to test one address against one document, the whole process would still take 194 days. So the first thing that needs to be done is create a search index on the HTML documents.

#### 7.1.1. Converting a text to a list of stemmed tokens

To create such an index, the document needs to be converted into tokens and those tokens need to be stemmed. Tokenization is the process of chopping a document into pieces of character sequences called tokens. Tokens can be loosely understood as the words that make up a document, but there are other cases, for example, dates like 1/1/1970, that can be understood as a token. An example for tokenization would be:

How long, O Catiline, will you abuse our patience?

Tokenized: How long O Catiline will you abuse our patience

In this example, the process of tokenization simply divided the words at whitespaces and eliminated the punctuation. However, some tokenizations are more complicated. Take, for example, “Mr. O’Neill thinks that the boys’ stories about Chile’s capital aren’t amusing.” Finding the correct tokenization here is more difficult because what is the correct tokenization of O’Neill: neill, oneill, o’neill, o’ neill, o neill, or of aren’t: aren’t, arent, are n’t, aren t (MANNING ET AL; 2009; pp. 22-24)?

The most common strategy tokenization algorithms use on this problem is to always split on none alphanumeric characters. Most tokenization algorithms also allow for provisions, depending on the language. But splitting on white spaces can also cause problems, for example, for a group of words that should be treated as one token. This can lead to bad search results such as when a search for

“York University” only returns results for “New York University”. A challenge that is specifically numerous in the German language is compound nouns. An example of this is *Lebensversicherungsgesellschaftsangestellter* (life insurance company employee), which contains four nouns. Search results are greatly improved when a compound splitter is used that subdivides compound nouns into multiple tokens. But regardless of how the tokenization algorithm works, it is imperative that the same algorithm is used for the documents and the search terms (MANNING ET AL; 2009; pp. 22-25).

Following the tokenization of the documents, it is important to drop stop words. Stop words are words that are so common in a language that they hold little to no value when selecting one document over another. Examples of this in the English language are: a, but, by, for, had, I, most, and so on. A stop wordlist can either be generated by counting the frequency of all words in a corpus and hand-selecting the words that go on the list out of the most frequent ones, or, as in the case of this thesis, the predefined stop word list of PostgreSQL can be used. With the help of a stop word list, the amount of postings that the database needs to store can be significantly reduced. The length of a stop list does vary from a very long list with 200 to 300 terms to small list with only 7 to 12 terms. Modern web search engines don’t use a stop list. As with the tokenization, if a stop list is used for documents, it is important that the same list is also used for search terms (MANNING ET AL; 2009; pp. 27-28).

Next comes token normalization. Normalization is used to make two character sequences that are not quite the same, but have the same meaning match, for example USA and U.S.A. One way to accomplish this is by using equivalence classes. For this method, all terms that are put together in one class are mapped to the same token. There are a couple of different approaches to create these equivalence classes, one is to replace all accents, diacritics, and, in the case of German, ß, are replaced by corresponding ASCII characters. Even though diacritics are in many cases the only distinguishing factor between two different meanings for a group of characters, the reason why this is still done is because many users tend to not use them when they use a search engine (MANNING ET AL; 2009; pp. 28-29).

Case-folding is another technique used to normalize tokens. In this strategy, all letters of a token are reduced to lower case. This, for example, allows “Automobile” written at the beginning of a sentence and therefore capitalized to also match the query “automobile”. It also helps with users’ search queries that misspell or incorrectly capitalize words. But this also creates problems because a

lot of proper nouns are derived from common nouns, capitalization being the only distinguishing factor between the two, for example, in company names (General Motors, The Associate Press), government organizations (the Fed vs. fed), and people's names (Bush, Black) (MANNING ET AL; 2009; pp. 30).

Truecasing is an alternative to case-folding in English. Instead of making all tokens lowercase, only some tokens are made lowercase. The simplest rule here is to make all tokens that are at the beginning of a sentence and all words occurring in a title lowercase. Words that are in the middle of a sentence are left capitalized. In most cases that will keep the distinction between to words. This method can be improved by machine learning algorithms that then take much more than only those basic heuristics into account. But also to mitigate for user input errors case-folding is still the most practical solution (MANNING ET AL; 2009; pp. 30).

The last step in the process is to stem or lemmatize a token. Both techniques try to accomplish the same reduction of a token to the base form of a word. Words differ for grammatical reasons, for example, "organize", "organizes", and "organizing", is the same word in different grammatical contexts. Also there can be derivationally related words that have similar meanings such as democracy, democratic, and democratization. From the perspective of a search engine user, it is preferable that in both cases the engine would consider words of all sets to generate results. So the goal of stemming and lemmatization is the same, to relate tokens to a common base form. In English for example:

am, are, is -> be

car, cars, car's, cars' -> car

If used on a complete sentence the results could look something like this:

The boy's cars are different colors -> the boy car be differ color

The difference between the two is how they try to accomplish this goal. Stemming is mostly a heuristic process that chops off the end of a word by a set of rules which try to achieve a base form of a word. While lemmatization works with a proper dictionary and morphological analysis of words in the aim to only remove the inflectional endings and return the base dictionary form of a word known as a lemma (MANNING ET AL; 2009; pp. 32-35).

To demonstrate the difference between the two, let us compare them through the token "saw". Using a stemmer on the token might just return "s", while the lemmatization of the word would either return "see" or "saw" depending on whether, in the context, it is a noun or a verb. While stemming does increase the recall of a search engine, it does also lose precision. Lemmatization increases precision, but reduces recall.

For this step, the thesis is bound to the tokenization process of PostgreSQL. PostgreSQL uses a stemmer for the practical reason that lemmatization needs complex and also time-intensive morphological language models (MANNING ET AL; 2009; pp. 32-35).

The option PostgreSQL leaves for tokenization is to provide a language. So a PostgreSQL query like this:

```
SELECT * from to_tsvector('german', Die Aufklärung, welche die Freiheiten
entdeckt hat, hat auch die Disziplinen erfunden.)
```

uses tokenization, normalization, removing stop words, case-folding, and stemming to create a result like this:

```
""aufklar':2 'disziplin':11 'entdeckt':6 'erfund':12 'freiheit':5"
```

Only the base words and their position within the original string are preserved (POSTGRESQL 9.3.9; documentation).

### 7.1.2. Creating a Token Index

Even though with tokenization the amount of data can be reduced, there is still a need for an index to search quickly through the tokens. PostgreSQL offers two types of Indexes for tsvector columns: a Generalized Search Tree (GiST) based index and a Generalized Inverted Index (GIN) based index. The GiST index is described as “lossy,” which means that the index itself may produce false matches for tokens. This makes it necessary to check the search term against the actual tokens of the matches produced by the index, which in turn slows the query speed down (POSTGRESQL 9.3.9; documentation).

A GIN index is not lossy, but its performance depends logarithmically on the number of unique tokens. In general, the following performance differences occur between the two types:

- GIN lookups are comparatively about three times faster.
- It takes about three times longer to build a GIN index.
- It is slower to update a GIN index compared to a GiST index.
- GIN indexes are about two or three times larger than GiST ones.

Because the data is static and there will be about 21k queries, one for every address, the GIN Index will be used (POSTGRESQL 9.3.9; documentation).

```

197 def CreateIndex():
198     conn, cur = DBConnect()
199
200     cur.execute("ALTER TABLE html ADD COLUMN textsearchable_index_col tsvector;")
201     conn.commit()
202     cur.execute("UPDATE html SET textsearchable_index_col = to_tsvector('german',
                                                                    stripped_html) WHERE Vienna = True;")
203
204     conn.commit()
205     cur.execute("CREATE INDEX textsearch_idx ON html USING
                                                                    gin(textsearchable_index_col);")
206
207     conn.commit()
208     cur.close()
209     conn.close()

```

**Figure 7.1** Index Creation

The whole process of how the index is created in the database is shown in **Figure 7.1**. First, the column of data type `tsvector`, `textsearchable_index_col`, is created in line 200. Then, in line 202, a `tsvector` is created from the content in the column `stripped_html` for all rows where Vienna is set to true. Lastly, in line 204, the index is created on the `textsearchable_index_col` column. Now it is possible to search through the column `stripped_html` without the slow pattern matching operator LIKE (PSYCOPG 2.5.3 LIBRARY), (POSTGRESQL 9.3.9; documentation).

## 7.2. Create a unique set of Addresses and prepare them for Search Queries

Since there are a variety of rules of how to tag addresses in OpenStreetMap and there is no consensus in the community, addresses can exist multiple times in the dataset. Because, for example, they are attached to every entrance of a building or tagged once just to a node and then to a way representing a building or, if addresses apply to multiple buildings, every building can have the address or just a relation that encapsulates all those buildings and so on. But for the geotagging process, only one instance of every address is needed. Even though it would not make a difference to look for the same address multiple times, it would increase the amount of necessary queries (OPENSTREETMAP WIKI; Addresses).

PostgreSQL provides a good way to make the addresses unique with an SQL command. With `SELECT DISTINCT ON`, one field or more that must be unique within the selection can be selected. The best way to progress from that is to transfer this unique set into a new table and this is what the code in **Figure 7.2** does.

```

054 def MakeAddressesUnique():
055     conn, cur = DBConnect()
056
057     conn.commit()
058     cur.execute("INSERT INTO
                  AddressesUnique (geom, street, street_number, pcode, AddDate) "
059                  "SELECT DISTINCT ON (street, street_number)
                  geom, street, street_number, pcode, AddDate FROM Addresses")
060     conn.commit()

```

**Figure 7.2** Populate Table AddressesUnique

The SQL statement inserts `AddressesUnique`, the selection of rows that are distinct in the columns `street` and `street_number`, into the table. The table `AddressesUnique` was created beforehand. As a result, the number of rows is reduced from 23830 to 21246 (PSYCOPG 2.5.3 LIBRARY).

The now unique addresses are read from the newly created and populated table, but in order to be suitable for a search query, some of them need to be modified. Most of this has to do with how the tokenization and stemming process works in PostgreSQL (POSTGRESQL 9.3.9; documentation).

```

157 def CleanStrings(lines):
158     for row in lines:
159         StreetName = row[0]
160         StreetNumber = row[1]
161         ID = row[2]
162         p = re.compile(r' ')
163         q = re.compile(r'[^-/a-zA-Z0-9_ ]')
164         r = re.compile(r'[0-9a-zA-Z] [-/] [0-9a-zA-Z]')
165         s = re.compile(r'')
166
167         if r.match(StreetNumber):
168             StreetNumber = p.sub(' ', StreetNumber)
169             StreetNumber = q.sub(' ', StreetNumber)
170             StreetName = s.sub(' ', StreetName)
171             row3 = p.sub(' & ', StreetName)
172             row4 = p.sub(' & ', StreetNumber)
173             lines.remove(row)
174             lines.insert(0, (StreetName, StreetNumber, ID, row3, row4))
175
176     return lines
177

```

**Figure 7.3** Preparing addresses for search queries

What can be seen in **Figure 7.3** is not only the preparation for the full text search query, but also for a following `LIKE` query. Again, regular expressions are used to manipulate the strings. All modifications have been developed by trial and error to make the addresses work with the various database queries. The regular expression in line 180 matches patterns like “8 – 9”, “4a – g”, and “7 / 8”, when there are spaces in between three defined character groups. This regular expression is used in line 167 to identify street numbers with these patterns and check if they match the pattern

the spaces in line 168 that are replaced with nothing. This creates “8-9”, “4a-g”, and “7/8” when applied to the above examples. This step is necessary because otherwise the symbols would be transformed into separated tokens. The regular expression defined in line 163 matches parenthesis and is applied to street numbers in line 169. The code removes the parentheses. This is necessary because parentheses create a lot of trouble in SQL statements. The expression in line 165 is designed for only one particular street in Vienna with the name D’Orsay-Gasse. The inverted comma in the name needs to be escaped because it also disrupts SQL statements. The expression is applied in line 160 replacing every single inverted comma with two inverted commas, thus escaping it in a SQL statement. Lastly, the expression defined in line 162 matches all spaces. The application of this expression in lines 171 and 172 is with the full text search already in mind. This is because tokens can be joined with an ampersand, creating only matches on documents if both tokens exist within the document. This is necessary for street names like “Kärtner Ring” or some street numbers named for example “Objekt 11”. Note that examples like “Objekt 11” are not changed in line 168 because they don’t fit the pattern defined in line 169 (PYTHON 2.7.10 LIBRARY; regular expression), (OPENSTREETMAP WIKI; Addresses).

### 7.3. Preparing the SQL Statement for Geotagging

Because the SQL statement for finding addresses within the HTML documents is relatively complex and considers possible abbreviations of an address, they are created in Python before they are executed on the database. The function responsible for forming SQL statements out of the address list created with the `CleanStrings()` method depicted in **Figure 7.3** is the `ConstructSQLStatmentSearchAddresses()` partly shown in **Figure 7.4**.



```

077 def ConstructSQLStatmentSearchAddresses(Values):
078     SQLStatmentdict = {}
079     conn, cur = DBConnect()
080
081     for line in Values:
082
083         if line[0][-6:] == 'traße':
084             SQLStatmentdict[line[2]] = cur.mogrify(
085                 "Select ID FROM HTML WHERE "
086                 "Vienna = TRUE AND "
087                 "(textsearchable_index_col @@ to_tsquery('german',
088                  '"+line[3]+' & '"+line[4]+"') AND "
089                 "stripped_html ILIKE '% "+line[0]+' '+line[1]+' %') "
090                 "OR "
091                 "(textsearchable_index_col @@ to_tsquery('german',
092                  '"+line[3]+' & '"+line[4]+"/*') AND "
093                 "stripped_html ILIKE '% "+line[0]+' '+line[1]+'/*') "
094                 "OR "
095                 "(textsearchable_index_col @@ to_tsquery('german',
096                  '"+line[3][:-4]+' & '"+line[4]+"/*') AND "
097                 "stripped_html ILIKE '% "+line[0][:-4]+' '+line[1]+'/*') "
098                 ";")

```

**Figure 7.4** SQL Statement Construction part one

This first part of the SQL statement construction in **Figure 7.4** shows the constructions if the address ends in “straße”. But before that, the `SQLStatmentdict`, a python dictionary, is created and will contain all SQL statements with the ID of the address as the key at the end. Even though the `ConstructSQLStatmentSearchAddresses()` function does not write anything to the database, a database connection is established in line 79 because the `.mogrify()` method of the cursor class is needed. Then starting in line 81, the function iterates through previously prepared addresses (PSYCOPG 2.5.3 LIBRARY).

The address is tested if it ends in “straße” in line 83. Using “straße” instead of “Straße” as a test ensures that words where “Straße” is a part of a word, like in “Hauptstraße,” or if “Straße” stands on its own are included in this `if` clause (PYTHON 2.7.10 LIBRARY; string).

What follows is a complex SQL statement that can be broken down into four blocks separated by the `OR`’s in the statement in lines 89, 92 and 95. The outer part of lines 85 and 86 are a `Select` for an `ID` from the `HTML` table where the field `Vienna` is set to `TRUE` and one of the four blocks of the inner part is true as well. All blocks consist of a query to the index described in chapter 7.1., and an `ILIKE` operator query on the table field containing the text that is stripped from HTML tags. Compared to the `LIKE` operator, the `ILIKE` operator also considers upper- and lowercases of a word. How one of the blocks works is that it narrows the possible documents down to only a handful with the help of the index query. Then to make sure that the document truly contains the address, the `ILIKE` query is

performed on the set of address selecting only those documents that clearly match the address. To give an example for “Kärntner Straße 37” the sql statement for line 87 and 88 would look like this:

```
"(textsearchable_index_col @@ to_tsquery('german','Kärntner & Straße & 37') AND
stripped_html ILIKE '% Kärntner Straße 37 %')"
```

The `to_tsquery` would match every document that contains all of the given words while `ILIKE` would only match the documents that contain this exact pattern of characters (POSTGRESQL 9.3.9; documentation).

The other blocks separated by `OR` work with possible abbreviations or deviations in the address pattern. The block in lines 91 and 92 appends a slash to the street number in the `to_tsquery` and `ILIKE` queries. This part of the query now also catches patterns in documents that not only specify the street number, but also the door number in the building or the staircase or both. The slash is directly followed by a wild card in both queries. The block in line 93 and 94 then works with the possible abbreviation of “straße” as “str.,” thereby, basically removing the last 4 letters of “straße” and adding a dot. Other than that, it is similar to the first block in lines 87 and 88. The last block in lines 96 and 97 combines the abbreviation of “straße” with the slash added to the street number (POSTGRESQL 9.3.9; documentation), (PSYCOPG 2.5.3 LIBRARY).

```

100     elif line[0][-4:] == 'asse':
101         SQLStatmentdict[line[2]] = cur.mogrify(
102             "Select ID FROM HTML WHERE "
103             "Vienna = TRUE AND "
104             "(textsearchable_index_col @@ to_tsquery('german',
105              '"+line[3]+' & '"+line[4]+"') AND "
106             "stripped_html ILIKE '% "+line[0]+' '"+line[1]+" %')"
107             "OR "
108             "(textsearchable_index_col @@ to_tsquery('german',
109              '"+line[3]+' & '"+line[4]+"/*') AND "
110             "stripped_html ILIKE '% "+line[0]+' '"+line[1]+"/*') "
111             "OR "
112             "(textsearchable_index_col @@ to_tsquery('german',
113              '"+line[3][:-4]+' & '"+line[4]+"') AND "
114             "stripped_html ILIKE '% "+line[0][:-4]+' . '"+line[1]+" %')"
115             "OR "
116             "(textsearchable_index_col @@ to_tsquery('german',
117              '"+line[3][:-4]+' & '"+line[4]+"/*') AND "
118             "stripped_html ILIKE '% "+line[0][:-4]+' . '"+line[1]+"/*') "
119             ";")
120
121     else:
122         SQLStatmentdict[line[2]] = cur.mogrify(
123             "Select ID FROM HTML WHERE "
124             "Vienna = TRUE AND "
125             "textsearchable_index_col @@ to_tsquery('german',
126              '"+line[3]+' & '"+line[4]+"') AND "
127             "stripped_html ILIKE '% "+line[0]+' '"+line[1]+" %'"
128             "OR "
129             "textsearchable_index_col @@ to_tsquery('german',
130              '"+line[3]+' & '"+line[4]+"/*') AND "
131             "stripped_html ILIKE '% "+line[0]+' '"+line[1]+"/*'"
132             ";")
133
134     return SQLStatmentdict

```

**Figure 7.5** SQL Statement Construction part two

The remaining part of the `ConstructSQLStatmentSearchAddresses()` function depicted in **Figure 7.5** works similarly to the just described part. The part from lines 100 to 115 works exactly like the one described before with the only difference being that it is for street names ending in “gasse”. The last part within the `else` clause catches all names that neither end in “gasse” nor “straße”. Compared to the other two, it does not include possible abbreviations of the street name in the SQL query, only the slash deviations (POSTGRESQL 9.3.9; documentation), (PSYCOPG 2.5.3 LIBRARY).

## 7.4. Joining Addresses with HTML Documents

Now that the SQL statements for every address have been created, they need to be executed on the database. But before that, there needs to be a table that can contain the join. The following SQL statement creates this table:

```
CREATE TABLE IF NOT EXISTS AddressesUniqueJoinedWithURL (id serial PRIMARY
KEY, AddressesUniqueID INTEGER, HTMLID INTEGER, Original BOOLEAN);
```

The table `AddressesUniqueJoinedWithURL` consists of four fields: an ID field, a field containing the ID of the address, a field containing the ID of and HTML file joined to the address, and a Boolean field used later to indicate that this is a direct connection different from indirect joins created later in the thesis (POSTGRESQL 9.3.9; documentation).

The execution of the SQL statements again happens in the python script and is shown in **Figure 7.6**.

```
133 def JoinAddressesUniqueWithURL(SQLStatementdict):
134     conn, cur = DBConnect()
135     i = 1
136     Starttime = time.time()
137     Starttime2 = time.time()
138     Numberofrows = len(SQLStatementdict)
139
140     for ID in SQLStatementdict:
141         cur.execute(SQLStatementdict[ID])
142         values = cur.fetchall()
143         if values:
144             args_str = ','.join(cur.mogrify("(%s,%s,TRUE)",
145                                     (ID,x[0])) for x in values)
146             cur.execute("INSERT INTO AddressesUniqueJoinedWithURL
147                           (AddressesUniqueID, HTMLID, Original) VALUES " + args_str)
148             conn.commit()
149
150             print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
151             delta_time = time.time() - Starttime
152             print "time till now %.2f Minutes"%(delta_time/60)
153             print "time till end %.2f Minutes"%(((delta_time/60)/i)*(Numberofrows-i))
154             i += 1
155             Starttime2 = time.time()
156     conn.close()
157     return
```

**Figure 7.6** Perform the join of addresses with HTML documents

Apart from the database connection established in line 134, the first lines up until line 138 of the `JoinAddressesUniqueWithURL()` function create some variables that help to keep track of time and calculate how long the function will run. Starting in line 140, the function iterates through the keys of the `SQLStatementdict` dictionary. As mentioned before, the ID of an address is used in the

dictionary as the key to the SQL statement created for this address. So when used as a key in line 141 the corresponding SQL statement is executed (POSTGRESQL 9.3.9; documentation), (PSYCOPG 2.5.3 LIBRARY).

The result of the query is handed over to the `values` variable in line 142 and the results is tested to check if it contains any rows in line 143. If it contains no rows, the next ID of the `SQLStatmentdict` dictionary is called. But if it contains any rows, lines 144 to 146 create an insert into the `AddressesUniqueJoinedWithURL` table. To achieve this, the script iterates through the result in line 144 and the created string contained in the `args_str` could, for example, if the address ID was 1, look like this:

```
(1,3,TRUE) ,(1,25,TRUE) ,(1,43,TRUE) ,(1,199,TRUE)
```

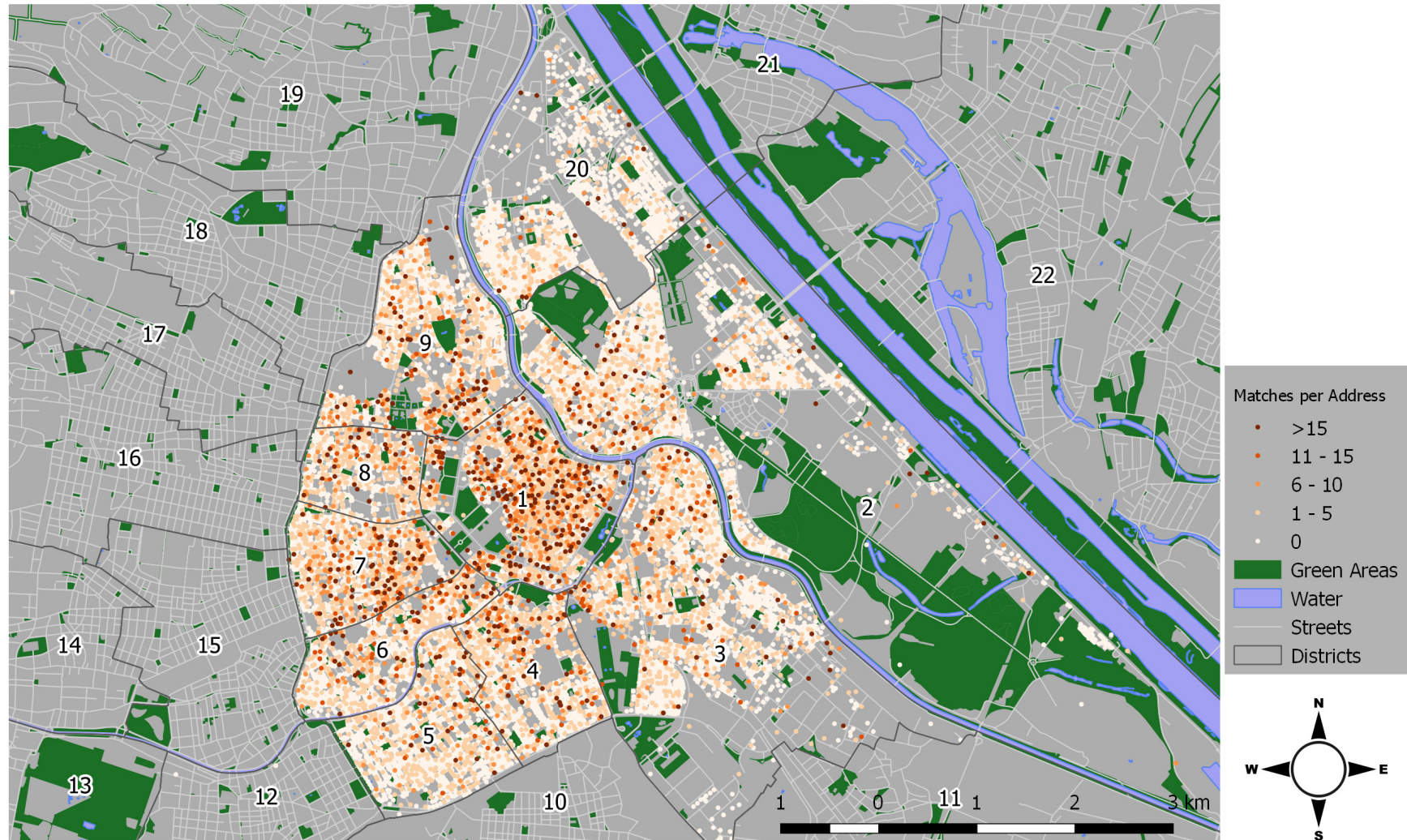
Now combining this string with the rest of the Insert SQL statement would look like this:

```
INSERT INTO AddressesUniqueJoinedWithURL(AddressesUniqueID, HTMLID, Original)
VALUES (1,3,TRUE) , (1,25,TRUE) , (1,43,TRUE) , (1,199,TRUE)
```

This would create 4 new rows in the `AddressesUniqueJoinedWithURL` table, containing the information about which address is joined to which HTML document (POSTGRESQL 9.3.9; documentation).

At the beginning of this chapter, there were around 700k unique HTML documents containing the string “Wien” somewhere and about 24k addresses from which a subset of 21,246 is unique. Now after the direct joins, there are 6284 unique addresses joined to 41,543 unique HTML documents in a total number of 52,586 joins. **Map 7.1** shows a spatial visualization of those joins. The **Table 7.1** shows a frequency distribution of those matches.

## Number of Direct HTML Document Matches per Address for the Inner Districts of Vienna Austria



Date: ETRS89 Projection: UTM 33N Sources: openstreetmap.org; open.wien.gv.at By: Alexander Czech / alexander.czech(at)gmail.com (CC-BY 4.0) 09.06.2015

**Map 7.1** Distribution of addresses joined to HTML documents



Matches per Address	Frequency
0	14962
1-10	5541
11-20	360
21-30	110
31-40	59
41-50	65
51-60	34
61-70	25
71-80	10
81-90	9
91-100	5
>100	66

**Table 7.1** Website match Frequency per Address

Rank	Streetname	Number	PostCode	Direct Website Matches
1	Nibelungengasse	13	1010	2197
2	Neubaugasse	1	1070	1158
3	Ebendorferstraße	7	1010	1152
4	Urban-Loritz-Platz	2a	1070	704
5	Brigittenauer Lände	38	1200	681
6	Johannesgasse	16	1010	468
7	Lothringerstraße	16	1030	468
8	Neubaugasse	8	1070	462
9	Radetzkystraße	2	1030	455
10	Rainergasse	38	1050	427

**Table 7.2** Top Ten matched Addresses

## 7.5. Discussion

Even though it is not the main objective of this thesis to compare different districts and regions of Vienna with one another, the **Map 7.1** still provides an opportunity to write about it. First of all, due to the fact that the addresses are not bought from a provider, like the Österreichische Post AG, the dataset is most likely incomplete (see **chapter 4.4.**). But still there is such an overwhelming amount of addresses that observations can be made. The obvious one is that the first district of Vienna produces the most matches. The reason for this is probably that this district hosts a mix of a lot of commercial companies, tourist attractions, and Austrian government buildings. The first district seems to be followed in matches by the seventh and eighth districts. Other interesting areas are in the ninth district around the University of Vienna and in the fourth around the Vienna University of

Technology. Also visible at the border between the sixth and seventh district is the Maria Hilfer Straße, one of the main shopping streets in Vienna.

Concerning the match frequency shown in **Table 7.1**, as expected, the overwhelming majority of address only turns up on a couple of HTML Documents each, most probably sites like imprints and legal disclaimers. **Table 7.2**, on the other hand shows the top ten addresses with the most matches. It might be assumed that there is no real information to gain out of so little information, other than that the address is named somewhere on the Internet. To broaden the information associated with an address in the next chapter, all web links to a webpage that contains an address are found and the websites containing those links are then also joined with the address.



## 8. Finding Links

The topic of this chapter is to broaden the amount of websites that are associated with an address. For this, all websites that link to a website that was geotagged in chapter 7 are also associated with this address. At the end, there is another look at the derived dataset and a discussion of the map created from the data.

### 8.1. Preparation

To successfully join links with HTML files that have already been geotagged, two python dictionaries are necessary. One contains all URLs from the HTML table and their IDs and second a dictionary contains all IDs of HTML documents matched to address IDs. The code in **Figure 8.1** creates these two dictionaries.

```

047 def URLsWithID(conn, cur):
048     cur.execute("SELECT URL, ID FROM html;")
049     data = cur.fetchall()
050     dictionary = dict(data)
051     return dictionary

013 def GetGeocodedHTMLIDs(conn, cur):
014
015     cur.execute("SELECT HTMLID, AddressesUniqueID FROM
                  AddressesUniqueJoinedWithURL WHERE Original = TRUE")
016     data = cur.fetchall()
017
018     datadict = {}
019     for row in data:
020         if row[0] in datadict:
021             datadict[row[0]].append(row[1])
022         else:
023             datadict[row[0]] = [row[1],]
024
025     return datadict

084 conn, cur = DBConnect()
085 URLDictionary = URLsWithID(conn, cur)
086 URLIDWITHAddressIDDictionary = GetGeocodedHTMLIDs(conn, cur)

```

**Figure 8.1** Creating the URL dictionary and the HTML joined to addresses dictionary

The creation of the URL dictionary is relatively straight forward and shown in line 47 to 51. With the cursor object, an SQL statement is executed on the database, fetching all URLs and IDs from the

table HTML. The database returns them to Python in the form of a list of tuples containing the URL and the ID. When this list is given to the `dict()` function in line 50, it is converted into a dictionary with the URLs as keys and the IDs as values. The dictionary is returned and saved to the variable `URLDictionary` in line 85 (PSYCOPG 2.5.3 LIBRARY).

What is slightly more complicated is the creation of the URL joined to addresses dictionary, because it is a many to many relationship. This means that one URL can be matched to more than one address and one address can be matched to more than one URL. The structure of `URLDictionary` is that HTML IDs act as keys to a list of addresses, because that is the relevant relation in this application. In line 15, the cursor object executes the SQL statement on the database, fetching the `HTMLID` and the `AddressesUniqueID` from the table `AddressesUniqueJoinedWithURL`. They are returned to python again in the form a list of tuples like in the `URLsWithID()` function. But this time instead of just creating a dictionary, the code iterates through the tuple pairs in line 19. The HTML ID of each row (`row[0]`) is tested to see if it already exists in the dictionary as a key in line 20. If so, the list of address IDs associated with the HTML ID is appended with one more address ID. But if the key does not exist, a new entry is created in the dictionary with the HTML ID as the key and the address ID as the first address in the list (PSYCOPG 2.5.3 LIBRARY).

The result is a dictionary with HTML IDs as keys and lists of addresses that are matched to this HTML ID as values. The dictionary is returned and saved to the variable `URLIDWITHAddressIDDictionary` in line 86 (PSYCOPG 2.5.3 LIBRARY).

## 8.1. Link Extraction

The next step is to extract all the links from all 8.4 million websites and, if necessary, convert them to full URLs. The part of the code depicted in **Figure 8.2** is responsible for accomplishing this.

```

027 def FindLinksInHtml(conn, cur, offset):
028     NoWhiteSpace = re.compile(r' ')
029     loadingtime = time.time()
030     cur.execute("SELECT id,url,html_file FROM html WHERE id > %s AND id <= %s
                                ORDER BY id;",(offset, offset+limit))
031     data = cur.fetchall()
032     print('Loading took %.2f Minutes' % ((time.time() - loadingtime) / 60))
033     regextime = time.time()
034     passeslist = []
035     linklist = []
036     for row in data:
037         links = re.findall(r'href=[\"]?([^\"]>+)', row[2])
038         for link in links:
039             try:
040                 linklist.append((row[0],
                                NoWhiteSpace.sub('%20',urlparse.urljoin(row[1], link))))
041             except:
042                 passeslist.append((row[1], link))
043     print('Regex took %.2f Minutes found links %s' %
                                (((time.time() - regextime) / 60), len(linklist)))
044     return linklist, passeslist

```

**Figure 8.2** Finding links and converting them

The method `FindLinksInHtml()` gets a connection object, a cursor object and an offset passed on to it. It loads the following columns: `id`, `url` and `html_file` from the HTML table. `html_file` is the field that contains the whole html file with all the tags in it, not the one created in chapter 6. Which html files are loaded is determined by the offset that changes for every call of the method. Thus, the method always reads the next slice of the HTML table.

The fetched data is saved to the `data` variable. Next, two result containers are created in line 34 `passeslist` and line 35 `linklist`. `passeslist` will hold all the found links that are either not properly formed or could not be converted into absolute URLs, while `linklist` will hold the information for all found links and in which html document they were found.

The script then starts to iterate through the fetched data in line 36. Each row contains an ID element in position 0, a URL element in position 1 and the html file in position 2. The html file is searched for links with the regular expression in line 37. The pattern matching will return all the text of an html link tag marked in this example, `<a href="http://www.w3.org/">` (PYTHON 2.7.10 LIBRARY; regular expression operations).

The `re.findall()` method will return these link strings for all the links in the given document in the form of a list. Iterating through this list is the next step of the script. In line 40 nested into each other there are two methods that process and covert the found links to absolute URLs. The first is the `urlparse.urljoin()` method. In this case, it takes the absolute URL of the page where the link was found (contained in `row[1]`) and creates an absolute URL from a link. This occurs regardless of whether it was a relative or absolute link before. For example, if the link found is `"/hello/world.htm"` and the URL of the page it was found on is `"http://www.w3.org/test/"`, the method would create the

following absolute URL “`http://www.w3.org/test/hello/world.htm`” out of both parts. Table 7.1 shows a couple of other examples of how `urlparse.urljoin()` works. No matter how complex the links or URLs are, the method derives the correct absolute URL (PYTHON 2.7.10 LIBRARY; regular expression operations), (PYTHON 2.7.10 LIBRARY; `urlparse`).

Link	URL	Result
<code>/hello/world.htm</code>	<code>http://www.w3.org/test/</code>	<code>http://www.w3.org/test/hello/world.htm</code>
<code>http://www.w3.org/test/hello/world.htm</code>	<code>http://www.w3.org/test/</code>	<code>http://www.w3.org/test/hello/world.htm</code>
<code>http://www.w3.org/test/</code>	<code>../hello/world.htm</code>	<code>http://www.w3.org/hello/world.htm</code>
<code>../test/</code>	<code>http://www.w3.org/test/hello/world.htm</code>	<code>http://www.w3.org/test/</code>

**Table 8.1** `urlparse.urljoin()` Examples

The second method simply replaces spaces in URLs. Spaces as an ASCII symbol are not part of the URL specification. Nevertheless, a lot of links do contain them and they mostly work fine because most modern browsers have error handling capabilities. But the URLs saved in the database are saved in the correct format for URLs, where white spaces are encoded with the percent encoding. To find those links pointing to those URLs, white spaces also need to be replaced with the percent encoding. This is what the `NoWhiteSpace.sub()` regular expression does. An example of such a conversion could be “`http://www.w3.org/hello world.htm`” is converted to “`http://www.w3.org/hello%20world.htm`”. The converted link is then appended to the `linklist` with the corresponding html file ID (PYTHON 2.7.10 LIBRARY; regular expression operations), (BERNERS-LEE, ET AL.; 2005; pp.11-14).

Because there are some malformed links, link conversion is within a `try` and `except` block. If the conversion fails, the link on which it fails is appended to the `passeslist`. Overall, there are 72 links in the 8.4 million documents that could not be converted. Examples of those can be seen in **Figure 8.3**.

```
'http://[www.boku.ac.at/fachstukofhnw.html']
'http://cialisqrx.com]buy')
('http://derstandard.at/1328507079451/Nachlese-Schneechaos-in-weiten-Teilen-
Oesterreichs'
```

**Figure 8.3** Malformed Link Examples

## 8.2. Geotagging the found linked websites

As the title suggest, this chapter is about how to join those linked websites to already geotagged ones and, in turn, geotagging the linked websites as well. As described before, this works by finding links to already geotagged websites on other websites. Matching those websites with the same address like the one they link to. This process is shown in **Figure 8.4**.

```

098     linklist,passes = FindLinksInHtml(conn, cur, offset)
099
100     passeslist += passes
101     URLIDStoLinkIDS = []
102     for row in linklist:
103         if row[1] in URLEDictionary:
104             URLIDStoLinkIDS.append((row[0],URLEDictionary[row[1]]))
105
106     Newlist = []
107     for row in URLIDStoLinkIDS:
108         if row[1] in URLIDWITHAddressIDDictionary:
109             for rowx in URLIDWITHAddressIDDictionary[row[1]]:
110                 Newlist.append((row[0],rowx))
111
112
113
114     r += len(Newlist)
115     WritetoAddressesUniqueJoinedWithURL(Newlist)

053 def WritetoAddressesUniqueJoinedWithURL(List):
054     conn, cur = DBConnect()
055     args_str = ','.join(cur.mogrify("(%s,%s,FALSE)", x) for x in List)
056     try:
057         cur.execute("INSERT INTO AddressesUniqueJoinedWithURL (HTMLID,
                                AddressesUniqueID, Original)VALUES " + args_str)
058         conn.commit()
059     except:
060         print "Error Inserting Joins"
061     cur.close()
062     conn.close()
063
064     return

```

**Figure 8.4** Geotag linked websites

Essential for doing this are the two dictionaries `URLEDictionary` and `URLIDWITHAddressIDDictionary` whose creations are described in subchapter 8.1., and `linklist`, the result of the previous subchapter. `URLEDictionary` contains all of the URLs in string form with their respective IDs. `URLIDWITHAddressIDDictionary` contains all URL IDs that are joined to address IDs. And `linklist` contains all IDs of websites and to which URLs those websites link. So what needs to be done for every link found on an HTML document is that the corresponding IDs have to be looked up in the `URLEDictionary`. There is a possibility that a link URL can't be found in `URLEDictionary`, because links

could also target none .at websites. If the ID is found, another lookup is done in the `URLIDWITHAddressIDDictionary` dictionary. If this HTML document is already joined to an address, the ID of the address is returned by the dictionary and the linked website is now also joined to this address.

To do this in code, the script starts to iterate through the `linklist` in line 102 and every found link URL is tested to see if it is contained in the `URLDictionary` in line 103. If the link URL is contained in `URLDictionary`, the ID of the website containing the link URL and the ID of the website the link is linking to, are appended to the list `URLIDStoLinkIDS`. The next step is that the script iterates through this list in line 107. If the URL ID a link points to is also found in the `URLIDWITHAddressIDDictionary` dictionary, the script iterates through all the addresses this website is associated with and appends a tuple consisting of the website ID where the link originated and the address ID to the `Newlist`, thus creating the desired join in line 110.

This `Newlist` containing these new joins is then handed over to the `WritetoAddressesUniqueJoinedWithURL()` method in line 115. This utilizes a couple of previously discussed techniques to write all joins to the database in one transaction. Especially the `cur.mogrify()` method in line 55. The value in the field `original` is set to `FALSE`. This makes it possible to discriminate between direct and indirect joins of websites to addresses (PSYCOPG 2.5.3 LIBRARY).

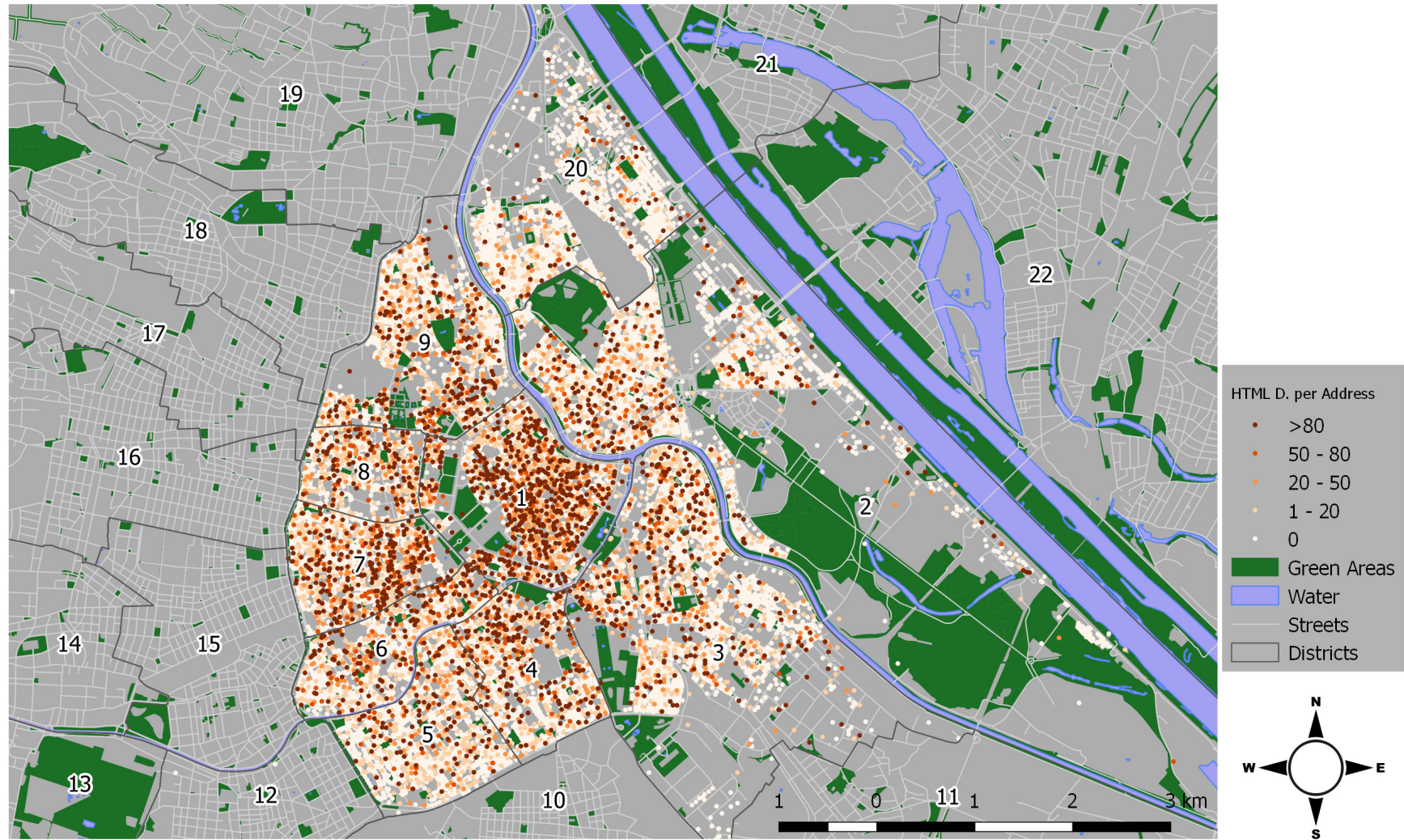
### 8.3. Discussion

In the previous chapter, there were 6,284 unique addresses joined to 41,543 unique HTML documents in a total of 52,586 joins. The number of addresses is constant, but there are now 269,083 unique HTML documents joined to 6284 addresses in a total of 2,062,981 joins. Those numbers are interesting because, while the number of total HTML documents only got about 6.5 times bigger, the number of joins in comparison massively increased by a factor of 40. That means there must be quite a lot of websites that are rather well interconnected to each other and websites that link to multiple addresses.

As in the previous chapter, this work yields another map shown as **Map 8.1**.



## Number of HTML Documents Associated with an Address for the Inner Districts of Vienna Austria



Date: ETRS89 Projection: UTM 33N Sources: openstreetmap.org; open.wien.gv.at By: Alexander Czech / alexander.czech(at)gmail.com (CC-BY 4.0) 22.06.2015

**Map 8.1** Result of joining linked HTML documents to addresses

Websites per Address	Frequency
0	14962
1-10	2264
11-20	820
21-30	427
31-40	294
41-50	243
51-60	189
61-70	142
71-80	114
81-90	83
91-100	77
>100	1631

**Table 8.2** Associated Website Frequencies per Address

Rank	Streetname	Number	Post Code	Direct Website Matches	Associated Website Matches
1	Viktorgasse	16	1040	323	91575
2	Urban-Loritz-Platz	2a	1070	704	38684
3	Stephansplatz	6	1010	370	33545
4	Neubaugasse	1	1070	1158	30878
5	Neubaugasse	8	1070	462	29442
6	Lothringerstraße	16	1030	468	29437
7	Siebenbrunnengasse	21	1050	374	25010
8	Brigittenauer Lände	38	1200	681	23174
9	Gumpendorfer Straße	10-12	1060	280	21385
10	Schottenring	17	1010	110	21230

**Table 8.3** Associated Website Frequencies per Address

When interpreting **Map 8.1**, the previous observations still seem to hold true. The 1<sup>st</sup> Viennese District is still the one with the strongest Internet presence. The 1<sup>st</sup> is followed by the 7<sup>th</sup> and 8<sup>th</sup> and the areas around the main University and Technology University of Vienna. For the dataset as a whole, this seems also to be true. When the coefficient of variation values from websites directly matched (10.03) and websites associated with addresses (10.74) are compared to each other. It can be deducted that the value dispersion does not change much. Thus, the addresses HTML document distribution is the same as before just with higher values. For a picture of the distribution see **Table 8.1**. The top ten addresses all have over 20,000 associated websites with the maximum of 91,575. See **Table 8.2** for the addresses with the most associations (Böhner; 1990; pp. 18-20)



## 9. The Vector Space Model

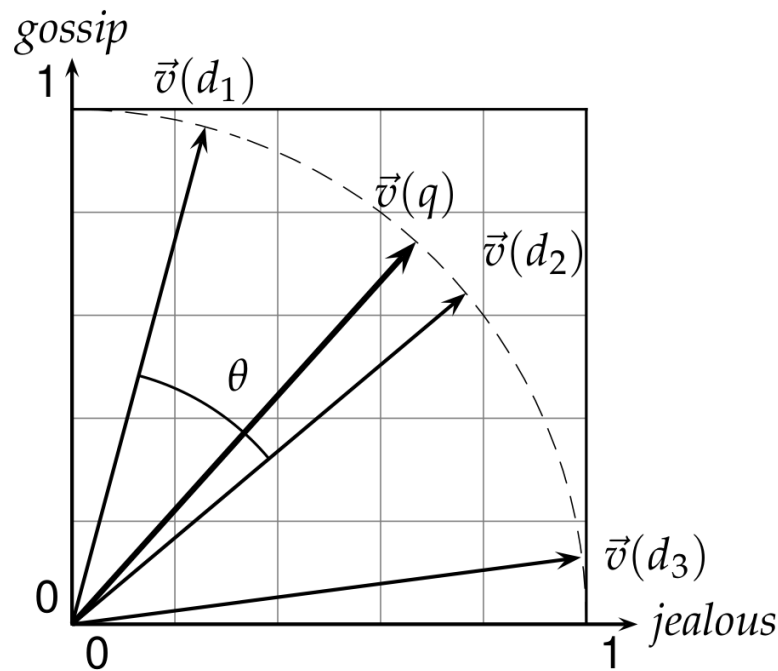
This chapter is an introduction into the vector space model, which is a way to classify documents within a high-dimensional vector space. The Vector Space Model can be used to compare document similarity and search queries. It is a useful tool to overcome the limitations of Boolean retrieval systems and its main component is a statistically weighted document vector for every document within a collection of documents.

### 9.1. The Document Vector

To understand the idea of the document vector space, the disadvantages of Boolean retrieval have to be considered. How Boolean retrieval for a big collection of document works is that if a term in a document matches a query used for retrieval, it is retrieved. But in many cases, such a query can be too restrictive. A query such as “ $T_1$  and  $T_2$  and  $T_3$ ” will only retrieve those documents that exactly match the query. An OR query for these terms “ $T_1$  or  $T_2$  or  $T_3$ ,” on the other hand, could be too loose. Furthermore, the list of documents is retrieved unordered. But it would be of interest to find the most relevant documents to a query. A possible way would be to simply count how many times the query term or terms are present in the document and order the retrieved documents by this count (SALTON.; 1991; pp. 974).

To do this effectively, the information retrieval system should not count all words in the document for every query again and again; rather it would be efficient to count them beforehand. Thus, a new representation of the document is created. If every individual term in the set of documents is seen as one dimension, all of the documents can now be seen as a vector in this high-dimensional space. In this representation, the relative order of words in the document is lost. The two documents “Mary is quicker than John” and “John is quicker than Mary” are represented as the same so called bag of words. To compensate for different lengths of documents and, thus, different lengths of the document vectors, the vectors are normalized to a length of 1 (MANNING ET AL.; 2009; pp. 120-122), (SALTON ET AL.; 1975; pp. 613-620).

To compare two documents to each other, it is now possible to use the vectors of both documents. **Figure 9.1** shows 3 normalized document vectors  $\vec{v}(d_{1-3})$  in a document 2D vector space. It only consists of the two words “gossip” and “jealous” (MANNING ET AL.; 2009; pp. 120-122).



**Figure 9.1** Cosine similarity example (MANNING ET AL.; 2009;)

The similarity between the two documents  $d_1$  and  $d_2$  can be determined by the cosine of the angle theta. The cosine of an angle is bigger the more acute the angle is. A angle of  $360^\circ/0^\circ$  results in a cosine of 1 while an angle of  $180^\circ$  results in -1 (MANNING ET AL.; 2009; pp. 120-122).

In this system, (search)-queries can be treated as just a bag of words as well and made into a document vector  $\vec{v}(q)$ . This vector can then be compared to the other document vectors. The dot product, which is equal to the cosine of the angle, for the query vector and all document vectors is created. The documents are then ordered by the cosine similarity to the query (MANNING ET AL.; 2009 p.123-124).

## 9.2. Term frequency Inverse Document Frequency

Still open is the question on how to weigh query terms. So far, words in documents and not queries, even though in the vector space model both can be seen as equal, are weighed by the term frequency. The more often a term occurs in the document, the more the vector is moved in the dimensional direction of the word. The reason why there is still a need to introduce some other form of term weighing is that not all words are equally important to a document (MANNING ET AL.; 2009 p.117).

There are, for example, stop words. Stop words is a term used for extremely common words that are no help when distinguishing documents from each other. A short example stop word list can be seen in **Figure 9.2**.

a      an      and      are      as      at      be      by      for  
 from   has   he      in      is      it      its      of      on  
 that   the   to      was   were   will   with

**Figure 9.2** Stop word list of 25 words that are common in the Reuters Corpus Volume 1 (MANNING ET AL.; 2009)

These words could be not included in the vector space, but the problem of how to weigh different terms still persists. For example, in a collection of documents about cake baking, the word sugar probably occurs in nearly every of those documents and has, therefore, a very low value in distinguishing the documents from one another. What needs to be done is to weigh the term frequency of words that are rare in the corpus higher and those that are frequent lower. Because the goal is to distinguish documents from each other, it is desirable to count in how many documents the term occurs, rather than how many times they occur overall. This can be further illustrated by **Table 9.1** another example from the Reuters Corpus. The collection frequency is how often the term occurs individually and the document frequency is in how many different documents the term occurs (MANNING ET AL.; 2009 p.117-118).

Word	cf	df
try	10422	8760
insurance	10440	3997

**Table 9.1** Collection frequency (cf) and document frequency (df) different behavior (Manning et al.; 2009)

The formula used to calculate the document frequency weight of a term, also called the inverse document frequency (idf), is:

$$idf_t = \log \frac{N}{df_t}$$

$idf_t$  (inverse document frequency of the term),  $N$ (total number of documents),  $df_t$ (Document frequency of the term)

**Table 9.2** shows some example Values for document frequency and the resulting inverse document frequency (SALTON; 1991; pp. 976).

Term	$df_t$	$idf_t$
car	18,165	1.65
auto	6,723	2.08
insurance	19,241	1.62
best	25,235	1.50

**Table 9.2** Examples for idf Values based on the Reuters Collection containing 806,791 documents (MANNING ET AL.; 2009)

Finally, the inverse document frequency can be combined with the term frequency by multiplication. Now, when combined with the content of sub chapter 9.2., a weighted and normalized vector of a document can be created (SALTON; 1991; pp. 976).

## 10. Categories for classification

This chapter is about defining classes in which the addresses can be categorized. For this, there is a short look at the Munich-Viennese school of social geography before mixing the findings with some newer approaches to define the classes.

### 10.1. Daseinsgrundfunktionen

The title of this chapter literally translates to “basic existence functions”. It is part of a concept developed by the so-called Munich-Viennese school of social geography. The premise of the school was that there are social groups that transform space for their needs, those needs being the “Daseinsgrundfunktionen.” It was an important step away from the previous approach that the natural/physical appearance of space determines the use of this space by humans (MAIER ET AL.; 1977; p. 18).

Translated into English, here are the seven “Daseinsgrundfunktionen”:

- To live somewhere (Wohnen)
- To work (Arbeiten)
- To supply (Sich versorgen und konsumieren)
- To be educated (Sich bilden)
- To relax (Sich erholen)
- To take part in traffic (Verkehrsteilnahme)
- To live in a community (Sich Fortpflanzen und in Gemeinschaft leben)

(RUPPERT AND SCHAFER; 1969; pp. 208-209)

These functions are of interest because, according to the principles of the Munich-Viennese school of social geography, these functions have a representation in space. Therefore, they could be detected in communication about a space (MAIER ET AL.; 1977; p. 100).

The concept of the Munich-Viennese school of social geography can be criticized. A main point of contention is that the concept of the social groups described within the theories is incompatible with the definition of social groups in other disciplines like sociology. In some cases, people would form a social group by just doing the same thing, like biking. Also, the basic functions of existence seem to be incompatible with sociology (WEICHERT; 2008; pp. 44-53).

The first approach was to just use the basic functions of existence as classes for the addresses. But first a transformation had to be made. Because the system is set up like an information retrieval system, the names of the classes needed to be cast more in the form of a search query than a

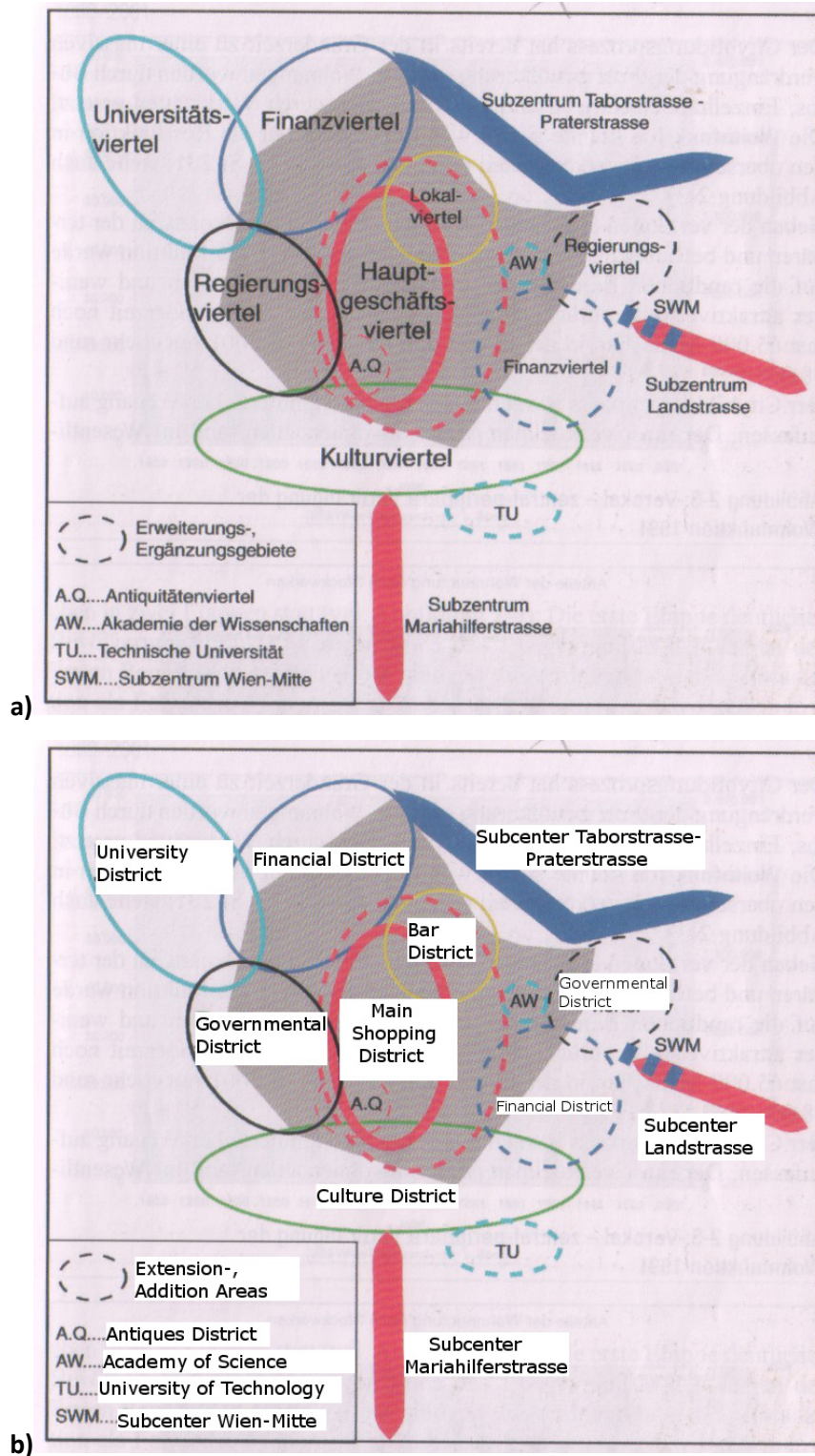
scientific term. But with this transformation into search queries, it became obvious that the classes would not cover or, in information retrieval terms, retrieve all places that are part of these classes. To illustrate this, the place where a doctor's office or a hospital is located would be part of the class "to live in community." But a search term corresponding to the class "community" would potentially not produce a good score with a doctor's office or a hospital. On the other extreme, a too narrow search term like "health care" would probably score very well with a doctor's website, but exclude everything else that is part of living in community (see **chapter 9.** about how document vectors works).

With these criteria in mind, in the end it was decided to only keep three of the Daseinsgrundfunktionen and develop appropriate search terms for them:

- Wohnen (To live somewhere)
- Arbeiten (To work)
- Sich bilden (To be educated)

## 10.2. Classes for addresses

With the problem of a too wide or narrow "search term" for classification purposes in mind, it was clear that finding enough classes to classify every possible entity was not an option. To broaden the scope of classes for entities, literature about functional urban geography was consulted to develop further classes. During this review **Map 10.1** came up. It shows the Viennese inner city divided up into functional quarters.



**Map 10.1 a),b)** Functional subdivision of the Viennese inner city a) German b) English translation (FASSMAN AND HATZ; 2002; p. 37)

This map is the second source from which classes are derived. Apart from listing a couple of functions that could work quite well with the search term paradigm, it provides the possibilities to compare the results produced by the classification other than the control group (see **chapter 13.**).

The last resource for classes was the trade groups found in HEINEBERG ET AL. (2014). From those two sources the following classes are derived:

- Kultur (culture)
- Einkaufen (shopping)
- Finanzen (finance)
- Regierung (government)
- Gaststätte (restaurant, bar)
- Hotel (hotel)

(HEINEBERG ET AL.; 2014; pp. 187-189), (FASSMAN AND HATZ; 2002; pp. 35-40)

The resulting list of classes is far from complete. Almost no services like barbers and plumbers or attorneys and doctors are represented by any of those classes. How complete the list is will also show the mapping of the control group. Everything that cannot be assigned a class will be put into the class “other.” How big the class “other” will be, after creating the control group, will reveal how much is not captured by the other classes.

Lastly, all the classes are transformed into search queries (see **Table 10.1**). Most of the queries are identical to their German class names. Exceptions are “Dienstgebäude” (governmental building) for “Regierung” (government). The idea behind this is that buildings that are associated with government can be named “Dienstgebäude” and therefore not only target buildings like the parliament but also other government agencies. For the finance class, the query “Kreditinstitut” (credit institution) was selected. The reason behind this has to do with the co-occurrence groups that are going to be explained in **Chapter 11.**, and because a query like “Finanzen” (finance) yielded a co-occurrence group that seemed too similar to the Ministry of Finance and government and the word “Bank” has more than one meaning in German.



Class	Query
<b>Wohnen (To live somewhere)</b>	Wohnen
<b>Arbeiten (To work)</b>	Arbeiten
<b>Sich bilden (To be educated)</b>	Bildung
<b>Kultur (culture)</b>	Kultur
<b>Einkaufen (shopping)</b>	Einkaufen
<b>Finanzen (finance)</b>	Kreditinstitut
<b>Gaststätte (restaurant, bar)</b>	Gaststätte
<b>Hotel (hotel)</b>	Hotel
<b>Regierung (government)</b>	Dienstgebäude

**Table 10.1** Classes and their corresponding queries

## 11. Co-occurrence Groups

This chapter gives an overview of natural language processing and part of speech (POS) tagging, in particular, and shows how these techniques are used to create a POS-tagged version of Wikipedia. Lastly, from the POS-tagged Wikipedia co-occurrence query, expansion groups are generated for the search terms defined in **Chapter 10**.

### 11.1. Introduction Natural Language Processing

The tool of computer linguistics is statistics. Computer linguistics attempts to create a statistical model of (natural) human language. The goal is that, with the statistical model, a computer could analyze a language or a text in this language and create some result about it, without the necessity of understanding language, like humans do. One of the prime examples for natural language processing (NLP) and computer linguistics is translation work. Other examples where computer linguistics is used today are in the creation of text summaries or detection of plagiarism (MANING AND SCHÜTZE; 1999; pp. 3-35).

The overarching instrument NLP uses is a text corpus. A corpus is a kind of annotated text that can be used as a knowledge base. It can be used to answer simple language questions like in what frequency some kind of word is used together with another (MANING AND SCHÜTZE; 1999; pp. 3-35).

### 11.2. Part-of-speech Tagging

Part-of-speech tagging, a discipline within the NLP field, is an important part of the further work in this thesis. It is a technique that determines whether every term is a noun, verb, adjective, etc. or if the word is part of a compound word. The results are then attached as a label to the word and saved. The annotated text is called a corpus (RUSSEL; 2014; p. 194).

An example for POS-tagged sentence is this:

The-**DT** representative-**NN** put-**VBD** chairs-**NNS** on-**IN** the-**DT** table-**NN**.

Every word has a label that indicates what kind of word it is. The meaning of the labels can be looked up in **Table 11.1**. The example sentence could also be tagged differently as in the next example:

The-**DT** representative-**JJ** put-**NN** chairs-**VBZ** on-**IN** the-**DT** table-**NN**.

Even though this way of reading is unlikely, the example shows that tagging always has some sort of ambiguity. A good tagger then determines which of the syntactic categories for a word is most likely for the word in this kind of a sentence (MANING AND SCHÜTZE; 1999; pp. 341-379).

Tag	Description	Example
CC	conjunction, coordinating	<i>and, or, but</i>
CD	cardinal number	<i>five, three, 13%</i>
DT	determiner	<i>the, a, these</i>
EX	existential there	<i><u>there</u> were six boys</i>
FW	foreign word	<i>mais</i>
IN	conjunction, subordinating or preposition	<i>of, on, before, unless</i>
JJ	adjective	<i>nice, easy</i>
JJR	adjective, comparative	<i>nicer, easier</i>
JJS	adjective, superlative	<i>nicest, easiest</i>
LS	list item marker	
MD	verb, modal auxiliary	<i>may, should</i>
NN	noun, singular or mass	<i>tiger, chair, laughter</i>
NNS	noun, plural	<i>tigers, chairs, insects</i>
NNP	noun, proper singular	<i>Germany, God, Alice</i>
NNPS	noun, proper plural	<i>we met two <u>Christmases</u> ago</i>
PDT	predeterminer	<i><u>both</u> his children</i>
POS	possessive ending	<i>'s</i>
PRP	pronoun, personal	<i>me, you, it</i>
PRP\$	pronoun, possessive	<i>my, your, our</i>
RB	adverb	<i>extremely, loudly, hard</i>
RBR	adverb, comparative	<i>better</i>
RBS	adverb, superlative	<i>best</i>
RP	adverb, particle	<i>about, off, up</i>
SYM	symbol	<i>%</i>
TO	infinitival to	<i>what <u>to</u> do?</i>
UH	interjection	<i>oh, oops, gosh</i>
VB	verb, base form	<i>think</i>
VBZ	verb, 3rd person singular present	<i>she <u>thinks</u></i>
VBP	verb, non-3rd person singular present	<i>I <u>think</u></i>
VBD	verb, past tense	<i>they <u>thought</u></i>
VBN	verb, past participle	<i>a <u>sunken</u> ship</i>
VBG	verb, gerund or present participle	<i><u>thinking</u> is fun</i>
WDT	wh-determiner	<i>which, whatever, whichever</i>
WP	wh-pronoun, personal	<i>what, who, whom</i>
WP\$	wh-pronoun, possessive	<i>whose, whosever</i>
WRB	wh-adverb	<i>where, when</i>

**Table 11.1** The Penn Treebank II POS tag set (Santorini 1990)

The first large tagged corpus was the Brown Corpus in 1971. It consists of about 1 million words and was first tagged by humans over a couple of years. The TAGGIT tagger was developed alongside the Brown Corpus. The tagger used lexical information to narrow down the tags a word could have and then apply rules to for tagging it. An example for such a rule could be that a noun very likely follows

an article and that a verb following an article is very unlikely. So if the tagger had found a word in the text by a lexical lookup that could be a noun or a verb and it is preceded by an article, then the tagger would decide it as a noun (MANING AND SCHÜTZE; 1999; pp. 341-379).

Taggers that were developed later and are, therefore, more advanced make use of Hidden Markov Chains. Markov Chains try to determine what type of word a word is by looking at the preceding 1 to 3 words. It calculates the combined possibility of these words occurring in this kind of order for all possible combinations and then chooses the combination with the highest possibility. These models need to be trained to “know” the possibility of a word sequence occurring (MANING AND SCHÜTZE; 1999; pp. 317-340).

The performance of a tagger mostly depends on four factors:

- The amount of training data the tagger has available
- How big the tag set is. The bigger the tag set, the less reliable a tagger gets
- How different the trainings corpus and dictionary that are used are from the corpus that needs to be tagged
- How well the tagger can handle words that are unknown to it

Most modern taggers reach an accuracy of 96% to 97%, which seems quite high, but in reality it means that in an average-length sentence of 20 words, there is one incorrectly tagged word (MANING AND SCHÜTZE; 1999; pp. 371-372).

A German language tagger that is natively available in python as a library is a Brill-Tagger. The Brill-Tagger works differently than the previously mentioned hidden Markov Chain tagger, instead it uses a learned rule base schema and a lexical lookup. The Brill-Tagger was originally developed for English and needed to be trained for German. It works with a pre-tagged corpus that is used as a lexical lookup and from which the rules are derived. It is trained in two-steps. First, it assigns all words their most common tags found in the trainings corpus. Then, because the tagger learns on a pre-tagged corpus, the errors that have been made are recorded. The tagger then tries to find rules that correct the mistakes. Each rule is tested against the pre-tagged corpus and the tagger weighs if this corrects more mistakes than it introduces. This process is iterated until the error rate plateaus (SCHNEIDER AND VOLK; 1998; pp. 2-3).

The Brill-Tagger generates two sets of rules: lexical rules and context rules. Lexical rules are used for unknown words. An example of a lexical rule (shown below) is that words that end in the 4-letter suffix -lich

**LICH HASSUF 4 RB**

The effect of this rule is that every unknown word that has the 4-letter suffix –lich is retagged as an adverb independent of what its first tag was (SCHNEIDER AND VOLK; 1998; p. 3).

An example for a context aware rule is:

**NN VB PREV-TAG TO**

This rule changes a word that is tagged as a noun to a verb if the word that precedes it is tagged with the infinitival “TO” (BRILL; 1992; p. 152-153).

The described Brill-Tagger for the German language achieved results of around 95 to 96% correctness. But a problem with these results is that the Brill-Tagger was validated on the same kind of text it was trained on. It was trained on the annual report from the University of Zurich. For the training phase, 25% of the corpus was withheld and used as a control group. In this control group, the tagger had an error rate of 5%, but it can be assumed that if the tagger was used not to tag annual reports from this specific university, but, for example, for journalistic sports publications, that the error rate would be higher (SCHNEIDER AND VOLK; 1998; p. 4-7), (MANNING AND SCHÜTZE; 1999; p. 343-344).

### **11.3. POS tagging Wikipedia**

The python library used for POS tagging uses the Penn Treebank II tag set. This is a tag set developed for the English language and, therefore, does not contain provisions for German language particularities, like tags for separated verb prefixes. A tag set providing language tags that were made for the German language is the Stuttgart-Tübingen Tag-Set (STTS) (SCHNEIDER AND VOLK; 1998; p.3).

But for the sake of simplicity, this paper will continue to use the Penn Treebank II because later only two broad word groups, verbs and nouns, are used to create the semantic vectors. It is therefore less relevant for this task that the tagger correctly identifies what kind of verb or noun a word is. That only broad classes are needed also helps with the second problem the tagger has, that it was created from a very specific kind of text. This was described at the end of the previous chapter (MANNING AND SCHÜTZE; 1999; pp. 343-344)

Because of its broad scope of themes, its huge amount of text and that it is freely available, the German Wikipedia is a good source for creating co-occurrence groups as they are described in the later subchapters 11.4 and 11.5. To generate these groups the words and sentences of Wikipedia need to be POS tagged. To POS tag Wikipedia, the content needs to be available as pure text. It is possible to download XML dumps from Wikipedia, but they need to be converted to pure text. KOPI,

a web portal used to identify plagiarism in English, German and Hungarian, has developed such a converter with the following adjustments:

- The conversion keeps article boundaries
- Only text information is extracted
- Info boxes get filtered out
- Comments, templates and math tags are also filtered out
- Other types of “written” information like tables are converted to text

KOPI publishes their converted dumps and makes them available under the Creative Commons license 3.0 BY-SA. The newest available German Wikipedia text version from 16.06.2014 was downloaded from their site (PATAKI ET AL.; 2012; pp. 48-49).

The downloaded dump is 7.22 gigabytes of text data when unzipped. It is unzipped into 1321 individual files, each in an individual subfolder. The POS tagging with the following code took about 2 days on a modern computer.

```

045 def createfilepathlist():
046     pathlist = []
047     subfolders = [x[0] for x in os.walk('./WikiText/')]
048     for subfolder in subfolders[1:]:
049         for filename in os.listdir(subfolder):
050             pathlist.append(subfolder+'/'+filename)
051
052     return pathlist
053
054
055 i = 0
056 filepathlist = createfilepathlist()
057 len_filelist = len(filepathlist)
058 Starttime = time.time()
059
060 if __name__ == '__main__':
061     print('Tagging new corpus')
062     pool = ThreadPool(4)
063     pool.map(POSTag, filepathlist)
064     pool.close()
065     pool.join()
066
067     print('+++++#####')
068     print('complete Operation took %s Minutes' % ((time.time() - Starttime) / 60))
069     print('+++++#####')
070

```

**Figure 11.1** Creating the file list

The code in **Figure 11.1** creates as its first task a list of all Wikipedia text files in line 56. This method is called `createfilepathlist()`. This function uses a couple of methods in python’s `os` library for example `os.walk()` in line 47 that returns all subfolders for a main folder. All subfolders are saved as a list to the variable `subfolder`. This list is iterated through, starting in line 48, except for the first

element because it is the main folder itself. Within the iteration, all objects in each subfolder are again called with `os.listdir()`. Each subfolder contains exactly one text and in line 50 every subfolder is combined with the file into one path and saved to the `pathlist`. This list is then returned in line 52 (PYTHON 2.7.10 LIBRARY; operating system).

Other variable in the lines 55 to 58 need to be set outside the main thread, too. They need to be accessible to the parallel threads created in the main thread. The main thread begins in line 60 and makes use of the multiprocessing library in python to POS tag the individual Wikipedia files in multiple threads. The `ThreadPool` is set to 4 workers in line 63. With the `pool.map()` function, the 4 threads are spawned. This method needs a function (`POSTag()`) and an iterable variable (`filepathlist`) to work. It takes the first item from the `filepathlist`, passes it to the `POSTag()` and starts an instance of the function in the first thread. Then the second item is handed to a new instance of the function in the second thread and so on. Whenever one thread is finished with the current item, it gets a new item from the `filepathlist` until all items from the list have been iterated through. Lines 65 and 66 make sure that all threads are completed and joined again before the script moves on (PYTHON 2.7.10 LIBRARY; multiprocessing).

```

011 def POSTag(filepath):
012
013     global i
014     global len_filelist
015     i += 1
016     newcorpustagged = []
017     Starttime2 = time.time()
018
019     with io.open(filepath, 'r', encoding='utf-8') as mfile:
020         data = mfile.read()
021         data = data.splitlines()
022
023
024     for section in data:
025         section = parse(section)
026         section = section.split()
027
028         for sentence in section:
029             sent = []
030             for token in sentence:
031                 sent.append((token[0], token[1]))
032             newcorpustagged.append(sent)
033
034
035     with io.open('./wikicorpuspickeld_2/%s_%s.pos' % (current_thread().ident, i),
036                  'wb') as fout:
037         pickle.dump(newcorpustagged, fout)
038
039     print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
040     delta_time = time.time() - Starttime
041     print("time till now %.2f Minutes" % (delta_time/60))
042     print "time till end %.2f Minutes" % (((delta_time/60)/(i*4))*(len_filelist-
043                                         (i*4)))

```

**Figure 11.2** Executing the POS tagger

The `POStag()` function in **Figure 11.2** defines the variables `i` and `len_filelist` as `global` variables to access them even though they were defined outside the function. The function also creates a new list `newcorpustagged` that will contain the part of the corpus that will be tagged by the function. It then proceeds to open the file at the passed `filepath` in line 19. Important for opening the file is to define the correct encoding of the file, in this case `'utf-8'`. The file is read in line 20 and saved to the variable `data`. Now that the text of the file is available as a string, this string is split at every new line (`\n`) symbol with `.splitlines()` in line 21. New lines occur whenever a paragraph in the original Wikipedia article ended. The reason to split the file into paragraphs is to POS tag the string not all at once, but bit by bit (PYTHON 2.7.10 LIBRARY; I/O),(PYTHON 2.7.10 LIBRARY; string).

The string is then POS tagged paragraph by paragraph starting in line 24. The `parse()` function in line 25 POS tags the paragraph. The paragraph is then split into individual sentences with `.split()` in line 26. The code iterates through all sentences in line 28 and then through all now-tagged words in line 30. From the individual tokens, only object 0 (word) and object 1 (POS tag) are of interest. That is why only those two are appended as a tuple to the new sentence list `sent` in line 31. This list is then appended to the meta list `newcorpustagged`. This continues until all sentences of all paragraphs have been tagged and appended to `newcorpustagged`, resulting in a list of sentences that, in turn, consist of a list of word and POS tag tuples (PATTERN; pattern.de).

The last step is to save `newcorpustagged` to the drive so that it can be recalled in later scripts. For this, the python input output library and the pickle library are used. Pickle allows for the serializing of python objects, so that they can be saved as a file. To avoid file name conflicts, the files that are saved in line 35 and 36 are named with the variable `i` and the ID of the current thread that is checked with `current_thread().ident`. As mentioned before, the `POStag()` function is executed in 4 Parallel threads, on the 1,321 Wikipedia text files. The result is 1,321 pickle objects that are saved to the `'./wikicorpuspickeld_2/'` directory (PYTHON 2.7.10 LIBRARY; I/O), (PYTHON 2.7.10 LIBRARY; pickle).

## 11.4. Co-occurrence

To simulate human knowledge about words in a machine-processing task, it is necessary to analyze the meaning of words. A thesaurus is a typical knowledge representation, in a sense, what words mean is described by other words. However, generating a thesaurus manually is very labor-intensive and is biased towards the manufacturer (ITO ET AL.; 2008; pp. 817-826).



To automatically generate a thesaurus and solve both of these problems, a couple of methods have been developed. One of them is co-occurrence. Broadly speaking, co-occurrence measures how often one word is used similarly to another word. How close both words have to be is defined by the window size. The windows size can range from only one word, resulting in only the words that directly proceed and precede a word, to up to 10 words. This becomes more nuanced if the frequency of the co- occurrence is also taken into account (ITO ET AL.; 2008; pp. 817-826).

Co-occurrence word information can be defined in a couple of ways and describes the relation of the co-occurring words to each other.

- a) The relation between a super-concept and a sub-concept word. Examples for this co-occurrence are “country name” and “Canada” or “clothes” and “trousers”
- b) The relation between verb and noun phrase. For example “run, dog, subject”
- c) Compound word relations like in “Canadian” and “Canadian Nationality” or “America” and “United States of America” as examples
- d) The synonymous relation between words “America” and “United States of America” are used as synonyms as well as “Cutter” and “Sports shirt”

To also capture how strongly two words correlate with each other, the frequency of their co-occurrence can be collected. A relation between the word “Chirp” and “Bird” is recorded two times in the corpus, so this relation has a co-occurrence frequency of 2 (ITO ET AL.; 2008; pp. 817-826).

Both aspects, the co-occurrence and the frequency of it, will be used to create a co-occurrence group similar to the document vector described earlier (KAZUHIRO ET AL.; 2003; pp. 957-960).

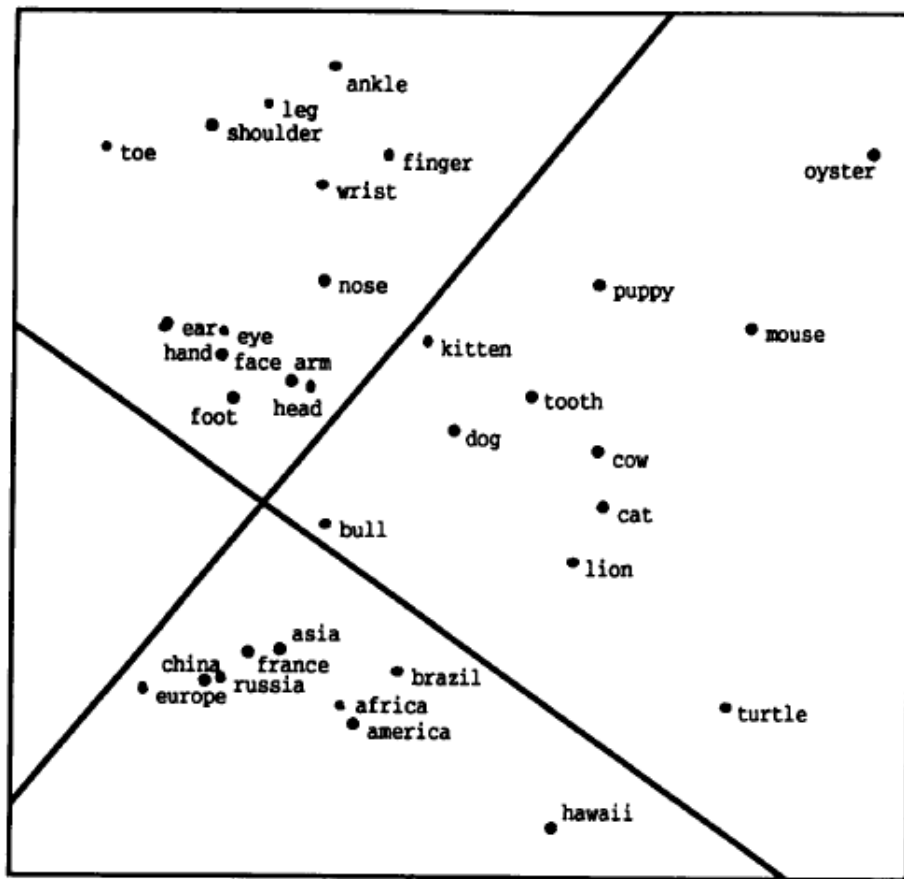
To explore how useful co-occurrence vectors are, a couple of experiments were conducted. For this, a corpus of 160 million words from Usenet Newsgroups was queried for co-occurrence. For every word that appeared at least 50 times within this corpus, word vectors similar to document vectors were calculated. For the calculation of the vectors, the co-occurrence frequency was used. In the next step, the Euclidian distance for each vector to each vector was calculated. Example results selected randomly from this processing can be seen in **Table 11.2**, where for each target the 5 nearest words are shown (LUND AND BURGESS; 1996; pp.203-205).

Target	n1	n2	n3	n4	n5
<b>Jugs</b>	Juice	Butter	Vinegar	Bottles	Cans
<b>Leningrad</b>	Rome	Iran	Dresden	Azerbaijan	Tibet
<b>Lipstick</b>	Lace	Pink	Cream	Purple	Soft
<b>Cardboard</b>	Plastic	Rubber	Glass	Thin	Tiny
<b>Triumph</b>	Beauty	Prime	Grand	Former	Rolling
<b>Monopoly</b>	Threat	Huge	Moral	Gun	Large

**Table 11.2** Five nearest neighbors for Target words (Source: Lund and Burgess 1996)

The relationship between two vectors appears to be both semantic (jugs-cans, cardboard-plastic) and associative (lipstick-lace, monopoly-threat), (LUND AND BURGESS; 1996; pp.203-205).

A second experiment tested if those vectors carry categorical information. The objective was to see if words that are perceived to be in the same category are also grouped by their corresponding vectors together in a category. Words that represent the categories animal names, body parts and geographical locations were selected for this test. The co-occurrence vectors for each word were extracted from the corpus. The Euclidean distance from every vector to every other vector was calculated and the resulting multidimensional space was scaled to a two-dimensional solution shown in **Figure 11.3** (LUND AND BURGESS; 1996; p. 205).



**Figure 11.3** Multidimensional scaling of co-occurrence vectors. (Source: Lund and Burgess 1996)

The results in **Figure 11.3** were enhanced by the lines added to clarify the differentiation of the categories. The geographic spaces are unlike either body parts or animals. The separation between body parts and animals also works well except for “tooth,” but intuitively it can be guessed that tooth is an important body part for animals. This is probably the reason why it gets clustered together with the animals. Overall, the experiment validates the assumption that words can be categorized to a certain degree using their co-occurrences without human supervision (LUND AND BURGESS; 1996; p.205).

### 11.5. Generating Co-occurrence query expansion groups from Wikipedia

The literature on query expansion is split. On the one hand, it is believed to introduce more noise, but increase the recall; on the other hand, query expansion could enhance document classifications. The automatic query expansion employed in this thesis based on co-occurrences is a simple one; there are more sophisticated methods. Examples of this are exploiting grammatical relationships between words, introducing a semantic term weight or utilizing Wikipedia to embed semantic

kernels into documents. To test which method works best, the addresses are going to be classified with both the query expansion and the search terms defined in **Chapter 10**. (see **Table 10.2**), (WANG AND DOMENICONI; 2008; pp. 713-721), (LUO ET AL.; 2011; pp. 12708-12716), (MANNING ET AL.; 2009; 189-194).

The code in **Figure 11.4** generates the co-occurrences used for the query expansion from the now POS-tagged Wikipedia.

```

025 def CoOccurrence(groups):
026     Starttime3 = time.time()
027     Fenster = 10
028     i = 1
029     S_list = stopwords_list()
030     word_dict = {}
031
032     Files = [x[2] for x in os.walk('./wikicorpuspickeld_2/')]
033     for file in Files[0]:
034
035         with io.open('./wikicorpuspickeld_2/'+file, 'rb') as fin:
036             loaded_corpus = pickle.load(fin)
037
038
039         for sentence in loaded_corpus:
040             for (index, tokentag) in enumerate(sentence):
041                 (token, tag) = tokentag
042                 token = token.lower()
043
044                 if token in groups:
045                     term = sentence[index-Fenster:index+Fenster]
046                     for (term_token, term_tag) in term:
047
048                         term_token = term_token.lower()
049                         if term_token not in S_list and NounVerb(term_tag):
050
051                             if token not in word_dict:
052                                 word_dict[token] = {}
053                             if term_token in word_dict[token]:
054                                 word_dict[token][term_token] += 1
055                             else:
056                                 word_dict[token][term_token] = 1
057
058             print i
059
060             delta_time = time.time() - Starttime3
061             print "time till end %.2f Minutes" %
062                 (((delta_time/60)/i)*(len(Files[0])-i))
063
064             i += 1
065
066     return word_dict
067
068 groups = [u'wohnen', u'arbeiten', u'bildung', u'einkaufen', u'gaststätte',
069           u'hotel', u'kreditinstitut', u'kultur', u'dienstgebäude',]
070
071 Starttime2 = time.time()
072 CoOccurrenceGroups = CoOccurrence(groups)

```

**Figure 11.4** Co-occurrence generation from Wikipedia

The `groups` are the lists of search terms (see **chapter 10**). They are passed as a python list to the function `CoOccurrence()`. The purpose of the functions is, if the one of the terms is found to look

forward and backward in the same sentence and record any verbs and nouns that occur within this window around the term. A couple of variables are defined to make this possible. Number one is `Fenster` containing the size of the window. The Second is `S_list`. Here the code calls a function not shown in **Figure 11.4** that delivers the German stop words found in the Natural Language Tool Kit (NLTK) in the form of a python list and, lastly, `word_dict` the dictionary that will contain the co-occurrences. The code from lines 32 to 36 opens the POS-tagged Wikipedia generated earlier in this chapter. Then, the script iterates through the separate sentences found in the POS-tagged Wikipedia. The tokens in the sentence are enumerated in line 40. The resulting `index` variable is used to save the position of the token in the sentence to make the window lookup in line 45 possible. So if the search term is in the `groups` list, the lookup gets triggered returning all words within the window size as a list of word and POS tag tuples in line 45. The words are then tested if they are stop words. Then they are tested again by the `NounVerb()` function to see if they are a verb or noun (see **Table 11.1**). If a word passes both tests, it is added to the `word_dict` dictionary. This happens in two phases. First, if the search term is not yet present in the dictionary as a key, it is added in line 52 containing a subdictionary as a value. Then, the word is either added as a new key to this subdictionary in line 56 or, if it is already present, plus one is added to the counter. This not only records the words, but also how many times they co-occur with the search term. This statistical connection will be used to create a weighted vector (see **chapter 9.2.** and **Chapter 12.2.3.**), (NLTK 3.0 library).

**Table 11.3** shows the now-produced co-occurrence groups. They still contain some non-information, like “=” and “]”, that is the result of tagging mistakes of the POS tagger and the way the text version of Wikipedia was formatted. These artifacts will be filtered out as soon as the groups are turned into vectors (see **chapter 12.2.3.**).

Search Term					
<b>hotel</b> <b>(25707)</b>	*	=	]	wurde	grand
	(2538)	(1926)	(1828)	(1378)	(1004)
<b>gaststätte</b> <b>(4111)</b>	wurde	heute	gebäude	straße	=
	(359)	(286)	(145)	(134)	(124)
<b>arbeiten</b> <b>(80323)</b>	=	beruf	rechtsanwalt	began	wurden
	(7759)	(4172)	t	(2687)	(2109)
			(3105)		
<b>bildung</b> <b>(47332)</b>	=	kultur	forschung	wissenschaft	bundeszentral
	(49506	(2334)	(2074)	(2059)	e
	)				(1786)
<b>wohnen</b> <b>(8240)</b>	=	bauen	menschen	arbeiten	haus
	(557)	(447)	(434)	(272)	(270)
<b>dienstgebäude</b> <b>(445)</b>	=	wurde	berlin	eisenbahndirektion	heute
	(111)	(29)	(26)	(20)	(19)
<b>kreditinstitut</b> <b>(1703)</b>	]	deutschland	schweiz	bank	österreich
	(970)	(360)	(146)	(81)	(77)
<b>einkaufen</b> <b>(1118)</b>	=	gehen	können	geht	konnten
	(218)	(74)	(62)	(46)	(29)
<b>kultur</b> <b>(80743)</b>	=	sehenswürdigkeiten	geschichte	kunst	wissenschaft
	(93309	(14092)	(5554)	(5180)	(2737)
	)				

**Table 11.3** Co-occurrence groups with top 5 terms and the number of their occurrences

## 12. Address Classification

This chapter is about bringing together various parts of the previous chapters; namely, the document vector and the co-occurrence groups derived from the POS tagged Wikipedia corpus. First, the final vector space that will contain all the document vectors will be created out of the combination of two other vector spaces. Then the vectors for the co-occurrence groups, the search terms and the HTML documents will be calculated and compared to each other. Lastly, the values created in this comparison are used to classify the addresses.

### 12.1. Creating the Vector Space

For classifying documents with document vectors, the vector space these document vectors can exist in must be created first. That means that a space exists that contains as many dimensions as individual terms or, in this case, stemmed tokens. The reason why individual stemmed tokens are used and not every individual word contained in the corpus is the same as described in **Chapter 7.1.1.** . The same word can differ for grammatical reasons or words can have similar meanings like in the examples “organize”, “organizes”, “organizing” and “democracy”, “democratic”, “democratization” (SALTON; 1991; pp. 974-980), (MANNING ET AL.; 2009; p. 123).

This chapter describes how two vector spaces, one from the HTML documents and one from the Wikipedia corpus, are created and merged into a combined vector space.

#### 12.1.1. Creating a unique set of HTML documents

Before the HTML file vector space can be created, there is an issue with the HTML documents itself that needs to be corrected first. Due to complications in the download process, namely, that it crashes or gets stuck a couple of times, it needs to be restarted (see **chapter 3.3.**). Because of how the index works and how the download script is coded, in the event of a crash it is unavoidable that some parts of already downloaded data will be downloaded again, thus creating duplicates. To create a dataset containing only unique HTML files, the following SQL command needs to be executed on the database:

```
INSERT INTO htmlunique SELECT DISTINCT ON (url) id, url, html_file, Vienna,
textsearchable_index_col, stripped_html, geocoded FROM html where geocoded
= TRUE
```

This selects the rows from the table that have a unique URL and are relevant because they are already geocoded and copies them exactly into the new table `htmlunique`. This ensures that the records work with the code and the database as they did before except for the index column. If now compared with the previous figures, there were 268,338 HTML files that have now been reduced to 256,180, a reduction of 4.53% (POSTGRESQL 9.3.9; documentation).

Even though this is only described now in the thesis, all previous chapters created tables, graphics and maps use the `htmlunique` table and not the `html` table. The reason for not doing this right after the import was that it is faster to create a set of unique files from 268,338 files than from 8.4 Million.



### 12.1.2. Creating the HTML documents Vector Space

```

045 def Delete_stopwords(Tokens):
046     return [token for token in Tokens if not token in
                                nltk.corpus.stopwords.words('german')]
047
048 GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
049 tokenizer = RegexpTokenizer(r'\w+')
050 token_dict = {}
051 HTMLIDS = get_html_ids()
052
053 lower = 0
054 upper = lower + 1000
055 Starttime = time.time()
056 parsedhtmls = 0
057
058 while lower <= len(HTMLIDS):
059
060     Starttime2 = time.time()
061     stripped_htmls_list = stripped_htmls(HTMLIDS[lower:upper])
062     for html in stripped_htmls_list:
063         parsedhtmls += 1
064
065         time_tokenize = time.time()
066         tokens = tokenizer.tokenize(html)
067         tokens = Delete_stopwords(tokens)
068         token_dict_file = {}
069
070         for token in tokens:
071
072             stemmedtoken = GermanStemmer.stem(token)
073
074             if stemmedtoken in token_dict_file:
075                 token_dict_file[stemmedtoken] += 1
076             else:
077                 token_dict_file[stemmedtoken] = 1
078
079         for key in token_dict_file:
080             if key in token_dict:
081                 doc_count = token_dict[key][0] + 1
082                 occurrence_count = token_dict[key][1] + token_dict_file[key]
083                 token_dict[key] = (doc_count, occurrence_count)
084             else:
085                 token_dict[key] = (1, token_dict_file[key])
086
087         print('Number of tokens in dict: %s' % len(token_dict))
088         print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
089         delta_time = time.time() - Starttime
090         print "time till now %.2f Minutes"%(delta_time / 60)
091         print "time till end %.2f Minutes"%(((delta_time/60)/(upper))*(len(HTMLIDS)-
                                (upper)))
092
093     lower += 1000
094     upper = lower + 1000
095
096 with io.open('./Vector/HTMLVectorSpace.pickle', 'wb') as fout:
097     pickle.dump(token_dict, fout)

```

**Figure 12.1** HTML Document vector space code

The first vector space created is the one derived from the HTML documents. To accomplish this, the code shown in **Figure 12.1** first fetches a batch of the HTML files that were stripped of their HTML

tags (line 61) as described in **Chapter 6.2**. The code then iterates through them starting in line 58. First, they get tokenized, split into word tokens, in line 66 with the help of the `tokenizer.tokenize()` function defined in line 49. This function returns the document as a list of tokens. From this list, with the help of the `Delete_stopwords()` method displayed in line 45 and 46, all German stop words that are defined in the Natural Language Tool Kit (NLTK) are deleted from the list (NLTK 3.0 LIBRARY; tokenize).

Then, the code iterates through all remaining tokens in the document starting in line 70. A token is then stemmed in line 72 with the NLTK snowball stemmer defined in line 48. It is then tested to see if the token already exists as a key in the token dictionary of this file `token_dict_file`. If so, plus one is added to the counter of the token. If not, the token is added as a new key in line 77 with a counter of 1. The dictionary is reset for every document, see line 64 (NLTK 3.0 LIBRARY; tokenize), (NLTK 3.0 LIBRARY; stem).

The keys and counters of `token_dict_file` are then fed into the `token_dict`, see lines 79 to 84. It is the same principal as used before with the `token_dict_file`. If a token already exists as a key in `token_dict`, the document frequency is increased by one. The counter for corpus frequency is increased by the counter value that the `token_dict_file` holds. The first count is the document frequency and the other one the collection frequency. Both are described in detail in **Chapter 9.2**. The document frequency can later be used to calculate the tf-idf vector of documents. When the code has parsed all HTML documents and added all individual tokens to the `token_dict`, the dictionary is serialized with pickle and saved to the drive in lines 96 and 97 (PYTHON 2.7.10 LIBRARY; I/O)(PYTHON 2.7.10 LIBRARY; pickle).

### 12.1.3. Creating the Wikipedia Vector Space

Because the co-occurrence groups for the query expansion are derived from a different corpus, a vector space for the Wikipedia corpus also needs to be created. The two vector spaces will then later be combined into one vector space. As can be seen in **Figure 12.2**, the Wikipedia vector space is similarly created to the previously described HTML documents vector space.

```

011 def Vector_Calculator():
012     Starttime3 = time.time()
013     i = 1
014     GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
015     token_dict_file = {}
016     p = re.compile(ur'^[a-zA-ZäöüßÄÖÜ]{2,}$', re.UNICODE)
017
018     Files = [x[2] for x in os.walk('./wikicorpuspickeld_2/')]
019     for file in Files[0]:
020         with io.open('./wikicorpuspickeld_2/'+file, 'rb') as fin:
021             loaded_corpus = pickle.load(fin)
022
023             for sentence in loaded_corpus:
024                 for (index, tagtuple) in enumerate(sentence):
025                     (token, tag) = tagtuple
026                     token = token.lower()
027                     if token not in stopword_list:
028                         if p.match(token):
029                             stemmedtoken = GermanStemmer.stem(token)
030
031                             if stemmedtoken in token_dict_file:
032                                 token_dict_file[stemmedtoken] += 1
033                             else:
034                                 token_dict_file[stemmedtoken] = 1
035
036             delta_time = time.time() - Starttime3
037             print "time till end %.2f Minutes" % (((delta_time/60)/i)*(len(Files[0])-i))
038
039             i += 1
040
041     return token_dict_file
042
043 stopword_list = []
044 for word in stopwords.words('german'):
045     stopword_list.append(unicode(word.decode('latin-1')))
046
047 Starttime = time.time()
048 Vectorraum = Vector_Calculator()
049
050 with io.open('./Vector/WikiVectorSpace2.pickle', 'wb') as fout:
051     pickle.dump(Vectorraum, fout)
052
053 print('Operation took %.2f Minutes' % ((time.time() - Starttime) / 60))

```

**Figure 12.2** Wikipedia vector space code

But unlike with HTML documents, vector space it is not necessary to record the document frequency for different tokens. The reason behind this is that the Wikipedia corpus is not the corpus that information retrieval algorithms are used on.

The POS tagged Wikipedia corpus is loaded into the code file by file in line 20 and 21. Because this corpus is structured as a list of sentences that contain a list of words and POS tag tuples, the tokens needs to be unpacked. This is done in lines 23 to 25. The resulting `token` variable then contains a string. This string is changed to all lowercase characters and checked against the stop word list. The next step in line 28 is to test if `token` passes the defined regular expression criteria: to only consist of letters and be at least 2 letters long. It is then stemmed in line 29. The token is then either newly added as a key with the value 1 to the `token_dict_file` in line 34 or, if it already exists, plus one is

added to the counter. This process is repeated until all Wikipedia corpus files are processed (PYTHON 2.7.10 LIBRARY; I/O), (PYTHON 2.7.10 LIBRARY; pickle), (PYTHON 2.7.10 LIBRARY; regular expression operations), (NLTK 3.0 LIBRARY; stem).

#### 12.1.4. Combined Vector Space

Now to combine both vectors spaces, the code in **Figure 12.3** is used.

```

004 with io.open('./Vector/WikiVectorSpace.pickle', 'rb') as fin:
005     WikiVectorSpace = pickle.load(fin)
006
007 with io.open('./Vector/HTMLVectorSpace.pickle', 'rb') as fin:
008     HTMLVectorSpace = pickle.load(fin)
009
010 CombinedVectorSpace = {}
011
012 for key in WikiVectorSpace:
013     if key in HTMLVectorSpace:
014         CombinedVectorSpace[key] = HTMLVectorSpace[key]
015
016 with io.open('./Vector/CombinedVectorSpace.pickle', 'wb') as fout:
017     pickle.dump(CombinedVectorSpace, fout)

```

**Figure 12.3** Combine Wikipedia and HTML File vector space

Both vector space dictionaries are deserialized in lines 04, 05 and lines 7, 8. Thereby, the `CombinedVectorSpace` dictionary that will contain the new vector space is created. The code then iterates through the keys found in `WikiVectorSpace`. As described in the previous chapters, the keys represent the individual tokens found in the respective corpora. In line 13, the code checks if the key is also present in `HTMLVectorSpace`. If this is true, the key is added as a key to the `CombinedVectorSpace` with the document frequency counter and the collection frequency counter stored in the `HTMLVectorSpace` as a value. The `CombinedVectorSpace` is then serialized in line 16 and 17 (PYTHON 2.7.10 LIBRARY; I/O), (PYTHON 2.7.10 LIBRARY; pickle).

To combine the two vector spaces with an intersection instead of a union has two main advantages. First, it filters out garbage tokens. Because the HTML documents are raw documents from the Internet, they contain nonsensical string combinations (xsdf, ddjdj, lkhj) even after the filtering. These should not or to a much lesser degree exist in the Wikipedia corpus. The second advantage is that, to a certain degree, foreign languages are filtered. Again the same reason as before the HTML documents could possibly contain all sorts of none German languages and up until now none German languages have not been filtered out.

As described later in this chapter, when the document vectors for the HTML documents are created, these now no longer existing tokens are simply ignored, like stop words. They have no influence on the resulting document vector. The combined vector space consists of 610753 tokens.

## 12.2. Calculating the idf-tf vectors

With the vector space created, the idf-tf vectors for HTML documents and co-occurrence groups for the query expansion can be produced. For this, first the inverse document frequency for every token is calculated. Then, the idf-tf vectors for the HTML documents and the co-occurrence groups are calculated.

### 12.2.1. Calculating Inverse Document Frequency per Term

The Inverse document frequency (idf) for every term is calculated with the code in **Figure 12.4**.

```

024 with io.open('./Vector/CombinedVectorSpace.pickle', 'rb') as fin:
025     CombinedVectorSpace = pickle.load(fin)
026
027 CombinedVectorSpaceIDFT = {}
028 DocumentCount = countrows()
029
030 for key in CombinedVectorSpace:
031     idft = numpy.log(numpy.divide(float(DocumentCount),
                                float((1+CombinedVectorSpace[key][0]))))
032
033     CombinedVectorSpaceIDFT[key] = CombinedVectorSpace[key][0],
                                CombinedVectorSpace[key][1], idft
034
035 with io.open('./Vector/CombinedVectorSpaceIDFT.pickle', 'wb') as fout:
036     pickle.dump(CombinedVectorSpaceIDFT, fout)
037
038 CombinedVectorSpaceIDFTKeyList = []
039
040 for key in CombinedVectorSpaceIDFT:
041     CombinedVectorSpaceIDFTKeyList.append(key)
042
043 CombinedVectorSpaceIDFTKeyList.sort()
044
045 with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'wb') as fout:
046     pickle.dump(CombinedVectorSpaceIDFTKeyList, fout)

```

**Figure 12.4** Code for inverse document frequency calculation

The vector space created in the last subchapter is loaded into the script as `CombinedVectorSpace`. A new dictionary `CombinedVectorSpaceIDFT` that will, at the end of the script, contain all tokens with their respective idf weight is created in line 27. Line 28 calls a function that returns the absolute

document count of all unique and geocoded html documents. As described in **chapter 9.2.**, the absolute document count is one of the variables used in the idf formula (PYTHON 2.7.10 LIBRARY; I/O), (PYTHON 2.7.10 LIBRARY; pickle), (SALTON; 1991; pp. 976).

The formula is:

$$idf_t = \log \frac{N}{df_t}$$

$idf_t$  (inverse document frequency of the term),  $N$ (total number of documents),  $df_t$ (document frequency of the term)

The next step is iterating through all tokens in the vector space dictionary. Every token in the the idf is calculated utilizing the formula. This happens in line 31 and the NumPy library is used for this. The reason for using this specialized library is that floating point calculations are problematic for computers and NumPy takes care of these problems.

The calculate idf stored in the variable `idft` is then added together with document frequency value and the collection frequency value to the new dictionary `CombinedVectorSpaceIDFT` with the token again serving as the key (NumPy 1.8.1 library;).

`CombinedVectorSpaceIDFT` is serialized and saved to the drive in lines 35 and 36. There is another step to creating the vector space. Because keys in python dictionaries are not always in the same order, a vehicle to preserve order needs to be created. This is done by adding all keys to a list in lines 40 and 41. A list can be sorted, resulting always in the same order. This is important because as soon as the vector space is presented mathematically in an array, every token has to always refer to the same dimensional position in the array. After the list is sorted, it is also serialized and saved to the drive in lines 35 and 36.

### 12.2.2. Calculating the Term Frequency-Inverse Document Frequency Vector for HTML Files

Now with the idf calculated for every token, the vectors for the documents can be calculated. First, all HTML documents have to be tokenized and the occurrence of the tokens within the documents have to be counted. This is done by the code in **Figure 12.5**.

```

061 createColumn()
062
063 offset = 0
064 htmls = ReadFromHTML(offset)
065 tokenizer = RegexpTokenizer(r'\w+')
066 GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
067 Starttime = time.time()
068 Length = countrows()
069
070 while htmls:
071     print len(htmls)
072     Starttime2 = time.time()
073     for html in htmls:
074         HTMLdict = {}
075         id, HTMLtext = html
076         tokens = tokenizer.tokenize(HTMLtext)
077
078         for token in tokens:
079             stemmedtoken = GermanStemmer.stem(token)
080             if stemmedtoken in CombinedVectorSpace:
081                 if stemmedtoken in HTMLdict:
082                     HTMLdict[stemmedtoken] += 1
083                 else:
084                     HTMLdict[stemmedtoken] = 1
085
086         htmlDictpickeld = pickle.dumps(HTMLdict)
087         UpdateHtmlUniquewithDict(htmlDictpickeld,id)
088
089     offset += 1000
090     print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
091     delta_time = time.time() - Starttime
092     print "time till now %.2f Minutes"%(delta_time / 60)
093     print "time till end %.2f Minutes"%(((delta_time/60)/offset)*(Length-offset))
094     htmls = ReadFromHTML(offset)

```

**Figure 12.5** HTML document tokenization

In line 61, a function is called to create a new column `VectorDICT` in the `htmlunique` table that will contain an HTML file-specific dictionary of tokens and their number occurrences in the file. The column is of the type `bytea`, a column type that PostgreSQL offers to store binary data. With that, it is possible to store pickled objects in the database (POSTGRESQL 9.3.9; documentation).

The tag stripped HTML documents are fetched from the database and the tokenization is started. It works analogously to the HTML vector space creation (see **chapter 12.1.2.**). The `HTMLtext` is split into tokens with the NLTK `RegexpTokenizer()` in line 76. The code then iterates through the tokens and stems them. Stop words and similar noise are not dismissed, but because they are not part of the vector space (see **chapter 12.1.**), they cannot be mapped to the final document vector. The stemmed tokens are added to the HTML document-specific dictionary as keys. If they already exist in the dictionary, plus one is added to the token counter. The dictionary is then serialized in line 86 and saved to the `VectorDICT` column of the corresponding html document (PYTHON 2.7.10 LIBRARY; pickle), (NLTK 3.0 LIBRARY; tokenize), (NLTK 3.0 LIBRARY; stem).

With the HTML files tokenized and the term frequency (tf) for the tokens set, the last step is to create the normalized tf-idf document vector.

```

049 offset = 0
050 length = countrows()
051 Starttime = time.time()
052 createColumn()
053
054 with io.open('./Vector/CombinedVectorSpaceIDFT.pickle', 'rb') as fin:
055     CombinedVectorSpaceIDFT = pickle.load(fin)
056
057 with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'rb') as fin:
058     CombinedVectorSpaceIDFTKeyList = pickle.load(fin)
059
060 while offset <= length:
061     dicts = VectorDICTReader(offset)
062     Starttime2 = time.time()
063     arraylist = []
064     for tuple in dicts:
065         array = []
066         id = tuple[0]
067         dictionary = pickle.loads(str(tuple[1]))
068
069         for key in CombinedVectorSpaceIDFTKeyList:
070
071             if key in dictionary:
072                 array.append(numpy.multiply(CombinedVectorSpaceIDFT[key][2],
073                                             dictionary[key]))
074
075             else:
076                 array.append(0)
077
078         array = numpy.array(array)
079         array = numpy.divide(array, numpy.linalg.norm(array))
080         array = pickle.dumps(array)
081         array = zlib.compress(array)
082         arraylist.append((psycpg2.Binary(array), id,))
083
084     UpdateHtmlUniquewithTFIDFlist(arraylist)
085     offset += 100
086     print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
087     delta_time = time.time() - Starttime
088     print "time till now %.2f Minutes"%(delta_time / 60)
089     print "time till end %.2f Minutes"%(((delta_time/60)/offset)*(length-offset))

```

**Figure 12.6** tf-idf vector for documents

The code in **Figure 12.6** creates another bytea column with the name `TFIDFVector` in the table `htmlunique`. Also loaded into the script is the combined vector space in lines 54 and 55 and the key list for the vector space in lines 57 and 58. The key list makes sure that the tokens are always called in the same order and therefore always related to the same dimension in the array created with this script. The dictionaries containing the tokens and their respective term frequency are loaded from the database in line 61. `arraylist` is created in line 63 and then the code begins to iterate through the fetched dictionary. For every dictionary, a new empty `array` is generated in line 65. The dictionary is deserialized in line 67 and then the code iterates through the `CombinedVectorSpaceIDFTKeyList`. If a key on the list is found in the dictionary belonging to the



html file, the term frequency is multiplied by the inverse document frequency and added to `array`. The `.append()` method makes sure that the product is added to the end of `array` and, because zero is added to `array` in line 74 in case the key is not found in the HTML file dictionary, the same token is always represented by the same position in `array` (PYTHON 2.7.10 LIBRARY; pickle), (PSYCOPG 2.5.3 LIBRARY), (PYTHON 2.7.10 LIBRARY; I/O).

The list array is then transformed into a NumPy array with `numpy.array()` in line 75. This makes it possible to use the NumPy library on the array. This library is then used in the next step to normalize the vector in line 76. Line 77 serializes the vector and, because a vector consisting of 610,753 dimensions takes up a lot of space, the serialized object is compressed with `zlib.compress()` in line 78. In line 79, `array` is made into a PostgreSQL binary data object and added together with the id to `arraylist`. `arraylist` is then passed to the `UpdateHtmlUniquewithTFIDFlist()` to write the normalized tf-idf vectors to the Database (NUMPY 1.8.1 LIBRARY;), (PSYCOPG 2.5.3 LIBRARY), (PYTHON 2.7.10 LIBRARY; zlib).

### 12.2.3. Calculating the tf-idf Vector for Wikipedia Co-Occurrences groups and search terms

Like with the HTML documents, the vector also has to be calculated for co-occurrence groups and search terms as well. This is the objective of the code in **Figure 12.7**.

```

008 with io.open('./Vector/CombinedVectorSpaceIDFT.pickle', 'rb') as fin:
009     CombinedVectorSpaceIDFT = pickle.load(fin)
010
011 with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'rb') as fin:
012     CombinedVectorSpaceIDFTKeyList = pickle.load(fin)
013
014 with io.open('./Co-Occurrence.pickle', 'rb') as fin:
015     CoOc = pickle.load(fin)
016
017 GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
018
019
020 TFIDF_CoOc = {}
021 for searchterm in CoOc:
022     TFIDF_CoOc[searchterm] = {}
023     TFIDF_CoOc[searchterm]['Stemmed'] = {}
024     for token, counter in CoOc[searchterm]:
025         token = GermanStemmer.stem(token)
026         if token in TFIDF_CoOc[searchterm]:
027             TFIDF_CoOc[searchterm]['Stemmed'][token] =
                 TFIDF_CoOc[searchterm]['Stemmed'][token]+counter
028         else:
029             TFIDF_CoOc[searchterm]['Stemmed'][token] = counter
030
031
032 for searchterm in TFIDF_CoOc:
033     array = []
034     dictionary = TFIDF_CoOc[searchterm]['Stemmed']
035     for key in CombinedVectorSpaceIDFTKeyList:
036         if key in dictionary:
037             array.append(numpy.multiply(CombinedVectorSpaceIDFT[key][2],
                                         dictionary[key]))
038         else:
039             array.append(0)
040     array = numpy.array(array)
041     array = numpy.divide(array, numpy.linalg.norm(array))
042     array = pickle.dumps(array)
043     array = zlib.compress(array)
044     TFIDF_CoOc[searchterm]['TFIDF_CoOc'] = array
045
046
047     STarray = []
048     searchtermstemmed = GermanStemmer.stem(searchterm)
049     for key in CombinedVectorSpaceIDFTKeyList:
050         if key is searchtermstemmed:
051             STarray.append(numpy.multiply(CombinedVectorSpaceIDFT[key][2], 1))
052         else:
053             STarray.append(0)
054
055     STarray = numpy.array(STarray)
056     STarray = numpy.divide(STarray, numpy.linalg.norm(STarray))
057     STarray = pickle.dumps(STarray)
058     STarray = zlib.compress(STarray)
059     TFIDF_CoOc[searchterm]['TFIDF_ST'] = STarray
060
061 with io.open('./Vector/TFIDF_CoOc.pickle', 'wb') as fout:
062     pickle.dump(TFIDF_CoOc, fout)

```

**Figure 12.7** Normalized tf-idf vector for co-occurrence groups and search terms

Lines 8 to 15 load the combined vector space, the key list for the combined vector space and the co-occurrence groups, the creation of which **Chapter 11.5.** describes. A new dictionary `TFIDF_CoOc` that will contain the vector arrays is created in line 20. The code then iterates through the keys of the

`CoOc` dictionary beginning in line 21. For every key or `searchterm`, a new sub dictionary is created within `TFIDF_CoOc`. In every sub dictionary, another sub dictionary is created in line 35 behind the key `'Stemmed'` that will contain the stemmed tokens and their counts. The next step is to stem the tokens, combine possible duplicates and save the results to the new `TFIDF_CoOc` in lines 24 to 29. The approach is similar to the work earlier described in this chapter (PYTHON 2.7.10 LIBRARY; I/O), (PYTHON 2.7.10 LIBRARY; pickle).

In comparison to **Table 11.3**, now **Table 12.1** contains the cleaned up and tokenized versions of the co-occurrence groups.

Search Term						
<b>hotel</b>	restaurant (578)	the (603)	heut (876)	grand (1004)	wurd (1378)	hotel (25707)
<b>gaststätte</b>	befindet (103)	wohnhaus (120)	strass (134)	gebaud (145)	heut (286)	wurd (359)
<b>arbeiten</b>	jahr (1300)	wurd (2109)	began (2687)	rechtsanwalt (3105)	beruf (4172)	arbeit (80323)
<b>bildung</b>	wurd (1283)	bundeszentral (1786)	wissenschaft (2059)	forschung (2074)	kultur (2334)	bildung (47332)
<b>wohnen</b>	einwohn (255)	haus (270)	arbeit (272)	mensch (434)	bau (447)	wohn (8240)
<b>dienstgebäude</b>	hannov (14)	beflagg (14)	munch (17)	heut (19)	berlin (26)	wurd (29)
<b>kreditinstitut</b>	vereinigt (73)	osterreich (77)	bank (81)	schweiz (146)	Deutschland (360)	kreditinstitut (1703)
<b>einkaufen</b>	geld (20)	kund (23)	konnt (29)	geht (46)	geh (74)	einkauf (1118)
<b>kultur</b>	gesellschaft (2618)	wissenschaft (2737)	kunst (5180)	geschichte (5544)	sehenswurd (14092)	kultur (80743)

**Table 12.1** Tokenized co-occurrence groups with top 5 terms and the number of their occurrences

Some predictions and observations can be made about **Table 12.1**. An interesting irregularity is the token “rechtsanwalt” (attorney) in the class “arbeiten” (working). This will make classification in this class interesting, because it might classify addresses that have an attorney’s office incorrectly.

With the term frequency and stemmed tokens available, the actual vector can be created. In this, the co-occurrence frequency will be used as the term frequency, thus making the co-occurrence groups a kind of pseudo-document on the topic of the search term used to create them. As described with the vector creation of the HTML documents, it is important that the sequence of the tokens is preserved. That is why the code iterates through the `CombinedVectorSpaceIDFTKeyList` in line 35. If a token on the list is also found in the dictionary containing the tokens of the co-occurrence group, then the frequency of the co-occurrence is multiplied with the inverse document frequency of this token and appended to `array` in line 37. If the token is not part of the co-

occurrence group, then zero is added to `array` in line 39. `array` is then made into a NumPy array, normalized, serialized and compressed in the lines 40 to 43. Finally, it is added to the sub dictionary corresponding to the `searchterm` with the key `'TFIDF_CoOc'` in line 44.

The whole process is repeated for the search terms as well. Because, as described in **Chapter 9.1.**, they are to be used as classifiers as well. The search term is stemmed and an array with just the search term in it is created. Again, this array is made into a NumPy array, normalized, serialized, compressed and added to the sub dictionary with the key `'TFIDF_ST'` (NumPy 1.8.1 library;), (Python 2.7.10 library; pickle), (Python 2.7.10 library; zlib).

As soon as this process is repeated for all keys in the `TFIDF_CoOc` dictionary, it is serialized and saved to the drive in line 61 and 62 (Python 2.7.10 library; I/O), (Python 2.7.10 library; pickle).

It is now possible to create similarity matrixes, shown in **Tables 12.2** and **12.3**, respectively. However, there is not really a point in creating the matrix between the different search terms because their vectors represent just one word. The comparison to the co-occurrence groups is interesting. It becomes clear that the co-occurrence groups are sometimes similar to each and therefore will create more noise but, on the other hand, the recall of each group is broadened (see **chapter 11.5.**).

Also similarities between co-occurrence that could have been suspected with the help of **Table 12.1** now become clear. There seems to be some similarity between “gaststätte” (restaurant) and “hotel” (hotel), as well as between “bildung” (education) and “kultur” (culture). Both don’t intuitively seem too surprising. But the similarity between “dienstgebäude” (government building) and “gaststätte” (restaurant) is. Intuitively, there seems to be no connection between them. Some same strange similarities also exist between the groups “hotel”, “dienstgebäude” and “wohnen”, “dienstgebäude”.

	hotel	gaststätte	arbeiten	bildung	wohnen	dienstgebäude	kreditinstitut	einkaufen	kultur
hotel	1.00	0.17	0.01	0.00	0.02	0.06	0.00	0.00	0.00
gaststätte	0.17	1.00	0.04	0.02	0.06	0.35	0.01	0.02	0.02
arbeiten	0.01	0.04	1.00	0.01	0.04	0.03	0.00	0.01	0.01
bildung	0.00	0.02	0.01	1.00	0.01	0.02	0.00	0.00	0.07
wohnen	0.02	0.06	0.04	0.01	1.00	0.11	0.00	0.02	0.01
dienstgebäude	0.06	0.35	0.03	0.02	0.11	1.00	0.01	0.02	0.02
kreditinstitut	0.00	0.01	0.00	0.00	0.00	0.01	1.00	0.00	0.00
einkaufen	0.00	0.02	0.01	0.00	0.02	0.02	0.00	1.00	0.00
kultur	0.00	0.02	0.01	0.07	0.01	0.02	0.00	0.00	1.00

**Table 12.2** Similarity matrix co-occurrence groups

	hotel	gaststätte	arbeiten	bildung	wohnen	dienstgebäude	kreditinstitut	einkaufen	kultur
hotel	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
gaststätte	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
arbeiten	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.05	0.00
bildung	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
wohnen	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
dienstgebäude	0.00	0.00	0.00	0.00	0.00	1.00	0.06	0.02	0.00
kreditinstitut	0.00	0.00	0.00	0.00	0.00	0.06	1.00	0.00	0.01
einkaufen	0.00	0.00	0.05	0.00	0.00	0.02	0.00	1.00	0.00
kultur	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	1.00

**Table 12.3** Similarity matrix search terms

#### 12.2.4. Cosine Similarity Calculations

To bring this chapter to a close the Cosine Similarity can now be calculated between HTML documents, co-occurrence groups and search terms. The code shown in **Figure 12.8** is utilized for this calculation.

```

065 with io.open('./Vector/TFIDF_CoOc.pickle', 'rb') as fin:
066     TFIDF_CoOc = pickle.load(fin)
067
068 Columnlist = []
069 SearchTermList = []
070 TFIDFworkingdict = {}
071
072 for searchterm in TFIDF_CoOc:
073     SearchTermList.append(searchterm)
074     Columnlist.append(TFIDF_CoOc[searchterm]+'_CoOc')
075     Columnlist.append(TFIDF_CoOc[searchterm]+'_ST')
076
077 for searchterm in SearchTermList:
078     TFIDFworkingdict[searchterm] =
079         pickle.loads(zlib.decompress(TFIDF_CoOc[searchterm]['TFIDF_CoOc'])),\
080         pickle.loads(zlib.decompress(TFIDF_CoOc[searchterm]['TFIDF_ST']))
081
082 for searchterm in Columnlist:
083     createColumn(searchterm)
084
085 sqlstring = Sqlstringconstructor(Columnlist)
086
087 offset = 0
088 range = 1000
089 length = countrows()
090
091 vectors = VectorDICTReader(range,offset)
092
093 Starttime = time.time()
094 while vectors:
095     Starttime2 = time.time()
096     updatelist = []
097     for vectortup in vectors:
098         cosinelist = []
099         id, vector = vectortup[0], pickle.loads(zlib.decompress(vectortup[1]))
100         for searchterm in SearchTermList:
101             cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][0],
102                                             vector),8)
103             cosinelist.append(cosine)
104             cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][1],
105                                             vector),8)
106             cosinelist.append(cosine)
107             cosinelist.append(id)
108             updatelist.append(cosinelist)
109         UpdateHtmlUniquewithCosinelist(sqlstring,updatelist)
110         offset += range
111         print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
112         delta_time = time.time() - Starttime
113         print "time till now %.2f Minutes"%(delta_time / 60)
114         print "time till end %.2f Minutes"%(((delta_time/60)/offset)*(length-offset))
115
116 vectors = VectorDICTReader(range,offset)

```

**Figure 12.8** Cosine similarity calculation

The first thing is to load the pickled dictionary containing the vectors of the co-occurrence groups and search terms in lines 65 and 66. To keep everything in sync, two new lists are needed `Columnlist` and `SearchTermList`. As described before, those two lists make sure that the iteration process stays synchronized. Both lists are filled with items lines 72 to 75. The `SearchTermList` contains the dictionary keys and the `Columnlist` contains the list columns where the corresponding

Euclidean dot product of the cosine similarity calculation is stored. In the database, there will be two columns for every search term, one for the co-occurrence groups ending in “\_CoOc” and one only for the search term ending in “\_ST”. All columns are created with lines 81 and 82. The third important variable is `TFIDFworkingdict`, a dictionary that will contain the uncompressed vectors for both search terms and co-occurrence groups, after the lines 77 to 79 have been executed. The reason to offload the vectors is to not have to uncompress them every time. The function `Sqlstringconstructor()` that is called in line 84 creates an SQL string that contains value place holders in the exact order in which the cosine similarity is later calculated in the code. Lastly, the first batch HTML document vectors is fetched from the database and stored in the variable `vectors` (PYTHON 2.7.10 LIBRARY; I/O), (PYTHON 2.7.10 LIBRARY; pickle), (PYTHON 2.7.10 LIBRARY; zlib).

The code then iterates through the HTML document vectors and creates cosine similarity products for every search term and co-occurrence group. For this, the individual vector is decompressed in line 98 and then the code iterates through the search term list, calling the respective vectors from the `TFIDFworkingdict` dictionary. For the calculation of the cosine, the `cosine_similarity()` function from the scikit-learn library is used. The results are added to the temporary `cosinelist`, which in turn is combined with the HTML document ID to the `updatelist`. All this happens in lines 99 to 106 and then the `updatelist` is passed on, together with the blueprint SQL statement, to the `UpdateHtmlUniquewithCosinelist()` function that adds the cosine similarity results to the respective columns of the HTML documents table (PYTHON 2.7.10 LIBRARY; pickle), (PYTHON 2.7.10 LIBRARY; zlib), (SCIKIT LEARN 0.14.1 LIBRARY; Pairwise metrics, Affinities and Kernels).

### 12.3. Address Classification

Classification happens in two steps because there are two problems to overcome. The first problem is how to best summarize the values for every class at every address. The second is to decide to which classes the address belongs according to the values. To illustrate the first problem further, there is **Table 12.4** with some example values.

	Class 1	Class 2	class 3
Website 1	8	0	1
Website 2	7	1	0
Website 3	9	0	2
Website 4	6	0	1
Website 5	5	10	2
Website 6	3	10	3
Website 7	5	9	0
Website 8	10	0	0
Website 9	9	0	0
Website 10	7	2	2
Mean	6.9	3.2	1.1
Proposal	8.3	9.6	2.25

**Table 12.4** Classification problem number one

**Table 12.4** is a fictional example of Websites 1 to 10 that are all associated with the same address. There are 3 classes and the websites are rated between highly associated with the class (10) and not associated with the class (0). Now, the classes that the address is associated with need to be selected. The first approach would be to create the mean value for every class leading to an unsatisfactory result for class 2. The data shows that there are 3 websites that suggest a strong association with class 2, but because the other websites show no association, the mean value is relatively low. This can be a problem with the real data as well. It is possible that an address is associated with 60 websites, 50 point towards the restaurant class and 10 towards a shopping class. Maybe there are just more websites describing the restaurant than the store. But dismissing the shopping class would probably be an error because there are 10 other websites indicating this class. To overcome this problem, the code in **Figure 12.9** is used implementing a simplified clustering algorithm.



```

062 def Breaks(valuelist, index):
063     firstrun = True
064     highlist = []
065     lowlist = []
066     newlist = []
067
068     for item in valuelist:
069         newlist.append(item[index])
070
071     while newlist:
072
073         high = max(newlist)
074         low = min(newlist)
075
076         if firstrun and len(newlist) == 1:
077             highlist.append(high)
078             newlist.remove(high)
079             lowlist.append(low)
080             firstrun = False
081
082         elif firstrun:
083             highlist.append(high)
084             newlist.remove(high)
085             lowlist.append(low)
086             newlist.remove(low)
087             firstrun = False
088
089         elif high == low:
090             if numpy.absolute(high-numpy.mean(highlist)) < numpy.absolute(high-
                                numpy.mean(lowlist)):
091                 highlist.append(high)
092                 newlist.remove(high)
093             else:
094                 lowlist.append(high)
095                 newlist.remove(high)
096         else:
097             if numpy.absolute(high-numpy.mean(highlist)) < numpy.absolute(high-
                                numpy.mean(lowlist)):
098                 highlist.append(high)
099                 newlist.remove(high)
100             else:
101                 lowlist.append(high)
102                 newlist.remove(high)
103
104             if numpy.absolute(low-numpy.mean(lowlist)) < numpy.absolute(low-
                                numpy.mean(highlist)):
105                 lowlist.append(low)
106                 newlist.remove(low)
107             else:
108                 highlist.append(low)
109                 newlist.remove(low)
110
111     return numpy.mean(highlist)

```

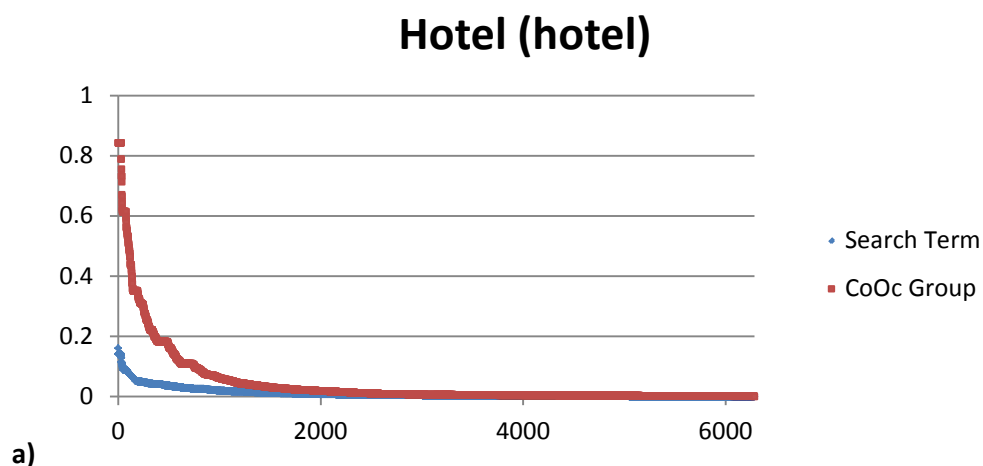
**Figure 12.9** Breaks code example

The implementation is loosely based on clustering values that are clumped together by similarity. The advantage of one-dimensional data is that minimum and maximum are known. And that is where the algorithm starts. After presorting the list in lines 68 and 69, which has to do with the format returned from the database, minimum and maximum are fetched from `newlist` in line 73 and 74 and saved to the variables `high` and `low`. If `firstrun` is `True` and `newlist` is only 1 long (i.e.

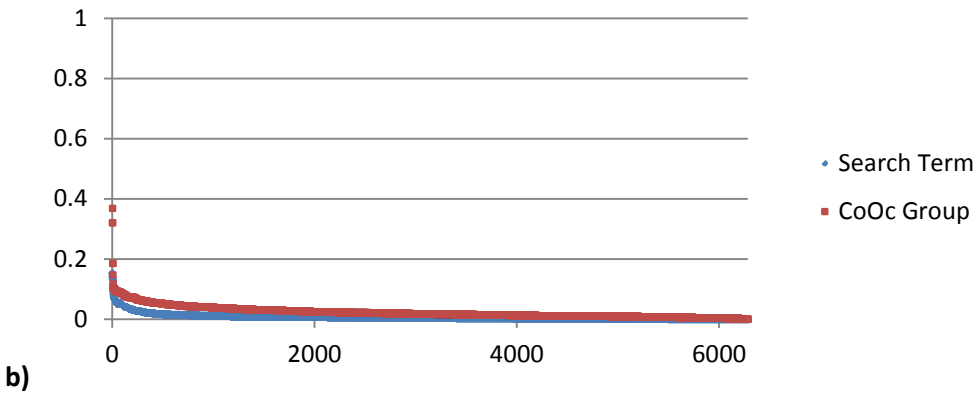
only one website is associated with the address), the special case in line 76 to 80 is invoked. This essentially does nothing but return the one value back in line 111. If `newlist` is longer though, then `highlist` and `lowlist` are appended with their first values `high` and `low`, lines 82 to 87, respectively. After this, `firststrun` is set to `False`. Both `high` and `low` are removed from `newlist` and the iteration begins again in lines 73 and 74 getting a new `high` and `low`, now the new highest and lowest value in the `newlist`. Both high and low have now been tested, whose mean value of `highlist` and `lowlist` is closer to their own value. They are then appended to the list that is more similar (closer) to them and removed from `newlist`. This process is repeated until no values are left in `newlist`. All of this happens in lines 96 to 109. One special case that can happen when, for example, only one value left in `newlist` is handled by the code in lines 89 to 95. In the end, the function returns the mean value of `highlist` (BAHRENBURG ET AL.; 2008; pp. 259 - 262), (NUMPY 1.8.1 LIBRARY).

The code clusters the data series in two parts: one containing the low values and one containing the high values and in the end dismissing the low values and just returning the mean of the high values. The results when used with the data in **Table 12.4** can be seen in the “proposal” row. Now the address gets high values in both class 1 and 2. A class that only contains low values will still return a low value as seen with class 3. But like in the example of class 2, if there are a few high values, a high overall class value is calculated (BAHRENBURG ET AL.; 2008; pp. 259 - 262), (NUMPY 1.8.1 LIBRARY).

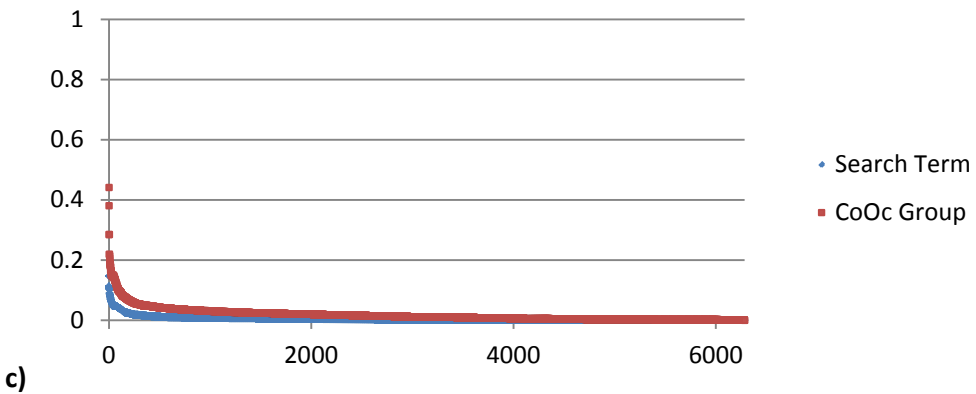
The resulting value distribution of those calculations for every class in **Table 10.1** are shown in **Figure 12.10 a) - i)**. Every value distribution features the values of the co-Occurrence groups and the search term vectors.



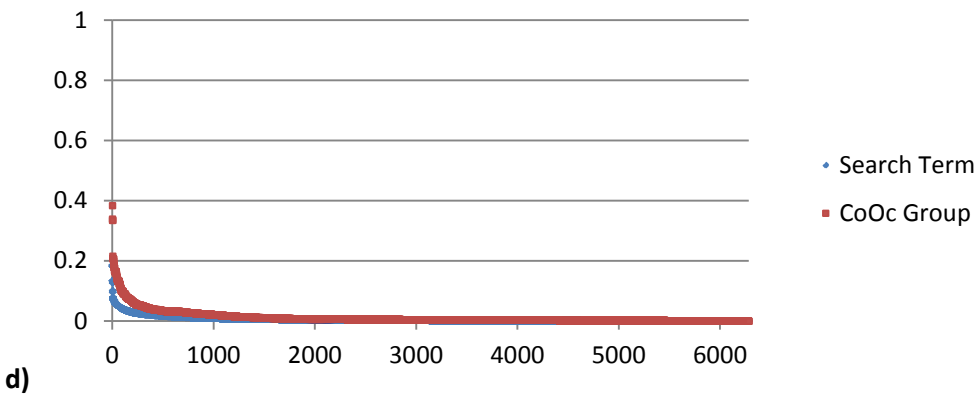
### Gaststätte (restaurant)



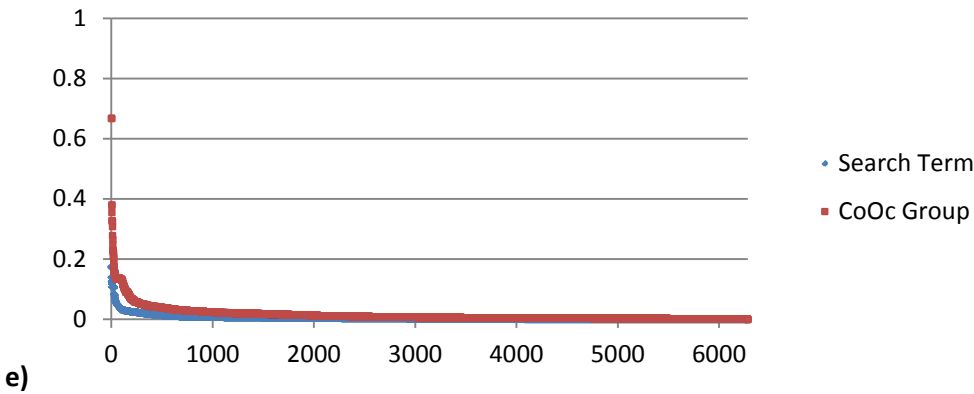
### Arbeiten (working)



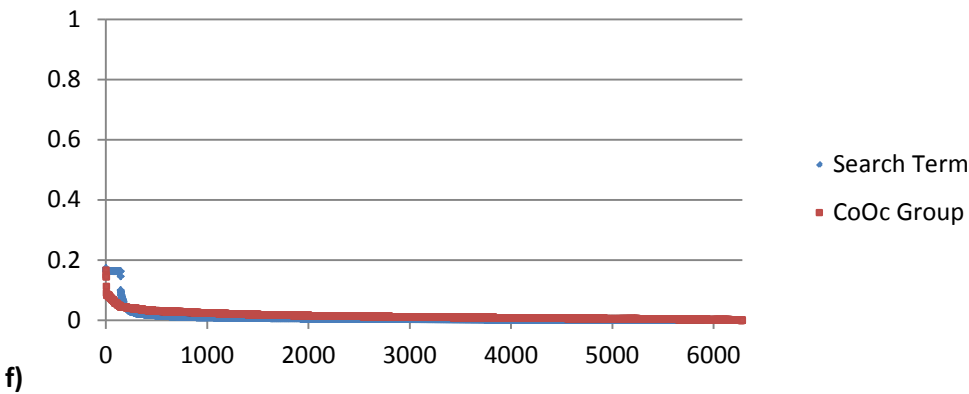
### Bildung (education)



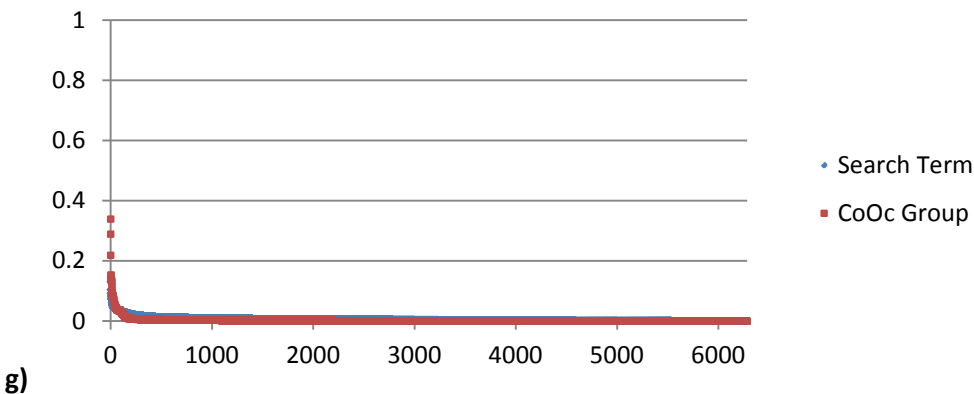
### Wohnen (live)

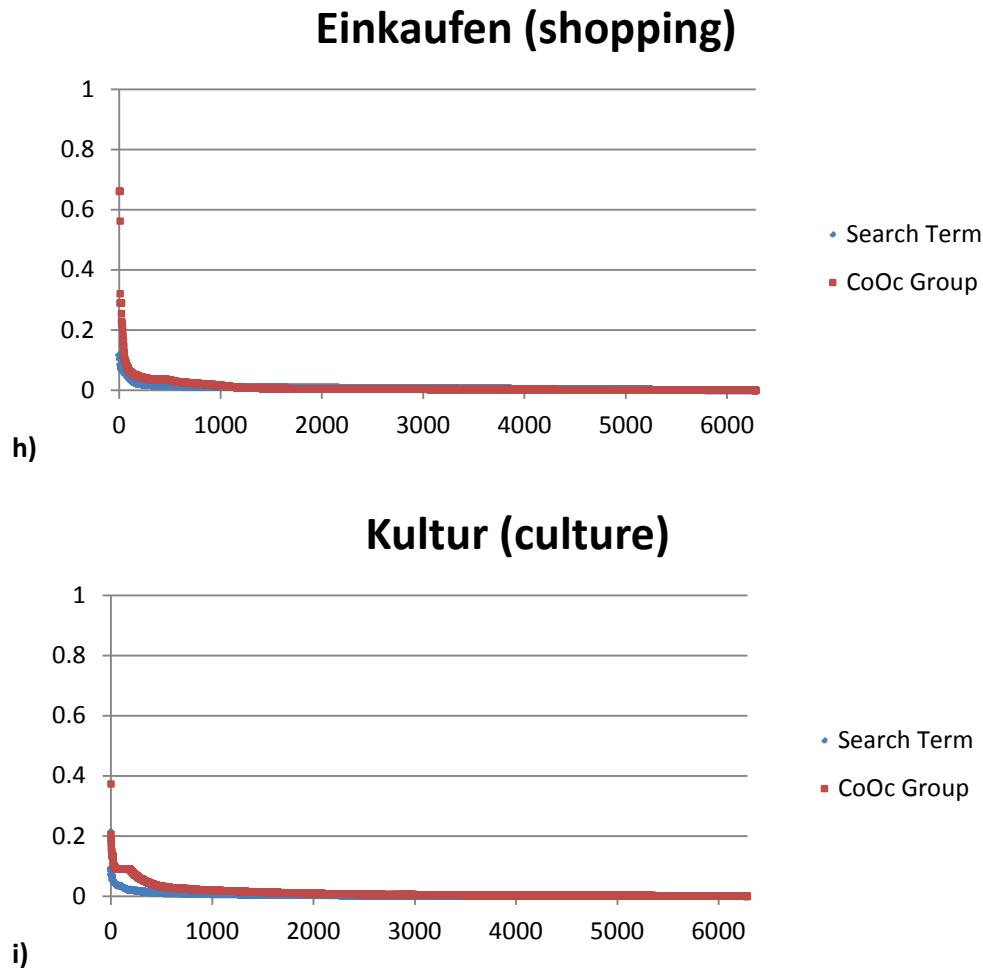


### Dienstgebäude (government building)



### Kreditinstitut (credit institution)





**Figure 12.10 a)-i)** Value distribution for all classes at the addresses for co-occurrence groups and search terms (N= 6284)

As can be seen in **Figure 12.10** and was to be expected, all the co-occurrence groups deliver higher values over just the search terms. Now for these value distributions a threshold needs to be determined

to decide at which value an address belongs to the class. Again, the same method as depicted in **Figure 12.9** is used to cluster the data into two sets. But this time, not the mean but the lowest value of the `highlist` is of interest, as it should constitute said threshold. Afterwards, every address possessing a value in this class above the threshold is set in the database as belonging to the class. The value ranges of “Gaststätte” co-occurrence group, “Einkaufen” co-occurrence group and “Kultur” search term had very strong outliers. As a result, the method classified only a handful of addresses. Therefore, the outliers in those classes have been reduced to the highest non-outlier value. These are the changes made:

```
"Update Addressesunique set gaststätte_cooc = 0.186 where gaststätte_cooc > 0.186"
```

3 rows affected

```
"Update Addressesunique set einkaufen_cooc = 0.322 where einkaufen_cooc > 0.322"
```

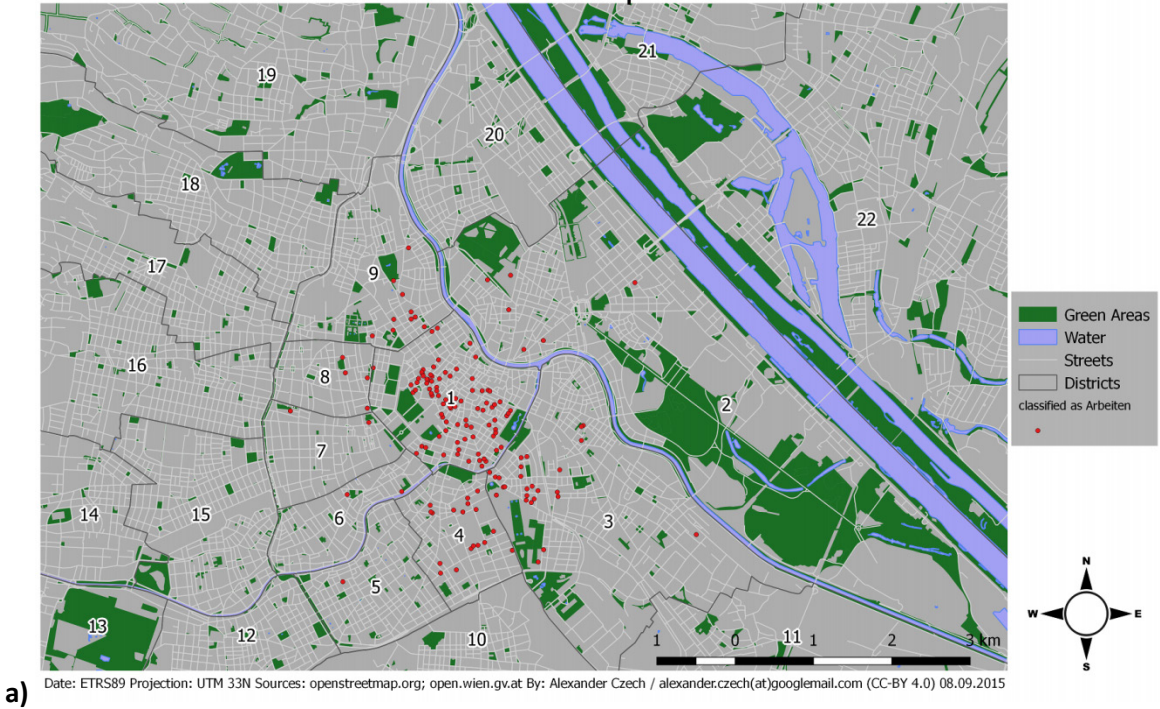
7 rows affected

```
"Update Addressesunique set kultur_st = 0.094 where kultur_st > 0.094"
```

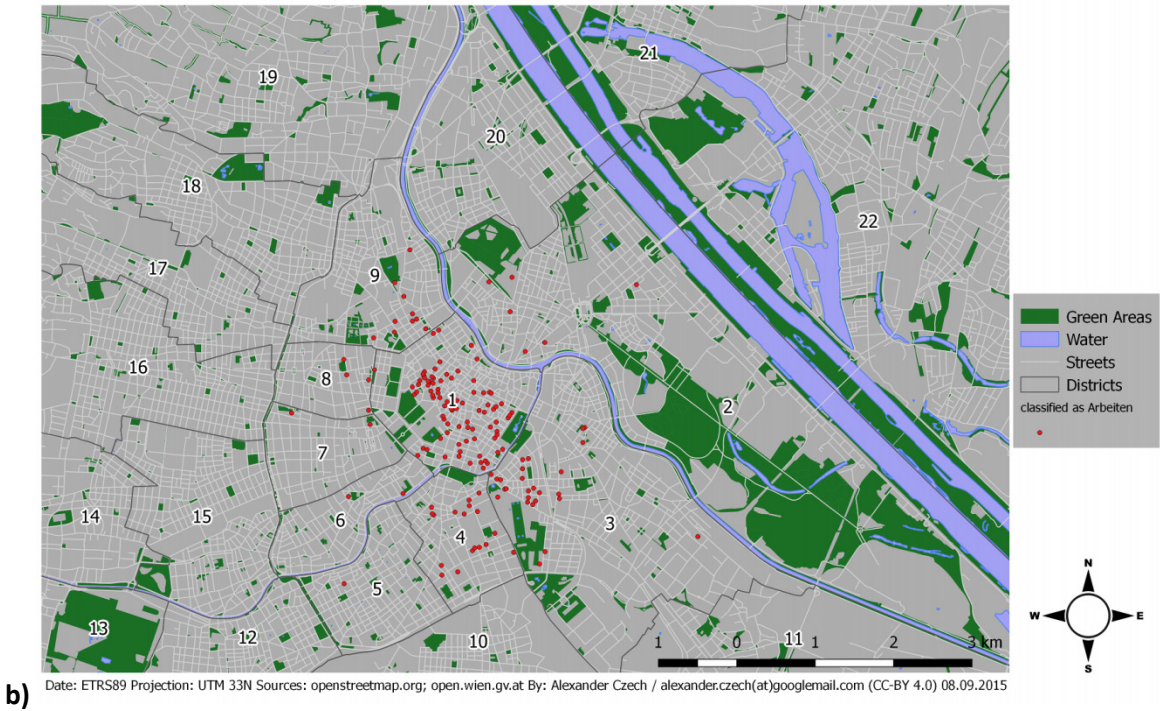
1 row affected

The results of the classification and the different categories can be seen on maps **Map 12.1 a) – r)**

Classification Arbeiten (working) with Co-Occurrence Group

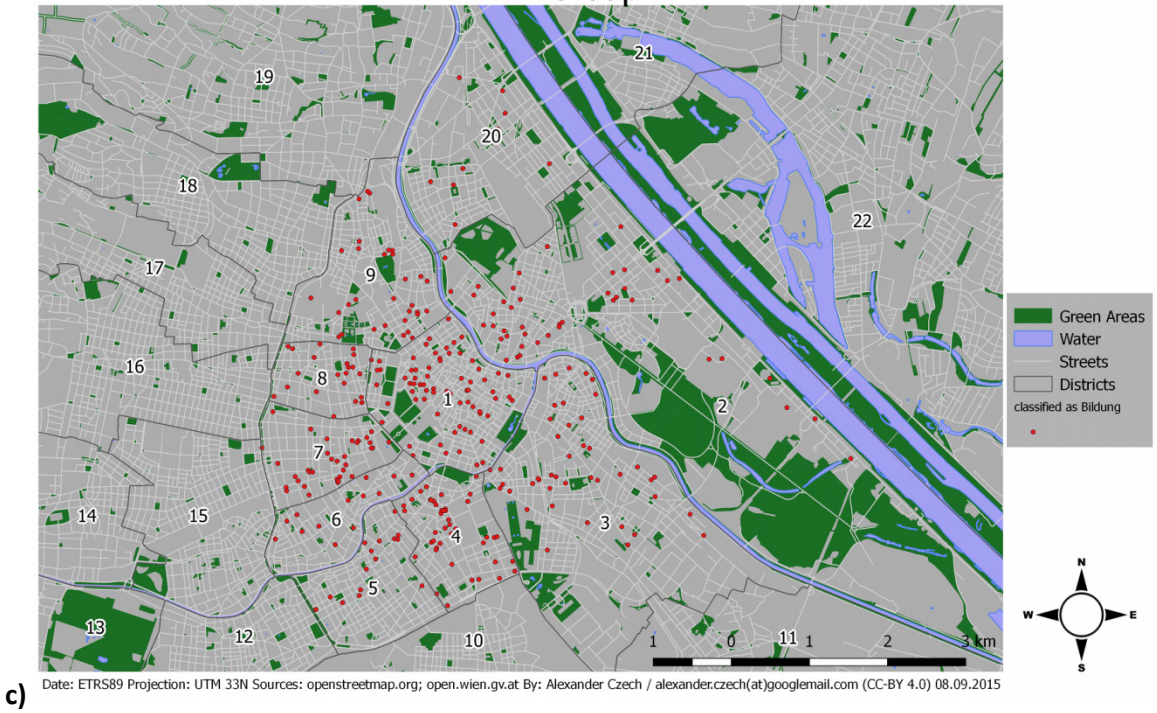


Classification Arbeiten (working) with Search Term

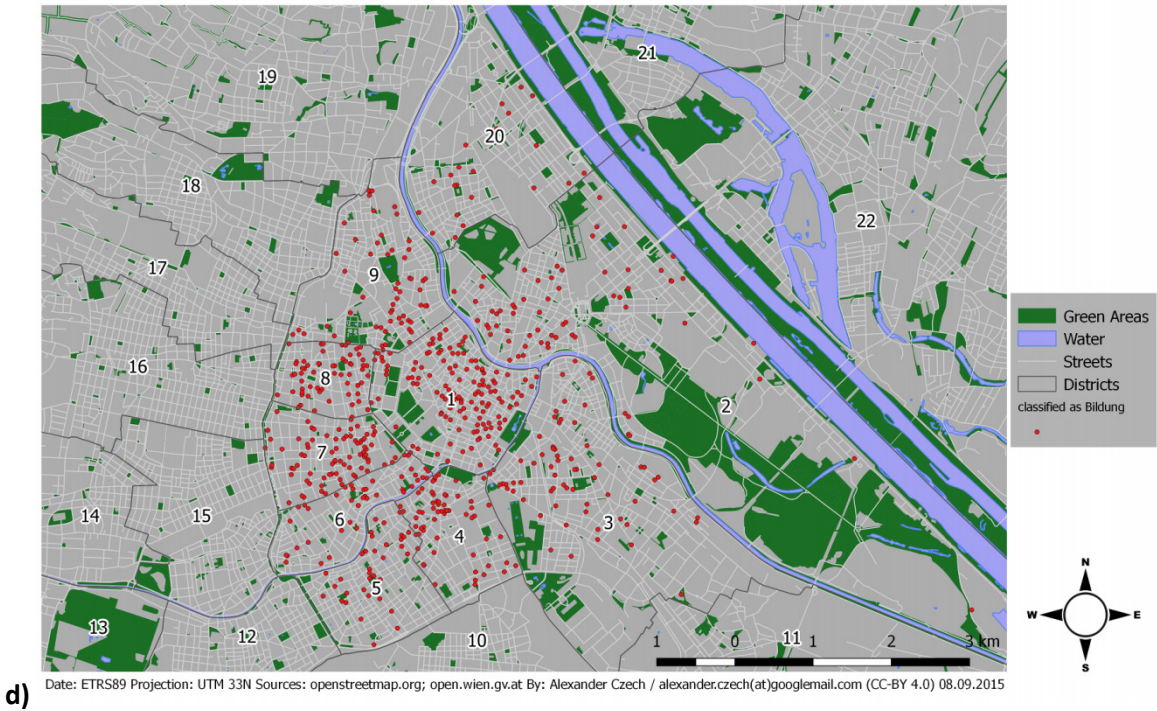




Classification Bildung (education) with Co-Occurrence Group

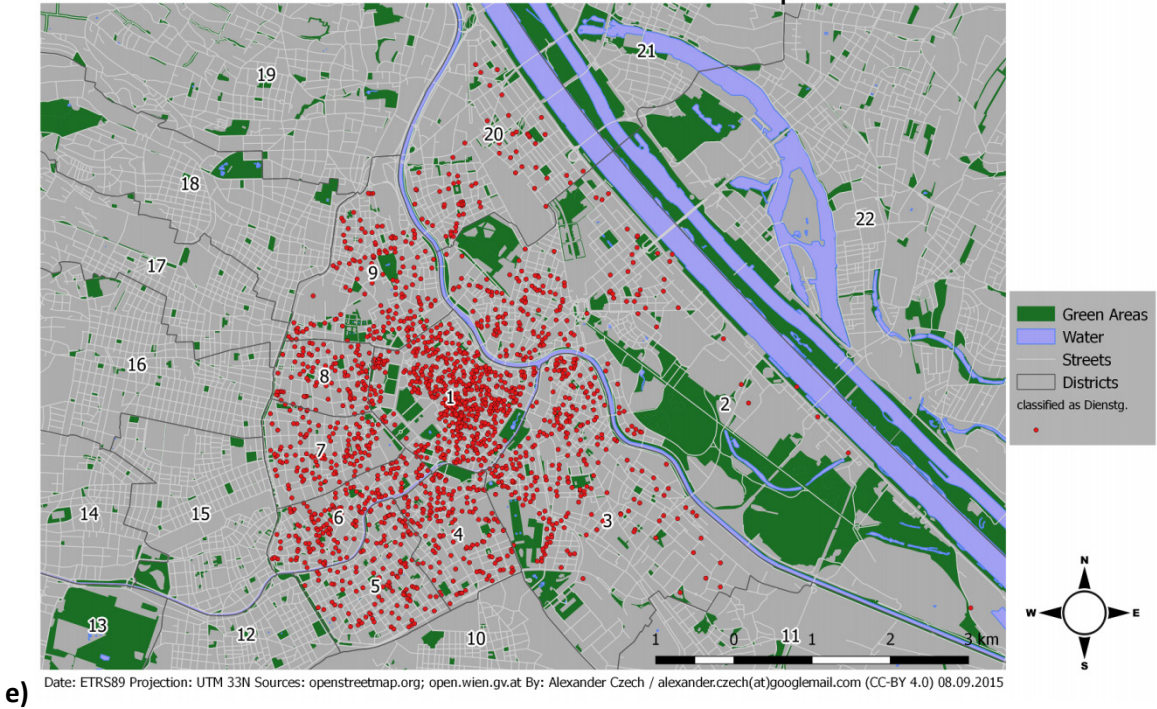


Classification Bildung (education) with Search Term

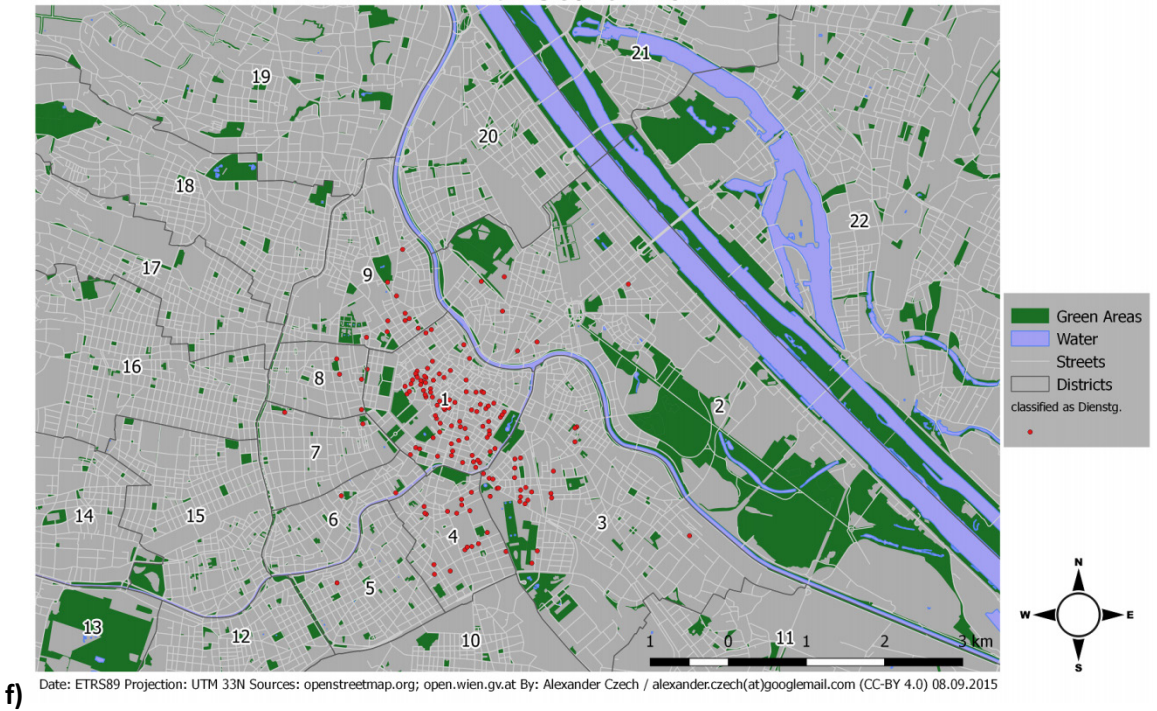




Classification Dienstgebäude (government building)  
with Co-Occurrence Group



Classification Dienstgebäude (government building)  
with Search Term

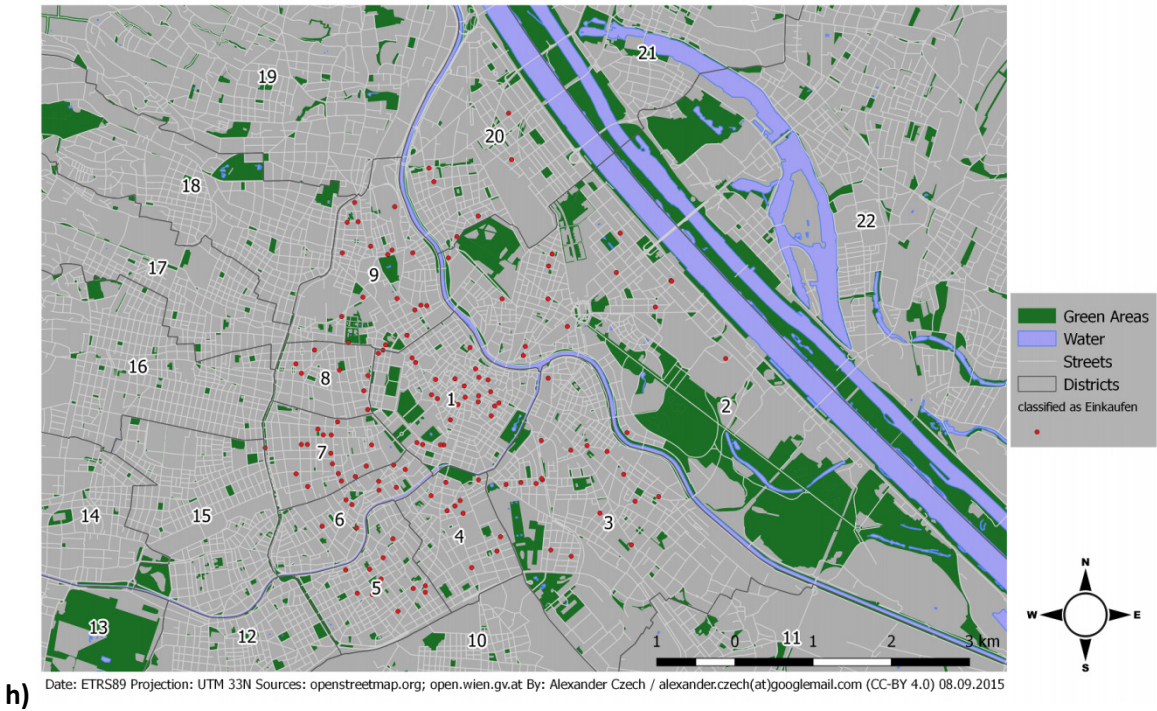




Classification Einkaufen (shopping) with Co-Occurrence Group

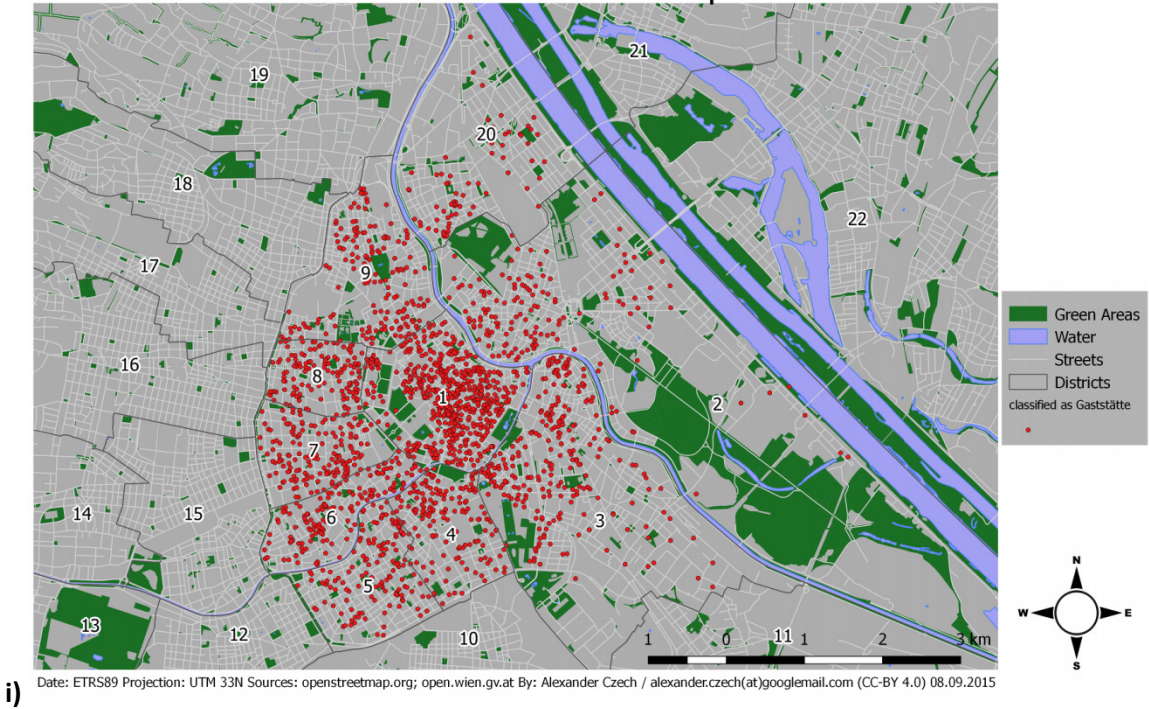


Classification Einkaufen (shopping) with Search Term

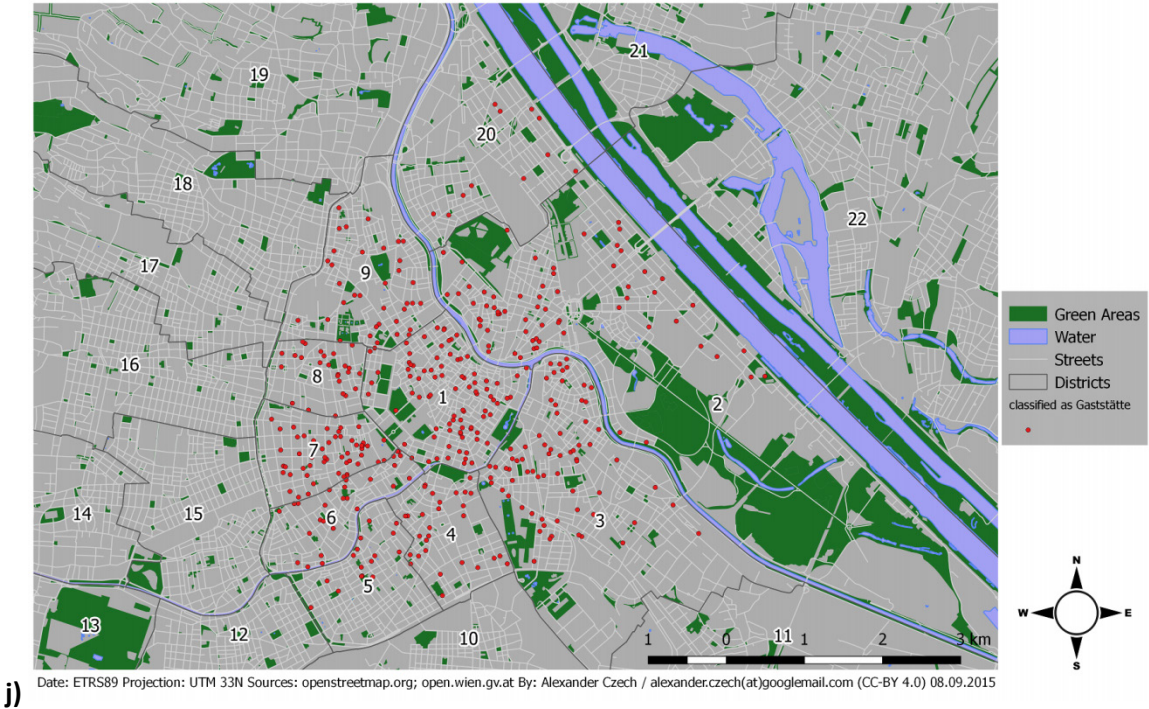




Classification Gaststätte (restaurant) with Co-Occurrence Group

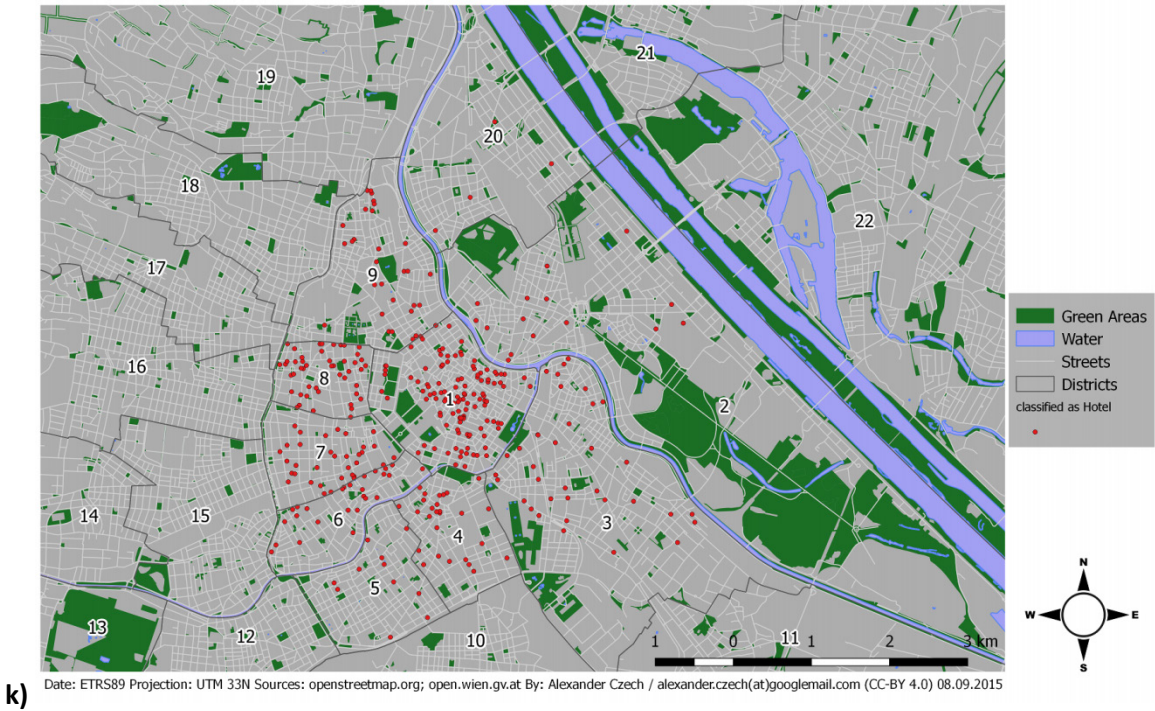


Classification Gaststätte (restaurant) with Search Term

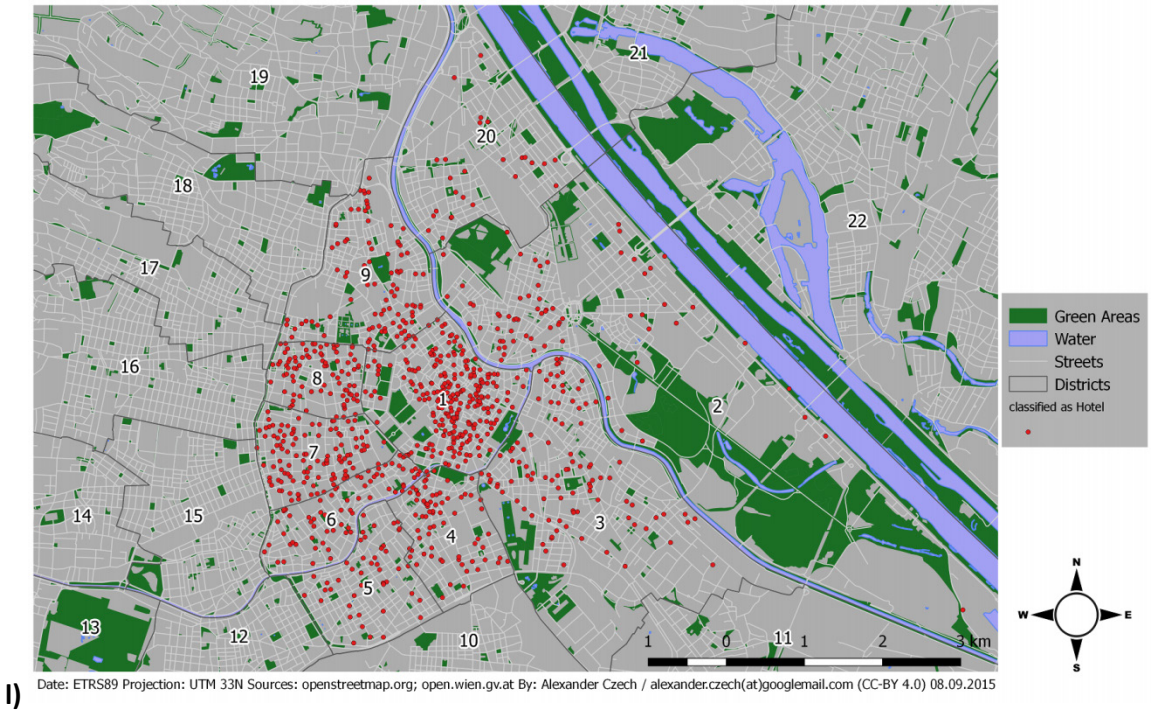




Classification Hotel (hotel) with Co-Occurrence Group

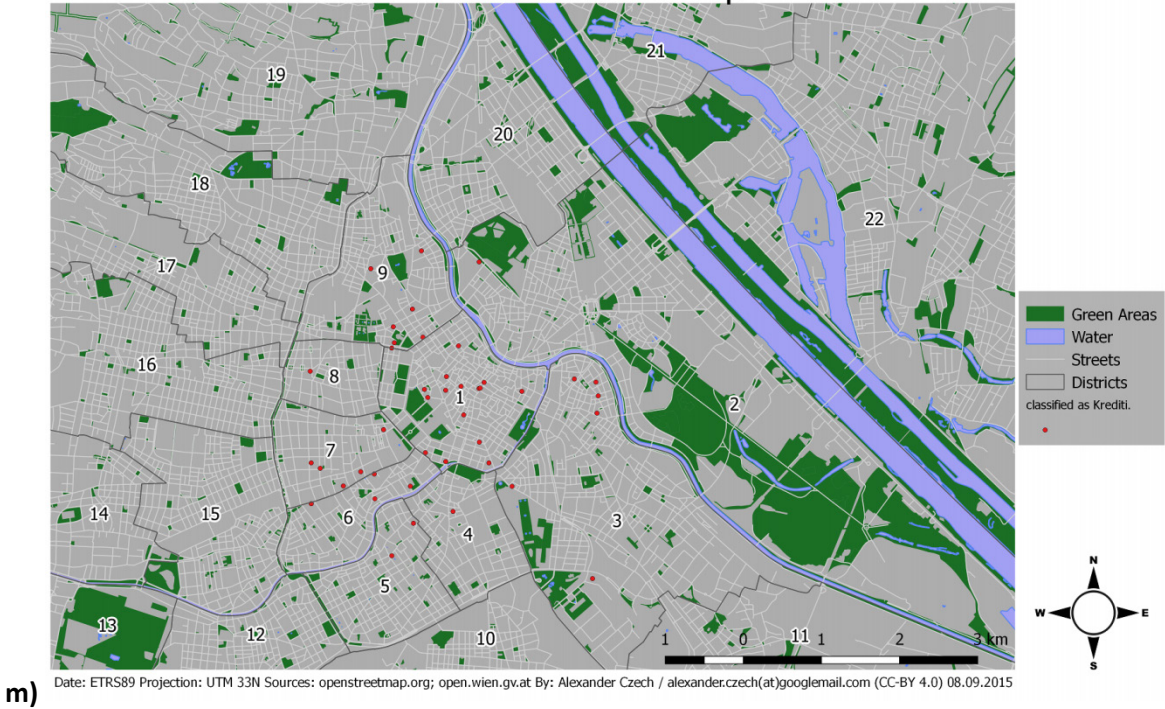


Classification Hotel (hotel) with Search Term

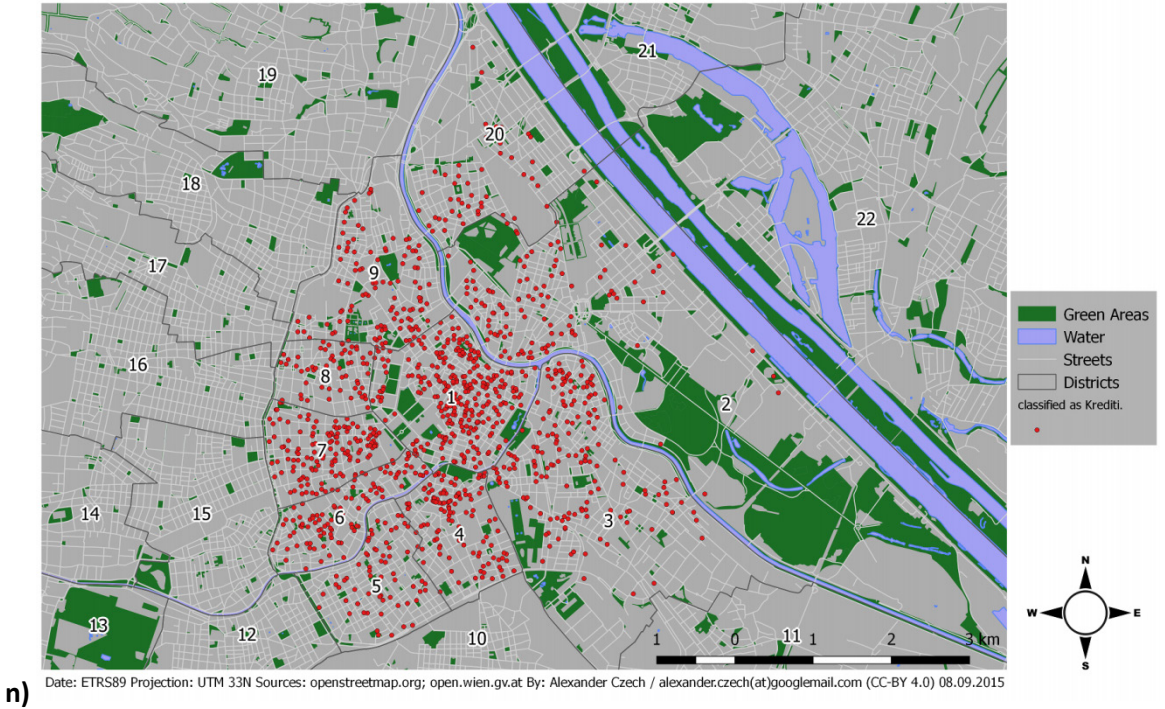




Classification Kreditinstitut (credit institution) with Co-Occurrence Group

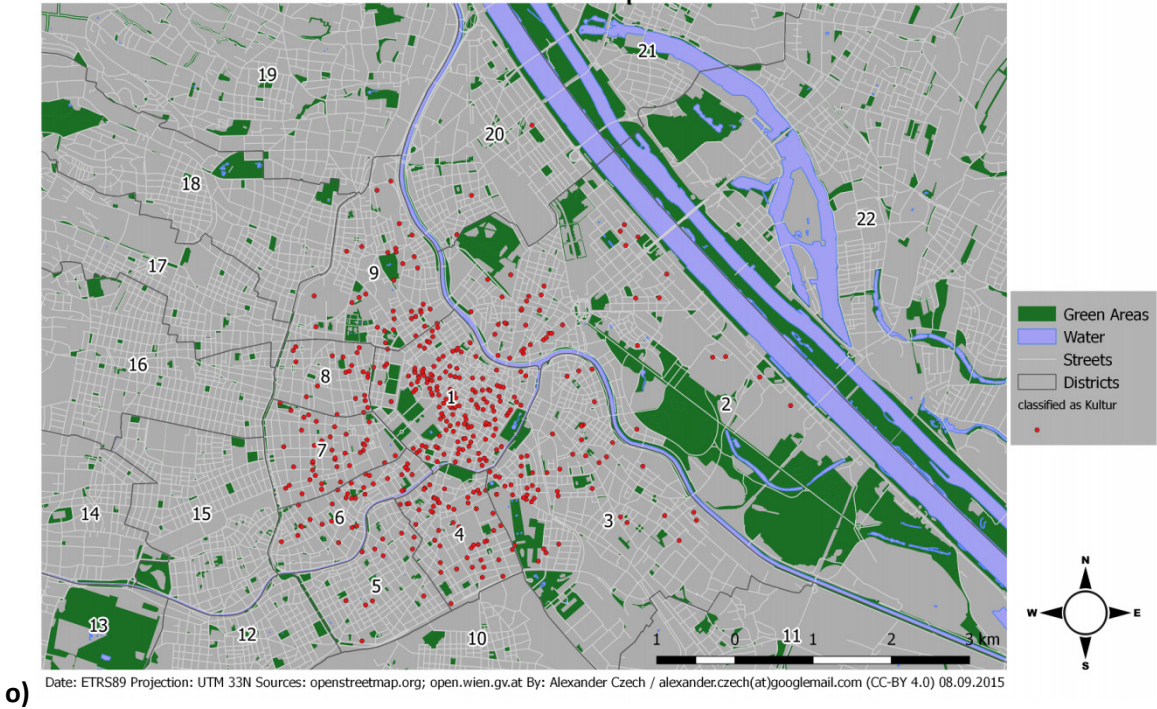


Classification Kreditinstitut (credit institution) with Search Term

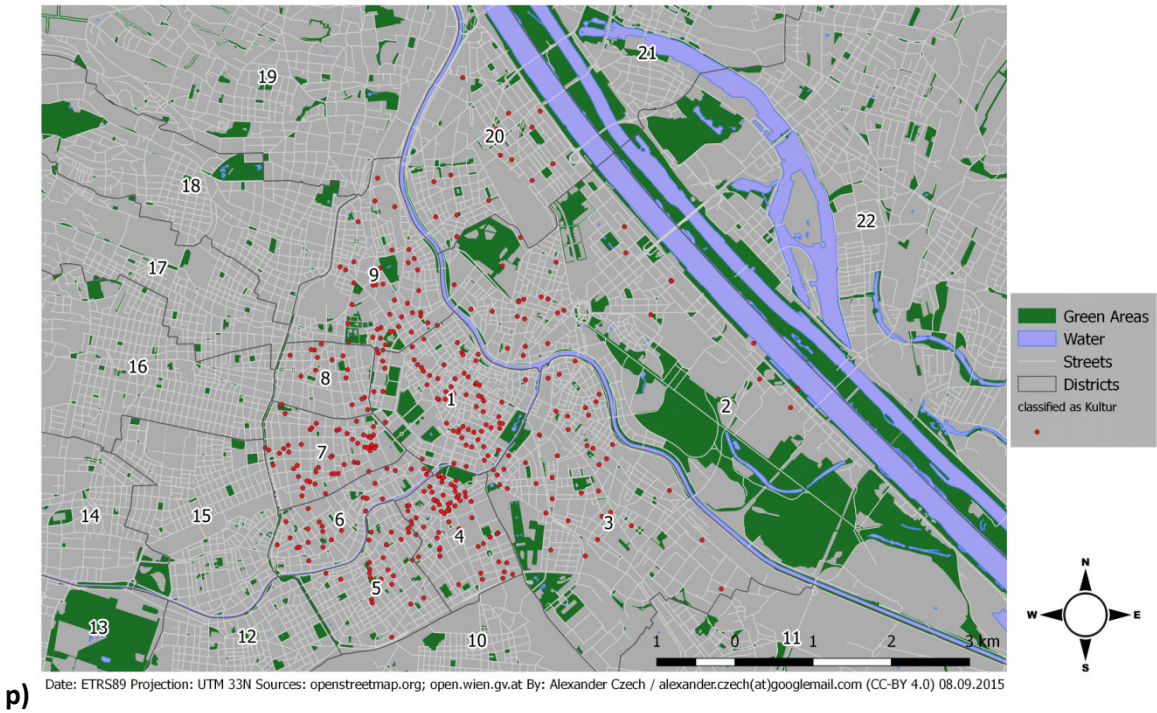




Classification Kultur (culture) with Co-Occurrence Group

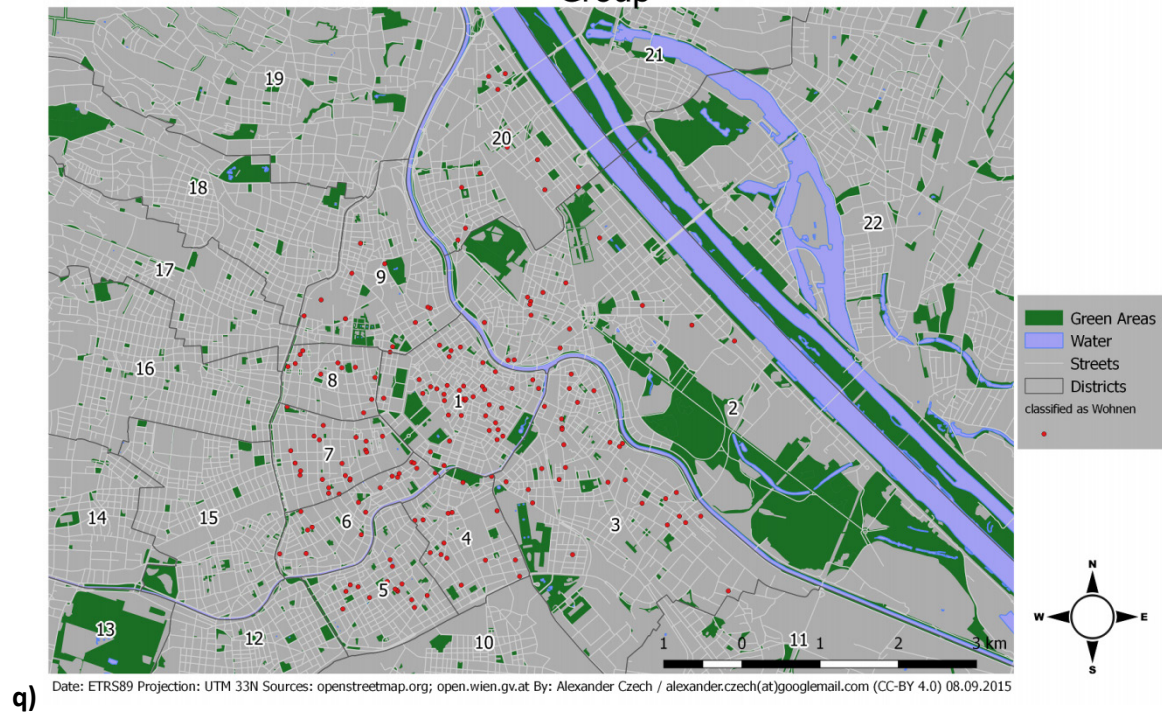


Classification Kultur (culture) with Search Term

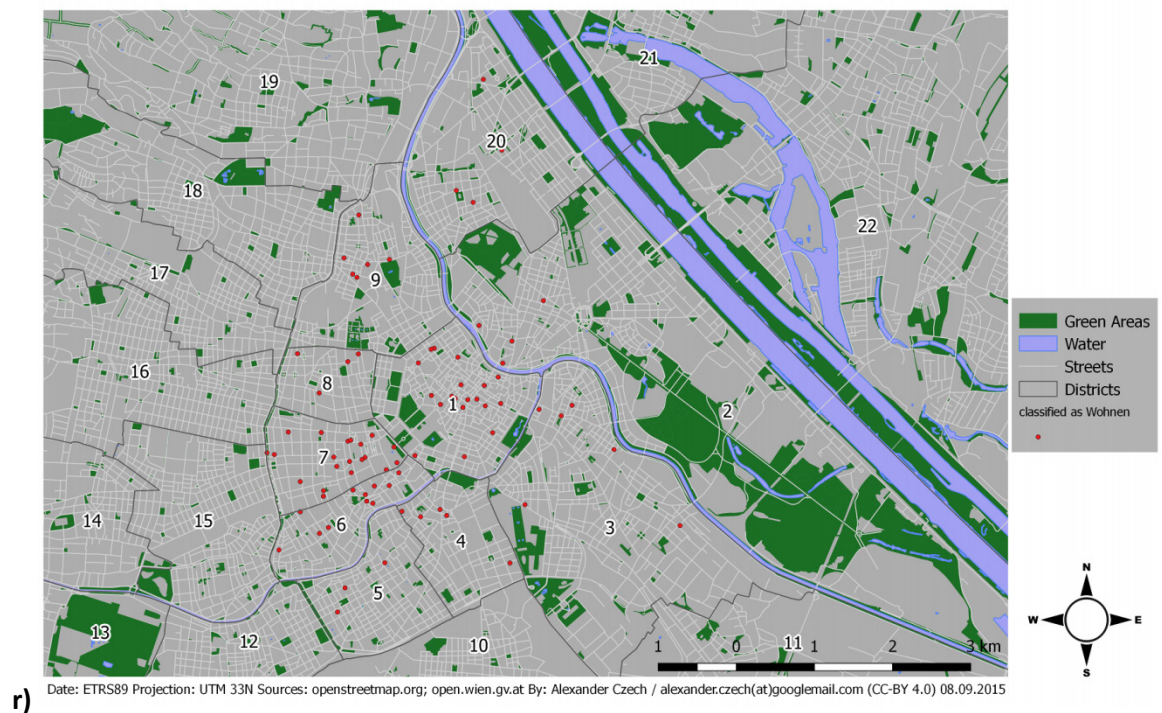




### Classification Wohnen (live) with Co-Occurrence Group



### Classification Wohnen (live) with Search Term



**Map 12.1 a) - r)** Classification Results for every individual category and method

Superficially, the classification outcome of many maps doesn't look as expected, especially maps like **e)** because there are definitely not so many governmental buildings in Vienna. Another good

example of where the classification probably did not work is maps **g)** and **h)**. There are most likely more places that are shopping-related in Vienna than those shown on the maps. On the other hand, maps like **c)**, **k)** and **i)** could be more plausible. The comparison with the control group in the next chapter will clarify this. Because the noisy results a comparison with **Map 10.1** makes little sense.



## 13. Mapping

The first part of this chapter is about how to select a representative random sample of addresses and by what criteria they are mapped. The Second part compares the mapping and the machine classification to each other and draws some first conclusions.

### 13.1. Selecting and Mapping a Control Group

There are now existing 18 categories for each class two categories. One classification is by search term vector and another is by co-occurrence group vector. To represent each category adequately, the random selection for mapping includes at least 10 addresses from each category. In practice, this means that from the database, all addresses belonging to one category are retrieved and a subset of 10 is selected with a programming function for random selection. This is repeated for all 18 categories. With 8 double selections, this resulted in 176 unique addresses, which constitute the control group. **Map 13.1** shows the spatial distribution of those 176 addresses (BAHRENBURG ET AL.; 2010; pp. 19-23).

Addresses Selected as Control Group



**Map 13.1** Randomly selected control group

Apart from creating a random sample, it is important to be as consistent as possible when mapping. The orientation for mapping the addresses is to map the functions that a place fulfills for most people, as described in the social geography. So if there is a shop, the function would be shopping and not working, even though there are people working there. This is because the number of people working is just a few in comparison to the amount people going there to shop. The same is true for other places like schools. Most people going there are students who are learning and not the teachers who are working. If there is more than one class present at an address, for example a café and a hotel, both are mapped. Lastly, because the classes (Wohnen, Arbeiten, Bildung, ...) cannot accommodate all possibilities, the class “other” is added to the mapping catalogue. Corner buildings are mapped according to where the entrance to the building with the door number is. To ensure an unbiased mapping, all vector classifications from the control group were hidden during the mapping process (BAHRENBURG ET AL.; 2010; pp. 19-23), (KRÜKER AND RAUH; 2005; pp.84-90), (MAIER ET AL.; 1977; pp. 100-157).

### 13.2. Comparing the Control Group to Vector Classification

With the control group mapped, it is now possible to make a comparison on the diagrams in **Figure 13.1 a)-i)**. The figure shows 3 values for every category. The value “Mapped and Identified” is the number of objects in the control group that were mapped as belonging to a class and identified by vector classification as belonging to the class. Value “Mapped” is the number of objects in the control group that, according to the mapping, belong to this class. Lastly, the Value “Identified” is the number of addresses in the control group that the vector classification identified as belonging to the class. There are a couple of key statics put on the figure as well. The First two precision and recall are common to evaluate information retrieval systems. Recall measures how many of the relevant documents were retrieved, in the case of the thesis how many of the addresses that where mapped as X where classified as X. Precision shows how many of the retrieved documents where relevant, in terms of the thesis this means correctly classified addresses compared to all classified addresses. The Values correspond to the data shown on the figure (MANNING ET AL; 2009; pp. 153-157).

The formula for recall is:

$$recall = \frac{MI}{M}$$

*MI* (Mapped and Identified), *M* (Mapped)

The formula for precision is:

$$precision = \frac{MI}{I}$$

*MI* (Mapped and Identified), *I* (Identified)

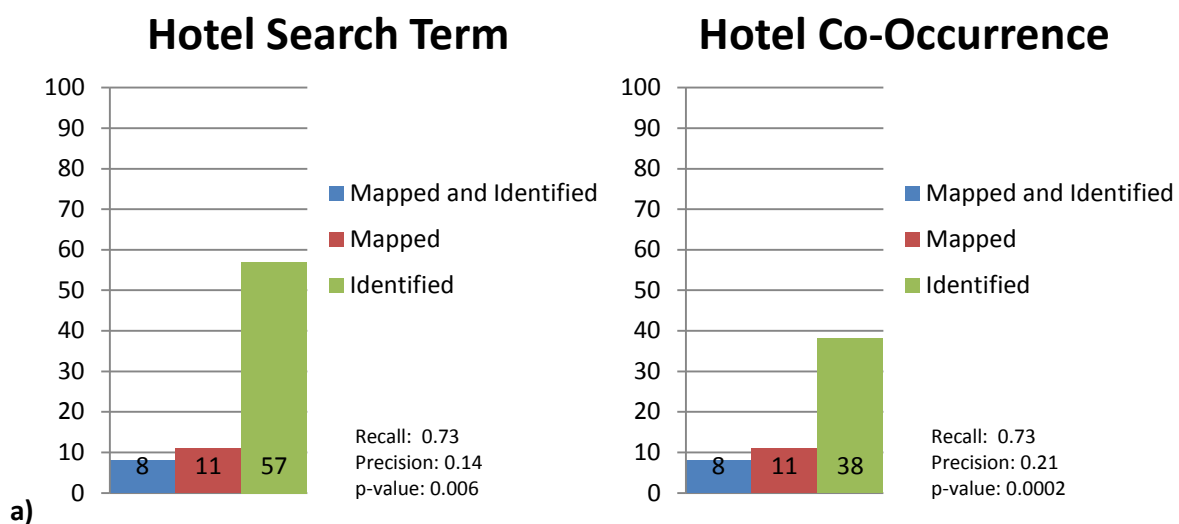
(MANNING ET AL; 2009; pp. 155)

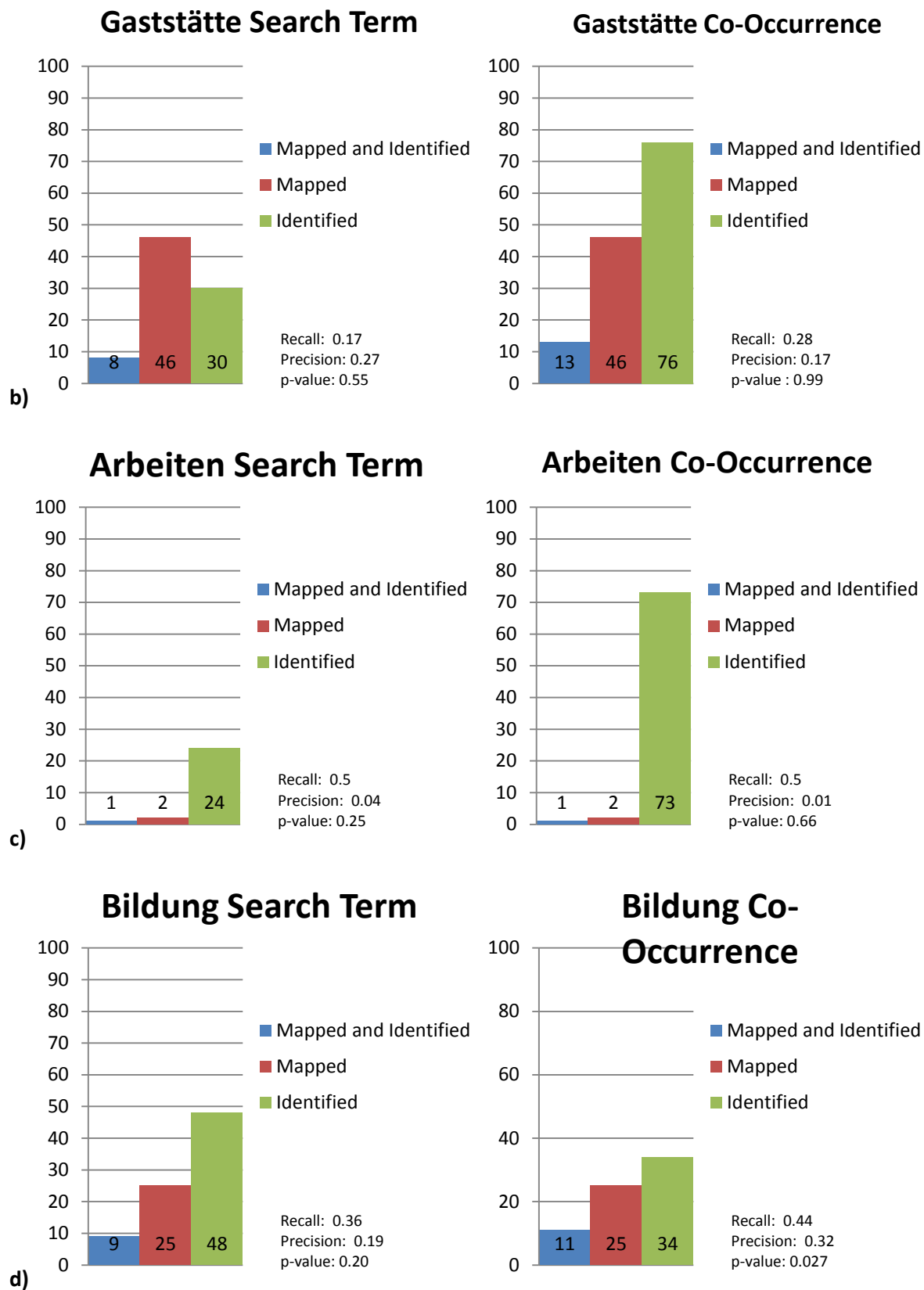
The last statistical value is the p-value. It shows how likely the null hypothesis is, i.e. that the correctly identified elements are identified simply by chance. The markup of every category is a finite population ( $N=176$ ) with discrete values of which some belong to the class and others do not. This resembles the urn problem. The urn problem is an urn with  $N$  marbles, of which  $M$  are black and from it  $n$  marbles are drawn. It is then determined how likely it was that with  $n$  draws  $k$  number of black marbles are drawn. This is solved with a hypergeometric function the formula of which is:

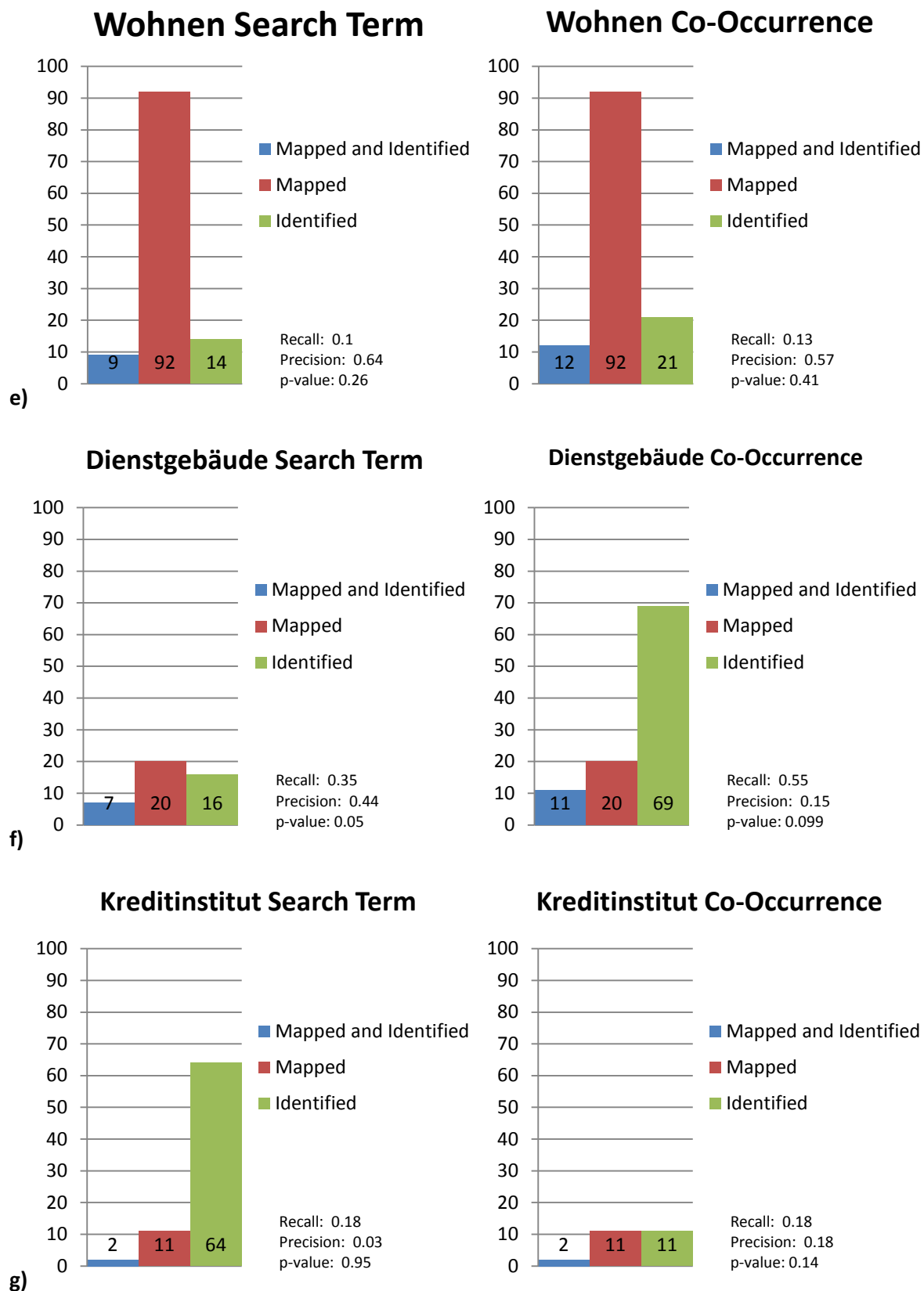
$$f(k) = \frac{\binom{M}{k} \times \binom{N-M}{n-k}}{\binom{N}{n}}$$

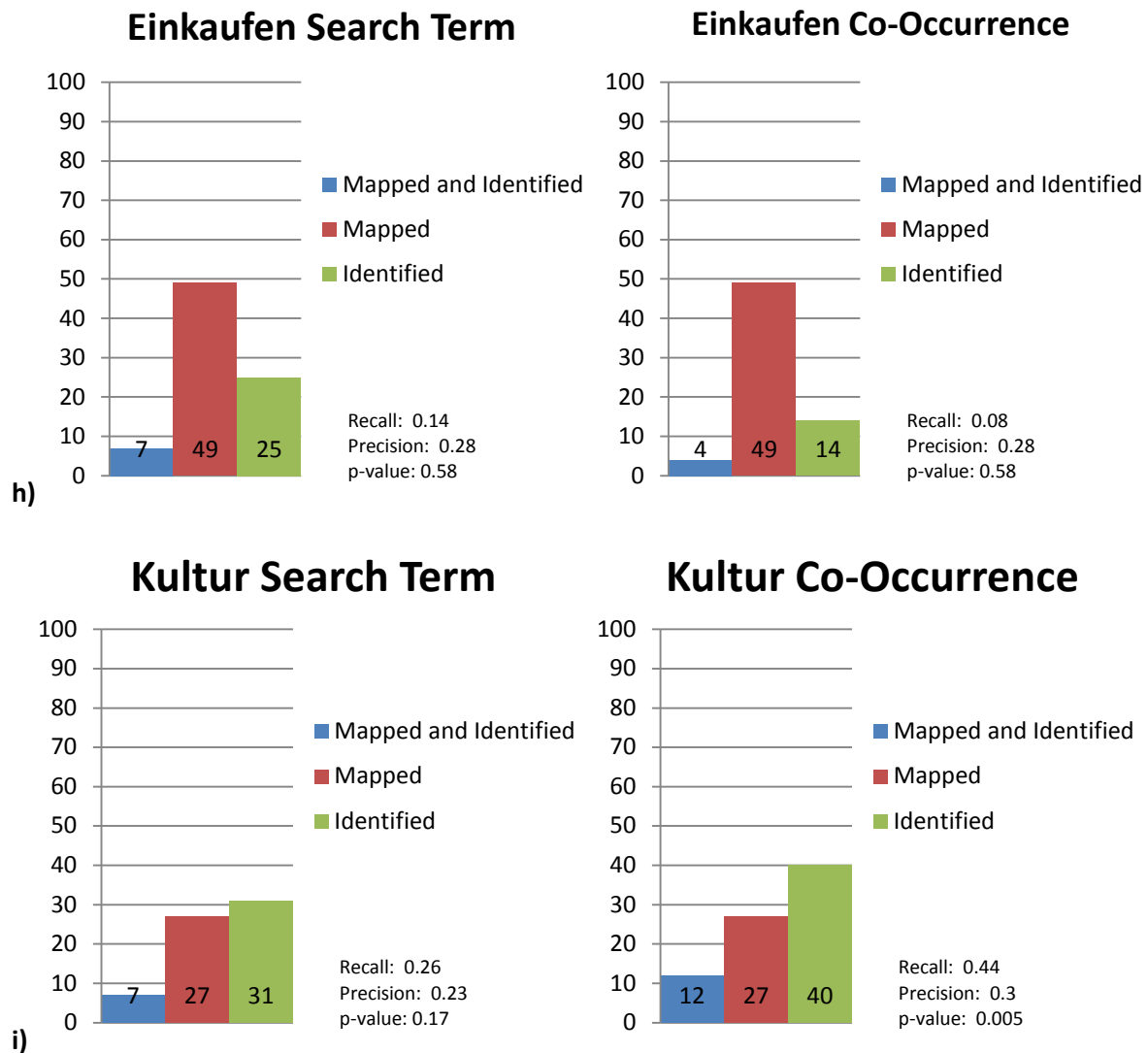
$N$  (Population Size),  $M$  (success states in population),  $n$  (size of sample),  $k$  (success states in sample)

In the context of a category,  $N$  is the size of the control group,  $M$  is the number mapped objects in the control group,  $n$  is the number of objects identified by vector classification and  $k$  is the number of correctly mapped and identified objects. However, this only gives the odds of exactly drawing  $k$  elements for the p-value, although the possibility of drawing  $k$  or more elements is needed. To achieve this, the odds for all values  $0$  to  $k-1$  are calculated, summed up and subtracted from  $1$ . The resulting value is the  $p$ -value the probability that the correct classification was just by chance (BAHRENBURG ET AL.; 2010; pp. 128-129).





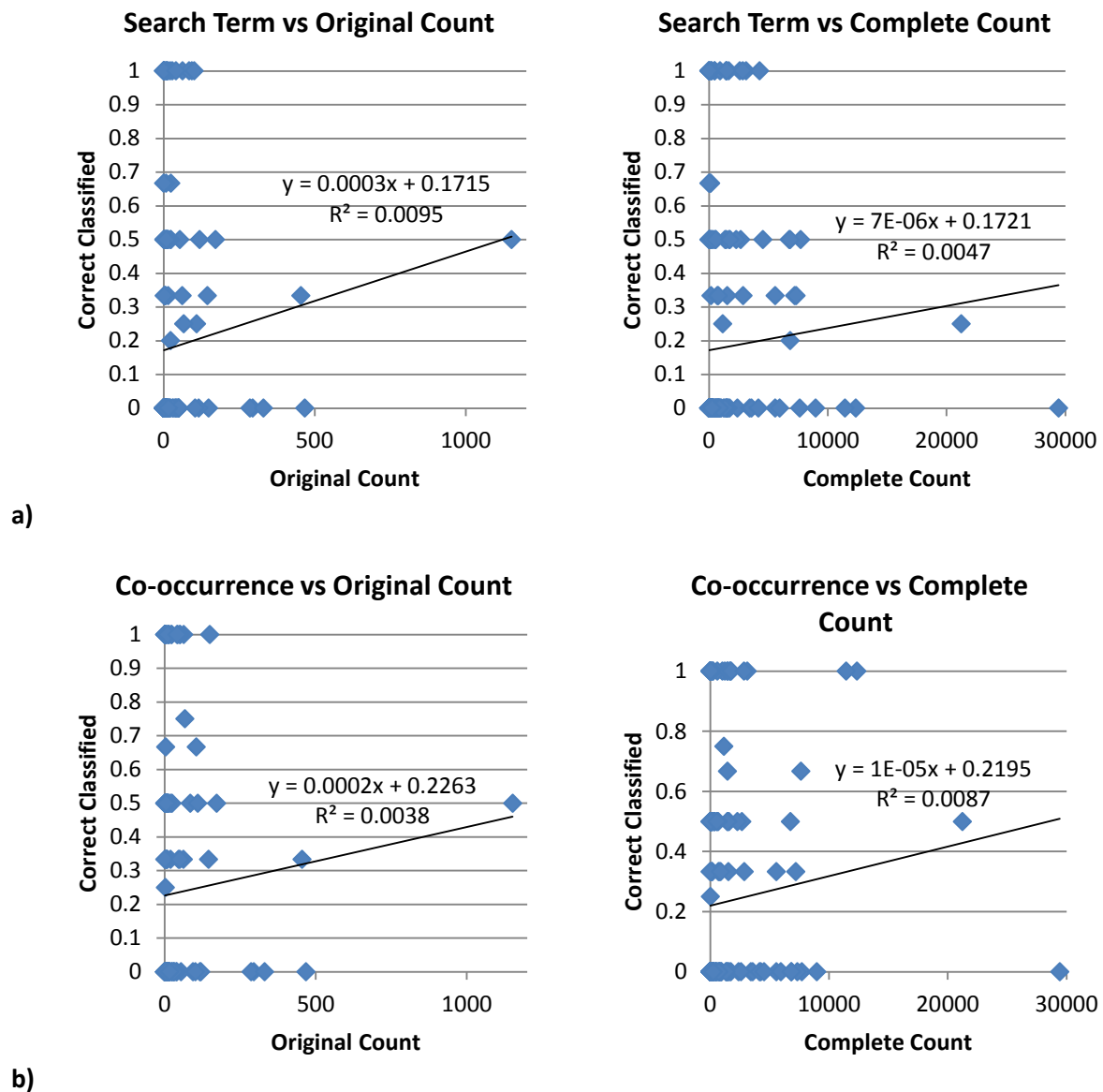




**Figure 13.1 a)-i)** Comparing control group to vector classification

Here two factors are striking: the recall is low and the precision aswell. The low precision was to be expected with respect to the results displayed on **Map 12.1 a)-r)**. Also interesting is the high variability of the  $p$  value within a class (see **d)** and **g)**) and between classes. Even though the vector classification does not work well enough to create a map of, for example, all hotels in Vienna, some vector classifications show, with a  $p \leq 0.1$ , that they picked up an underlying structure. It is clear that there are big class to class differences on how well the vector classification works. It would also be interesting to see if there is also a connection between how many HTML documents are associated with an address and the classification performance. A value that should reflect the correctness is calculated for every address. If, for example, a cultural place is mapped in the control group and either the search term vector or the Co-occurrence Vector has also classified this place as a cultural place, then this is counted as correct for either the co-occurrence or the search term classification.

But if one or both of the vector classifications has not mapped this place, this is counted as incorrect for the respective group. In the end, the correct count is divided by the sum of the correct and incorrect counts. This yields a value between 1 and 0. 1 means all classifications are correct and 0 means none are correct. The method excludes false positives. The value is then correlated with the original count and complete count (see **chapters 7.4.** and **8.2.**). The result can be seen in **Figure 13.2 a)-b).**



**Figure 13.2 a),b)** Correlation between original count, complete count and vector classification

All four scatter plots and  $r^2$  values in **Figure 13.2** clearly show that there is no correlation between how many documents are associated with a website and how well the classification worked (BAHRENBURG ET AL.; 2010; pp. 183-191).

	Search Vector	Term Co-Occurrence Vector	Group
Mean p	0.34	0.32	
Mean Precision	0.25	0.246	
Mean Recall	0.31	0.37	
Addresses with at least 1 correct Classification	50	62	
Mean Correctness	0.19	0.25	

**Table 13.1** Search term vector co-occurrence group vector fitness comparison

A comparison between search term vectors and co-occurrence group vectors can be seen in **Table 13.1**. It summarizes the values of this subchapter. Overall, the co-occurrence group classification performs better than the search term classification. Both don't show good mean p values, but the variability between the different classes is very high. The mean precision is slightly in favor of the search term method, but only by 0.04. In particular, this point is interesting because the concern with the co-occurrence groups was that they would create more noise. The co-occurrence groups also produce more addresses with at least 1 correct classification and have a slightly higher mean correctness.

The others category was used 92 times because something at an address could not fit within one of the 9 classes. Mostly as predicted it was expected in **Chapter 10.2**. services could not be classified. Subjectively a high proportion of tertiary and quaternary services like doctors, attorneys and, engineers have been mapped.

Finally, a short paragraph about the quality of the address data set, even though it is no longer a valid random sample because the addresses have been filtered by being matched to HTML documents. Of the 176 mapped addresses, only two were wrong. One did not exist, but if it had existed, the geo-coordinates would have been correct. The address in question is Daffingerstraße 1. On the whole side of Daffingerstraße where number 1 would have been, there was no entrance door. Other web map applications point to the same coordinates, so maybe it is an address that exists on paper but not in reality. The second error was Freyung 4, an address that exists, but had the wrong coordinates. Even with this no longer representative dataset, the quality of the address data set seemed to be very good.



## 14. Conclusion

For the conclusion, the three research questions formulated in the Introduction need to be examined.

4. How can unstructured information be retrieved and made usable?
5. How can this information be linked to places?
6. How can context be derived from this now structured and geotagged information?

This thesis shows ways and methods of how to engage and solve the first two questions. There are extensive explanations and instructions in chapters 3, 5, and 6 about how to transfer a selective subset of raw crawled data from an amazon s3 bucket into a database and how to process this data to make it suitable for analyses. Chapter 4 shows how to create an addresses data set that is useable for geocoding and this dataset is used on the HTML document dataset in chapter 7. Finally, the data associated with an address is expanded by also including all documents that are linked to a document that is joined with this address in chapter 8. With this, the first two questions could sufficiently be answered.

To derive context from this processed information, the vector space model was selected. This method is used in the field of information retrieval for document classification and retrieval. The concept was to create a classifier with which the addresses could be classified. The addresses should be assigned to one or more of the classes developed in chapter 12. The classes are derived from the “Daseinsgrundfunktionen” and other concepts related to space use, found in functional urban geography. For each of the 9 classes, two classifiers were created- one just relying on one term for the whole class, in the context of this work the so-called “search term classifier,” and the other based on the query expansion method. For this, a co-occurrence group for each of the 9 classes is created from a part of speech tagged Wikipedia and this group is used as the classifier. The classification was conducted in chapter 12 and a control group of 176 addresses was mapped to evaluate the machine classification.

The results of this evaluation are mixed. This was to be expected because already **Map 12.1 a) - r)** did not match the subjective expectations. When viewed individually, the results from class to class vary a lot (see **Figure 13.1 a-i)**). The two worst performing classes are “Arbeiten,” “Wohnen” and “Einkaufen”.

The class “Arbeiten” probably performed badly because it is a wide term that, when stemmed, just gets wider. In German, it can mean labor, work in physics, a school assessment, a work of art and academic writing. Also, the co-occurrence group picked up the German term for lawyer quite a lot, probably skewing the classification (many places with law offices were mapped). Lastly, in a west

European inner city environment, places that are devoted to production and labor, like, for example, factories, no longer exist.

The classification for “Wohnen” mostly performed badly because a lot of buildings had the function living, but the classification didn’t pick it up. The probable reason for not being picked up is that the function living is not something that gets advertised on websites. The only exception for this is if the flat or house is for rent or for sale. Except for that, “Wohnen” has the highest precision.

The class “Einkaufen” in the group of bad performing classes is a bit surprising. This is because the subjective expectation was that there is a motivation to communicate that shopping is possible at a place. Already in the classification process there were problems leading to the elimination of strong outlier values. But that did not help much, still only a few places were classified as belonging to the class “Einkaufen”.

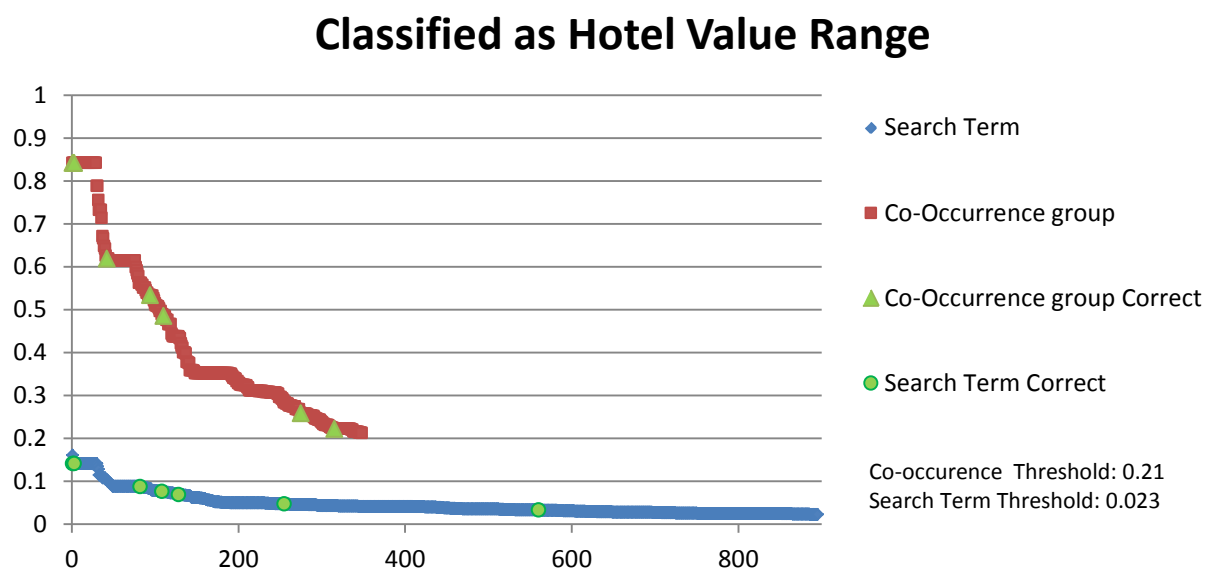
It is likely that the wrong terms were chosen for the classes “Kreditinstitut” and “Gaststätte.” Instead of “Kreditinstitut”, the term “Bank” could have been a better choice. Even though it overlaps in German with a word for siting furniture, it is probably much more common. The attempt to include bar, café, and restaurant with “Gaststätte” in one term is probably the reason why this did not work. Instead, using just one of the three could have yielded better results.

The classes “Kultur,” “Bildung” and “Dienstgebäude” worked comparatively well, with either the search term or the co-occurrence group classification. Noticeable about all three is that they are not “commercial” classes within limits, entrance to a Museum or a theater is mostly not free and education can be “bought” at some places. The classes “Kultur” and “Bildung” also span a multitude of different places that can be assigned to either class. For both classes, also the co-occurrence group classification worked better than the search term classification.

The distinction between the two classification methods for “Dienstgebäude” is not so clear. The class has results that would have been naively expected for the co-occurrence group and search term classification. The search term classification results in higher precision, but also a reduced recall. Comparatively, the co-occurrence group classification delivers a higher recall and a lower precision. Overall, the co-occurrence group classification as discussed at the end of chapter 13 performs noticeably better than the search term classification. It has a higher recall. It classifies 62 of 176 addresses with at least one correct result. It has a higher mean correctness of 0.25 to 0.19 and a nearly equal precision. Also the mean p-value is lower and, therefore, better for the co-occurrence group classification.

A problem with the evaluation of the results has so far only been addressed within the context of the class “Wohnen”. The problem with mapping places that have no website or other form of Internet presence is that any classification attempt with this method is impossible.

Lastly, there is the class of “Hotel” that, by a wide margin, performs the best in the recall domain. It is a narrow term describing a place in common language and about a class that subjectively relies on a visible web presence. But this is also the term that most clearly shows why the 3<sup>rd</sup> research question cannot be answered by this thesis. The recall is high but the precision is low. Since the recall does not perfectly identify or nearly identify all mapped objects correctly, it can be assumed that the only strategy in context of this thesis is to have higher thresholds for classification. But to reduce the rate of false positives could also reduce the number of correctly mapped places. To substantiate this point, **Figure 14.1** shows the search term classification value and the co-occurrence group classification value of the correctly identified addresses with the value range for both as a backdrop. The values are distributed over the whole class range.



**Figure 14.1** Correctly Classified Hotel addresses compared to classification value range

The tool developed in the second part of this thesis is too blunt for classification work. Nevertheless, it produces interesting results because what the p-values from the classification show is that there is a connection between the information associated with a place and what actually is at this place. The tool works well enough to detect that, but is too blunt to generate any useable information from it. In conclusion, this thesis creates an entry point into how to aggregate spatial information from unordered and non-georeferenced web-based information. However, the tools to analyze this information need more refinement.

## 15. References

### 15.1. Scientific References

Bahrenberg, G.; Giese, E.; Mevenkamp, N.; Nipper, J. (2008): Statistische Methoden in der Geographie. Band2: Multivariate Statistik, 3. Auflage; Gebr. Brontraeger Verlagsbuchhandlung; Berlin, Stuttgart

Bahrenberg, G.; Giese, E.; Mevenkamp, N.; Nipper, J. (2010): Statistische Methoden in der Geographie. Band1: Univariate und bivariate Statistik, 5. Auflage; Gebr. Brontraeger Verlagsbuchhandlung; Berlin, Stuttgart

Bayer, R.; Unterauer, K. (1977); Prefix B-Trees; In: ACM Transactions on Database Systems, Vol. 2, No. 1. March 1977, pp. 11-26

Berners-Lee, T.; Fielding, R.; Masinter, L. (2005); Uniform Resource Identifier (URI): Generic Syntax; Network Working Group

Böhner, J. (1990); Statistik für Geographen oder: „Jetzt rechne ich selbst“

Bollier, D.; Firestone, C., M. (2010); The promise and peril of big data; Washington, DC, USA: Aspen Institute, Communications and Society Program.

Bray, T.; Paoli, J.; Sperberg-McQueen, C.; Maler, E.; Yergeau, F. (2008); "Character and Entity References"; <http://www.w3.org/TR/REC-xml/#sec-references>; Cited: 03.07.2015

Brill, E. (1992); A Simple Rule-Based Part of Speech Tagger; In: HLT '91 Proceedings of the workshop on Speech and Natural Language; pp. 112-116; Association for Computational Linguistics; Stroudsburg

Brin, S.; Page, L.(1998); The Anatomy of a Large-Scale Hypertextual Web Search Engine; Proc. Seventh World Wide Web Conf. (WWW7); International World Wide Web Conference Committee (IW3C2), 1998, pp. 107-117

Crampton, J. W.; Graham, M.; Poorthuis, A.; Shelton, T.; Stephens, M.; Wilson, M. W.; Zook, M. (2012); Beyond the Geotag. Deconstructing Big Data and Leveraging the Potential of the Geoweb; Lexington, KY: Department of Geography, University of Kentucky.

Cooley, R; Mobasher, B.; Srivastava, J. (1997); Web mining: Information and pattern discovery on the world wide web; In: Tools with Artificial Intelligence, 1997. Proceedings., Ninth IEEE International Conference on, pp. 558-567; IEEE.

Fassman, H.; Hatz, G. (2002); Wien Stadtgeographische Exkursion; im Auftrag des Ortsausschusses des 28. Deutschen Schulgeographentages (Wien 2002); Ed. Hölzel GmbH; Wien

Haklay, M. (2010); How good is volunteered geographical information? A comparative study of OpenStreetMap and Ordnance Survey datasets; In: Environment and planning B: Planning & design, vol. 37, pp. 682-703

Heineberg, H.; Kraas, F.; Krajewski, C. (2014); Stadtgeographie 4. Auflage; Ferdinand Schöningh, UTB

Ito, M.; Nakayama, K.; Hara, T.; Nishio, S. (2008); Association thesaurus construction methods based on link co-occurrence analysis for wikipedia. In: Proceedings of the 17th ACM conference on Information and knowledge management, pp. 817-826; ACM

Kazuhiro, M.; El-Sayed, A.; Masao F.; Kazuhiko, T.; Masaki O.; Jun-ichi A. (2003); Word classification and hierarchy using co-occurrence word information; In: Information Processing & Management, 2003, Vol. 40, Issue 6, pp. 957-972

Kruker, V., M.; Rauh, J. (2005); Arbeitsmethoden der Humangeographie; Wiss. Buchges.; Darmstadt

Lund, K.; Burgess, C. (1996); Producing high-dimensional semantic spaces from lexical co-occurrence; In: Behavior Research Methods, Instruments, & Computers, Vol. 28, Issue 2, pp. 203-208

Maier, J.; Paesler, R.; Ruppert, K.; Schaffer, F. (1977); Sozialgeographie; Das Geographische Seminar ; Westermann; Braunschweig

- Manning, C., D.; Schütze, H. (1999); Foundations of Statistical Natural Language Processing; The MIT Press, Massachusetts, London, England
- Manning, C., D.; Raghavan, P.; Schütze, H. (2009); An Introduction to Information Retrieval; Cambridge; Cambridge University Press
- Nayak, R.; Witt, R.; Tonev, A. (2002); Data Mining and XML documents; In: Proceedings International Conference on Internet Computing, IC'2002 3, pp. 660-666; Las Vegas, Nevada
- Neis, P.; Zipf, A. (2012) Analyzing the Contributor Activity of a Volunteered Geographic Information Project — The Case of OpenStreetMap In: ISPRS International Journal of Geo-Information, vol. 1, pp. 146-165
- Pataki, M.; Vajna, M.; Marosi, A. (2012); Wikipedia as Text; In: Ercim News - Special theme: Big Data. 2012, Vols. 89; pp. 48-49
- Pladino, S.; Bojic I.; Sobolevsky, S.; Ratti, C.; González, M. C. (2015); Urban magnetism through the lens of geo-tagged photography; In: EPJ Data Science 4(1), pp. 1-17
- Luo, Q.; Chen, E.; Xiong, H. (2011); A semantic term weighting scheme for text categorization; In: Expert Systems with Applications, vol. 38, issue 10, pp. 12708-12716
- Robertson, S. (2013); Common Crawl URL Index; [https://github.com/trivio/common\\_crawl\\_index](https://github.com/trivio/common_crawl_index)  
Cited: 01.07.2015
- Ruppert, .K; Schaffer, F. (1969) Zur Konzeption der Sozialgeographie; In: Geographische Rundschau, vol. 21, Issue 6, pp. 205-214.
- Russel, A., R. (2014): Mining the Social Web (second edition); O'Reilly Media Inc.; Beijing, Cambridge, Farnham, Köln, Sebastopol, Tokyo
- Salton, G.; Wong, A.; Yang, C., S. (1975) A Vector Space Model for Automatic Indexing; In: Communications of the ACM, Vol. 18, Issue 11, pp. 613-620

- Salton, G. (1991) Developments in Automatic Text Retrieval; In: Science, Vol. 253, pp 974-980
- Santorini, B. (1990); Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3<sup>rd</sup> Revision, 2<sup>nd</sup> printing);
- Schneider, G.; Volk, M. (1998); Adding manual constraints and lexical look-up to a brill-tagger for German; Zurich Open Repository and Archive; University of Zurich; Zurich
- Shelton, T.; Poorthuis, A.; Zook, M. (2015); Social media and the city: Rethinking urban socio-spatial inequality using user-generated geographic information; Landscape and Urban Planning
- Spiegler, S. (2013); Statistics of the Common Crawl Corpus 2012
- Tan, A., H. (1999); Text mining: The state of the art and the challenges; In: Proceedings of the PAKDD 1999 Workshop on Knowledge Discovery from Advanced Databases , vol. 8, pp. 65-71
- Teske, D. (2014); Geocoder Accuracy Ranking; In: Process Design for Natural Scientists, CCIS 500, pp. 161-174; Springer-Verlag; Berlin, Heidelberg
- Wang, P.; Domeniconi, C. (2008); Building semantic kernels for text classification using Wikipedia; In: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining pp. 713-721; ACM
- Weichert, P. (2008); Entwicklungslinien der Sozialgeographie von Hans Bobek bis Benno Werlen; Franz Steiner Verlag, Stuttgart
- Zaki, M., J.; Meira Jr, W. (2014); Data mining and analysis: fundamental concepts and algorithms. Cambridge University Press.

## 15.2. Programming Library References and Technical Documentations

Beautiful Soup 4.3.2 library; <http://www.crummy.com/software/BeautifulSoup/bs4/doc/>

NLTK 3.0 library;

stem; <http://www.nltk.org/api/nltk.stem.html#module-nltk.stem>

tokenize; <http://www.nltk.org/api/nltk.tokenize.html#module-nltk.tokenize>

NumPy 1.8.1 library; <http://docs.scipy.org/doc/numpy-1.8.1/reference/>

Pattern library; pattern.de; <http://www.clips.ua.ac.be/pages/pattern-de>

PostGIS 2.1.3; documentation; <http://postgis.net/docs/manual-2.1/>

PostgreSQL 9.3.9; documentation; <http://www.postgresql.org/docs/9.3/static/>

Psycogp 2.5.3 library; <http://initd.org/psycogp/docs/>

Python 2.7.10 library;

Built-in types; <https://docs.python.org/2/library/stdtypes.html>

I/O; <https://docs.python.org/2/library/io.html>

gzip; <https://docs.python.org/2/library/gzip.html>

multiprocessing; <https://docs.python.org/2/library/multiprocessing.html>

operating system; <https://docs.python.org/2/library/os.html>

pickle; <https://docs.python.org/2/library/pickle.html>

regular expression operations; <https://docs.python.org/2/library/re.html>

xml.sax; <https://docs.python.org/2/library/xml.sax.html>

string; <https://docs.python.org/2/library/string.html>

threading; <https://docs.python.org/2/library/threading.html>

Unicode; <https://docs.python.org/2/howto/unicode.html>

urlparse; <https://docs.python.org/2/library/urlparse.html>

zlib; <https://docs.python.org/2/library/zlib.html>

Scikit learn 0.14.1 library; Pairwise metrics, Affinities and Kernels;  
<http://scikit-learn.org/stable/modules/metrics.html>

Unicode 7.0.0; <http://www.unicode.org/versions/Unicode7.0.0/>



### 15.3. Websites

Archive.org; <https://archive.org/web/researcher/ArcFileFormat.php>; Cited: 15.06.2015

Cloudmade; <http://downloads.cloudmade.com>; Data Downloaded 01.06.2014

Common Crawl; <http://commoncrawl.org/the-data/get-started/> Cited: 15.6.02015

Common Crawl atlassian;  
<https://commoncrawl.atlassian.net/wiki/display/CRWL/About+the+Data+Set> Cited: 15.06.2015

Common Crawl Blog; <http://blog.commoncrawl.org>;

Community questions; <http://blog.commoncrawl.org/2011/11/answers-to-recent-community-questions/>; Cited: 15.06.2015

Crawl data; <http://blog.commoncrawl.org/2012/07/2012-crawl-data-now-available/>; Cited: 15.06.2015

URL Index; <http://blog.commoncrawl.org/2013/01/common-crawl-url-index/>; Cited: 15.06.2015

Dudenkorpus;

<http://www.duden.de/sprachwissen/sprachratgeber/die-laengsten-woerter-im-dudenkorpus>; Cited: 03.07.2015

nic.at; <https://www.nict.at>;

.at Report 2012

[https://www.nic.at/fileadmin/www.nic.at/documents/at-report/at-report\\_2012-1\\_EN.pdf](https://www.nic.at/fileadmin/www.nic.at/documents/at-report/at-report_2012-1_EN.pdf); Cited: 16.06.2015

Registration Guidelines

<https://www.nic.at/en/service/legal-information/registration-guidelines/#c4341> Cited: 16.06.2015

Openstreetmap.org; <https://www.openstreetmap.org>; Cited 25.06.2015

OpenStreetMap Wiki; [https://wiki.openstreetmap.org/wiki/Main\\_Page](https://wiki.openstreetmap.org/wiki/Main_Page);

OSM XML [https://wiki.openstreetmap.org/wiki/OSM\\_XML](https://wiki.openstreetmap.org/wiki/OSM_XML) Cited 25.06.2015

Addresses <https://wiki.openstreetmap.org/wiki/Addresses>

W<sup>3</sup>Techs; [http://w3techs.com/technologies/history\\_overview/top\\_level\\_domain/ms/y](http://w3techs.com/technologies/history_overview/top_level_domain/ms/y) Cited:  
15.6.2015

Ich habe mich bemüht, sämtliche Inhaber der Bildrechte ausfindig zu machen und ihre Zustimmung zur Verwendung der Bilder in dieser Arbeit eingeholt. Sollte dennoch eine Urheberrechtsverletzung bekannt werden, ersuche ich um Meldung bei mir.

## Annex

### Source Code

#### Threading Example

```

001 import remote_copy_external,
002 import threading
003 import time
004 import pickle
005 import os
006
007
008 class myThread (threading.Thread):
009     def __init__(self, urlstump, threadid):
010         threading.Thread.__init__(self)
011         self.urlstump = urlstump
012         self.id = threadid
013     def run(self):
014         threadLimiter.acquire()
015         print 'checking for ' + str(self.urlstump)
016         current_urllist.append(self.urlstump)
017         remote_copy_external.external('AWS-PUBLIC-KEY',
018                                     'AWS-PRIVATE-KEY', 'tldat', 'Data2//'+str(self.urlstump),
019                                     self.urlstump, parallelconnections, True)
020
021         print "Exit Thread: %d of %d" %(self.id, NummberIDs)
022         urllist.remove(self.urlstump)
023         current_urllist.remove(self.urlstump)
024         threadLimiter.release()
025
026 def iterate_ipickle(ipickle):
027     if ipickle < 10:
028         return ipickle +1
029     else:
030         return 0
031
032 duds = len(threading.enumerate())
033 threadnumber = 20
034 parallelconnections = 50
035 threadLimiter = threading.BoundedSemaphore(threadnumber)
036 print 'starting threads %s'%(duds)
037
038 current_urllist = []
039 running = True
040
041 NummberIDs = len(urllist)
042 ID = 1
043 Passnumber = 1
044 threadlist = list(urllist)
045 ipickle = 0
046
047 while running:
048     print ''
049     print 'running threads %s'%(len(threading.enumerate()))
050     urllist_pickle = list(urllist)
051     ipickle = iterate_ipickle(ipickle)
052     picklelist(urllist_pickle, 'obj_%s.pickle'%(ipickle))
053
054     if duds+threadnumber > len(threading.enumerate()):
055         if threadlist:
056             element = threadlist[0]
057             threadlist.remove(element)
058             myThread(element, ID).start()
059             ID += 1
060             continue
061         if not threadlist:
062             pass
063

```

```

064     elif threadlist:
065         ipickle = iterate_ipickle(ipickle)
066         picklelist(urllist_pickle, 'obj_%.pickle'%(ipickle))
067         time.sleep(1)
068         print current_urllist
069         continue
070
071     elif not threadlist:
072         ipickle = iterate_ipickle(ipickle)
073         picklelist(urllist_pickle, 'obj_%.pickle'%(ipickle))
074         pass
075
076     while duds < len(threading.enumerate()):
077         time.sleep(10)
078
079         print ''
080         print 'Current Passnumber: %d'%(Passnumber)
081         print current_urllist
082         urllist_pickle = list(urllist)
083         ipickle = iterate_ipickle(ipickle)
084         picklelist(urllist_pickle, 'obj_%.pickle'%(ipickle))
085         pass
086
087     if not urllist:
088         running = False
089         print ''
090         print 'Script did run Passnumber %d'%(Passnumber)
091         pass
092     else:
093         threadlist = list(urllist)
094         urllist_pickle = list(urllist)
095         ipickle = iterate_ipickle(ipickle)
096         picklelist(urllist_pickle, 'obj_%.pickle'%(ipickle))
097         print ''
098         print 'Script did run Passnumber %d'%(Passnumber)
099         Passnumber += 1

```

## DBconnector

```

001 import psycopg2
002
003
004 def DBConnect():
005
006     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres
password=#####")
007     cur = conn.cursor()
008     return conn,cur
009
010
011 def CreateTable():
012
013     conn,cur = DBConnect()
014     cur.execute("CREATE TABLE IF NOT EXISTS Addresses (id serial PRIMARY KEY,geom
geometry,street text, Street_number text, pcode integer);")
015     conn.commit()
016     conn.close()
017
018
019 def WriteToTableMany(Values):
020
021     conn,cur = DBConnect()
022     cur.executemany("INSERT INTO Addresses (geom, street, Street_number, pcode, AddDate)
VALUES (%s, %s, %s, %s, 24022015)", (Values))
023     conn.commit()
024     conn.close()
025
026 def addcolumn():
027     conn,cur = DBConnect()
028
029     cur.execute( """
030
DO $$

```

```

031             BEGIN
032             BEGIN
033                 ALTER TABLE Addresses ADD COLUMN AddDate INT;
034             EXCEPTION
035                 WHEN duplicate_column THEN RAISE NOTICE 'column Addresses
already exists in AddDate.';
036             END;
037         END;
038         $$
039     """
040 )
041     conn.commit()
042     conn.close()
043
044 def finddoubles(addresslist):
045     newlist = []
046     conn, cur = DBConnect()
047     for address in addresslist:
048         cur.execute('SELECT ID FROM Addresses where street = %s and Street_number = %s and
pcode = %s', (address[1], address[2], address[3],))
049         data = cur.fetchall()
050         if not data:
051             newlist.append(address)
052
053     WriteToTableMany(newlist)
054     return newlist
055
056 def ReadFromTable():
057
058     conn, cur = DBConnect()
059     cur.execute("SELECT * FROM Addresses;")
060     data = list(cur.fetchall())
061     conn.close()
062
063     return data
064
065 def DeleteFromTable():
066
067     conn, cur = DBConnect()
068     cur.execute("DELETE * FROM Addresses;")
069     conn.commit()
070     conn.close()
071
072
073 def DropTable():
074     conn, cur = DBConnect()
075     cur.execute("DROP TABLE Addresses")
076     conn.commit()
077     conn.close()

```

## OSM Parser

```

001 # -*- coding: utf-8 -*-
002 import sys, numpy, DBconnector
003 from xml.sax import make_parser, handler
004
005 endaddress = '0'
006 startaddress = '0'
007 start = '0'
008 end = '0'
009 addresslist = []
010
011 class startendfinder(handler.ContentHandler):
012     def __init__(self):
013         self.address = ['lat_lon', 'pcode', 'street', 'number']
014
015         self.plz = False
016         self.street = False
017         self.number = False
018         self.nodemode = False
019         self.waymode = False
020         self.relationmode = False

```

```

021         self.nodedict = {}
022         self.wayid = False
023         self.relationid = False
024         self.relationdict = {}
025         self.waydict = {}
026         self.ndlist = []
027         self.memberlist = []
028         self.latlist = []
029         self.lonlist = []
030
031     def startElement(self, name, attrs):
032         if name in ('node'):
033             self.address[0] = 'POINT(%s %s)' % (attrs.get('lon'), attrs.get('lat'))
034             self.nodedict[int(attrs.get('id'))] = (float(attrs.get('lat')),
float(attrs.get('lon')))
035
036             self.nodemode = True
037
038         elif name in ('way'):
039             self.wayid = int(attrs.get('id'))
040             self.waymode = True
041
042         elif name in ('relation'):
043             self.relationid = int(attrs.get('id'))
044             self.relationmode = True
045
046         if self.relationmode == True:
047             if name == 'member':
048                 self.memberlist.append((int(attrs.get('ref')), attrs.get('type')))
049             if name == 'tag':
050                 k, v = (attrs.get('k'), attrs.get('v'))
051
052                 if k == 'addr:street':
053                     self.address[1] = unicode(v)
054                     self.street = True
055
056                 if k == 'addr:housenumber':
057                     self.address[2] = unicode(v)
058                     self.number = True
059
060                 if k == 'addr:postcode':
061                     try:
062                         if int(v) <= 1099:
063                             self.address[3] = int(v)
064                             self.plz = True
065                         elif int(v) >= 1200 and int(v) <= 1209:
066                             self.address[3] = int(v)
067                             self.plz = True
068                     except:
069                         pass
070
071         if self.waymode == True:
072             if name == 'nd':
073                 self.ndlist.append(int(attrs.get('ref')))
074             if name == 'tag':
075                 k, v = (attrs.get('k'), attrs.get('v'))
076
077                 if k == 'addr:street':
078                     self.address[1] = unicode(v)
079                     self.street = True
080
081                 if k == 'addr:housenumber':
082                     self.address[2] = unicode(v)
083                     self.number = True
084
085                 if k == 'addr:postcode':
086                     try:
087                         if int(v) <= 1099:
088                             self.address[3] = int(v)
089                             self.plz = True
090                         elif int(v) >= 1200 and int(v) <= 1209:
091                             self.address[3] = int(v)
092                             self.plz = True
093                     except:
094                         pass
095

```

```

096         if self.nodemode == True:
097             if name == 'tag':
098                 k, v = (attrs.get('k'), attrs.get('v'))
099
100                 if k == 'addr:street':
101                     self.address[1] = unicode(v)
102                     self.street = True
103
104                 if k == 'addr:housenumber':
105                     self.address[2] = unicode(v)
106                     self.number = True
107
108                 if k == 'addr:postcode':
109                     try:
110                         if int(v) <= 1099:
111                             self.address[3] = int(v)
112                             self.plz = True
113                         elif int(v) >= 1200 and int(v) <= 1209:
114                             self.address[3] = int(v)
115                             self.plz = True
116                     except:
117                         pass
118
119     def endElement(self, name):
120         if name in ('node'):
121             if self.plz is True and self.street is True and self.number is True:
122                 addresslist.append(tuple(self.address))
123
124             self.nodemode = False
125             self.plz = False
126             self.street = False
127             self.number = False
128             self.address = ['lat_lon', 'pcode', 'street', 'number']
129
130         if name in ('way'):
131             for nd in self.ndlist:
132                 self.latlon = self.nodedict[nd]
133                 self.latlist.append(self.latlon[0])
134                 self.lonlist.append(self.latlon[1])
135
136             self.waydict[self.wayid] = (numpy.mean(self.latlist),
numpy.mean(self.lonlist))
137
138             if self.plz is True and self.street is True and self.number is True:
139                 self.address[0] = 'POINT(%s %s)' %
(numpy.mean(self.lonlist), numpy.mean(self.latlist))
140                 addresslist.append(tuple(self.address))
141
142             self.latlist = []
143             self.lonlist = []
144             self.waymode = False
145             self.plz = False
146             self.street = False
147             self.number = False
148             self.address = ['lat_lon', 'pcode', 'street', 'number']
149             self.ndlist = []
150
151         if name in ('relation'):
152             for member in self.memberlist:
153
154                 if member[1] == 'node':
155                     self.latlist.append(self.nodedict[member[0]][0])
156                     self.lonlist.append(self.nodedict[member[0]][1])
157                 elif member[1] == 'way':
158                     self.latlist.append(self.waydict[member[0]][0])
159                     self.lonlist.append(self.waydict[member[0]][1])
160                 elif member[1] == 'relation':
161                     self.latlist.append(self.relationdict[member[0]][0])
162                     self.lonlist.append(self.relationdict[member[0]][1])
163
164             self.relationdict[self.relationid] = (numpy.mean(self.latlist),
numpy.mean(self.lonlist))
165
166             if self.plz is True and self.street is True and self.number is True:
167                 self.address[0] = 'POINT(%s %s)' % (numpy.mean(self.lonlist),
numpy.mean(self.latlist))

```

```

168         addresslist.append(tuple(self.address))
169
170         self.latlist = []
171         self.lonlist = []
172         self.waymode = False
173         self.plz = False
174         self.street = False
175         self.number = False
176         self.address = ['lat_lon', 'pcode', 'street', 'number']
177         self.memberlist = []
178
179
180 if __name__ == '__main__':
181     parser = make_parser()
182     parser.setContentHandler(startendfinder())
183     parser.parse('./vienna.osm')
184
185
186     DBconnector.CreateTable()
187     DBconnector.WriteToTableMany(addresslist)
188

```

## Disassemble HTML

```

001 import psycpg2
002 import gzip
003 from bs4 import UnicodeDammit
004 import os
005 import time
006
007
008
009
010
011
012 def DBConnect():
013     conn = psycpg2.connect("dbname=Master_DB_spatial2 user=postgres password=#####")
014     cur = conn.cursor()
015     return conn, cur
016
017
018 def CreateTable():
019     conn, cur = DBConnect()
020     cur.execute("CREATE TABLE IF NOT EXISTS html (id serial PRIMARY KEY, url TEXT,
html_file text);")
021     conn.commit()
022     conn.close()
023
024
025 def WriteManyToTable(Values):
026     conn, cur = DBConnect()
027     args_str = ','.join(cur.mogrify("(%s,%s)", x) for x in Values)
028     cur.execute("INSERT INTO html (url, html_file) VALUES " + args_str)
029     conn.commit()
030     conn.close()
031
032
033 def ReadFromTable():
034     conn, cur = DBConnect()
035     cur.execute("SELECT * FROM html;")
036     lines = cur.fetchall()
037     conn.close()
038     return lines
039
040
041 def Database_export(file):
042     start1 = time.time()
043     i = ''
044     tup_i = ()
045     url = ''
046     html_written = 0
047     mode = False
048     for line in file.readlines():

```



```

049         splitline = line.split(' ')
050         try:
051             if splitline[0][:7] == 'http://' and splitline[3] == 'text/html' and
len(splitline) == 5:
052
053                 if len(i) > 0:
054                     tup_i = tup_i + ((url, i),)
055                     url = ''
056                     i = ''
057
058                 url = splitline[0]
059                 mode = True
060
061                 if len(tup_i) >= 100:
062                     print('empty tup_i')
063                     try:
064                         WriteManyToTable(tup_i)
065                     except:
066                         pass
067                     tup_i = ()
068                     print ("html files written to DB %s" % html_written)
069
070                 html_written += 1
071
072             elif splitline[0][:7] == 'http://' and splitline[3] != 'text/html' and
len(splitline) == 5:
073
074                 if len(i) > 0:
075                     tup_i = tup_i + ((url, i),)
076                     url = ''
077                     i = ''
078
079                 mode = False
080
081                 if len(tup_i) >= 100:
082                     print('empty tup_i')
083                     try:
084                         WriteManyToTable(tup_i)
085                     except:
086                         pass
087                     tup_i = ()
088                     print ("html files written to DB %s" % html_written)
089             except:
090                 pass
091
092             if mode == True:
093                 try:
094                     i += unicode(line, "utf-8")
095                 except:
096                     #i += UnicodeDammit(line).unicode markup # Benoetigt Extrem Viele
resourcen
097                     pass
098
099             if len(i) > 0:
100                 tup_i = tup_i + ((url, i),)
101
102             try:
103                 WriteManyToTable(tup_i)
104             except:
105                 pass
106             html_written += len(tup_i)
107             print('parsing %s took %s Minutes \n' % (file, (time.time() - start1) / 60))
108             start = time.time()
109
110
111
112             print ("html files written to DB %s" % html_written)
113             print('export to database took %s Minutes' % ((time.time() - start) / 60))
114             print('complete Operation took %s Minutes' % ((time.time() - start1) / 60))
115             print('#####\n')
116
117
118 def open_Paths(PATH):
119     for path, dirs, files in os.walk(PATH):
120         for filename in files:
121             try:

```

```

122         fullpath = os.path.join(path, filename)
123         print('#####')
124         print('%s Size: %s MB' % (fullpath, os.path.getsize(fullpath) / 1048576))
125         file = gzip.open(fullpath, 'rb')
126         Database_export(file)
127         file.close()
128     except:
129         pass
130
131
132 def current_database():
133     conn, cur = DBConnect()
134     cur.execute('SELECT current_database()')
135     DB_name = cur.fetchone()
136     print('#####')
137     print('Connecting to %s' % DB_name)
138     print('#####')
139
140 if __name__ == '__main__':
141     current_database()
142     raw_input('Please Press the anykey')
143
144     CreateTable()
145     startall = time.time()
146
147     PATH = 'E:\Data'
148     open_Paths(PATH)
149
150     print('+++++#####')
151     print('complete Operation took %s Minutes' % ((time.time() - startall) / 60))
152     print('+++++#####')
153
154     lines = ReadFromTable()
155     print(len(lines))
156     for line in lines:
157         print (line[2])
158     raw_input('Please Press the anykey')

```

## SetVienna

```

001 import psycopg2
002 import gzip
003 from bs4 import UnicodeDammit
004 import os
005 import time
006
007
008
009
010
011
012 def DBConnect():
013     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=#####")
014     cur = conn.cursor()
015     return conn, cur
016
017
018 def CreateTable():
019     conn, cur = DBConnect()
020     cur.execute("CREATE TABLE IF NOT EXISTS html (id serial PRIMARY KEY, url TEXT,
html_file text);")
021     conn.commit()
022     conn.close()
023
024
025 def WriteManyToTable(Values):
026     conn, cur = DBConnect()
027     args_str = ','.join(cur.mogrify("(%s,%s)", x) for x in Values)
028     cur.execute("INSERT INTO html (url, html_file) VALUES " + args_str)
029     conn.commit()
030     conn.close()
031

```

```

032
033 def ReadFromTable():
034     conn, cur = DBConnect()
035     cur.execute("SELECT * FROM html;")
036     lines = cur.fetchall()
037     conn.close()
038     return lines
039
040
041 def Database_export(file):
042     start1 = time.time()
043     i = ''
044     tup_i = ()
045     url = ''
046     html_written = 0
047     mode = False
048     for line in file.readlines():
049         splitline = line.split(' ')
050         try:
051             if splitline[0][:7] == 'http://' and splitline[3] == 'text/html' and
len(splitline) == 5:
052
053                 if len(i) > 0:
054                     tup_i = tup_i + ((url, i),)
055                     url = ''
056                     i = ''
057
058                 url = splitline[0]
059                 mode = True
060
061                 if len(tup_i) >= 100:
062                     print('empty tup_i')
063                     try:
064                         WriteManyToTable(tup_i)
065                     except:
066                         pass
067                     tup_i = ()
068                     print("html files written to DB %s" % html_written)
069
070                 html_written += 1
071
072             elif splitline[0][:7] == 'http://' and splitline[3] != 'text/html' and
len(splitline) == 5:
073
074                 if len(i) > 0:
075                     tup_i = tup_i + ((url, i),)
076                     url = ''
077                     i = ''
078
079                 mode = False
080
081                 if len(tup_i) >= 100:
082                     print('empty tup_i')
083                     try:
084                         WriteManyToTable(tup_i)
085                     except:
086                         pass
087                     tup_i = ()
088                     print("html files written to DB %s" % html_written)
089             except:
090                 pass
091
092             if mode == True:
093                 try:
094                     i += unicode(line, "utf-8")
095                 except:
096                     #i += UnicodeDammit(line).unicode_markup # Benoetigt Extrem Viele
ressourcen
097                 pass
098
099             if len(i) > 0:
100                 tup_i = tup_i + ((url, i),)
101
102             try:
103                 WriteManyToTable(tup_i)
104             except:

```

```

105         pass
106     html_written += len(tup_i)
107     print('parsing %s took %s Minutes \n' % (file, (time.time() - start1) / 60))
108     start = time.time()
109
110
111
112     print ("html files written to DB %s" % html_written)
113     print('export to database took %s Minutes' % ((time.time() - start) / 60))
114     print('complete Operation took %s Minutes' % ((time.time() - start1) / 60))
115     print('#####\n')
116
117
118 def open_Paths(PATH):
119     for path, dirs, files in os.walk(PATH):
120         for filename in files:
121             try:
122                 fullpath = os.path.join(path, filename)
123                 print ('#####')
124                 print('%s Size: %s MB' % (fullpath, os.path.getsize(fullpath) / 1048576))
125                 file = gzip.open(fullpath, 'rb')
126                 Database_export(file)
127                 file.close()
128             except:
129                 pass
130
131
132 def current_database():
133     conn,cur = DBConnect()
134     cur.execute('SELECT current_database()')
135     DB_name = cur.fetchone()
136     print('#####')
137     print('Connecting to %s' % DB_name)
138     print('#####')
139
140 if __name__ == '__main__':
141     current_database()
142     raw_input('Please Press the anykey')
143
144     CreateTable()
145     startall = time.time()
146
147     PATH = './Data'
148     open_Paths(PATH)
149
150     print('+++++#####')
151     print('complete Operation took %s Minutes' % ((time.time() - startall) / 60))
152     print('+++++#####')
153
154     lines = ReadFromTable()
155     print(len(lines))
156     for line in lines:
157         print (line[2])
158     raw_input('Please Press the anykey')

```

## Tag Stripper

```

001 # -*- coding: UTF-8 -*-
002 import psycopg2
003 import time
004 from HTMLParser import HTMLParser
005 import re
006
007
008 def DBConnect():
009     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=#####")
010     cur = conn.cursor()
011     return conn, cur
012
013
014 def ReadFromHTML(offset):
015     conn, cur = DBConnect()

```

```

016     cur.execute("SELECT id,html_file FROM html WHERE vienna = TRUE ORDER BY id limit 1000
offset %s ;" % offset)
017     data = cur.fetchall()
018     cur.close()
019     conn.close()
020
021     return data
022
023
024 def UpdateHTMLwithStrippedHTML(Values):
025     conn, cur = DBConnect()
026     cur.executemany("UPDATE html SET stripped_html = %s WHERE vienna = TRUE AND id in
(%s)", Values)
027     conn.commit()
028     cur.close()
029     conn.close()
030
031
032 def createColumn():
033     conn, cur = DBConnect()
034     cur.execute("ALTER TABLE html DROP COLUMN IF EXISTS stripped_html;")
035     conn.commit()
036     cur.execute("ALTER TABLE html ADD COLUMN stripped_html TEXT;")
037     conn.commit()
038     cur.close()
039     conn.close()
040
041
042 def remove_tags(text):
043     text = TAG_RE.sub('', text)
044     text = Short.sub('', text)
045     text = eszt.sub('ß', text)
046     text = ae.sub('ä', text)
047     text = AE.sub('Ä', text)
048     text = oe.sub('ö', text)
049     text = OE.sub('Ö', text)
050     text = ue.sub('ü', text)
051     text = UE.sub('Ü', text)
052     return text
053
054
055
056
057 TAG_RE = re.compile(r'<[>]+>')
058 Short = re.compile(r'\S{68,}')
059 eszt = re.compile(r'&szlig;')
060 ae = re.compile(r'&auml;')
061 AE = re.compile(r'&Auml;')
062 oe = re.compile(r'&ouml;')
063 OE = re.compile(r'&Ouml;')
064 ue = re.compile(r'&uuml;')
065 UE = re.compile(r'&Uuml;')
066
067
068
069 createColumn()
070 Starttime = time.time()
071 strippedlist = []
072 offset = 0
073 Starttime2 = time.time()
074 htmls = ReadFromHTML(offset)
075 Numberofrows = 8406507
076
077
078
079 while htmls:
080
081     timeregex = time.time()
082     print("starting Regex")
083     for row in htmls:
084         id = row[0]
085         stripped_html = remove_tags(row[1])
086         strippedlist.append((stripped_html, id))
087     print('Regex took %.2f Minutes' % ((time.time() - timeregex) / 60))
088
089     print("Starting DB Update")

```

```

090     timeDBupdate = time.time()
091     UpdateHTMLwithStrippedHTML(strippedlist)
092     print('DB Update took %.2f Minutes' % ((time.time() - timeDBupdate) / 60))
093
094     print offset
095     print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
096     delta_time = time.time() - Starttime
097     print "time till now %.2f Minutes"%(delta_time / 60)
098     print "time till end %.2f Minutes"%(((delta_time/60)/(offset+1000))*(Numberofrows-
(offset+1000)))
099
100
101     Starttime2 = time.time()
102     strippedlist = []
103     offset += 1000
104     htmls = ReadFromHTML(offset)
105
106
107
108 # for row in ReadFromHTML():
109 #     print row[0]
110 #     print row[1]
111 #     print row[2]
112
113
114 print('+++++#####')
115 print('complete Operation took %s Minutes' % ((time.time() - Starttime) / 60))
116 print('+++++#####')

```

## Geo Tagging

```

001 # -*- coding: utf-8 -*-
002 import psycopg2
003 import time
004 import re
005
006 def DBConnect():
007     conn = psycopg2.connect("dbname=TEST_DB user=postgres
password=#####")#Master_DB_spatial2
008     cur = conn.cursor()
009     return conn, cur
010
011
012 def CreateTables():
013     conn, cur = DBConnect()
014     cur.execute("CREATE TABLE IF NOT EXISTS AddressesUnique (id serial PRIMARY KEY,geom
geometry, street text, street_number text, pcode integer);")
015     cur.execute("CREATE TABLE IF NOT EXISTS AddressesUniqueJoinedWithURL (id serial
PRIMARY KEY, AddressesUniqueID INTEGER, HTMLID INTEGER, Original BOOLEAN);")
016     conn.commit()
017     conn.close()
018
019 def addcolumn():
020     conn,cur = DBConnect()
021
022     cur.execute("""
023
024         DO $$
025         BEGIN
026             ALTER TABLE AddressesUnique ADD COLUMN AddDate INT;
027             EXCEPTION
028                 WHEN duplicate_column THEN RAISE NOTICE 'column
AddressesUnique already exists in AddDate.';
029         END;
030     END;
031     $$
032     """
033     )
034     conn.commit()
035
036     cur.execute("""
037         DO $$

```

```

038             BEGIN
039                 BEGIN
040                     ALTER TABLE AddressesUniqueJoinedWithURL ADD COLUMN AddDate INT;
041                     EXCEPTION
042                         WHEN duplicate_column THEN RAISE NOTICE 'column
AddressesUniqueJoinedWithURL already exists in AddDate.';
043                 END;
044             END;
045             $$
046             """
047         )
048     conn.commit()
049
050
051     conn.close()
052
053
054 def MakeAddressesUnique():
055     conn, cur = DBConnect()
056
057     conn.commit()
058     cur.execute("INSERT INTO AddressesUnique (geom, street, street_number, pcode, AddDate) "
059                "SELECT DISTINCT ON (street, street_number)
geom, street, street_number, pcode, AddDate FROM Addresses")
060     conn.commit()
061     return
062
063
064 def ReadFromTableAddressesUnique():
065     conn, cur = DBConnect()
066     cur.execute("SELECT street, street_number, id FROM AddressesUnique;")
067     return cur.fetchall()
068
069
070 def FindVienna():
071     conn, cur = DBConnect()
072     cur.execute("""UPDATE html SET Vienna = TRUE WHERE html_file LIKE '%%' || ' %s ' ||
'%%';""", % 'Wien')
073     conn.commit()
074     conn.close()
075
076
077 def ConstructSQLStatementSearchAddresses(Values):
078     SQLStatementdict = {}
079     conn, cur = DBConnect()
080
081     for line in Values:
082
083         if line[0][-6:] == 'traße':
084             #Berücksichtigt mögliche groß und klein schreibung von Straße
085             SQLStatementdict[line[2]] = cur.mogrify(
086                 "Select ID FROM HTML WHERE "
087                 "Vienna = TRUE AND "
088                 "(textsearchable_index_col @@
to_tsquery('german', '"+line[3]+' & '+line[4]+'') AND "
089                 "stripped_html ILIKE '% "+line[0]+' '+line[1]+' %') "
090                 "OR"
091                 "(textsearchable_index_col @@
to_tsquery('german', '"+line[3]+' & '+line[4]+'/:*') AND " # für address format 18/9
092                 "stripped_html ILIKE '% "+line[0]+' '+line[1]+'/:*') "
093                 "OR"
094                 "(textsearchable_index_col @@
to_tsquery('german', '"+line[3][:-4]+' & '+line[4]+'') AND "
095                 "stripped_html ILIKE '% "+line[0][:-4]+' & '+line[1]+' %') "
096                 "OR"
097                 "(textsearchable_index_col @@
to_tsquery('german', '"+line[3][:-4]+' & '+line[4]+'/:*') AND " # für address format 18/9
098                 "stripped_html ILIKE '% "+line[0][:-4]+' & '+line[1]+'/:*') "
099                 ";")
100
101         elif line[0][-4:] == 'asse':
102             #Berücksichtigt mögliche groß und klein schreibung von Gasse
103             SQLStatementdict[line[2]] = cur.mogrify(

```

```

104         "(textsearchable_index_col @@
to_tsquery('german','"+line[3]+' & '+line[4]+'') AND "
105         "stripped_html ILIKE '% "+line[0]+' '+line[1]+' %' )"
106         "OR"
107         "(textsearchable_index_col @@
to_tsquery('german','"+line[3]+' & '+line[4]+'/*') AND " # für address format 18/9
108         "stripped_html ILIKE '% "+line[0]+' '+line[1]+'/*' )"
109         "OR"
110         "(textsearchable_index_col @@
to_tsquery('german','"+line[3][:-4]+' & '+line[4]+'') AND "
111         "stripped_html ILIKE '% "+line[0][:-4]+' '+line[1]+' %' )"
112         "OR"
113         "(textsearchable_index_col @@
to_tsquery('german','"+line[3][:-4]+' & '+line[4]+'/*') AND " # für address format 18/9
114         "stripped_html ILIKE '% "+line[0][:-4]+' '+line[1]+'/*' )"
115         ";"
116
117
118     else:
119         # Nimmt den Rest auf
120         SQLStatmentdict[line[2]] = cur.mogrify(
121             "Select ID FROM HTML WHERE "
122             "Vienna = TRUE AND "
123             "textsearchable_index_col @@
to_tsquery('german','"+line[3]+' & '+line[4]+'') AND "
124             "stripped_html ILIKE '% "+line[0]+' '+line[1]+' %' "
125             "OR "
126             "textsearchable_index_col @@
to_tsquery('german','"+line[3]+' & '+line[4]+'/*') AND " # für address format 18/9
127             "stripped_html ILIKE '% "+line[0]+' '+line[1]+'/*' "
128             ";"
129
130     return SQLStatmentdict
131
132
133 def JoinAddressesUniqueWithURL(SQLStatmentdict):
134     conn, cur = DBConnect()
135     i = 1
136     Starttime = time.time()
137     Starttime2 = time.time()
138     Numberofrows = len(SQLStatmentdict)
139     cur.execute("TRUNCATE AddressesUniqueJoinedWithURL RESTART IDENTITY;")
140     for ID in SQLStatmentdict:
141         cur.execute(SQLStatmentdict[ID])
142         values = cur.fetchall()
143         if values:
144             args_str = ','.join(cur.mogrify("(%s,%s,TRUE)", (ID,x[0])) for x in values)
145             cur.execute("INSERT INTO AddressesUniqueJoinedWithURL (AddressesUniqueID,
HTMLID, Original) VALUES " + args_str)
146             conn.commit()
147
148             print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
149             delta_time = time.time() - Starttime
150             print("time till now %.2f Minutes"%(delta_time/60))
151             print("time till end %.2f Minutes"%(((delta_time/60)/i)*(Numberofrows-i))
152             i += 1
153             Starttime2 = time.time()
154     conn.close()
155     return
156
157 def CleanStrings(lines):
158     for row in lines:
159         StreetName = row[0]
160         StreetNumber = row[1]
161         ID = row[2]
162         p = re.compile(r' ')
163         q = re.compile(r'[^-/a-zA-Z0-9_]')
164         r = re.compile(r'[0-9a-zA-Z] [-/] [0-9a-zA-Z]')
165         s = re.compile(r'""')
166
167         if r.match(StreetNumber):
168             # Match adress nummer die so aussehen
169             "8 - 9" "4a - g" und "7 / 8" angepasst durch fehlern die ausgeworfen wurden
170             StreetNumber = p.sub(' ', StreetNumber) # ersetzt die leer zeichen mit
171             nichts
172             StreetNumber = q.sub(' ', StreetNumber ) # Klammer in der Nummer

```



```

170         StreetName = s.sub("'", StreetName) # fügt einen weiteren goute ' hinzu um denn
ersten im to tsquery und ilike zu escapen
171         row3 = p.sub(' & ', StreetName) # ersetzt Leer zeichen im straßen nahmen mit ' & '
sonst gehen sie nicht durch to tsquery
172         row4 = p.sub(' & ', StreetNumber ) # ersetzt Leer zeichen im Straßen Nummern namen
mit ' & ' bsp.: "Objekt 11" wird zu "Objekt & 11" sonst gehen sie nicht durch to_tsquery
173
174         lines.remove(row)
175         lines.insert(0,(StreetName,StreetNumber,ID,row3,row4))
176
177     return lines
178
179
180 def CreateIndex():
181     conn, cur = DBConnect()
182
183     cur.execute("ALTER TABLE html ADD COLUMN textsearchable_index_col tsvector;")
184     conn.commit()
185     cur.execute("UPDATE html SET textsearchable_index_col = to_tsvector('german',
stripped_html) WHERE Vienna = True;")
186     conn.commit()
187     cur.execute("CREATE INDEX textsearch_idx ON html USING
gin(textsearchable_index_col);")
188     conn.commit()
189     cur.close()
190     conn.close()
191
192
193
194 Starttime = time.time()
195 CreateTables()
196 addcolumn()
197 print("creating Index")
198 CreateIndex()
199 print("Make Addresses Unique")
200 MakeAddressesUnique()
201 lines = ReadFromTableAddressesUnique()
202 print lines
203
204 lines = CleanStrings(lines)
205 print lines
206 print len(lines)
207 SQLStatmentdict = ConstructSQLStatmentSearchAddresses(lines)
208 print SQLStatmentdict
209 JoinAddressesUniqueWithURL(SQLStatmentdict)
210
211
212
213 print('+++++#####')
214 print('complete Operation took %s Minutes' % ((time.time() - Starttime) / 60))
215 print('+++++#####')

```

## Find Links

```

001 # -*- coding: utf-8 -*-
002 import psycopg2
003 import time
004 import re
005 import urlparse
006
007
008 def DBConnect():
009     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres
password=#####")
010     cur = conn.cursor()
011     return conn, cur
012
013 def GetGeocodedHTMLIDs(conn, cur):
014
015     cur.execute("SELECT HTMLID, AddressesUniqueID FROM AddressesUniqueJoinedWithURL WHERE
Original = TRUE")
016     data = cur.fetchall()

```

```

017
018 datadict = {}
019 for row in data:
020     if row[0] in datadict:
021         datadict[row[0]].append(row[1])
022     else:
023         datadict[row[0]] =[row[1],]
024
025     return datadict
026
027 def FindLinksInHtml(conn, cur, offset):
028     NoWhiteSpace = re.compile(r' ')
029     loadingtime = time.time()
030     cur.execute("SELECT id,url,html_file FROM html WHERE id > %s AND id <= %s ORDER BY
id;",(offset, offset+limit))
031     data = cur.fetchall()
032     print('Loading took %.2f Minutes' % ((time.time() - loadingtime) / 60))
033     regextime = time.time()
034     passeslist = []
035     linklist=[]
036     for row in data:
037         links = re.findall(r'href=[\'"]?([^\'" >]+)', row[2])
038         for link in links:
039             try:
040                 linklist.append((row[0],NoWhiteSpace.sub('%20', urlparse.urljoin(row[1],
link)))) #makes the links absolut and removes Whitespaces
041             except:
042                 passeslist.append((row[1], link))
043     print('Regex took %.2f Minutes found links %s' % (((time.time() - regextime) / 60),
len(linklist)))
044     return linklist, passeslist
045
046
047 def URLsWithID(conn, cur):
048     cur.execute("SELECT URL,ID FROM html;")
049     data = cur.fetchall()
050     dictionary = dict(data)
051     return dictionary
052
053 def WritetoAddressesUniqueJoinedWithURL(List):
054     conn, cur = DBConnect()
055     args_str = ','.join(cur.mogrify("(%s,%s,FALSE)", x) for x in List)
056     try:
057         cur.execute("INSERT INTO AddressesUniqueJoinedWithURL (HTMLID, AddressesUniqueID,
Original)VALUES " + args_str)
058         conn.commit()
059     except:
060         print "Error Inserting Joins"
061     cur.close()
062     conn.close()
063
064     return
065
066
067 def get_html_ids():
068     conn, cur = DBConnect()
069     cur.execute("SELECT id FROM html")
070     data = cur.fetchall()
071
072     cur.close()
073     conn.close()
074
075     return data
076
077
078
079
080 conn, cur = DBConnect()
081 Starttime = time.time()
082
083
084 print('getting URLEDictionary and URLIDWITHAddressIDDictionary')
085 URLEDictionary = URLsWithID(conn, cur)
086 URLIDWITHAddressIDDictionary = GetGeocodedHTMLIDs(conn, cur)
087
088 limit = 1000

```

```

089 offset = 0
090 passeslist = []
091
092 Numberofrows = 8406507
093
094 r = 0
095 while offset <= Numberofrows:
096     Starttime2 = time.time()
097     print offset
098     linklist,passes = FindLinksInHtml(conn, cur, offset)
099
100     passeslist += passes
101     URLIDStoLinkIDS = []
102     for row in linklist:
103         if row[1] in URLDictionary:
104             URLIDStoLinkIDS.append((row[0],URLDictionary[row[1]]))
105
106     Newlist = []
107     for row in URLIDStoLinkIDS:
108         if row[1] in URLIDWITHAddressIDDictionary:
109             for rowx in URLIDWITHAddressIDDictionary[row[1]]:
110                 Newlist.append((row[0],rowx))
111
112
113
114     r += len(Newlist)
115     WritetoAddressesUniqueJoinedWithURL(Newlist)
116
117     print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
118     delta_time = time.time() - Starttime
119     print('time till now %.2f Minutes'%(delta_time / 60))
120     print('time till end %.2f Minutes'%(((delta_time/60)/(offset+1000))*(Numberofrows-
121 (offset+1000))))
122     offset += limit
123
124 cur.close()
125 conn.close()
126 print('Passes %s' % len(passeslist))
127 print passeslist
128 print r
129 print('+++++')
130 print('complete Operation took %s Minutes' % ((time.time() - Starttime) / 60))
131 print('+++++')

```

## Wikipedia POS Tagging

```

001 # -*- coding: utf-8 -*-
002 import time
003 from pattern.de import parse
004 import os
005 import io
006 import cPickle as pickle
007 from multiprocessing import Pool as ThreadPool
008 from threading import current_thread
009
010
011 def POStag(filepath):
012
013     global i
014     global len_filelist
015     i += 1
016     newcorpustagged = []
017     Starttime2 = time.time()
018
019     with io.open(filepath, 'r', encoding='utf-8') as mfile:
020         data = mfile.read()
021         data = data.splitlines()
022
023
024     for section in data:
025         section = parse(section)

```

```

026         section = section.split()
027
028         for sentence in section:
029             sent = []
030             for token in sentence:
031                 sent.append((token[0], token[1]))
032             newcorpustagged.append(sent)
033
034
035     with io.open('./wikicorpuspickeld_2/%s_%s.pos' % (current_thread().ident, i), 'wb')
as fout:
036         pickle.dump(newcorpustagged, fout)
037
038
039     print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
040     delta_time = time.time() - Starttime
041     print("time till now %.2f Minutes" % (delta_time/60))
042     print("time till end %.2f Minutes" % (((delta_time/60)/(i*4))*(len_filelist-(i*4))))
043
044
045 def createfilepathlist():
046     pathlist = []
047     subfolders = [x[0] for x in os.walk('./WikiText/')]
048     for subfolder in subfolders[1:]:
049         for filename in os.listdir(subfolder):
050             pathlist.append(subfolder+'/'+filename)
051
052     return pathlist
053
054
055 i = 0
056 filepathlist = createfilepathlist()
057 len_filelist = len(filepathlist)
058 Starttime = time.time()
059
060 if __name__ == '__main__':
061
062     print('Tagging new corpus')
063     pool = ThreadPool(4)
064     pool.map(POSTag, filepathlist)
065     pool.close()
066     pool.join()
067
068     print('#####')
069     print('complete Operation took %s Minutes' % ((time.time() - Starttime) / 60))
070     print('#####')

```

## Co-occurrence Group Generation

```

001 # -*- coding: utf-8 -*-
002 import time
003 from nltk.corpus import stopwords
004 import os
005 import io
006 import cPickle as pickle
007
008
009 def NounVerb(tag):
010     noun_verb_list = ['nn', 'nns', 'np', 'nnp', 'nps', 'vb', 'vzb', 'vbp', 'vbd',
011                       'vbn', 'vbg']
012
013     if tag.lower() in noun_verb_list:
014         return True
015
016     return False
017
018 def stopwords_list():
019     new_list = []
020     for word in stopwords.words('german'):
021         new_list.append(unicode(word.decode('latin-1')))
022     return new_list

```

```

023
024
025 def CoOccurrence(groups):
026     Starttime3 = time.time()
027     Fenster = 10
028     i = 1
029     S_list = stopwords_list()
030     word_dict = {}
031
032     Files = [x[2] for x in os.walk('./wikicorpuspickeld_2/')]
033     for file in Files[0]:
034
035         with io.open('./wikicorpuspickeld_2/'+file, 'rb') as fin:
036             loaded_corpus = pickle.load(fin)
037
038
039         for sentence in loaded_corpus:
040             for (index, tokentag) in enumerate(sentence):
041                 (token, tag) = tokentag
042                 token = token.lower()
043
044                 if token in groups:
045                     term = sentence[index-Fenster:index+Fenster]
046                     for (term_token, term_tag) in term:
047
048                         term_token = term_token.lower()
049                         if term_token not in S_list and NounVerb(term_tag):
050
051                             if token not in word_dict:
052                                 word_dict[token] = {}
053                             if term_token in word_dict[token]:
054                                 word_dict[token][term_token] += 1
055                             else:
056                                 word_dict[token][term_token] = 1
057
058             print i
059
060             delta_time = time.time() - Starttime3
061             print "time till end %.2f Minutes" % (((delta_time/60)/i)*(len(Files[0])-i))
062             i += 1
063
064         return word_dict
065
066 groups = [u'wohnen', u'arbeiten', u'bildung', u'einkaufen', u'gaststätte', u'hotel',
067 u'kreditinstitut', u'kultur', u'dienstgebäude',]
068
069 Starttime2 = time.time()
070 CoOccurrenceGroups = CoOccurrence(groups)
071
072 directory = './topics/'
073 if not os.path.exists(directory):
074     os.makedirs(directory)
075
076 with io.open(directory+'CoOccurrenceGroups.pickle', 'wb') as fout:
077     pickle.dump(CoOccurrenceGroups, fout)
078
079 print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
080 delta_time = time.time() - Starttime

```

## Inverse Document Frequency

```

001 import cPickle as pickle
002 import io
003 import psycpg2
004 import numpy
005 import random
006 import sys
007
008
009 def DBConnect():

```

```

010     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres
password=####")#Master_DB_spatial2
011     cur = conn.cursor()
012     return conn, cur
013
014
015 def countrows():
016     conn, cur = DBConnect()
017     cur.execute("SELECT count(*) FROM HtmlUnique;")
018     data = cur.fetchone()
019     cur.close()
020     conn.close()
021     return data[0]
022
023
024 with io.open('./Vector/CombinedVectorSpace.pickle', 'rb') as fin:
025     CombinedVectorSpace = pickle.load(fin)
026
027 CombinedVectorSpaceIDFT = {}
028 DocumentCount = countrows()
029
030 for key in CombinedVectorSpace:
031     idft = numpy.log(numpy.divide(float(DocumentCount),
float((1+CombinedVectorSpace[key][0]))))
032
033     CombinedVectorSpaceIDFT[key] = CombinedVectorSpace[key][0],
CombinedVectorSpace[key][1], idft
034
035 with io.open('./Vector/CombinedVectorSpaceIDFT.pickle', 'wb') as fout:
036     pickle.dump(CombinedVectorSpaceIDFT, fout)
037
038 CombinedVectorSpaceIDFTKeyList = []
039
040 for key in CombinedVectorSpaceIDFT:
041     CombinedVectorSpaceIDFTKeyList.append(key)
042
043 CombinedVectorSpaceIDFTKeyList.sort()
044
045 with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'wb') as fout:
046     pickle.dump(CombinedVectorSpaceIDFTKeyList, fout)

```

## Wikipedia Vector Space

```

001 # -*- coding: utf-8 -*-
002 import time
003 from nltk.corpus import stopwords
004 import nltk
005 import re
006 import os
007 import io
008 import cPickle as pickle
009
010
011 def Vector_Calculator():
012     Starttime3 = time.time()
013     i = 1
014     GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
015     token_dict_file = {}
016     p = re.compile(ur'^[a-zA-Z\s]{6,}$', re.UNICODE)
017
018     Files = [x[2] for x in os.walk('E:/Tools/Topic Generation/wikicorpuspickeld_2/')]
019     for file in Files[0]:
020         with io.open('E:/Tools/Topic Generation/wikicorpuspickeld_2/'+file, 'rb') as fin:
021             loaded_corpus = pickle.load(fin)
022
023             for sentence in loaded_corpus:
024                 for (index, tagtuple) in enumerate(sentence):
025                     (token, tag) = tagtuple
026                     token = token.lower()
027                     if token not in stopword_list:
028                         if p.match(token):
029                             stemmedtoken = GermanStemmer.stem(token)

```

```

030
031         if stemmedtoken in token_dict_file:
032             token_dict_file[stemmedtoken] += 1
033         else:
034             token_dict_file[stemmedtoken] = 1
035
036     delta_time = time.time() - Starttime3
037     print "time till end %.2f Minutes" % (((delta_time/60)/i)*(len(Files[0])-i))
038     i += 1
039
040     return token_dict_file
041
042 stopword_list = []
043 for word in stopwords.words('german'):
044     stopword_list.append(unicode(word.decode('latin-1')))
045
046 Starttime = time.time()
047 Vectorraum = Vector_Calculator()
048
049 with io.open('./Vector/WikiVectorSpace2.pickle', 'wb') as fout:
050     pickle.dump(Vectorraum, fout)
051 print('Operation took %.2f Minutes' % ((time.time() - Starttime) / 60))

```

## HTML Vector Space

```

001 # -*- coding: UTF-8 -*-
002 import nltk
003 from nltk.tokenize import RegexpTokenizer
004 import psycopg2
005 import time
006 import cPickle as pickle
007 import io
008 import re
009
010
011 def DBConnect():
012     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres
password=#####")#Master_DB_spatial2
013     cur = conn.cursor()
014     return conn, cur
015
016
017 def stripped_htmls(ID_list):
018     conn, cur = DBConnect()
019     data = []
020     for ID in ID_list:
021         cur.execute("SELECT stripped_html FROM HtmlUnique WHERE ID = %s ", ID)
022         html = cur.fetchone()
023         if html:
024             data.append(html)
025         else:
026             error += 1
027     print "Number of Errors %s" % error
028     cur.close()
029     conn.close()
030
031     return data
032
033
034 def get_html_ids():
035     conn, cur = DBConnect()
036     cur.execute("select DISTINCT HTMLID FROM AddressesUniqueJoinedWithURL ORDER BY
HTMLID")
037     data = cur.fetchall()
038
039     cur.close()
040     conn.close()
041
042     return data
043
044
045 def Delete_stopwords(Tokens):

```

```

046     return [token for token in Tokens if not token in
nlk.corpus.stopwords.words('german')]
047
048 GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
049 tokenizer = RegexpTokenizer(r'\w+')
050 token_dict = {}
051 HTMLIDS = get_html_ids()
052
053 lower = 0
054 upper = lower + 1000
055 Starttime = time.time()
056 parsedhtmls = 0
057
058 while lower <= len(HTMLIDS):
059
060     Starttime2 = time.time()
061     stripped_htmls_list = stripped_htmls(HTMLIDS[lower:upper])
062     for html in stripped_htmls_list:
063         parsedhtmls += 1
064
065         time_tokenize = time.time()
066         tokens = tokenizer.tokenize(html)
067         tokens = Delete_stopwords(tokens)
068         token_dict_file = {}
069
070         for token in tokens:
071
072             stemmedtoken = GermanStemmer.stem(token)
073
074             if stemmedtoken in token_dict_file:
075                 token_dict_file[stemmedtoken] += 1
076             else:
077                 token_dict_file[stemmedtoken] = 1
078
079         for key in token_dict_file:
080             if key in token_dict:
081                 doc_count = token_dict[key][0] + 1
082                 occurrence_count = token_dict[key][1] + token_dict_file[key]
083                 token_dict[key] = (doc_count, occurrence_count)
084             else:
085                 token_dict[key] = (1, token_dict_file[key])
086
087         print('Number of tokens in dict: %s' % len(token_dict))
088         print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
089         delta_time = time.time() - Starttime
090         print("time till now %.2f Minutes"%(delta_time / 60))
091         print("time till end %.2f Minutes"%(((delta_time/60)/(upper))*(len(HTMLIDS)-(upper))))
092
093         lower += 1000
094         upper = lower + 1000
095
096 with io.open('./Vector/HTMLVectorSpace.pickle', 'wb') as fout:
097     pickle.dump(token_dict, fout)
098
099 print 'parsed HTML Files %s' % parsedhtmls

```

## Combined Vector Space

```

001 import io
002 import cPickle as pickle
003
004 with io.open('./Vector/WikiVectorSpace.pickle', 'rb') as fin:
005     WikiVectorSpace = pickle.load(fin)
006
007 with io.open('./Vector/HTMLVectorSpace.pickle', 'rb') as fin:
008     HTMLVectorSpace = pickle.load(fin)
009
010 CombinedVectorSpace = {}
011
012 for key in WikiVectorSpace:
013     if key in HTMLVectorSpace:
014         CombinedVectorSpace[key] = HTMLVectorSpace[key]

```



```

015
016 with io.open('./Vector/CombinedVectorSpace.pickle', 'wb') as fout:
017     pickle.dump(CombinedVectorSpace, fout)

```

## HTML Tokenization

```

001 import nltk
002 from nltk.tokenize import RegexpTokenizer
003 import psycopg2
004 import time
005 import cPickle as pickle
006 import io
007 import re
008
009
010 def DBConnect():
011     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres
password=#####")#Master_DB_spatial2
012     cur = conn.cursor()
013     return conn, cur
014
015
016 def createColumn():
017     conn, cur = DBConnect()
018     cur.execute("ALTER TABLE HtmlUnique DROP COLUMN IF EXISTS VectorDICT;")
019     conn.commit()
020     cur.execute("ALTER TABLE HtmlUnique ADD COLUMN VectorDICT bytea;")
021     conn.commit()
022     cur.close()
023     conn.close()
024
025
026 def ReadFromHTML(offset):
027     conn, cur = DBConnect()
028     cur.execute("SELECT id,stripped_html FROM ORDER BY id limit 1000 offset %s ;",
(offset,))
029     data = cur.fetchall()
030     cur.close()
031     conn.close()
032
033     return data
034
035
036 def Delete_stopwords(Tokens):
037     return [token for token in Tokens if not token in
nltk.corpus.stopwords.words('german')]
038
039
040 def UpdateHtmlUniquewithDict(Dict,ID):
041     conn, cur = DBConnect()
042     cur.execute("UPDATE HtmlUnique SET VectorDICT = %s WHERE id = %s",
(psycopg2.Binary(Dict), ID,))
043     conn.commit()
044     cur.close()
045     conn.close()
046
047
048 def countrows():
049     conn, cur = DBConnect()
050     cur.execute("select count(id) from HtmlUnique;")
051     data = cur.fetchone()
052     cur.close()
053     conn.close()
054     return data[0]
055
056
057 with io.open('./Vector\CombinedVectorSpace.pickle', 'rb') as fin:
058     CombinedVectorSpace = pickle.load(fin)
059
060
061 createColumn()
062

```

```

063 offset = 0
064 htms = ReadFromHTML(offset)
065 tokenizer = RegexpTokenizer(r'\w+')
066 GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
067 Starttime = time.time()
068 Length = countrows()
069
070 while htms:
071     print len(htms)
072     Starttime2 = time.time()
073     for html in htms:
074         HTMLdict = {}
075         id, HTMLtext = html
076         tokens = tokenizer.tokenize(HTMLtext)
077
078         for token in tokens:
079             stemmedtoken = GermanStemmer.stem(token)
080             if stemmedtoken in CombinedVectorSpace:
081                 if stemmedtoken in HTMLdict:
082                     HTMLdict[stemmedtoken] += 1
083                 else:
084                     HTMLdict[stemmedtoken] = 1
085
086         htmlDictpickeld = pickle.dumps(HTMLdict)
087         UpdateHtmlUniquewithDict(htmlDictpickeld, id)
088
089     offset += 1000
090     print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
091     delta_time = time.time() - Starttime
092     print "time till now %.2f Minutes"%(delta_time / 60)
093     print "time till end %.2f Minutes"%(((delta_time/60)/offset)*(Length-offset))
094     htms = ReadFromHTML(offset)

```

## TFIDF Vector HTML Documents

```

001 import cPickle as pickle
002 import io
003 import psycopg2
004 import numpy
005 import time
006 import zlib
007
008 def DBConnect():
009     conn = psycopg2.connect("dbname=TEST_DB user=postgres password=#####")
010     cur = conn.cursor()
011     return conn, cur
012
013
014 def countrows():
015     conn, cur = DBConnect()
016     cur.execute("SELECT count(*) FROM HtmlUnique;")
017     data = cur.fetchone()
018     cur.close()
019     conn.close()
020     return data[0]
021
022 def VectorDICTReader(offset):
023     conn, cur = DBConnect()
024     cur.execute("SELECT id,VectorDICT FROM HtmlUnique order by id limit 100 offset %s"
025 ;", (offset,))
026     data = cur.fetchall()
027     cur.close()
028     conn.close()
029     return data
030
031 def UpdateHtmlUniquewithTFIDFlist(Values):
032     conn, cur = DBConnect()
033     cur.executemany("UPDATE HtmlUnique SET TFIDFVector = %s WHERE id = %s", Values)
034     conn.commit()
035     cur.close()
036     conn.close()

```

```

037
038
039 def createColumn():
040     conn, cur = DBConnect()
041     cur.execute("ALTER TABLE HtmlUnique DROP COLUMN IF EXISTS TFIDFVector;")
042     conn.commit()
043     cur.execute("ALTER TABLE HtmlUnique ADD COLUMN TFIDFVector bytea;")
044     conn.commit()
045     cur.close()
046     conn.close()
047
048
049 offset = 0
050 length = countrows()
051 Starttime = time.time()
052 createColumn()
053
054 with io.open('./Vector/CombinedVectorSpaceIDFT.pickle', 'rb') as fin:
055     CombinedVectorSpaceIDFT = pickle.load(fin)
056
057 with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'rb') as fin:
058     CombinedVectorSpaceIDFTKeyList = pickle.load(fin)
059
060 while offset <= length:
061     dicts = VectorDICTReader(offset)
062     Starttime2 = time.time()
063     arraylist = []
064     for tuple in dicts:
065         array = []
066         id = tuple[0]
067         dictionary = pickle.loads(str(tuple[1]))
068
069         for key in CombinedVectorSpaceIDFTKeyList:
070
071             if key in dictionary:
072                 array.append(numpy.multiply(CombinedVectorSpaceIDFT[key][2],
dictionary[key]))
073             else:
074                 array.append(0)
075             array = numpy.array(array)
076             array = numpy.divide(array, numpy.linalg.norm(array))
077             array = pickle.dumps(array)
078             array = zlib.compress(array)
079             arraylist.append((psycopg2.Binary(array), id,))
080
081     UpdateHtmlUniquewithTFIDFList(arraylist)
082     offset += 100
083     print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
084     delta_time = time.time() - Starttime
085     print("time till now %.2f Minutes"%(delta_time / 60))
086     print("time till end %.2f Minutes"%(((delta_time/60)/offset)*(length-offset))

```

## TFIDF Vector for Search Term and Co-ccurence Groups

```

001 import cPickle as pickle
002 import io
003 import nltk
004 import numpy
005 import zlib
006
007
008 with io.open('./Vector/CombinedVectorSpaceIDFT.pickle', 'rb') as fin:
009     CombinedVectorSpaceIDFT = pickle.load(fin)
010
011 with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'rb') as fin:
012     CombinedVectorSpaceIDFTKeyList = pickle.load(fin)
013
014 with io.open('./Co-Occurrence.pickle', 'rb') as fin:
015     CoOc = pickle.load(fin)
016
017 GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
018

```

```

019
020 TFIDF_CoOc = {}
021 for searchterm in CoOc:
022     TFIDF_CoOc[searchterm] = {}
023     TFIDF_CoOc[searchterm]['Stemmed'] = {}
024     for token, counter in CoOc[searchterm]:
025         token = GermanStemmer.stem(token)
026         if token in TFIDF_CoOc[searchterm]:
027             TFIDF_CoOc[searchterm]['Stemmed'][token] =
TFIDF_CoOc[searchterm]['Stemmed'][token]+tuple[1]
028         else:
029             TFIDF_CoOc[searchterm]['Stemmed'][token] = tuple[1]
030
031
032 for searchterm in TFIDF_CoOc:
033     array = []
034     dictionary = TFIDF_CoOc[searchterm]['Stemmed']
035     for key in CombinedVectorSpaceIDFTKeyList:
036         if key in dictionary:
037             array.append(numpy.multiply(CombinedVectorSpaceIDFT[key][2],
dictionary[key]))
038         else:
039             array.append(0)
040     array = numpy.array(array)
041     array = numpy.divide(array,numpy.linalg.norm(array))
042     array = pickle.dumps(array)
043     array = zlib.compress(array)
044     TFIDF_CoOc[searchterm]['TFIDF_CoOc'] = array
045
046
047     STarray = []
048     searchtermstemmed = GermanStemmer.stem(searchterm)
049     for key in CombinedVectorSpaceIDFTKeyList:
050         if key in searchtermstemmed:
051             STarray.append(numpy.multiply(CombinedVectorSpaceIDFT[key][2], 1))
052         else:
053             STarray.append(0)
054
055     STarray = numpy.array(STarray)
056     STarray = numpy.divide(STarray,numpy.linalg.norm(STarray))
057     STarray = pickle.dumps(STarray)
058     STarray = zlib.compress(STarray)
059     TFIDF_CoOc[searchterm]['TFIDF_ST'] = STarray
060
061 with io.open('./Vector/TFIDF_CoOc.pickle', 'wb') as fout:
062     pickle.dump(TFIDF_CoOc, fout)

```

## Cosine Similarity

```

001 import cPickle as pickle
002 import io
003 import psycopg2
004 import time
005 import zlib
006 from sklearn.metrics.pairwise import cosine_similarity
007
008
009 def DBConnect():
010     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=#####")
011     cur = conn.cursor()
012     return conn, cur
013
014
015 def VectorDICTReader(range,offset):
016     conn, cur = DBConnect()
017     cur.execute("SELECT id,TFIDFVector FROM HtmlUnique order by id limit %s offset %s
;",(range,offset,))
018     data = cur.fetchall()
019     cur.close()
020     conn.close()
021     return data
022

```

```

023
024 def createColumn(column):
025     conn, cur = DBConnect()
026     sqlstring = "ALTER TABLE HtmlUnique DROP COLUMN IF EXISTS %s;" % column
027     cur.execute(sqlstring)
028     conn.commit()
029     sqlstring = "ALTER TABLE HtmlUnique ADD COLUMN %s FLOAT;" % column
030     cur.execute(sqlstring)
031     conn.commit()
032     cur.close()
033     conn.close()
034
035
036 def Sqlstringconstructor(Columnlist):
037
038     sqlstring = "UPDATE HtmlUnique SET "
039     for key in Columnlist[:-1]:
040         sqlstring += key+' = %s, '
041     sqlstring += Columnlist[-1]+' = %s'
042     sqlstring += ' WHERE id = %s'
043     print sqlstring
044     return sqlstring
045
046
047 def UpdateHtmlUniquewithCosinelist(sqlstring,Values):
048     conn, cur = DBConnect()
049     cur.executemany(sqlstring, Values)
050     conn.commit()
051     cur.close()
052     conn.close()
053
054
055 def countrows():
056     conn, cur = DBConnect()
057     cur.execute("SELECT count(*) FROM HtmlUnique;")
058     data = cur.fetchone()
059     cur.close()
060     conn.close()
061     return data[0]
062
063 #####
064
065 with io.open('./Vector/TFIDF_CoOc.pickle', 'rb') as fin:
066     TFIDF_CoOc = pickle.load(fin)
067
068 Columnlist = []
069 SearchTermList = []
070 TFIDFworkingdict = {}
071
072 for searchterm in TFIDF_CoOc:
073     SearchTermList.append(searchterm)
074     Columnlist.append(TFIDF_CoOc[searchterm]+'_CoOc')
075     Columnlist.append(TFIDF_CoOc[searchterm]+'_ST')
076
077 for searchterm in SearchTermList:
078     TFIDFworkingdict[searchterm] =
079 pickle.loads(zlib.decompress(TFIDF_CoOc[searchterm]['TFIDF_CoOc'])),\
080 pickle.loads(zlib.decompress(TFIDF_CoOc[searchterm]['TFIDF_ST']))
081
082 for searchterm in Columnlist:
083     createColumn(searchterm)
084
085 sqlstring = Sqlstringconstructor(Columnlist)
086
087 offset = 0
088 range = 1000
089 length = countrows()
090
091 vectors = VectorDICTReader(range,offset)
092
093 Starttime = time.time()
094 while vectors:
095     Starttime2 = time.time()
096     updatelist = []
097     for vectortup in vectors:

```

```

097     cosinelist = []
098     id, vector = vectortup[0], pickle.loads(zlib.decompress(vectortup[1]))
099     for searchterm in SearchTermList:
100         cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][0], vector),8)
101         cosinelist.append(cosine)
102         cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][1], vector),8)
103         cosinelist.append(cosine)
104
105     cosinelist.append(id)
106     updatelist.append(cosinelist)
107     UpdateHtmlUniquewithCosinelist(sqlstring,updatelist)
108     offset += range
109     print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
110     delta_time = time.time() - Starttime
111     print "time till now %.2f Minutes"%(delta_time / 60)
112     print "time till end %.2f Minutes"%(((delta_time/60)/offset)*(length-offset))
113
114     vectors = VectorDICTReader(range,offset)

```

## Address Classification

```

001 import cPickle as pickle
002 import io
003 import psycopg2
004 import time
005 import zlib
006 from sklearn.metrics.pairwise import cosine_similarity
007
008
009 def DBConnect():
010     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=#####")
011     cur = conn.cursor()
012     return conn, cur
013
014
015 def VectorDICTReader(range,offset):
016     conn, cur = DBConnect()
017     cur.execute("SELECT id,TFIDFVector FROM HtmlUnique order by id limit %s offset %s"
018 ;", (range,offset,))
019     data = cur.fetchall()
020     cur.close()
021     conn.close()
022     return data
023
024 def createColumn(column):
025     conn, cur = DBConnect()
026     sqlstring = "ALTER TABLE HtmlUnique DROP COLUMN IF EXISTS %s;" % column
027     cur.execute(sqlstring)
028     conn.commit()
029     sqlstring = "ALTER TABLE HtmlUnique ADD COLUMN %s FLOAT;" % column
030     cur.execute(sqlstring)
031     conn.commit()
032     cur.close()
033     conn.close()
034
035
036 def Sqlstringconstructor(Columnlist):
037
038     sqlstring = "UPDATE HtmlUnique SET "
039     for key in Columnlist[:-1]:
040         sqlstring += key+' = %s, '
041     sqlstring += Columnlist[-1]+' = %s'
042     sqlstring += ' WHERE id = %s'
043     print sqlstring
044     return sqlstring
045
046
047 def UpdateHtmlUniquewithCosinelist(sqlstring,Values):
048     conn, cur = DBConnect()
049     cur.executemany(sqlstring, Values)
050     conn.commit()

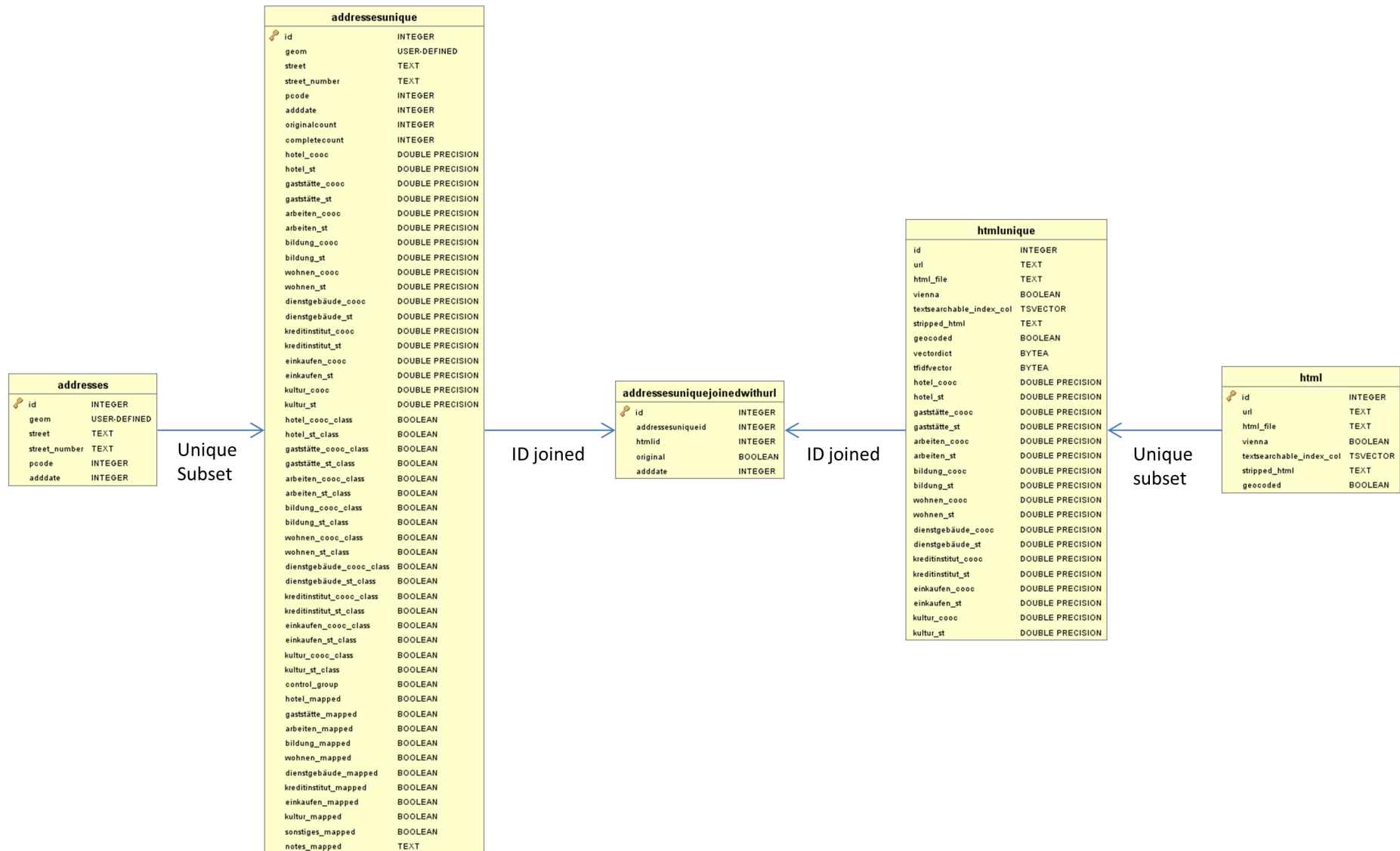
```

```

051     cur.close()
052     conn.close()
053
054
055 def countrows():
056     conn, cur = DBConnect()
057     cur.execute("SELECT count(*) FROM HtmlUnique;")
058     data = cur.fetchone()
059     cur.close()
060     conn.close()
061     return data[0]
062
063 #####
064
065 with io.open('./Vector/TFIDF_CoOc.pickle', 'rb') as fin:
066     TFIDF_CoOc = pickle.load(fin)
067
068 Columnlist = []
069 SearchTermList = []
070 TFIDFworkingdict = {}
071
072 for searchterm in TFIDF_CoOc:
073     SearchTermList.append(searchterm)
074     Columnlist.append(TFIDF_CoOc[searchterm]+'_CoOc')
075     Columnlist.append(TFIDF_CoOc[searchterm]+'_ST')
076
077 for searchterm in SearchTermList:
078     TFIDFworkingdict[searchterm] =
pickle.loads(zlib.decompress(TFIDF_CoOc[searchterm]['TFIDF_CoOc'])),\
079
pickle.loads(zlib.decompress(TFIDF_CoOc[searchterm]['TFIDF_ST']))
080
081 for searchterm in Columnlist:
082     createColumn(searchterm)
083
084 sqlstring = Sqlstringconstructor(Columnlist)
085
086 offset = 0
087 range = 1000
088 length = countrows()
089
090 vectors = VectorDICTReader(range,offset)
091
092 Starttime = time.time()
093 while vectors:
094     Starttime2 = time.time()
095     updatelist = []
096     for vectortup in vectors:
097         cosinelist = []
098         id, vector = vectortup[0], pickle.loads(zlib.decompress(vectortup[1]))
099         for searchterm in SearchTermList:
100             cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][0], vector),8)
101             cosinelist.append(cosine)
102             cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][1], vector),8)
103             cosinelist.append(cosine)
104
105         cosinelist.append(id)
106         updatelist.append(cosinelist)
107     UpdateHtmlUniquewithCosinelist(sqlstring,updatelist)
108     offset += range
109     print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
110     delta_time = time.time() - Starttime
111     print "time till now %.2f Minutes"%(delta_time / 60)
112     print "time till end %.2f Minutes"%(((delta_time/60)/offset)*(length-offset))
113
114     vectors = VectorDICTReader(range,offset)

```

## Database Schema





## Mapping Results

ID	Street	street_n umber	post code	hotel	gaststätte	arbeiten	bildung	wohnen	dienstgebäude	kreditinstitut	einkaufen	kultur	other	Notes
87	Aegidigasse	7-11	1060					t						
143	Albertgasse	1a	1080										t	Rechtsanwalt, Immobilien, PR-Agentur, edv, Ärzte Flugambulanz
329	Alser Straße	71	1080										t	Pensionistenclub
440	Am Hof	13	1010								t		t	Gewürz geschafft, Kosmetikgeschäft, Consulting, Medienproduktion,
707	Argentinierstraß e	14	1040						t					Griechische Botschaft
724	Argentinierstraß e	30	1040									t		Funkhaus
725	Argentinierstraß e	30A	1040		t							t		Funkhaus, kaffee
732	Argentinierstraß e	39	1040					t				t		Rumänisches Kulturinstitut, Ingenieurbüro
834	Augasse	11	1090								t			Supermarkt
849	Augustinerstraße	1	1010		t					t		t		Museum, restaurant, museum geschäft
918	Bacherplatz	11	1050										t	Film und Design verleih
933	Bäckerstraße	10	1010					t			t			Schmuckgeschäft
1245	Baumgasse	83	1030	t										Hotel
1297	Bechardgasse	18	1030					t						
1372	Bennogasse	1	1080							t			t	Bank, post
1373	Bennogasse	10	1080					t					t	Arzt, edv
1440	Berggasse	32	1090					t					t	Schuhservice, Vorbeugung von sexuellem Missbrauch von Kindern
1952	Bräuhausgasse	37	1050								t		t	Ingenieurbüro, Steuerberatung, Kindergarten,

														Investmentberatung
2084	Brigittaplatz	18	1200					t			t		t	Aquarien Fachgeschäft, Elektrik fachgeschäft, psychotherapie, Heilpraktiker
2091	Brigittaplatz	9	1200					t			t			Küchenmöbel
2262	Burggasse	28-32	1070					t					t	Werbeagentur, Fotografie, film, Kommunikation
2288	Burggasse	56	1070										t	Fotografie
2312	Burggasse	81	1070		t			t			t			Hochzeitsgeschäft, Schneiderei, Bäcker, Gemüse Händler, Café
2576	Daffingerstraße	1	1030											Nichts
2677	Dannebergplatz	12	1030					t						
2816	Diehlgasse	28	1050					t						
3009	Dorotheergasse	2	1010		t			t					t	Restaurant, architekt,
3036	Dr.-Karl-Renner- Ring	3	1010						t					Parlament
3114	Dresdner Straße	68	1200										t	Baugesellschaft, Pipelines, EDV, arzt, telefon, ingenieur, Reisebüro
3131	Dresdner Straße	87	1200	t					t		t		t	MA 42, österreichische Patentamt, hotel management, trafik
3195	Ebendorferstraße	7	1010				t	t						Österreichischer Austauschdienst, wohnraumverwaltung GmbH
3227	Einsiedlergasse	19	1050	t										Appartemenzhaus
3296	Elisabethstraße	13	1010					t					t	Arzt, physiotherapie, rechtsanwalt, Fischerei Gesellschaft, Model Management, Immobilien, Kunden Forschung, marketing
3709	Erzherzog- Johann-Platz	1	1040				t							Universität
3724	Eslarngasse	11	1030					t						

3774	Esteplatz	5	1030					t					t	Rechtsanwalt, Kochkurse
3857	Fasangasse	35-37	1030		t		t	t					t	Gaststätte, arzt, reinigung, bücherei, beauty
3862	Fasangasse	4	1030				t					t		Kloster, handelsakademie
3941	Favoritenstraße	52	1040	t										Hotel
4122	Fischhof	3	1010								t		t	Kleidungsgeschäft, business center
4145	Fleischmarkt	24	1010	t	t									Hotel, restaurant
4173	Florianigasse	16	1080					t					t	Rechtsanwalt, FPÖ, Aktionsgemeinschaft unabhängiger, architekt
4181	Florianigasse	24	1080					t					t	Hilfswerk, architekt, Immobilien, ingenieur
4283	Frankenberggasse	11	1040								t			Schmuckmanufaktur
4411	Franzensgasse	4	1050					t					t	Esoterik
4448	Freyung	4	1010		t							t	t	Friseur, Beauty, restaurant, Auktionshaus, events
4451	Freyung	6a	1010				t					t		Kirche, kirche Museum
4468	Friedrich-Engels-Platz	21	1200					t			t		t	Schuhgeschäft, schuhservice, friseur, installateur
4490	Friedrichstraße	10	1010							t		t	t	Immobilien, erste Stiftung, art collection, reisebüro, Möbel geschäft, Verlag
5115	Gonzagagasse	1	1010										t	Arzt, Consulting, Logistik, rechtsanwalt, Eventmanagement, Werbeagentur, Design, wirkwaren und Strickwaren
5169	Graben	7	1010								t			Apotheke, Kleidungsgeschäft, Kosmetikgeschäft,
5395	Große Pfarrgasse	2	1020		t			t						Restaurant
5503	Große Stadtgutgasse	31	1020	t	t									Hotel, restaurant
5520	Grünangergasse	6	1010					t			t	t		Galerie, Schmuckgeschäft,
5751	Gumpendorfer	9	1060		t		t	t				t	t	Restaurant, bar,institut

	Straße													Kultur Konzepte, Gemüse groß Händler
5929	Hahngasse	2	1090				t							Universität
6594	Herrengasse	13	1010						t			t		Außenministerium, Kunstraum Niederösterreich
6605	Herrengasse	7	1010						t					Bundesinnenministerium
6731	Himmelfortgasse	4	1010				t		t					Bundeskanzleramt Finanzen Archiv, Bundesfinanzministerium Bibliothek
6808	Hofenedergasse	3	1020					t					t	Rechtsanwalt, sicherheitstechnik, city wheel
6937	Hollandstraße	15	1020		t			t			t		t	Restaurant, apotheke, Werbeagentur
6985	Hornbostelgasse	16-18	1060					t					t	Ingenieurbüro, grafiker
7267	Johann-Böhm- Platz	1	1020			t	t			t			t	Österreichischer Gewerkschaftsbund, bfi, EDV, post, bank, Verlag
7433	Josefsplatz	5	1010								t	t		Antiquitäten
7502	Josefstädter Straße	7	1080								t		t	Friseur, Consulting, Kleidungsgeschäft
7573	Judenplatz	8	1010		t		t					t		Museum, kaffee, Bibliothek, Archiv
7649	Kaiserstraße	26	1070					t						Psychotherapie, edv
7896	Karlsplatz	5	1010									t		Kunst museum
7929	Kärntner Ring	42190	1010		t						t		t	Shopping center, Rechtsanwalt, consulting, Steuerberater, Versicherungen, Film Produktion,
7947	Kärntner Straße	21-23	1010		t					t	t		t	Kaffee, Kleidungsgeschäft, Kreditkarten firma, Immobilienfirma, consulting
8214	Kirchengasse	1	1070	t							t		t	Kleidungsgeschäft, Hotel, rechtsanwalt, Consulting
8251	Kirchengasse	44	1070		t			t			t			Kaffee, möbelgeschäft

8637	Kohlmarkt	11	1010					t			t	t	t	Zahnarzt, rechtsanwalt, Atelier, Kleidungsgeschäft, apotheke, schmuckgeschäft, parfumerie
8702	Kollergasse	14	1030										t	Holzgroßhandel
9167	Lammgasse	8	1080				t	t						Universitäts Institut
9362	Landstraßer Hauptstraße	8	1030					t			t		t	Arzt, notar, möbelgeschäft, elektronikgeschäft
9397	Lange Gasse	25	1080		t			t					t	Schneiderei, restaurant
9480	Lassallestraße	7b	1020										t	Edv, Consulting, und Schinnerl buero, marketing, M and A
9514	Laudongasse	26	1080					t			t		t	Parkett geschäft, physiotherapeut, spiegelgeschäft
9564	Laudongasse	8	1080	t	t									Hotel, restaurant
9792	Leopold-Rister-Gasse	5	1050					t					t	Beauty, Consulting, Arzt
10411	Linke Wienzeile	102	1060		t							t	t	Restaurants, Beratungszentrum, Kulturzentrum
10548	Lothringerstraße	16	1030					t					t	Rechtsanwalt, consulting, Finanzberatung,,
10608	Löwengasse	37	1030								t		t	Second hand geschäft, Kleidungsgeschäft, bäcker, friseur
10988	Margaretenstraße	33	1040					t			t			Möbelgeschäft
11006	Margaretenstraße	52	1040		t			t			t		t	Architekt, Restaurant, Kleidungsgeschäft
11118	Mariahilfer Straße	111	1060								t			Kleidungsgeschäft
11123	Mariahilfer Straße	117	1060		t			t			t			Restaurant, beauty, Kleidungsgeschäft, arzt, psychotherapie, consulting
11146	Mariahilfer Straße	22	1070		t		t		t	t	t			Restaurant, Kleidungsgeschäft, Technik geschäft, Bank, militärische schule, militärisches Immobilien Management Zentrum

11160	Mariahilfer Straße	4	1070					t			t		t	Kleidungsgeschäft, Medien firma, cortical.io
11165	Mariahilfer Straße	49	1060								t			Kleidungsgeschäft, werkzeuggeschäft,, Kosmetikgeschäft, Sex geschäft
11193	Mariahilfer Straße	76	1070					t			t		t	Rechtsanwalt, Kleidungsgeschäfte, orthopädie
11540	Messeplatz	1	1020		t									Messe Wien
11658	Minoritenplatz	3	1010				t		t					Bildungsministerium
11692	Mittersteig	13	1040										t	Architekt, IT, Großküche, Sachverständiger
12125	Naglergasse	25	1010								t			Stahlgroßhandel, Badutensilien geschäft, Filmproduktion
12283	Neubaugasse	38	1070									t	t	Theater, Tanzstudio, kaffeemaschinen Vertrieb
12289	Neubaugasse	43	1070				t						t	Fortbildung, Finanzberatung, zeitarbeit, Consulting, reisen, edv, rechtsanwalt
12391	Neulinggasse	29	1030		t				t					Botschaft Elfenbeinküste, Consulting, restaurant
12424	Neustiftgasse	101	1070		t			t						Kaffee, Reisebüro
12438	Neustiftgasse	115a	1070					t					t	Verlag, PR Agentur, erneuerbare energien,Ingenieur
12472	Neustiftgasse	23	1070		t			t			t			Eisgeschäft, Kleidungsgeschäft, Tier Erziehung
12656	Nordbergstraße	6	1090		t								t	Physiotherapie, arzt, restaurant
12948	Obere Donaustraße	12	1020		t			t				t	t	Hundertwasser Geburtstags, friseur, gaststätte
13126	Opernring	2	1010		t						t	t		Opa, restaurant, Bücher und musik geschäft
13233	Paniglasse	14	1040		t									Restaurant
13247	Pannaschgasse	6	1050				t	t						Bibliothek

13495	Petersplatz	7	1010							t				Bank
13681	Plößlgasse	4	1040				t	t						Sprachreisen
13777	Postgasse	19	1010					t					t	Arzt, Garage
13833	Praterstraße	14	1020					t			t		t	Kleidungsgeschäft, Büchergeschäft, elterntreff.
13939	Prinz-Eugen- Straße	16	1040					t	t				t	Botschaft San Marino, Steuerberatung ärzte
14004	Rabengasse	3	1030					t				t		Theater, spö
14022	Radetzkyplatz	2	1030		t								t	Restaurant, Zahnarzt
14026	Radetzkystraße	1	1030				t		t					Berufsrettung
14037	Radetzkystraße	2	1030		t				t		t		t	Gesundheitsministerium, am bäcker, friseur, restaurant, kindergarten
14049	Radetzkystraße	3	1030		t			t			t			Gaststätte, bäckerei
14247	Rathausstraße	11	1010		t			t					t	Restaurant, Cafe, rechtsanwalt, Consulting, Steuerberater, edv
14526	Reisnerstraße	40	1030					t						Rechtsanwälte, consulting, sport, Inkasso, mode
14580	Rembrandtstraß e	5	1020					t						
14610	Rennweg	16	1030	t	t									Hotel, restaurant
14644	Rennweg	51	1030	t	t									Hotel, restaurant, bar
14700	Resselgasse	4	1040				t							Universität
14958	Rotenkreuzgasse	11	1020					t						
14977	Rotenlöwengass e	19	1090					t					t	Steuerberatung, consulting, medizinprodukte
15129	Rudolfsplatz	2	1010		t			t					t	Bar, sport, werbung, Immobilien,
15275	Salmgasse	8	1030					t					t	Ingenieur
15575	Schiffamtsgasse	11	1020					t				t	t	Kunstraum, beauty
15626	Schimmelgasse	42222	1030					t						
15825	Schnirchgasse	14	1030						t					Gesundheitsdienst stadt wien

16040	Schönburgstraße	42285	1040						t					Botschaft belgien
16045	Schönlaterngasse	11	1010		t			t					t	Gaststätte, bar, arzt, Immobilien, gebäudereinigung
16096	Schottenfeldgasse	42096	1070		t						t		t	Kaffee, Kleidungsgeschäft, architekt, arzt, Consulting
16189	Schottenring	16	1010										t	Business center
16190	Schottenring	17	1010						t	t	t	t	t	Versicherung, eisenwarengeschäft, münzgeschäft, bank, Honorarkonsulat von rumänien, Consulting, Finanzberatung,
16233	Schreygasse	14	1020					t						
16273	Schrotzbergstraße	6	1020					t					t	Beauty, Eventmanagement
16304	Schubertring	11	1010	t						t	t		t	Hotel, Bäcker, Arzt, Steuerberater, investment Beratung,
16462	Schwarzenbergplatz	16	1010				t		t		t		t	Rechtsanwalt, französische OSZE, französische UN, Privat Bank, Arzt, super markt
16508	Schwarzspanierstraße	13	1090		t		t	t				t		Albert-schweitzer-haus
16561	Schwindgasse	14	1040					t				t	t	Kroatischer Kulturverein, Ingenieurbüro, Eventmanagement
16606	Sechskrügelgasse	2	1030		t			t			t		t	Growshop, Kleidungsgeschäft, kaffee, sportgeschäft, Pflanzengeschäft, Beauty, apotheke
16687	Seidengasse	27	1070					t						
16709	Seidengasse	9	1070										t	Verlag, Dachverband soziale Einrichtungen, Consulting, Krankenhaus bedarf, Logistik, customer management, hotel grosshandel, EDV
16768	Seilerstätte	11	1010					t					t	Aluminium Großhandel, Verlag, arzt



16825	Sensengasse	3	1090				t					t	Ärzte Zentrum, Bibliothek
17050	Sigmundsgasse	16	1070					t				t	Immobilien, Schmuckdesign
17064	Simon-Denk- Gasse	42159	1090				t	t					Bibliothek
17094	Singerstraße	27	1010									t	Rechtsanwalt, Beauty, Consulting,
17193	Sonnenfelsgasse	19	1010				t		t				Österreichische Akademie der Wissenschaft, Institut für Informationsverarbeitung, Publikums Forschung, Seelsorgezentrum
17952	Stubenring	1	1010				t		t				Bundesministerium für Wissenschaft Forschung und Wirtschaft, Bundesministerium für Land und Forstwirtschaft
18020	Stumpergasse	65	1060					t		t			Bank
18179	Taborstraße	81	1020			t		t				t	Personalvermittlung
18304	Theobaldgasse	9	1060		t			t			t		Schmuckgeschäft, Kleidungsgeschäft, bar
18643	Türkenstraße	8	1090									t	Architekt, psychologie, Kommunikation
18959	Veithgasse	3	1030					t				t	Serbische orthodoxe Kirche
18961	Veithgasse	5	1030					t	t				Australische Botschaft
19446	Währinger Straße	68	1090		t			t			t		Restaurant, elektrofachgeschäft
19478	Walfischgasse	13	1010				t		t				Tüv österreich
19498	Wallensteinplatz	8	1200		t					t		t	Konzerte, bar, Bank, post
19610	Wallnerstraße	3	1010								t		Kleidungsgeschäft, rechtsanwalt, Consulting, Steuerberater, friseur, vermögensberatung
19696	Wassergasse	2	1030					t				t	Psychotherapie
19765	Webgasse	43	1060					t			t		Kaffeeesgeschäft, arzt, sportstudio, Psychotherapie
20126	Widerhofergasse	3	1090					t				t	Steuerberatung

<b>20139</b>	Wiedner Gürtel	16	1040					t						Geschlossenes lokal
<b>20253</b>	Wiedner Hauptstraße	46	1040					t						Rechtsanwalt, Foto Büro
<b>20448</b>	Wimmergasse	7	1050					t					t	Hausverwaltung
<b>20468</b>	Windmühlgasse	15	1060					t			t		t	Kleidungsgeschäft, blumengeschäft, psychotherapie
<b>20481</b>	Windmühlgasse	32	1060		t			t				t	t	Restaurant, bar, Ingenieur, Künstlermanagement
<b>20838</b>	Ziegelofengasse	37	1050		t			t						Restaurant, Filmmacher, Ausstellungs utensilien
<b>20987</b>	Zirkusgasse	28	1020					t					t	Autowerkstatt
<b>21021</b>	Zollergasse	13	1070										t	Sport, Design, Fotografie, neos
<b>21049</b>	Zollergasse	5	1070		t			t					t	Café, Bar, Marketing
<b>21132</b>	Lindengasse	22	1070					t			t		t	Geschäft, Kleidungsgeschäft, Finanzberatung,
<b>21193</b>	Schönbrunner Straße	47	1050					t					t	Werbeagentur, Maler
<b>21218</b>	Webgasse	6	1060					t					t	Beauty

## Lebenslauf

Name: Alexander Czech B.Sc.  
Geburtsort: Erfurt

## Bildungsweg

---

10/2011 - 10/2015	Kartographie und Geoinformation M.Sc. Universität Wien
10/2006 - 9/2011	Geographie B.Sc. Universität Hamburg

## Relevante Berufserfahrung

---

06/2012 -	Institut für Energiesysteme und Elektrische Antriebe, Technische Universität Wien, Freier GIS-Analyst / Consultant
10/2008 – 9/2011	Institut für Geographie, Universität Hamburg, Tutor
1/2011 – 8/2011	Cluster of Excellence “Integrated Climate System Analysis and Prediction, Universität Hamburg, Wissenschaftliche Hilfskraft
11/2009 – 9/2010	Institut für Biologie Universität Hamburg; Wissenschaftliche Hilfskraft

Ich versichere:

- dass ich die Masterarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.
- dass alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Publikationen entnommen sind, als solche kenntlich gemacht sind.
- dass ich dieses Masterarbeitsthema bisher weder im In- noch im Ausland (einer Beurteilerin/ einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- dass diese Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Datum

Unterschrift