"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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Folder and file structure</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>Subdivide files into individual HTML files</td>
<td>33</td>
</tr>
<tr>
<td>6.</td>
<td>HTML Tag Stripper</td>
<td>37</td>
</tr>
<tr>
<td>6.1</td>
<td>Find Vienna</td>
<td>37</td>
</tr>
<tr>
<td>6.2</td>
<td>Remove HTML Tags</td>
<td>38</td>
</tr>
<tr>
<td>7.</td>
<td>Geotagging</td>
<td>42</td>
</tr>
<tr>
<td>7.1</td>
<td>Creating an index in PostgreSQL</td>
<td>42</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Converting a text to a list of stemmed tokens</td>
<td>42</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Creating a Token Index</td>
<td>45</td>
</tr>
<tr>
<td>7.2</td>
<td>Create a unique set of Addresses and prepare them for Search Queries</td>
<td>46</td>
</tr>
<tr>
<td>7.3</td>
<td>Preparing the SQL Statement for Geotagging</td>
<td>48</td>
</tr>
<tr>
<td>7.4</td>
<td>Joining Addresses with HTML Documents</td>
<td>52</td>
</tr>
<tr>
<td>7.5</td>
<td>Discussion</td>
<td>55</td>
</tr>
<tr>
<td>8.</td>
<td>Finding Links</td>
<td>57</td>
</tr>
<tr>
<td>8.1</td>
<td>Preparation</td>
<td>57</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Link Extraction</td>
<td>58</td>
</tr>
<tr>
<td>8.2</td>
<td>Geotagging the found linked websites</td>
<td>61</td>
</tr>
<tr>
<td>8.3</td>
<td>Discussion</td>
<td>62</td>
</tr>
<tr>
<td>9.</td>
<td>The Vector Space Model</td>
<td>65</td>
</tr>
<tr>
<td>9.1</td>
<td>The Document Vector</td>
<td>65</td>
</tr>
<tr>
<td>9.2</td>
<td>Term frequency Inverse Document Frequency</td>
<td>66</td>
</tr>
<tr>
<td>10.</td>
<td>Categories for classification</td>
<td>69</td>
</tr>
<tr>
<td>10.1</td>
<td>Daseinsgrundfunktionen</td>
<td>69</td>
</tr>
<tr>
<td>10.2</td>
<td>Classes for addresses</td>
<td>70</td>
</tr>
<tr>
<td>11.</td>
<td>Co-occurrence Groups</td>
<td>74</td>
</tr>
<tr>
<td>11.1</td>
<td>Introduction Natural Language Processing</td>
<td>74</td>
</tr>
<tr>
<td>11.2</td>
<td>Part-of-speech Tagging</td>
<td>74</td>
</tr>
<tr>
<td>11.3</td>
<td>POS tagging Wikipedia</td>
<td>77</td>
</tr>
</tbody>
</table>
Table of Contents

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
## Table of Contents

- 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-occurrence Groups ....................................................... xxxvii
- Cosine Similarity ...................................................................................................................... xxxviii
- Address Classification ............................................................................................................... xl
- Database Schema ...................................................................................................................... xlii
- Mapping Results ....................................................................................................................... xliii
- Lebenslauf ................................................................................................................................. liii
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>ARC File Example (archive.org)</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Example Binary Tree (Manning et al., 2009)</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Example B-Tree (Manning et al., 2009)</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>Index Check Example DerStandard.at</td>
<td>14</td>
</tr>
<tr>
<td>3.5</td>
<td>Index Check Example .at TLD</td>
<td>14</td>
</tr>
<tr>
<td>3.6</td>
<td>Code Example URL Stump Creation</td>
<td>15</td>
</tr>
<tr>
<td>3.7</td>
<td>Threading Code</td>
<td>16</td>
</tr>
<tr>
<td>3.8</td>
<td>Starting and Controlling Threads</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>OSM Data Example (OpenStreetMap Wiki; OSM XML)</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Example Address Tag (OpenStreetMap.org)</td>
<td>21</td>
</tr>
<tr>
<td>4.3</td>
<td>OpenStreetMap XML parser start element part one</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>OpenStreetMap XML parser start element part two</td>
<td>23</td>
</tr>
<tr>
<td>4.5</td>
<td>OpenStreetMap XML parser start element part three</td>
<td>24</td>
</tr>
<tr>
<td>4.6</td>
<td>OpenStreetMap XML parser end element part one</td>
<td>24</td>
</tr>
<tr>
<td>4.7</td>
<td>OpenStreetMap XML parser end element part two</td>
<td>25</td>
</tr>
<tr>
<td>4.8</td>
<td>OpenStreetMap XML parser end element part three</td>
<td>26</td>
</tr>
<tr>
<td>4.9</td>
<td>Database Import and Calling Database Import from the Parser</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Schematic Example File Structure</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>Database Transfer Script part one</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>Database Transfer Script open_paths() method</td>
<td>33</td>
</tr>
<tr>
<td>5.4</td>
<td>Transfer Script database_export() method part one</td>
<td>33</td>
</tr>
<tr>
<td>5.5</td>
<td>Transfer Script database_export() method part two</td>
<td>34</td>
</tr>
<tr>
<td>5.6</td>
<td>Transfer Script database_export() method part three</td>
<td>35</td>
</tr>
<tr>
<td>5.7</td>
<td>Transfer Script database_export() method part four</td>
<td>35</td>
</tr>
<tr>
<td>5.8</td>
<td>WriteManyToTable() method</td>
<td>36</td>
</tr>
<tr>
<td>6.1</td>
<td>Set Vienna</td>
<td>37</td>
</tr>
<tr>
<td>6.2</td>
<td>Fetching HTML Documents to Strip Tags</td>
<td>39</td>
</tr>
<tr>
<td>6.3</td>
<td>Regular Expression Tag Stripper</td>
<td>40</td>
</tr>
<tr>
<td>6.4</td>
<td>Writing Stripped HTML Documents to the Database</td>
<td>41</td>
</tr>
<tr>
<td>7.1</td>
<td>Index Creation</td>
<td>46</td>
</tr>
<tr>
<td>7.2</td>
<td>Populate Table AddressesUnique</td>
<td>47</td>
</tr>
<tr>
<td>7.3</td>
<td>Preparing Addresses for Search Queries</td>
<td>47</td>
</tr>
<tr>
<td>7.4</td>
<td>SQL Statement Construction part one</td>
<td>49</td>
</tr>
<tr>
<td>7.5</td>
<td>SQL Statement Construction part two</td>
<td>51</td>
</tr>
<tr>
<td>7.6</td>
<td>Perform the Join of Addresses with HTML Documents</td>
<td>52</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.1</td>
<td>Creating the URL dictionary and the HTML joined to addresses dictionary</td>
<td>57</td>
</tr>
<tr>
<td>8.2</td>
<td>Finding links and converting them</td>
<td>59</td>
</tr>
<tr>
<td>8.3</td>
<td>Malformed link examples</td>
<td>60</td>
</tr>
<tr>
<td>8.4</td>
<td>Geotag linked websites</td>
<td>61</td>
</tr>
<tr>
<td>9.1</td>
<td>Cosine similarity example (Manning et al.; 2009;)</td>
<td>66</td>
</tr>
<tr>
<td>9.2</td>
<td>Stop word list of 25 words that are common in the Reuters Corpus Volume 1 (Manning et al.; 2009)</td>
<td>67</td>
</tr>
<tr>
<td>11.1</td>
<td>Creating the file list</td>
<td>78</td>
</tr>
<tr>
<td>11.2</td>
<td>Executing the POS tagger</td>
<td>79</td>
</tr>
<tr>
<td>11.3</td>
<td>Multidimensional scaling of co-occurrence vectors. (Source: Lund and Burgess 1996)</td>
<td>83</td>
</tr>
<tr>
<td>11.4</td>
<td>Co-occurrence generation from Wikipedia</td>
<td>84</td>
</tr>
<tr>
<td>12.1</td>
<td>HTML Document vector space code</td>
<td>89</td>
</tr>
<tr>
<td>12.2</td>
<td>Wikipedia vector space code</td>
<td>91</td>
</tr>
<tr>
<td>12.3</td>
<td>Combine Wikipedia and HTML file vector space</td>
<td>92</td>
</tr>
<tr>
<td>12.4</td>
<td>Code for inverse document frequency calculation</td>
<td>93</td>
</tr>
<tr>
<td>12.5</td>
<td>HTML Document tokenization</td>
<td>95</td>
</tr>
<tr>
<td>12.6</td>
<td>TF-IDF vector for documents</td>
<td>96</td>
</tr>
<tr>
<td>12.7</td>
<td>Normalized TF-IDF vector for co-occurrence groups and search terms</td>
<td>98</td>
</tr>
<tr>
<td>12.8</td>
<td>Cosine similarity calculation</td>
<td>102</td>
</tr>
<tr>
<td>12.9</td>
<td>Breaks code example</td>
<td>105</td>
</tr>
<tr>
<td>12.10</td>
<td>Value distribution for all classes at the addresses for co-occurrence groups and search terms (N= 6284)</td>
<td>109</td>
</tr>
<tr>
<td>13.1</td>
<td>Comparing control group to vector classification</td>
<td>126</td>
</tr>
<tr>
<td>13.2</td>
<td>Correlation between original count, complete count and vector classification</td>
<td>127</td>
</tr>
<tr>
<td>14.1</td>
<td>Correctly Classified Hotel addresses compared to classification value range</td>
<td>131</td>
</tr>
</tbody>
</table>
Table of Maps

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

**Table 3.1** A), B) Representativeness of TLDS in the Common Crawl Corpus 2012 (Spielger; 2013) ............................ 11

**Table 7.1** Website match Frequency per Address ........................................................................................................ 55

**Table 7.2** Top Ten matched Addresses .......................................................................................................................... 55

**Table 8.1** urlparse.urljoin() Examples .......................................................................................................................... 60

**Table 8.2** Associated Website Frequencies per Address .................................................................................................. 64

**Table 8.3** Associated Website Frequencies per Address .................................................................................................. 64

**Table 9.1** Collection frequency (cf) and document frequency (df) different behavior (Manning et al.; 2009).............. 67

**Table 9.2** Examples for idf Values based on the Reuters Collection containing 806,791 documents (Manning et al.; 2009) ........................................................................................................................................ 68

**Table 10.1** Classes and their corresponding queries ....................................................................................................... 73

**Table 11.1** The Penn Treebank II POS tag set (Santorini 1990) .............................................................................. 75

**Table 11.2** Five nearest neighbors for target words (Source: Lund and Burgess 1996) ............................................. 82

**Table 11.3** Co-occurrence groups with top 5 terms and the number of their occurrences ........................................ 86

**Table 12.1** Tokenized co-occurrence groups with top 5 terms and the number of their occurrences ..................... 99

**Table 12.2** Similarity matrix co-occurrence groups ...................................................................................................... 100

**Table 12.3** Similarity matrix search terms ..................................................................................................................... 101

**Table 12.4** Classification problem number one ........................................................................................................... 104

**Table 13.1** Search term vector co-occurrence group vector fitness comparison ....................................................... 128
List of Abbreviations

CF - Corpus Frequency
DF - Document Frequency
HTML - HyperText Markup Language
IDF - Inverse Document Frequency
IDF - TF - Inverse Document Frequency - Term Frequency
NLP - Natural Language Processing
NLTK - Natural Language Tool Kit
OSM - OpenStreetMap
POS - Part of Speech
SLD - Second Level Domain
SQL - Structured Query Language
TF - Term Frequency
TLD - Top Level Domain
URL - Uniform Resource Locator
UTF-8 - (Universal Coded Character Set + Transformation Format—8-bit)
VGI - volunteer geographic information
XML - 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 1st to 9th and 20th 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

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
tzu dem Top-level Domain-Bereich von .at gehören und die Adressen sind beschränkt auf den 1. bis 9.

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-Modell 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 used be 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.
Introduction

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 1st to 9th and 20th 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 shortcomings 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 Common Crawl Dataset

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).
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 ATLASSIAN), (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 p 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).

<table>
<thead>
<tr>
<th>TLD</th>
<th>Rel. freq. W3 survey</th>
<th>Rel. freq. CC corpus</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>.gov</td>
<td>0.001</td>
<td>0.002</td>
<td>2.0</td>
</tr>
<tr>
<td>.nz</td>
<td>0.001</td>
<td>0.000</td>
<td>1.0</td>
</tr>
<tr>
<td>.edu</td>
<td>0.003</td>
<td>0.004</td>
<td>1.2</td>
</tr>
<tr>
<td>.uk</td>
<td>0.010</td>
<td>0.010</td>
<td>1.0</td>
</tr>
<tr>
<td>.nl</td>
<td>0.004</td>
<td>0.003</td>
<td>1.2</td>
</tr>
<tr>
<td>.se</td>
<td>0.005</td>
<td>0.003</td>
<td>1.2</td>
</tr>
<tr>
<td>.ca</td>
<td>0.004</td>
<td>0.006</td>
<td>1.5</td>
</tr>
<tr>
<td>.ch</td>
<td>0.003</td>
<td>0.002</td>
<td>1.5</td>
</tr>
<tr>
<td>.cz</td>
<td>0.005</td>
<td>0.007</td>
<td>1.5</td>
</tr>
<tr>
<td>.org</td>
<td>0.021</td>
<td>0.095</td>
<td>4.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TLD</th>
<th>Rel. freq. W3 survey</th>
<th>Rel. freq. CC corpus</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>.in</td>
<td>0.000</td>
<td>0.000</td>
<td>1.0</td>
</tr>
<tr>
<td>.tk</td>
<td>0.001</td>
<td>0.000</td>
<td>1.0</td>
</tr>
<tr>
<td>.th</td>
<td>0.001</td>
<td>0.001</td>
<td>1.0</td>
</tr>
<tr>
<td>.kz</td>
<td>0.001</td>
<td>0.001</td>
<td>1.0</td>
</tr>
<tr>
<td>.co</td>
<td>0.003</td>
<td>0.000</td>
<td>1.0</td>
</tr>
<tr>
<td>.az</td>
<td>0.001</td>
<td>0.001</td>
<td>1.0</td>
</tr>
<tr>
<td>.asia</td>
<td>0.001</td>
<td>0.001</td>
<td>1.0</td>
</tr>
<tr>
<td>.pk</td>
<td>0.001</td>
<td>0.001</td>
<td>1.0</td>
</tr>
<tr>
<td>.ve</td>
<td>0.001</td>
<td>0.001</td>
<td>1.0</td>
</tr>
<tr>
<td>.ir</td>
<td>0.006</td>
<td>0.005</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 3.1 a), b) Representativeness of TLDs in the Common Crawl Corpus 2012 (SPIEGLER; 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), (W3TECHS).

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).

![Example B-Tree](image)

**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 inverted 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).

![Figure 3.4 Index check example derstandard.at](image)

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.

![Figure 3.5 Index check example .at TLD](image)

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.

```python
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     completlist = []
041     for first in list1:
042         for second in list2:
043             for third in list3:
044                 completlist.append('at.' + first + second + third)
045     return completlist
```

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.
```python
class myThread (threading.Thread):
    def __init__ (self, urlstump, threadid):
        threading.Thread.__init__(self)
        self.urlstump = urlstump
        self.id = threadid
    def run(self):
        threadLimiter.acquire()
        print('checking for ' + str(self.urlstump))
        current_urllist.append(self.urlstump)
        remote_copy_external.external('AWS-PUBLIC-KEY', 'AWS-PRIVATE-KEY','tldat','Data2//' + str(self.urlstump),
        self.urlstump,parallelconnections,True)
        print("Exit Thread: %d of %d" % (self.id, NummerIDs)
        urllist.remove(self.urlstump)
        current_urllist.remove(self.urlstump)
        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).
while running:
    print ''
    print 'running threads %s' % (len(threading.enumerate()))
    urllist_pickle = list(urllist)
ipickle = iterate_ipickle(ipickle)
picklelist(urllist_pickle, 'obj_%s.pickle' % ipickle)

if duds + threadnumber > len(threading.enumerate()):
    if threadlist:
        element = threadlist[0]
        threadlist.remove(element)
        myThread(element, ID).start()
        ID += 1
        continue
    if not threadlist:
        pass

elif threadlist:
ipickle = iterate_ipickle(ipickle)
picklelist(urllist_pickle, 'obj_%s.pickle' % ipickle)
time.sleep(1)
print current_urllist
continue

elif not threadlist:
ipickle = iterate_ipickle(ipickle)
picklelist(urllist_pickle, 'obj_%s.pickle' % ipickle)
pass

while duds < len(threading.enumerate()):
time.sleep(10)
    print ''
    print 'Current Passnumber: %d' % (Passnumber)
    print current_urllist
    urllist_pickle = list(urllist)
ipickle = iterate_ipickle(ipickle)
picklelist(urllist_pickle, 'obj_%s.pickle' % ipickle)
    pass

if not urllist:
    running = False
    print ''
    print 'Script did run Passnumber %d' % (Passnumber)
    pass
else:
    threadlist = list(urllist)
    urllist_pickle = list(urllist)
ipickle = iterate_ipickle(ipickle)
picklelist(urllist_pickle, 'obj_%s.pickle' % ipickle)
    print ''
    print 'Script did run Passnumber %d' % (Passnumber)
    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 overview of 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 (NEILS 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 decompress 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).
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

Figure 4.1 OSM Data Example (OPENSTREETMAP WIKI: OSM XML)
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 itself. 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).

```python
class startendfinder(handler.ContentHandler):
    def __init__(self):
        self.address = ['lat', 'lon', 'pcode', 'street', 'number']
        self.nodedict = {}
        self.waydict = {}
        self.relationdict = {}
        self.plz = False
        self.street = False
        self.number = False
        self.nodemode = False
        self.waymode = False
        self.relationmode = False
        self.wayid = False
        self.relationid = False
        self.ndlist = []
        self.memberlist = []
        self.latlist = []
        self.lonlist = []

    def startElement(self, name, attrs):
        if name in ('node',):
            self.address[0] = 'POINT(%s %s)' % (attrs.get('lon'), attrs.get('lat'))
            self.nodedict[int(attrs.get('id'))] = (float(attrs.get('lat')), float(attrs.get('lon')))
            self.nodemode = True

        elif name in ('way',):
            self.wayid = int(attrs.get('id'))
            self.waymode = True

        elif name in ('relation',):
            self.relationid = int(attrs.get('id'))
            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 a 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).

```
if self.nodemode == True:
    if name == 'tag':
        k, v = (attrs.get('k'), attrs.get('v'))
    if k == 'addr:street':
        self.address[1] = unicode(v)
        self.street = True
    if k == 'addr:housenumber':
        self.address[2] = unicode(v)
        self.number = True
    if k == 'addr:postcode':
        try:
            if int(v) <= 1099:
                self.address[3] = int(v)
                self.plz = True
            elif int(v) >= 1200 and int(v) <= 1209:
                self.address[3] = int(v)
                self.plz = True
        except:
            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 1st to 9th, or 20th 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).
if self.relationmode == True:
    if name == 'member':
        self.memberlist.append(((int(attrs.get('ref'))),attrs.get('type')))  
    if name == 'tag':
        k, v = (attrs.get('k'), attrs.get('v'))
        ...
if self.waymode == True:
    if name == 'nd':
        self.ndlist.append(int(attrs.get('ref')))
    if name == 'tag':
        k, v = (attrs.get('k'), attrs.get('v'))
        ...

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).

def endElement(self, name):
    if name in ('node'):
        if self.plz is True and self.street is True and self.number is True:
            addresslist.append(tuple(self.address))
            self.nodemode = False
            self.plz = False
            self.street = False
            self.number = False
            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.

```python
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             self.waydict[self.wayid] = (numpy.mean(self.latlist)),
136                 numpy.mean(self.lonlist))
137             if self.plz is True and self.street is True and self.number is True:
138                 self.address[0] = 'POINT(%s %s)' % (numpy.mean(self.lonlist),
139                     numpy.mean(self.latlist))
140                 addresslist.append(tuple(self.address))
141         self.latlist = []
142         self.lonlist = []
143         self.waymode = False
144         self.plz = False
145         self.street = False
146         self.number = False
147         self.address = ['lat_lon', 'pcode', 'street', 'number']
148         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
If `name` is `'relation'`, for each member in `self.memberlist`:

```python
if member[1] == 'node':
    self.latlist.append(self.nodedict[member[0]][0])
    self.lonlist.append(self.nodedict[member[0]][1])
elif member[1] == 'way':
    self.latlist.append(self.waydict[member[0]][0])
    self.lonlist.append(self.waydict[member[0]][1])
elif member[1] == 'relation':
    self.latlist.append(self.relationdict[member[0]][0])
    self.lonlist.append(self.relationdict[member[0]][1])
self.relationdict[self.relationid] = (numpy.mean(self.latlist),
                                       numpy.mean(self.lonlist))
```

```python
if self.plz is True and self.street is True and self.number is True:
    self.address[0] = 'POINT(%s %s)' % (numpy.mean(self.lonlist),
                                         numpy.mean(self.latlist))
    addresslist.append(tuple(self.address))
```

```python
self.latlist = []
self.lonlist = []
self.waymode = False
self.plz = False
self.street = False
self.number = False
self.address = ['lat_lon', 'pcode', 'street', 'number']
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:

```python
if __name__ == '__main__':
    parser = make_parser()
    parser.setContentHandler(startendfinder())
    parser.parse('./vienna.osm')

DBconnector.CreateTable()
DBconnector.WriteToTableMany(addresslist)
```

DBconnector file:

```python
import psycopg2
def DBConnect():
    conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres\password=########")
    cur = conn.cursor()
    return conn, cur
def CreateTable():
    conn, cur = DBConnect()
    cur.execute("CREATE TABLE IF NOT EXISTS Addresses (id serial PRIMARY KEY, \geom geometry, street text, Street_number text, pcode integer, AddDate integer);")
    conn.commit()
    conn.close()
def WriteToTableMany(Values):
    conn, cur = DBConnect()
    cur.executemany("INSERT INTO Addresses (geom, street, Street_number, pcode, \AddDate) VALUES (%s, %s, %s, %s, 24022015)", (Values))
    conn.commit()
    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 \texttt{DBConnect()} method that creates a database connection object \texttt{conn} in line 6 and derives out of this connection object a cursor object \texttt{cur} in line 7. Both the cursor object and the connection object are returned to the \texttt{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 \texttt{conn.commit()} in line 15 and the connection closed in line 16 (PSYCOPG 2.5.3 LIBARY).

\texttt{WriteToTableMany()} is then called in line 184 and the \texttt{addresslist} is passed to it. Again, a connection and cursor object are created, but this time the \texttt{cur.executemany()} function is used. This function allows an iterateable python object to be passed to the psycopg2 library. The preceding SQL-like string, with \texttt{\_\_} as placeholders, works as a blue print for every value in the iterateable python object. After all items in the \texttt{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 LIBARY).
From Openstreetmap 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 20th 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 12th district, one in the 13th (this one cannot be seen on the map) and one in the 17th. 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 Ordinance Survey meridian 2 dataset to OSM data. It shows that relatively wealthy and densely populate 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 .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).

```python
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     CreateTable()
144     startall = time.time()
145     PATH = './Data'
146     open_Paths(PATH)
147     print('########################################################################')
148     print('complete Operation took %s Minutes' % ((time.time() - startall) / 60))
149     print('########################################################################')
150     lines = ReadFromTable()
151     print(len(lines))
152     for line in lines:
153         print(line[2])
154     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).
def open_Paths(PATH):
    for path, dirs, files in os.walk(PATH):
        for filename in files:
            try:
                fullpath = os.path.join(path, filename)
                print('####################')
                print('%s Size: %.2f MB' % (fullpath, os.path.getsize(fullpath) / 1048576))
                file = gzip.open(fullpath, 'rb')
                Database_export(file)
                file.close()
            except:
                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.

def Database_export(file):
    start1 = time.time()
    i = ''
    tup_i = ()
    url = ''
    html_written = 0
    mode = False
    for line in file.readlines():
        spliline = line.split(' ')
        try:
            if spliline[0][:7] == 'http://' and spliline[3] == 'text/html'
                and len(spliline) == 5:
                if len(i) > 0:
                    tup_i = tup_i + ((url, i),)
                    url = ''
                    i = ''
                url = spliline[0]
                mode = True
                if len(tup_i) >= 100:
                    print('empty tup_i')
                    try:
                        WriteManyToTable(tup_i)
                    except:
                        pass
                    tup_i = ()
                    print("html files written to DB %s" % html_written)
                    html_written += 1
            except:
                pass
        except:
            pass

Figure 5.4 Transfer Script Database_export() method part one
Lines 42 to 47 define several controlling variables. Most important are $i$ and $\text{tup}_i$. The variable $i$ will contain all lines of the current HTML document, while $\text{tup}_i$ 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 $\text{splitline}$ 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 $\text{tup}_i$ 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 $\text{tup}_i$ in line 54 and $\text{url}$ and $i$ reset (PYTHON 2.7.10 LIBRARY; string).

The newly current URL is saved to $\text{url}$ and $\text{mode}$ is set to `True`. What follows in line 61 is a test if the amount of already parsed HTML documents contained in $\text{tup}_i$ 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 $\text{tup}_i$ is set to an empty tuple.

```python
if mode == True:
    try:
        i += unicode(line, "utf-8")
    except:
        #i += UnicodeDammit(line).unicode_markup
        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 $\text{mode}$ 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 (Universal Coded Character Set + Transformation Format—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 LIBARY;).

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
stretches 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).

```
elif splitline[0][:7] == 'http://' and 
    splitline[3] != 'text/html' and len(splitline) == 5:
    if len(i) > 0:
        tup_i = tup_i + ((url, i),)
        url = ''
        i = ''
    mode = False

    if len(tup_i) >= 100:
        try:
            WriteManyToTable(tup_i)
        except:
            pass
        tup_i = ()
```

**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).

```
if len(i) > 0:
    tup_i = tup_i + ((url, i),)
try:
    WriteManyToTable(tup_i)
except:
    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 \( \text{url} \) are appended to \( \text{tup}_i \) and \( \text{tup}_i \) 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.

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

```python
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 def setvienna(lower, upper):
017     conn, cur = DBConnect()
018     cur.execute("UPDATE html SET Vienna = TRUE WHERE html_file LIKE \\
019                 '%%' || ' %s ' || '%%' AND ID >= %s AND ID <\$s RETURNING ID;" % \\
020                 ('Wien', lower, upper))
021     conn.commit()
022     conn.close()
023     return
024
025 addvienna()
026 lower = 0
027 increment = 10000
028 starttime = time.time()
029 while lower <= 8406507:
030     setvienna(lower, lower+increment)
031     delta_time = time.time() - starttime
032     print lower+increment
033     print "time till now %.2f Minutes" % (delta_time / 60)
034     print "time till end %.2f Minutes" % \\
035         (((delta_time/60)/(lower+increment))*(8406507-(lower+increment)))
036     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).
def ReadFromHTML(offset):
    conn, cur = DBConnect()
    cur.execute("SELECT id,html_file FROM html WHERE vienna = TRUE ORDER BY id
    limit 1000 offset %s ;" % offset)
    data = cur.fetchall()
    cur.close()
    conn.close()
    return data

def createColumn():
    conn, cur = DBConnect()
    cur.execute("ALTER TABLE html DROP COLUMN IF EXISTS stripped_html;")
    conn.commit()
    cur.execute("ALTER TABLE html ADD COLUMN stripped_html TEXT;")
    conn.commit()
    cur.close()
    conn.close()

createColumn()
Starttime = time.time()
strippedlist = []
offset = 0
Starttime2 = time.time()
htmls = ReadFromHTML(offset)
NumberOfrows = 8406507
while htmls:
    timeregex = time.time()
    print("starting Regex")
    for row in htmls:
        id = row[0]
        stripped html = remove_tags(row[1])
        strippedlist.append((stripped_html, id))
    print('Regex took %.2f Minutes' % ((time.time() - timeregex) / 60))
    offset += 1000
    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).
def remove_tags(text):
    text = TAG_RE.sub('', text)
    text = Short.sub('!', text)
    text = eszt.sub('ß', text)
    text = ae.sub('ä', text)
    text = AE.sub('Ä', text)
    text = oe.sub('ö', text)
    text = OE.sub('Ö', text)
    text = UE.sub('Ü', text)
    return text

TAG_RE = re.compile(r'<[^>]+>')</n
Short = re.compile(r'^\S{68,}$')

eszt = re.compile(r'^\&szlig;$')

ae = re.compile(r'^\&auml;$')

AE = re.compile(r'^\&Auml;$')

oe = re.compile(r'^\&ouml;$')

OE = re.compile(r'^\&Ouml;$')

ue = re.compile(r'^\&uuml;$')

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 `^` 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).

```python
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 mashing 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 know 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 PostgresSQL 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).
def CreateIndex():
    conn, cur = DBConnect()
    cur.execute("ALTER TABLE html ADD COLUMN textsearchable_index_col tsvector;")
    conn.commit()
    cur.execute("UPDATE html SET textsearchable_index_col = to_tsvector('german', stripped_html) WHERE Vienna = True;")
    conn.commit()
    cur.execute("CREATE INDEX textsearch_idx ON html USING gin(textsearchable_index_col);")
    conn.commit()
    cur.close()
    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.
def MakeAddressesUnique():
    conn, cur = DBConnect()
    conn.commit()
    cur.execute("INSERT INTO AddressesUnique(geom,street,street_number,pcode,AddDate)
                  SELECT DISTINCT ON (street,street_number)
                      geom,street,street_number,pcode,AddDate FROM Addresses")
    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).

def CleanStrings(lines):
    for row in lines:
        StreetName = row[0]
        StreetNumber = row[1]
        ID = row[2]
        p = re.compile(r' ')  # Matches spaces
        q = re.compile(r'[^a-zA-Z0-9_]')  # Matches non-alphanumeric characters
        r = re.compile(r'^[0-9a-zA-Z][-/][0-9a-zA-Z]$')  # Matches patterns like 8 - 9'
        s = re.compile(r''')

        if r.match(StreetNumber):
            StreetNumber = p.sub(' ', StreetNumber)
            StreetNumber = q.sub('', StreetNumber)
            StreetName = s.sub('""', StreetName)
            row3 = p.sub(' & ', StreetName)
            row4 = p.sub(' & ', StreetNumber)
            lines.remove(row)
            lines.insert(0,(StreetName,StreetNumber,ID,row3,row4))

    return lines

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 “Kärntner 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 `ConstructSQLStatementSearchAddresses()` partly shown in Figure 7.4.
def ConstructSQLStatementSearchAddresses(Values):

    SQLStatmentdict = {}
    conn, cur = DBConnect()

    for line in Values:

        if line[0][-6:] == 'straße':
            SQLStatmentdict[line[2]] = cur.mogrify(
            "Select ID FROM HTML WHERE "
            "Vienna = TRUE AND "
            "stripped_html ILIKE '%"+line[0]+" ')+line[1]+" %')"
            "OR"
            "stripped_html ILIKE '%"+line[0]+" :"+line[1]+" %')"
            "OR"
            "stripped_html ILIKE '%"+line[0]+" :"+line[1]+" %')"
            "OR"
            "stripped_html ILIKE '%"+line[0]+" :"+line[1]+" %')"
            ";"

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 ConstructSQLStatementSearchAddresses() 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).
elif line[0][-4:] == 'asse':
    SQLStatmentdict[line[2]] = cur.mogrify(
        "Select ID FROM HTML WHERE "
        "Vienna = TRUE AND "
        "(textsearchable_index_col @@ to_tsquery('german',
        "stripped_html ILIKE '%line[0]'+ 'line[1]+%')"  
        "OR"
        "(textsearchable_index_col @@ to_tsquery('german',
        "stripped_html ILIKE '%line[0]'+ 'line[1]+%')"
        "OR"
        "(textsearchable_index_col @@ to_tsquery('german',
        "stripped_html ILIKE '%line[0]+[:-4]'+ 'line[1]+%')"
        "OR"
        "(textsearchable_index_col @@ to_tsquery('german',
        "stripped_html ILIKE '%line[0]+[:-4]'+ 'line[1]+%/')"
        ";")

else:
    SQLStatmentdict[line[2]] = cur.mogrify(
        "Select ID FROM HTML WHERE "
        "Vienna = TRUE AND "
        "textsearchable_index_col @@ to_tsquery('german',
        "stripped_html ILIKE '%line[0]+[:-4]'+ 'line[1]+%')"
        "OR"
        "textsearchable_index_col @@ to_tsquery('german',
        "stripped_html ILIKE '%line[0]+[:-4]'+ 'line[1]+%/')"
        "OR"
        "textsearchable_index_col @@ to_tsquery('german',
        "stripped_html ILIKE '%line[0]+[:-4]'+ 'line[1]+%/')"
        ";")

    return SQLStatmentdict

Figure 7.5 SQL Statement Construction part two

The remaining part of the ConstructSQLStatementSearchAddresses() 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:

```sql
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.

```python
def JoinAddressesUniqueWithURL(SQLStatmentdict):
    conn, cur = DBConnect()
    i = 1
    Starttime = time.time()
    Starttime2 = time.time()
    Numberofrows = len(SQLStatmentdict)
    for ID in SQLStatmentdict:
        cur.execute(SQLStatmentdict[ID])
        values = cur.fetchall()
        if values:
            args_str = ','.join(cur.mogrify("(%s,%s,TRUE)", (ID,x[0])) for x in values)
            cur.execute("INSERT INTO AddressesUniqueJoinedWithURL (AddressesUniqueID, HTMLID, Original) VALUES " + args_str)
            conn.commit()
        print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
        delta_time = time.time() - Starttime
        print "time till now %.2f Minutes"%(delta_time/60)
        print "time till end %.2f Minutes"%(((delta_time/60)/i)*(Numberofrows-i))
        i += 1
    Starttime2 = time.time()
    conn.close()
    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 `SQLStatmentdict` 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 `SQLStatementdict` 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:

\[(1,3,\text{TRUE}),(1,25,\text{TRUE}),(1,43,\text{TRUE}),(1,199,\text{TRUE})\]

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,\text{TRUE}),(1,25,\text{TRUE}),(1,43,\text{TRUE}),(1,199,\text{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 where around unique 700k 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.
Map 7.1 Distribution of addresses joined to HTML documents
Table 7.1 Website match Frequency per Address

<table>
<thead>
<tr>
<th>Matches per Address</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14962</td>
</tr>
<tr>
<td>1-10</td>
<td>5541</td>
</tr>
<tr>
<td>11-20</td>
<td>360</td>
</tr>
<tr>
<td>21-30</td>
<td>110</td>
</tr>
<tr>
<td>31-40</td>
<td>59</td>
</tr>
<tr>
<td>41-50</td>
<td>65</td>
</tr>
<tr>
<td>51-60</td>
<td>34</td>
</tr>
<tr>
<td>61-70</td>
<td>25</td>
</tr>
<tr>
<td>71-80</td>
<td>10</td>
</tr>
<tr>
<td>81-90</td>
<td>9</td>
</tr>
<tr>
<td>91-100</td>
<td>5</td>
</tr>
<tr>
<td>&gt;100</td>
<td>66</td>
</tr>
</tbody>
</table>

Table 7.2 Top Ten matched Addresses

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

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.

```python
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     cur.execute("SELECT HTMLID, AddressesUniqueID FROM AddressesUniqueJoinedWithURL WHERE Original = TRUE")
015     data = cur.fetchall()
016     datadict = {}
017     for row in data:
018         if row[0] in datadict:
019             datadict[row[0]].append(row[1])
020         else:
021             datadict[row[0]] = [row[1],]
022     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
Finding Links

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 LIBARY).

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 LIBARY).

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 `URLIDWITHAddressIDDictonary` in line 86 (PSYCOPG 2.5.3 LIBARY).

**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.
def FindLinksInHtml(conn, cur, offset):
    NoWhiteSpace = re.compile(r' ')
    loadingtime = time.time()
    cur.execute("SELECT id,url,html_file FROM html WHERE id > %s AND id <= %s ORDER BY id;", (offset, offset+limit))
    data = cur.fetchall()
    regextime = time.time()
    passeslist = []
    linklist = []
    for row in data:
        links = re.findall(r'href="[^"]?(^[^" >]+)', row[2])
        for link in links:
            try:
                linklist.append((row[0], NoWhiteSpace.sub('%20', urlparse.urljoin(row[1], link))))
            except:
                passeslist.append((row[1], link))
    print('Regex took %.2f Minutes found links %s' % ((time.time() - regextime) / 60, len(linklist)))
    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
Finding Links

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).

<table>
<thead>
<tr>
<th>Link</th>
<th>URL</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>/hello/world.htm</td>
<td><a href="http://www.w3.org/test/">http://www.w3.org/test/</a></td>
<td><a href="http://www.w3.org/test/hello/world.htm">http://www.w3.org/test/hello/world.htm</a></td>
</tr>
<tr>
<td><a href="http://www.w3.org/test/hello/world.htm">http://www.w3.org/test/hello/world.htm</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/test/">http://www.w3.org/test/</a></td>
<td>../hello/world.htm</td>
<td><a href="http://www.w3.org/hello/world.htm">http://www.w3.org/hello/world.htm</a></td>
</tr>
<tr>
<td>../../test/</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 passeslist += passes
100 URLIDstoLinkIDS = []
101 for row in linklist:
102  if row[1] in URLDictionary:
103   URLIDstoLinkIDS.append((row[0], URLDictionary[row[1]]))
104 Newlist = []
105 for row in URLIDstoLinkIDS:
106  if row[1] in URLIDWITHAddressIDDictionary:
107    for rowx in URLIDWITHAddressIDDictionary[row[1]]:
108      Newlist.append((row[0], rowx))
109 r += len(Newlist)
110 WritetoAddressesUniqueJoinedWithURL(Newlist)
```

```
def WritetoAddressesUniqueJoinedWithURL(List):
  conn, cur = DBConnect()
  args_str = ','.join(cur.mogrify("(%s,%s,FALSE)", x) for x in List)
  try:
    cur.execute("INSERT INTO AddressesUniqueJoinedWithURL (HTMLID,
                     AddressesUniqueID, Original)VALUES " + args_str)
    conn.commit()
  except:
    print "Error Inserting Joins"
    cur.close()
    conn.close()
  return
```

Figure 8.4 Geotag linked websites

Essential for doing this are the two dictionaries `URLDictionary` and `URLIDWITHAddressIDDictionary` whose creations are described in subchapter 8.1., and `linklist`, the result of the previous subchapter. `URLDictionary` 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 `URLDictionary`. There is a possibility that a link URL can’t be found in `URLDictionary`, because links
could also target none .at websites. If the ID is found, another lookup is done in the
\texttt{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 \texttt{linklist} in line 102 and every found link
URL is tested to see if it is contained in the \texttt{URLDictionary} in line 103. If the link URL is contained in
\texttt{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 \texttt{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
\texttt{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 \texttt{Newlist}, thus creating the desired join in line 110.

This \texttt{Newlist} containing these new joins is then handed over to the
\texttt{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
\texttt{cur.mogrify()} method in line 55. The value in the field original is set to \texttt{FALSE}. This makes it
possible to discriminate between direct and indirect joins of websites to addresses (\texttt{PSYCOPG 2.5.3}
\texttt{LIBARY}).

### 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 \textbf{Map 8.1}. 
Map 8.1 Result of joining linked HTML documents to addresses
When interpreting Map 8.1, the previous observations still seem to hold true. The 1st Viennese District is still the one with the strongest Internet presence. The 1st is followed by the 7th and 8th 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)

<table>
<thead>
<tr>
<th>Websites per Address</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14962</td>
</tr>
<tr>
<td>1-10</td>
<td>2264</td>
</tr>
<tr>
<td>11-20</td>
<td>820</td>
</tr>
<tr>
<td>21-30</td>
<td>427</td>
</tr>
<tr>
<td>31-40</td>
<td>294</td>
</tr>
<tr>
<td>41-50</td>
<td>243</td>
</tr>
<tr>
<td>51-60</td>
<td>189</td>
</tr>
<tr>
<td>61-70</td>
<td>142</td>
</tr>
<tr>
<td>71-80</td>
<td>114</td>
</tr>
<tr>
<td>81-90</td>
<td>83</td>
</tr>
<tr>
<td>91-100</td>
<td>77</td>
</tr>
<tr>
<td>&gt;100</td>
<td>1631</td>
</tr>
</tbody>
</table>

Table 8.2 Associated Website Frequencies per Address

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

Table 8.3 Associated Website Frequencies per Address
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 documents 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).
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. An angle of 360°/0° results in a cosine of 1 while an angle of 180° 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.

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).

<table>
<thead>
<tr>
<th>Word</th>
<th>cf</th>
<th>df</th>
</tr>
</thead>
<tbody>
<tr>
<td>try</td>
<td>10422</td>
<td>8760</td>
</tr>
<tr>
<td>insurance</td>
<td>10440</td>
<td>3997</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>Term</th>
<th>(df_t)</th>
<th>(idf_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>car</td>
<td>18,165</td>
<td>1.65</td>
</tr>
<tr>
<td>auto</td>
<td>6,723</td>
<td>2.08</td>
</tr>
<tr>
<td>insurance</td>
<td>19,241</td>
<td>1.62</td>
</tr>
<tr>
<td>best</td>
<td>25,235</td>
<td>1.50</td>
</tr>
</tbody>
</table>

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 SCHAFFER; 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.
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).
Categories for classification

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)


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

**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).

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

Table 11.1 The Peen 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
Co-occurrence Groups

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 Makrov
Chains. Makrov 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;

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

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 Makrov 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 Peen 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,
Co-occurrence Groups

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.

```python
def createfilepathlist():
    pathlist = []
    subfolders = [x[0] for x in os.walk('./WikiText/')]  # line 47
    for subfolder in subfolders[1:]:
        for filename in os.listdir(subfolder):
            pathlist.append(subfolder + '/' + filename)
    return pathlist

i = 0
filepathlist = createfilepathlist()
len_filelist = len(filepathlist)
Starttime = time.time()

if __name__ == '__main__':
    print('Tagging new corpus')
    pool = ThreadPool(4)
    pool.map(POStag, filepathlist)
    pool.close()
    pool.join()
    print('+++++########+++')
    print('complete Operation took %s Minutes' % ((time.time() - Starttime) / 60))
    print('+++++########+++')
```

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

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

Table 11.2 Five nearest neighbors for Target words (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.

```python
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         with io.open('./wikicorpuspickeld_2/'+file, 'rb') as fin:
035             loaded_corpus = pickle.load(fin)
036
037     for sentence in loaded_corpus:
038         for (index, tokentag) in enumerate(sentence):
039             (token, tag) = tokentag
040             token = token.lower()
041
042             if token in groups:
043                 term = sentence[index-Fenster:index+Fenster]
044                 for (term_token, term_tag) in term:
045                     term_token = term_token.lower()
046                     if term_token not in S_list and NounVerb(term_tag):
047                         if token not in word_dict:
048                             word_dict[token] = {}
049                         if term_token in word_dict[token]:
050                             word_dict[token][term_token] += 1
051                         else:
052                             word_dict[token][term_token] = 1
053
054         print i
055         delta_time = time.time() - Starttime3
056         print "time till end %.2f Minutes" % (((delta_time/60)/i)*[(len(Files[0])-i)]
057         i += 1
058     return word_dict
059
060     groups = ['wohnen', 'arbeiten', 'bildung', 'einkaufen', 'gaststätte',  
061         'hotel', 'kreditinstitut', 'kultur', 'dienstgebäude',]
062
063     Starttime2 = time.time()
064     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.).
<table>
<thead>
<tr>
<th>Search Term</th>
<th>*</th>
<th>=</th>
<th>weiß</th>
<th>wurde</th>
<th>grand</th>
</tr>
</thead>
<tbody>
<tr>
<td>hotel</td>
<td>(25707)</td>
<td>(2538)</td>
<td>(1926)</td>
<td>(1828)</td>
<td>(1378)</td>
</tr>
<tr>
<td>gaststätte</td>
<td>wurde</td>
<td>heute</td>
<td>gebäude</td>
<td>straße</td>
<td>=</td>
</tr>
<tr>
<td>arbeiten</td>
<td>=</td>
<td>beruf</td>
<td>rechtsanwalt</td>
<td>began</td>
<td>wurden</td>
</tr>
<tr>
<td>bildung</td>
<td>=</td>
<td>kultur</td>
<td>forschung</td>
<td>wissenschaft</td>
<td>bundeszentrale</td>
</tr>
<tr>
<td>wohnen</td>
<td>=</td>
<td>bauen</td>
<td>menschen</td>
<td>arbeiten</td>
<td>haus</td>
</tr>
<tr>
<td>dienstgebäude</td>
<td>=</td>
<td>wurde</td>
<td>berlin</td>
<td>eisenbahndirektion</td>
<td>heute</td>
</tr>
<tr>
<td>kreditinstitut</td>
<td>]</td>
<td>deutschland</td>
<td>schweiz</td>
<td>bank</td>
<td>österreich</td>
</tr>
<tr>
<td>einkaufen</td>
<td>=</td>
<td>gehen</td>
<td>können</td>
<td>geht</td>
<td>konnten</td>
</tr>
<tr>
<td>kultur</td>
<td>=</td>
<td>sehenswürdigkeiten</td>
<td>geschichte</td>
<td>kunst</td>
<td>wissenschaft</td>
</tr>
</tbody>
</table>

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

```python
045  def Delete_stopwords(Tokens):
046      return [token for token in Tokens if not token in
047                        nltk.corpus.stopwords.words('german')]
048
049  GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
050  tokenizer = RegexpTokenizer(r'\w+')
051  token_dict = {}
052  HTMLIDS = get_html_ids()
053  lower = 0
054  upper = lower + 1000
055  Starttime = time.time()
056  parsedhtmls = 0
057  while lower <= len(HTMLIDS):
058      Starttime2 = time.time()
059      stripped_htmls_list = stripped_htmls(HTMLIDS[lower:upper])
060      for html in stripped_htmls_list:
061          parsedhtmls += 1
062          time_tokenize = time.time()
063          tokens = tokenizer.tokenize(html)
064          tokens = Delete_stopwords(tokens)
065          token_dict_file = {}
066          for token in tokens:
067              stemmedtoken = GermanStemmer.stem(token)
068              if stemmedtoken in token_dict_file:
069                  token_dict_file[stemmedtoken] += 1
070              else:
071                  token_dict_file[stemmedtoken] = 1
072              for key in token_dict_file:
073                  if key in token_dict:
074                      doc_count = token_dict[key][0] + 1
075                      occurrence_count = token_dict[key][1] + token_dict_file[key]
076                      token_dict[key] = (doc_count, occurrence_count)
077                  else:
078                      token_dict[key] = (1, token_dict_file[key])
079              print('Number of tokens in dict: %s' % len(token_dict))
080              print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
081              delta_time = time.time() - Starttime
082              print "time till now %.2f Minutes"%(delta_time / 60)
083              print "time till end %.2f Minutes"%(((delta_time/60)/upper)*(len(HTMLIDS)-
084                                                          upper))
085          lower += 1000
086          upper = lower + 1000
087      with io.open('./Vector/HTMLVectorSpace.pickle', 'wb') as fout:
088          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.
def Vector_Calculator():
    Starttime3 = time.time()
    i = 1
    GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
    token_dict_file = {}
    p = re.compile(ur'^[a-zA-ZäöüßÄÖÜ]{2,}$', re.UNICODE)
    Files = [x[2] for x in os.walk('./wikicorpuspickeld_2/')]
    for file in Files[0]:
        with io.open('./wikicorpuspickeld_2/' + file, 'rb') as fin:
            loaded_corpus = pickle.load(fin)
            for sentence in loaded_corpus:
                for (index, tagtuple) in enumerate(sentence):
                    (token, tag) = tagtuple
                    token = token.lower()
                    if token not in stopword_list:
                        stemmedtoken = GermanStemmer.stem(token)
                        if stemmedtoken in token_dict_file:
                            token_dict_file[stemmedtoken] += 1
                        else:
                            token_dict_file[stemmedtoken] = 1
            delta_time = time.time() - Starttime3
            print '% time till end %.2f Minutes' % (((delta_time/60)/i)*len(Files[0]) - i)
            i += 1
    return token_dict_file

stopword_list = []
for word in stopwords.words('german'):
    stopword_list.append(unicode(word.decode('latin-1')))
Starttime = time.time()
Vectorraum = Vector_Calculator()
with io.open('./Vector/WikiVectorSpace2.pickle', 'wb') as fout:
    pickle.dump(Vectorraum, fout)
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.

```python
with io.open('./Vector/WikiVectorSpace.pickle', 'rb') as fin:
    WikiVectorSpace = pickle.load(fin)

with io.open('./Vector/HTMLVectorSpace.pickle', 'rb') as fin:
    HTMLVectorSpace = pickle.load(fin)

CombinedVectorSpace = {}

for key in WikiVectorSpace:
    if key in HTMLVectorSpace:
        CombinedVectorSpace[key] = HTMLVectorSpace[key]

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

```
with io.open('./Vector/CombinedVectorSpace.pickle', 'rb') as fin:
    CombinedVectorSpace = pickle.load(fin)

CombinedVectorSpaceIDFT = {}
DocumentCount = countrows()

for key in CombinedVectorSpace:
    idft = numpy.log(numpy.divide(float(DocumentCount),
                                  float(1+CombinedVectorSpace[key][0])))
    CombinedVectorSpaceIDFT[key] = CombinedVectorSpace[key][0],
                                   CombinedVectorSpace[key][1], idft

with io.open('./Vector/CombinedVectorSpaceIDFT.pickle', 'wb') as fout:
    pickle.dump(CombinedVectorSpaceIDFT, fout)

CombinedVectorSpaceIDFTKeyList = []

for key in CombinedVectorSpaceIDFT:
    CombinedVectorSpaceIDFTKeyList.append(key)

CombinedVectorSpaceIDFTKeyList.sort()

with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'wb') as fout:
    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 \( idf_t \) is then added together with document frequency value and the collection frequency value to the new dictionary \( \text{CombinedVectorSpaceIDFT} \) with the token again serving as the key (NumPy 1.8.1 library;).

\( \text{CombinedVectorSpaceIDFT} \) 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.
Address Classification

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.

```python
offset = 0
length = countrows()
Starttime = time.time()
createColumn()

with io.open('./Vector/CombinedVectorSpaceIDFT.pickle', 'rb') as fin:
    CombinedVectorSpaceIDFT = pickle.load(fin)

with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'rb') as fin:
    CombinedVectorSpaceIDFTKeyList = pickle.load(fin)

while offset <= length:
dicts = VectorDICTReader(offset)
Starttime2 = time.time()
arraylist = []
for tuple in dicts:
    array = []
id = tuple[0]
dictionary = pickle.loads(str(tuple[1]))
    for key in CombinedVectorSpaceIDFTKeyList:
        if key in dictionary:
            array.append(numpy.multiply(CombinedVectorSpaceIDFT[key][2],
                                         dictionary[key]))
        else:
            array.append(0)
    array = numpy.divide(array, numpy.linalg.norm(array))
    array = zlib.compress(array)
    arraylist.append((psycopg2.Binary(array), id,))

UpdateHtmlUniquewithTFIDFlist(arraylist)
offset += 100

print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
delta_time = time.time() - Starttime
print 'time till now %.2f Minutes' % (delta_time / 60)
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.
Address Classification

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

**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.
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”.

<table>
<thead>
<tr>
<th></th>
<th>hotel</th>
<th>gaststätte</th>
<th>arbeiten</th>
<th>bildung</th>
<th>wohnen</th>
<th>dienstgebäude</th>
<th>kreditinstitut</th>
<th>einkaufen</th>
<th>kultur</th>
</tr>
</thead>
<tbody>
<tr>
<td>hotel</td>
<td>1.00</td>
<td>0.17</td>
<td>0.01</td>
<td>0.00</td>
<td>0.02</td>
<td>0.06</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>gaststätte</td>
<td>0.17</td>
<td>1.00</td>
<td>0.04</td>
<td>0.02</td>
<td>0.06</td>
<td>0.35</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>arbeiten</td>
<td>0.01</td>
<td>0.04</td>
<td>1.00</td>
<td>0.01</td>
<td>0.04</td>
<td>0.03</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>bildung</td>
<td>0.00</td>
<td>0.02</td>
<td>0.01</td>
<td>1.00</td>
<td>0.01</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.07</td>
</tr>
<tr>
<td>wohnen</td>
<td>0.02</td>
<td>0.06</td>
<td>0.04</td>
<td>0.01</td>
<td>1.00</td>
<td>0.11</td>
<td>0.00</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>dienstgebäude</td>
<td>0.06</td>
<td>0.35</td>
<td>0.03</td>
<td>0.02</td>
<td>0.11</td>
<td>1.00</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>kreditinstitut</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>einkaufen</td>
<td>0.00</td>
<td>0.02</td>
<td>0.01</td>
<td>0.00</td>
<td>0.02</td>
<td>0.02</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>kultur</td>
<td>0.00</td>
<td>0.02</td>
<td>0.01</td>
<td>0.07</td>
<td>0.01</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 12.2 Similarity matrix co-occurrence groups
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.
with io.open('./Vector/TFIDF_CoOc.pickle', 'rb') as fin:
    TFIDF_CoOc = pickle.load(fin)

Columnlist = []
SearchTermList = []
TFIDFworkingdict = {}

for searchterm in TFIDF_CoOc:
    SearchTermList.append(searchterm)
    Columnlist.append(TFIDF_CoOc[searchterm]+'_CoOc')
    Columnlist.append(TFIDF_CoOc[searchterm]+'_ST')

for searchterm in SearchTermList:
    TFIDFworkingdict[searchterm] =
        pickle.loads(zlib.decompress(TFIDF_CoOc[searchterm][0]),
        pickle.loads(zlib.decompress(TFIDF_CoOc[searchterm][1])))

for searchterm in Columnlist:
    createColumn(searchterm)

sqlstring = Sqlstringconstructor(Columnlist)

offset = 0
range = 1000
length = countrows()

vectors = VectorDICTReader(range, offset)

Starttime = time.time()
while vectors:
    Starttime2 = time.time()
    updatelist = []
    for vectortup in vectors:
        cosinelist = []
        id, vector = vectortup[0], pickle.loads(zlib.decompress(vectortup[1]))
        for searchterm in SearchTermList:
            cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][0],
            vector), 8)
            cosinelist.append(cosine)
        cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][1],
            vector), 8)
        cosinelist.append(cosine)
        cosinelist.append(id)
        updatelist.append(cosinelist)
    offset += range
    print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
    delta_time = time.time() - Starttime
    print "time till now %.2f Minutes" % (delta_time / 60)
    print "time till end %.2f Minutes" % (((delta_time/60)/offset)*(length-offset))
    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`.

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.

### 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.
Table 12.4 Classification problem number one

<table>
<thead>
<tr>
<th>Class 1</th>
<th>Class 2</th>
<th>Class 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Website 1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Website 2</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Website 3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Website 4</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Website 5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Website 6</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Website 7</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>Website 8</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Website 9</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Website 10</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Mean</td>
<td>6.9</td>
<td>3.2</td>
</tr>
<tr>
<td>Proposal</td>
<td>8.3</td>
<td>9.6</td>
</tr>
</tbody>
</table>

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.
def Breaks(valuelist, index):
    firstrun = True
    highlist = []
    lowlist = []
    newlist = []

    for item in valuelist:
        newlist.append(item[index])

    while newlist:
        high = max(newlist)
        low = min(newlist)

        if firstrun and len(newlist) == 1:
            highlist.append(high)
            lowlist.append(low)
            firstrun = False
        elif firstrun:
            highlist.append(high)
            lowlist.append(low)
            newlist.remove(high)
            newlist.remove(low)
            firstrun = False
        elif high == low:
            if numpy.abs(high-numpy.mean(highlist)) < numpy.abs(high-numpy.mean(lowlist)):
                highlist.append(high)
                newlist.remove(high)
                else:
                    lowlist.append(high)
                    newlist.remove(high)
            else:
                if numpy.abs(high-numpy.mean(highlist)) < numpy.abs(high-numpy.mean(lowlist)):
                    highlist.append(high)
                    newlist.remove(high)
                    else:
                        lowlist.append(high)
                        newlist.remove(high)
                else:
                    if numpy.abs(low-numpy.mean(lowlist)) < numpy.abs(low-numpy.mean(highlist)):
                        lowlist.append(low)
                        newlist.remove(low)
                        else:
                            highlist.append(low)
                            newlist.remove(low)
                            else:
                                highlist.append(low)
                                newlist.remove(low)
    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 \texttt{newlist} is longer though, then \texttt{highlist} and \texttt{lowlist} are appended with their first values \texttt{high} and \texttt{low}, lines 82 to 87, respectively. After this, \texttt{firstrun} is set to \texttt{False}. Both \texttt{high} and \texttt{low} are removed from \texttt{newlist} and the iteration begins again in lines 73 and 74 getting a new \texttt{high} and \texttt{low}, now the new highest and lowest value in the \texttt{newlist}. Both \texttt{high} and \texttt{low} have now been tested, whose mean value of \texttt{highlist} and \texttt{lowlist} is closer to their own value. They are then appended to the list that is more similar (closer) to them and removed from \texttt{newlist}. This process is repeated until no values are left in \texttt{newlist}. All of this happens in lines 96 to 109. One special case that can happen when, for example, only one value left in \texttt{newlist} is handled by the code in lines 89 to 95. In the end, the function returns the mean value of \texttt{highlist} (BAHRENBERG 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 \textbf{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 (BAHRENBERG ET AL.; 2008; pp. 259 - 262), (NUMPY 1.8.1 LIBRARY).

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

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure.png}
\caption{Hotel (hotel)}
\end{figure}
Address Classification

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, “Einkauf” 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:
Address Classification

"Update Addresses unique set gaststätte_cooc = 0.186 where gaststätte_cooc > 0.186"
3 rows affected

"Update Addresses unique set einkaufen_cooc = 0.322 where einkaufen_cooc > 0.322"
7 rows affected

"Update Addresses unique 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 Bildung (education) with Co-Occurrence Group

c)

Classification Bildung (education) with Search Term

d)
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 Kultur (culture) with Co-Occurrence Group

Classification Kultur (culture) 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 (BAHRENBERG ET AL.; 2010; pp. 19-23).

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 (Bahnernberg et al.; 2010; pp. 19-23), (Kruker 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:

\[
\text{recall} = \frac{MI}{M}
\]

$MI$ (Mapped and Identified), $M$ (Mapped)
The formula for precision is:

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

$MI$ (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 (BAHRENBERG ET AL.; 2010; pp. 128-129).
Mapping

**Wohnen Search Term**
- Recall: 0.1
- Precision: 0.64
- p-value: 0.26

**Dienstgebäude Search Term**
- Recall: 0.35
- Precision: 0.44
- p-value: 0.05

**Kreditinstitut Search Term**
- Recall: 0.18
- Precision: 0.03
- p-value: 0.95

**Wohnen Co-Occurrence**
- Recall: 0.13
- Precision: 0.57
- p-value: 0.41

**Dienstgebäude Co-Occurrence**
- Recall: 0.55
- Precision: 0.15
- p-value: 0.099

**Kreditinstitut Co-Occurrence**
- Recall: 0.18
- Precision: 0.18
- p-value: 0.14
Here two factors are striking: the recall is low and the precision as well. 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 ≤ 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).

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 (BAHRENBERG ET AL.; 2010; pp. 183-191).
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 usable 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 where 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 3rd 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


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


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


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


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

Manning, C., D.; Raghavan, P.; Schütze, H. (2009); An Introduction to Information Retrieval; Cambridge; Cambridge University Press


Robertson, S. (2013); Common Crawl URL Index; https://github.com/trivio/common_crawl_index Cited: 01.07.2015


Santorini, B. (1990); Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision, 2nd 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


Teske, D. (2014); Geocoder Accuracy Ranking; In: Process Design for Natural Scientists, CCIS 500, pp. 161-174; Springer-Verlag; Berlin, Heidelberg


Weichert, P. (2008); Entwicklungenlinien der Sozialgeographie von Hans Bobek bis Benno Werlen; Franz Steiner Verlag, Stuttgart


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/

Psycopg 2.5.3 library; http://initd.org/psycopg/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;

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: 15.6.2015

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; Citied: 03.07.2015

nic.at; https://www.nict.at;
    .at Report 2012
    Registration Guidelines

Openstreetmap.org; https://www.openstreetmap.org; Cited 25.06.2015
OpenStreetMap Wiki; https://wiki.openstreetmap.org/wiki/Main_Page;
  Addresses https://wiki.openstreetmap.org/wiki/Addresses

W³Techs;  http://w3techs.com/technologies/history_overview/top_level_domain/ms/y  Cited: 15.6.2015

Annex

Source Code

Threading Example

```python
import remote_copy_external,
import threading
time
import pickle
import os

class myThread (threading.Thread):
    def __init__(self, urlstump, threadid):
        threading.Thread.__init__(self)
        self.urlstump = urlstump
        self.id = threadid
    def run(self):
        threadLimiter.acquire()
        print('checking for ' + str(self.urlstump)
        current_urllist.append(self.urlstump)
        remote_copy_external('AWS-PUBLIC-KEY',
                          'AWS-PRIVATE-KEY',
                          'tldat',
                          'Data2/'+'str(self.urlstump),
                          self.urlstump,parallelconnections,True)
        threadLimiter.release()

def iterate_ipickle(ipickle):
    if ipickle < 10:
        return ipickle +1
    else:
        return 0

duds = len(threading.enumerate())
threadnumber = 20
parallelconnections = 50
threadLimiter = threading.BoundedSemaphore(threadnumber)

print 'starting threads %s' % (duds)

current_urllist = []
running = True

for threadnumber in range(0, threadnumber):
    threadlist = list(urllist)
    ipickle = 0

    while running:
        print 'running threads %s' % (len(threading.enumerate()))
        print
        picklelist = list(urllist)
        ipickle = iterate_ipickle(ipickle)
        picklelist(urllist_pikkle, 'obj_%s.pickle'(ipickle))

        if duds+threadnumber > len(threading.enumerate()):
            if threadlist:
                element = threadlist[0]
                threadlist.remove(element)
                myThread(element,ID).start()
                ID += 1
            continue
        if not threadlist:
            pass
```

xiii
Annex

```python
elif threadlist:
    ipickle = iterate_ipickle(ipickle)
    picklelist(urllist_pickle, 'obj_%s.pickle' % (ipickle))
    time.sleep(1)
    print current_urllist
    continue

elif not threadlist:
    ipickle = iterate_ipickle(ipickle)
    picklelist(urllist_pickle, 'obj_%s.pickle' % (ipickle))
    time.sleep(1)
    print current_urllist
    continue

while duds < len(threading.enumerate()):
    time.sleep(10)
    print ''
    print 'Current Passnumber: %d' % (Passnumber)
    print current_urllist
    continue

if not urllist:
    running = False
    print ''
    print 'Script did run Passnumber %d' % (Passnumber)
    pass
else:
    threadlist = list(urllist)
    urllist_pickle = list(urllist)
    ipickle = iterate_ipickle(ipickle)
    picklelist(urllist_pickle, 'obj_%s.pickle' % (ipickle))
    print ''
    print 'Script did run Passnumber %d' % (Passnumber)
    Passnumber += 1

DBconnector

```
BEGIN
ALTER TABLE Addresses ADD COLUMN AddDate INT;
EXCEPTION
WHEN duplicate_column THEN RAISE NOTICE 'column Addresses already exists in AddDate.';
END;
$$
)

def finddoubles(addresslist):
    newlist = []
    conn, cur = DBConnect()
    for address in addresslist:
        cur.execute("SELECT ID FROM Addresses where street = %s and Street_number = %s and pcode = %s", (address[1], address[2], address[3],))
        data = cur.fetchall()
        if not data:
            newlist.append(address)
    WriteToTableMany(newlist)
    return newlist

def ReadFromTable():
    conn, cur = DBConnect()
    cur.execute("SELECT * FROM Addresses;")
    data = list(cur.fetchall())
    conn.close()
    return data

def DeleteFromTable():
    conn, cur = DBConnect()
    cur.execute("DELETE * FROM Addresses;")
    conn.commit()
    conn.close()

def DropTable():
    conn, cur = DBConnect()
    cur.execute("DROP TABLE Addresses")
    conn.commit()
    conn.close()

# OSM Parser

# -*- coding: utf-8 -*-
import sys, numpy, DBconnector
from xml.sax import make_parser, handler
endaddress = '0'
startaddress = '0'
start = '0'
end = '0'
addresslist = []
class startendfinder(handler.ContentHandler):
def __init__(self):
    self.address = ['lat_lon', 'pcode', 'street', 'number']
    self.plz = False
    self.street = False
    self.number = False
    self.nodemode = False
    self.waymode = False
    self.relationmode = False
def startElement(self, name, attrs):
    if name in ('node'):
        self.address[0] = 'POINT(%s %s) %s' % (attrs.get('lon'), attrs.get('lat'))
        self.nodedict[int(attrs.get('id'))] = {float(attrs.get('lat')),
                                         float(attrs.get('lon'))}
        self.nodemode = True
    elif name in ('way'):
        self.wayid = int(attrs.get('id'))
        self.waymode = True
    elif name in ('relation'):
        self.relationid = int(attrs.get('id'))
        self.relationmode = True
    if self.relationmode == True:
        if name == 'member':
            self.memberlist.append((int(attrs.get('ref'))), attrs.get('type'))
        if name == 'tag':
            k, v = (attrs.get('k'), attrs.get('v'))
            if k == 'addr:street':
                self.address[1] = unicode(v)
                self.street = True
            if k == 'addr:housenumber':
                self.address[2] = unicode(v)
                self.number = True
            if k == 'addr:postcode':
                try:
                    if int(v) <= 1099:
                        self.address[3] = int(v)
                        self.plz = True
                    elif int(v) >= 1200 and int(v) <= 1209:
                        self.address[3] = int(v)
                        self.plz = True
                    except:
                        pass
                if self.waymode == True:
                    if name == 'nd':
                        self.ndlist.append(int(attrs.get('ref')))
                    if name == 'tag':
                        k, v = (attrs.get('k'), attrs.get('v'))
                        if k == 'addr:street':
                            self.address[1] = unicode(v)
                            self.street = True
                        if k == 'addr:housenumber':
                            self.address[2] = unicode(v)
                            self.number = True
                        if k == 'addr:postcode':
                            try:
                                if int(v) <= 1099:
                                    self.address[3] = int(v)
                                    self.plz = True
                                elif int(v) >= 1200 and int(v) <= 1209:
                                    self.address[3] = int(v)
                                    self.plz = True
                                except:
                                    pass
if self.nodemode == True:
    if name == 'tag':
        k, v = (attr.get('k'), attr.get('v'))
        if k == 'addr:street':
            self.address[0] = unicode(v)
            self.street = True
        elif k == 'addr:house_number':
            self.address[1] = unicode(v)
            self.number = True
        if k == 'addr:postcode':
            try:
                if int(v) <= 1099:
                    self.address[3] = int(v)
                    self.plz = True
                elif int(v) >= 1200 and int(v) <= 1209:
                    self.address[3] = int(v)
                    self.plz = True
            except:
                pass

def endElement(self, name):
    if name in ('node',):
        if self.plz is True and self.street is True and self.number is True:
            addresslist.append(tupple(self.address))
            self.nodemode = False
            self.plz = False
            self.street = False
            self.number = False
            if name in ('way',)
                for nd in self.ndlist:
                    self.latlon = self.nodedict[nd]
                    self.latlist.append(self.latlon[0])
                    self.lonlist.append(self.latlon[1])
                self.waydict[self.wayid] = (numpy.mean(self.latlist)),
            nodyn.mean(self.lonlist))
            if self.plz is True and self.street is True and self.number is True:
                self.address[0] = 'POINT(%s %s)'
                (numpy.mean(self.lonlist), numpy.mean(self.latlist))
                addresslist.append(tupple(self.address))
            self.latlist = []
            self.lonlist = []
            self.waymode = False
            self.plz = False
            self.street = False
            self.number = False
            self.ndlist = []

    if name in ('relation',)
        for member in self.memberlist:
            if member[1] == 'node':
                self.latlist.append(self.nodedict[member[0]][0])
                self.lonlist.append(self.nodedict[member[0]][1])
            elif member[1] == 'way':
                self.latlist.append(self.waydict[member[0]][0])
                self.lonlist.append(self.waydict[member[0]][1])
            elif member[1] == 'relation':
                self.latlist.append(self.relationdict[member[0]][0])
                self.lonlist.append(self.relationdict[member[0]][1])
            self.relationdict[member.relationid] = (numpy.mean(self.latlist),
            numpy.mean(self.lonlist))
        if self.plz is True and self.street is True and self.number is True:
            self.address[0] = 'POINT(%s %s)' % (numpy.mean(self.lonlist),
            numpy.mean(self.latlist))

Annex
Annex

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

Disassemble HTML

001 import psycopg2
002 import gzip
003 from bs4 import UnicodeDammit
004 import os
005 import time
006
007 def DBConnect():
008     conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=########")
009     cur = conn.cursor()
010     return conn, cur
011
012 def CreateTable():
013     conn, cur = DBConnect()
014     cur.execute("CREATE TABLE IF NOT EXISTS html (id serial PRIMARY KEY, url TEXT, html_file text);")
015     conn.commit()
016     conn.close()
017
018 def WriteManyToTable(Values):
019     conn, cur = DBConnect()
020     args_str = ','.join(cur.mogrify("(\s%s\s)\s", x) for x in Values)
021     cur.execute("INSERT INTO html (url, html_file) VALUES " + args_str)
022     conn.commit()
023     conn.close()
024
025 def ReadFromTable():
026     conn, cur = DBConnect()
027     lines = cur.fetchall()
028     conn.close()
029     return lines
030
031 def Database_export(file):
032     start1 = time.time()
033     i = ''
034     tup_i = ()
035     url = ''
036     html_written = 0
037     mode = False
038     for line in file.readlines():
splitline = line.split(' ')

try:
    if splitline[0][:7] == 'http://' and splitline[3] == 'text/html' and len(splitline) == 5:
        if len(i) > 0:
            url = splitline[0]
            mode = True
            if len(tup_i) >= 100:
                print('empty tup_i')
                try:
                    WriteManyToTable(tup_i)
                except:
                    pass
                    tup_i = ()
                    print ('html files written to DB %s' % html_written)
            html_written += 1
            elif splitline[0][:7] == 'http://' and splitline[3] != 'text/html' and len(splitline) == 5:
                if len(i) > 0:
                    url = splitline[0]
                    mode = False
                    if len(tup_i) >= 100:
                        print('empty tup_i')
                        try:
                            WriteManyToTable(tup_i)
                        except:
                            pass
                            tup_i = ()
                            print ('html files written to DB %s' % html_written)
                except:
                    pass
        if mode == True:
            try:
                i += unicode(line, "utf-8")
            except:
                #i += UnicodeDammit(line).unicode_markup  # Benoetigt Extrem Viele resourcen
                pass
        if len(i) > 0:
            try:
                WriteManyToTable(tup_i)
            except:
                pass
                html_written += len(tup_i)
                print('parsing %s took %s Minutes \n' % (file, (time.time() - start1) / 60))
                start = time.time()
                html_written += len(tup_i)
                print ('export to database took %s Minutes' % ((time.time() - start) / 60))
                html_written += len(tup_i)
                print ('COMPLETE Operation took %s Minutes' % ((time.time() - start) / 60))
                html_written += len(tup_i)
    def open_Paths(PATH):
        for path, dirs, files in os.walk(PATH):
            for filename in files:
                try:
                    xix
fullpath = os.path.join(path, filename)
print ('##############################')
print('%s Size: %s MB' % (fullpath, os.path.getsize(fullpath) / 1048576))
file = gzip.open(fullpath, 'rb')
Database_export(file)
file.close()
except:
    pass

def current_database():
    conn, cur = DBConnect()
    cur.execute('SELECT current_database()')
    DB_name = cur.fetchone()
    print('Connecting to %s % DB_name)
    print('##############################')
if __name__ == '__main__':
    current_database()
    raw_input('Please Press the anykey')
    CreateTable()
startall = time.time()
PATH = 'E:\Data'
open_Paths(PATH)
print('+++++++++++++++')
print('complete Operation took %s Minutes' % ((time.time() - startall) / 60))
print('+++++++++++++++')
lines = ReadFromTable()
for line in lines:
    print (line[2])
    raw_input('Please Press the anykey')

import psycopg2
import gzip
from bs4 import UnicodeDammit
import os
import time

def DBConnect():
    conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=########")
    cur = conn.cursor()
    return conn, cur

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

def WriteManyToTable(Values):
    conn, cur = DBConnect()
    args_str = ', '.join(cur.mogrify("(\%s,\%s)", x) for x in Values)
    cur.execute("INSERT INTO html (url, html_file) VALUES " + args_str)
    conn.commit()
    conn.close()
def ReadFromTable():
    conn, cur = DBConnect()
    cur.execute("SELECT * FROM html;")
    lines = cur.fetchall()
    conn.close()
    return lines

def Database_export(file):
    start1 = time.time()
    i = ''
    tup_i = ()
    url = ''
    html_written = 0
    mode = False
    for line in file.readlines():
        splitline = line.split(' ')
        try:
            if splitline[0][7:] == 'http://' and splitline[3] == 'text/html' and len(splitline) == 5:
                if len(i) > 0:
                    tup_i = tup_i + ((url, i),)
                    i = ''
                url = splitline[0]
                mode = True
        except:
            pass
        try:
            if len(tup_i) >= 100:
                print('empty tup_i')
            except:
                WriteManyToTable(tup_i)
            pass
            tup_i = ()
            print("html files written to DB %s" % html_written)
        except:
            pass
        html_written += 1
    try:
        WriteManyToTable(tup_i)
    except:
        pass
    if mode == True:
        try:
            i += unicode(line, "utf-8")
        except:
            # i += UnicodeDammit(line).unicode_markup  # Benoetigt Extrem Viele
            pass
    if len(i) > 0:
        tup_i = tup_i + ((url, i),)
    try:
        WriteManyToTable(tup_i)
    except:
Annex

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

Tag Stripper

# -*- coding: UTF-8 -*-
import psycopg2
import time
from HTMLParser import HTMLParser
import re

def DBConnect():
    conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=########")
    cur = conn.cursor()
    return conn, cur

def ReadFromHTML(offset):
    conn, cur = DBConnect()
ANNEX

```python
cur.execute("SELECT id,html_file FROM html WHERE vienna = TRUE ORDER BY id limit 1000 offset %s ;"
offset %s)
data = cur.fetchall()
cur.close()
conn.close()

```

```python
def UpdateHTMLwithStrippedHTML(Values):
    conn, cur = DBConnect()
    cur.execute("UPDATE html SET stripped_html = %s WHERE vienna = TRUE AND id in (%s)", Values)
    conn.commit()
cur.close()
conn.close()
return data
```

```python
def createColumn():
    conn, cur = DBConnect()
    cur.execute("ALTER TABLE html DROP COLUMN IF EXISTS stripped_html;")
    conn.commit()
cur.execute("ALTER TABLE html ADD COLUMN stripped_html TEXT;")
    conn.commit()
cur.close()
conn.close()
```

```python
def remove_tags(text):
    text = TAG_RE.sub('', text)
    text = Short.sub(' ', text)
    text = eszt.sub('ß', text)
    text = ae.sub('ä', text)
    text = AE.sub('Ä', text)
    text = oe.sub('ö', text)
    text = OE.sub('Ö', text)
    text = ue.sub('ü', text)
    text = UE.sub('Ü', text)
    return text
```

```python
TAG_RE = re.compile(r'<[^>]+>')
Short = re.compile(r'\S{68,}')
eszt = re.compile(r'ß')
ae = re.compile(r'ä')
AE = re.compile(r'Ä')
oe = re.compile(r'ö')
OE = re.compile(r'Ö')
ue = re.compile(r'ü')
UE = re.compile(r'Ü')
```

```python
createColumn()
Starttime = time.time()
strippedlist = []
offset = 0
Starttime2 = time.time()
htmls = ReadFromHTML(offset)
Numberofrows = 8406507
while htmls:
    timeregex = time.time()
    print("starting Regex")
    for row in htmls:
        id = row[0]
        stripped_html = remove_tags(row[1])
        strippedlist.append((stripped_html, id))
    print("Regex took %.2f Minutes" % ((time.time() - timeregex) / 60))
print("Starting DB Update")
```
timeDBupdate = time.time()
UpdateHTMLwithStrippedHTML(strippedlist)
print('DB Update took %.2f Minutes' % ((time.time() - timeDBupdate) / 60))
print offset
print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
delta_time = time.time() - Starttime
time till now %.2f Minutes"%(delta_time / 60)
time till end %.2f Minutes"%(delta_time/60)/(offset+1000))*(NumberOfrows-
(offset+1000))
Starttime2 = time.time()
strippedlist = []
offset += 1000
htmls = ReadFromHTML(offset)

# for row in ReadFromHTML():
#     print row[0]
#     print row[1]
#     print row[2]

print('+++++########+++++')
print('complete Operation took %s Minutes' % ((time.time() - Starttime) / 60))
print('+++++########+++++')

import psycopg2
import time
import re

def DBConnect():
    conn = psycopg2.connect("dbname=TEST_DB user=postgres
password=############") #Master_DB_spatial2
    cur = conn.cursor()
    return conn, cur

def CreateTable():
    conn, cur = DBConnect()
    cur.execute("CREATE TABLE IF NOT EXISTS AddressesUnique (id serial PRIMARY KEY,geom
geometry, street text, street_number text, pcode integer);")
    cur.execute("CREATE TABLE IF NOT EXISTS AddressesUniqueJoinedWithURL (id serial
PRIMARY KEY, AddressesUniqueID INTEGER, HTMLID INTEGER, Original BOOLEAN);")
    conn.commit()
    conn.close()

def addcolumn():
    conn, cur = DBConnect()
cur.execute(""
DO $$
BEGIN
ALTER TABLE AddressesUnique ADD COLUMN AddDate INT;
EXCEPTION
WHEN duplicate_column THEN RAISE NOTICE 'column AddressesUnique already exists in AddDate.';
END;
$$
"")
    conn.commit()
cur.execute(""
DO $$
END;
$$
"")
BEGIN
ALTER TABLE AddressesUniqueJoinedWithURL ADD COLUMN AddDate INT;
EXCEPTION
WHEN duplicate_column THEN RAISE NOTICE 'column AddressesUniqueJoinedWithURL already exists in AddDate.';
END;
$$
)
conn.commit()
conn.close()
def MakeAddressesUnique():
    conn, cur = DBConnect()
    conn.commit()
    cur.execute("INSERT INTO AddressesUnique(geom, street, street_number, pcode, AddDate)
    SELECT DISTINCT ON (street, street_number)
    geom, street, street_number, pcode, AddDate FROM Addresses")
    conn.commit()
    return
def ReadFromTableAddressesUnique():
    conn, cur = DBConnect()
    cur.execute("SELECT street, street_number, id FROM AddressesUnique;")
    return cur.fetchall()
def FindVienna():
    conn, cur = DBConnect()
    cur.execute("UPDATE html SET Vienna = TRUE WHERE html_file LIKE '%%' || ' %s ' || '%%';" % 'Wien')
    conn.commit()
    conn.close()
def ConstructSQLStatmentSearchAddresses(Values):
    SQLStatmentdict = {}
    conn, cur = DBConnect()
    for line in Values:
        if line[0][-6:] == 'straße':
            #Berücksichtigt mögliche groß und klein schreibung von Straße
            SQLStatmentdict[line[2]] = cur.mogrify("Select ID FROM HTML WHERE Vienna = TRUE AND 
"OR "stripped_html ILIKE '%"+line[0]+" '+line[1]+"/:%')" 
"OR "stripped_html ILIKE '%"+line[0]+" '+line[1]+"/:*') AND "'", line[2])
        elif line[0][-4:] == 'asse':
            #Berücksichtigt mögliche groß und klein schreibung von Gasse
            SQLStatmentdict[line[2]] = cur.mogrify("Select ID FROM HTML WHERE Vienna = TRUE AND 
"OR "stripped_html ILIKE '%"+line[0]+" '+line[1]+"/:%')" 
"OR "stripped_html ILIKE '%"+line[0]+" '+line[1]+"/:*') AND "'", line[2])

Annex
Annex

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

xxvi
Annex

streetname = s.sub('"', StreetName) # fügt einen weiteren quote ' hinzu um den ersten im to_tsquery und ilike zu escapen
row3 = p.sub(' & ', StreetName) # ersetzt Leerzeichen in den Straßen namen mit ' & ' sonst gehen sie nicht durch to_tsquery
row4 = p.sub(' & ', StreetNumber) # ersetzt Leerzeichen in den Straßennummern namen mit ' & ' bsp.: "Objekt 11" wird zu "Objekt & 11" sonst gehen sie nicht durch to_tsquery

lines.remove(row)
lines.insert(0, (StreetName, StreetNumber, ID, row3, row4))

return lines

def CreateIndex():
    conn, cur = DBConnect()
    cur.execute("ALTER TABLE html ADD COLUMN textsearchable_index_col tsvector;")
    conn.commit()
    cur.execute("UPDATE html SET textsearchable_index_col = to_tsvector('german', stripped_html) WHERE Vienna = True;")
    conn.commit()
    cur.execute("CREATE INDEX textsearch_idx ON html USING gin(textsearchable_index_col);")
    conn.commit()
    cur.close()
    conn.close()
    Starttime = time.time()
    CreateTables()
    addcolumn()
    print("creating Index")
    CreateIndex()
    print("Make Addresses Unique")
    MakeAddressesUnique()
    lines = ReadFromTableAddressesUnique()
    print lines
    lines = CleanStrings(lines)
    print lines
    print(len(lines)
    SQLStatmentdict = ConstructSQLStatmentSearchAddresses(lines)
    print SQLStatmentdict
    JoinAddressesUniqueWithURL(SQLStatmentdict)

    print("+++++########+++++")
    print("complete Operation took %s Minutes" % ((time.time() - Starttime) / 60))
    print("+++++########+++++")

Find Links

# -*- coding: utf-8 -*-
import psycopg2
import time
import re
import urlparse

def DBConnect():
    conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=###########")
    cur = conn.cursor()
    return conn, cur

def GetGeocodedHTMLIDs(conn, cur):
    cur.execute("SELECT HTMLID, AddressesUniqueID FROM AddressesUniqueJoinedWithURL WHERE Original = TRUE")
    data = cur.fetchall()
Annex

017 datadict = {}
018 for row in data:
019     if row[0] in datadict:
020         datadict[row[0]] = [row[1],]
021     else:
022         datadict[row[0]] = [row[1],]
023 return datadict
024
025 def FindLinksInHtml(conn, cur, offset):
026     NoWhiteSpace = re.compile(r' ')
027     loadingtime = time.time()
028     cur.execute("SELECT id,url,html_file FROM html WHERE id > %s AND id <= %s ORDER BY id;", (offset, offset+limit))
029     data = cur.fetchall()
030     regextime = time.time()
031     passeslist = []
032     linklist = []
033     for row in data:
034         links = re.findall(r'href=["']?((["']>|\?)\)+', row[2])
035         for link in links:
036             try:
037                 linklist.append((row[0], NoWhiteSpace.sub('%20', urlparse.urljoin(row[1], link))))  # makes the links absolut and removes Whitespaces
038             except:
039                 passeslist.append((row[1], link))
040     print('Regex took %.2f Minutes found links %s' % ((time.time() - regextime) / 60), len(linklist))
041 return linklist, passeslist
042
043 def URLsWithID(conn, cur):
044     cur.execute("SELECT URL,ID FROM html;")
045     data = cur.fetchall()
046     dictionary = dict(data)
047 return dictionary
048
049 def WriteToAddressesUniqueJoinedWithURL(List):
050     conn, cur = DBConnect()
051     args_str = ','.join(cur.mogrify("(%s,%s,FALSE)", x) for x in List)
052     try:
053         cur.execute("INSERT INTO AddressesUniqueJoinedWithURL (HTMLID, AddressesUniqueID, Original)VALUES " + args_str)
054     except:
055         print "Error Inserting Joins"
056     conn.commit()
057     conn.close()
058 return
059
060 def get_html_ids():
061     conn, cur = DBConnect()
062     cur.execute("SELECT id FROM html")
063     data = cur.fetchall()
064     cur.close()
065     conn.close()
066 return data
067
068 conn, cur = DBConnect()
069 Starttime = time.time()
070 print('getting URLDictonary and URLIDWITHAddressIDDictonary')
071 URLDictonary = URLsWithID(conn, cur)
072 URLIDWITHAddressIDDictonary = GetGeocodedHTMLIDs(conn, cur)
073 limit = 1000
074
075 limit = 1000
076
077 print('getting URLDictonary and URLIDWITHAddressIDDictonary')
078 URLDictonary = URLsWithID(conn, cur)
079 URLIDWITHAddressIDDictonary = GetGeocodedHTMLIDs(conn, cur)
def POStag(filepath):
    global i
    global len_filelist
    i += 1
    newcorpusstaged = []
    Starttime2 = time.time()
    with io.open(filepath, 'r', encoding='utf-8') as mfile:
        data = mfile.read()
        data = data.splitlines()
    for section in data:
        section = parse(section)
```
with io.open('./wikicorpuspickeld_2/%s_%s.pos' % (current_thread().ident, i,), 'wb') as fout:
    pickle.dump(newcorpustagged, fout)

print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
delta_time = time.time() - Starttime
print('time till now %.2f Minutes' % (delta_time / 60))
print('time till end %.2f Minutes' % (((delta_time / 60) / (i * 4)) * (len_filelist - (i * 4))))

def createfilepathlist():
    pathlist = []
    subfolders = [x[0] for x in os.walk('./WikiText/')] for subfolder in subfolders[1:]:
        for filename in os.listdir(subfolder):
            pathlist.append(subfolder + '/' + filename)
    return pathlist

i = 0
filepathlist = createfilepathlist() len_filelist = len(filepathlist)
Starttime = time.time()
if __name__ == '__main__':
    print('Tagging new corpus')
    pool = ThreadPool(4)
    pool.map(POStag, filepathlist)
    pool.close()
    pool.join()
    print('+++++########+++++')
    print('complete Operation took %s Minutes' % ((time.time() - Starttime) / 60))
    print('+++++########+++++')

Co-occurrence Group Generation
```

```
# -*- coding: utf-8 -*-
import time
from nltk.corpus import stopwords
import os
import io
import cPickle as pickle

def NounVerb(tag):
    noun_verb_list = ['u.nn', 'u.nnsg', 'u.nnps', 'u.vb', 'u.vbz', 'u.vbp', 'u.vbd', 'u.vbn', 'u.vbg']
    if tag.lower() in noun_verb_list:
        return True
    return False

def stopwords_list():
    new_list = []
    for word in stopwords.words('german'):
        new_list.append(unicode(word.decode('latin-1')))
    return new_list
```
def CoOccurrence(groups):
    Starttime3 = time.time()
    Fenster = 10
    i = 1
    S_list = stopwords_list()
    word_dict = {}
    Files = [x[2] for x in os.walk('./wikicorpuspickeld_2/')]
    for file in Files[0]:
        with io.open('./wikicorpuspickeld_2/'+file, 'rb') as fin:
            loaded_corpus = pickle.load(fin)

    for sentence in loaded_corpus:
        for (index, tokentag) in enumerate(sentence):
            token = tokentag
            token = token.lower()

            if token in groups:
                term = sentence[index-Fenster:index+Fenster]
                for (term_token, term_tag) in term:
                    term_token = term_token.lower()
                    if term_token not in S_list and NounVerb(term_tag):
                        if token not in word_dict:
                            word_dict[token] = {}
                        if term_token in word_dict[token]:
                            word_dict[token][term_token] += 1
                        else:
                            word_dict[token][term_token] = 1
            print i

    delta_time = time.time() - Starttime3
    print "time till end %.2f Minutes" % (((delta_time/60)/i)*(len(Files[0])-i))
    i = i+1
    return word_dict

groups = [u'wohnen', u'arbeiten', u'bildung', u'einkaufen', u'gaststätte', u'hotel', u'kreditinstitut', u'kultur', u'dienstgebäude',]
Starttime2 = time.time()
CoOccurrenceGroups = CoOccurrence(groups)
directory = './topics/
if not os.path.exists(directory):
    os.makedirs(directory)
with io.open(directory+'CoOccurrenceGroups.pickle', 'wb') as fout:
    pickle.dump(CoOccurrenceGroups, fout)
print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
delta_time = time.time() - Starttime
import psycopg2

c = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=#####")
cur = c.cursor()

return c, cur

def countrows():
    c, cur = DBConnect()
    cur.execute("SELECT count(*) FROM HtmlUnique;")
    data = cur.fetchone()
    cur.close()
    c.close()

    return data[0]

if __name__ == "__main__":
    with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'wb') as fout:
        pickle.dump(CombinedVectorSpaceIDFTKeyList, fout)

Wikipedia Vector Space

# -*- coding: utf-8 -*-

import time
from nltk.corpus import stopwords
import nltk
import re
import os
import io
import cPickle as pickle

def Vector_Calculator():
    Starttime3 = time.time()
    i = 1
    GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
    token_dict_file = {}
    p = re.compile(ur'^[a-zA-Z\u00e0-\u00ff]([a-zA-Z\u00e0-\u00ff]*)$', re.UNICODE)
    Files = [x[2] for x in os.walk('E:/Tools/Topic Generation/wikicorpuspickeld_2/')]
    for file in Files:
        with io.open('E:/Tools/Topic Generation/wikicorpuspickeld_2/' + file, 'rb') as fin:
            loaded_corpus = pickle.load(fin)

            for sentence in loaded_corpus:
                for (index, tagtuple) in enumerate(sentence):
                    (token, tag) = tagtuple
                    token = token.lower()
                    if token not in stopword_list:
                        if p.match(token):
                            stemmedtoken = GermanStemmer.stem(token)
```python
if stemmedtoken in token_dict_file:
    token_dict_file[stemmedtoken] += 1
else:
    token_dict_file[stemmedtoken] = 1

delta_time = time.time() - Starttime3
print "time till end %.2f Minutes" % (((delta_time/60)/i)*(len(Files[0])-i))
i += 1

return token_dict_file

stopword_list = []
for word in stopwords.words('german'):
    stopword_list.append(unicode(word.decode('latin-1')))

Starttime = time.time()
Vectorraum = Vector_Calculator()

with io.open('/Vector/WikiVectorSpace2.pickle', 'wb') as fout:
    pickle.dump(Vectorraum, fout)
print('Operation took %.2f Minutes' % ((time.time() - Starttime) / 60))
```

**HTML Vector Space**

```python
# -*- coding: UTF-8 -*-
import nltk
from nltk.tokenize import RegexpTokenizer
import psycopg2
import time
import cPickle as pickle
import io
import re

def DBConnect():
    conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=##########")
    cur = conn.cursor()
    return conn, cur

def stripped_htmls(ID_list):
    conn, cur = DBConnect()
    data = []
    for ID in ID_list:
        cur.execute("SELECT stripped_html FROM HtmlUnique WHERE ID = %s", (ID))
        html = cur.fetchone()
        if html:
            data.append(html)
        else:
            error += 1
    print "Number of Errors %s" % error
    cur.close()
    conn.close()
    return data

def get_html_ids():
    conn, cur = DBConnect()
    cur.execute("select DISTINCT HTMLID FROM AddressesUniqueJoinedWithURL ORDER BY HTMLID")
    data = cur.fetchall()
    cur.close()
    conn.close()
    return data

def Delete_stopwords(Tokens):
```
return [token for token in Tokens if not token in nltk.corpus.stopwords.words('german')]

GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
tokenizer = RegexpTokenizer(r'\\w+')
token_dict = {}

HTMLIDS = get_html_ids()

lower = 0
upper = lower + 1000
Starttime = time.time()
parsedhtmls = 0

while lower <= len(HTMLIDS):
    Starttime2 = time.time()
    stripped_htmls_list = stripped_htmls(HTMLIDS[lower:upper])
    for html in stripped_htmls_list:
        parsedhtmls += 1
        time_tokenize = time.time()
        tokens = tokenizer.tokenize(html)
        tokens = Delete_stopwords(tokens)
        token_dict_file = {}

        for token in tokens:
            stemmedtoken = GermanStemmer.stem(token)
            if stemmedtoken in token_dict_file:
                token_dict_file[stemmedtoken] += 1
            else:
                token_dict_file[stemmedtoken] = 1

        for key in token_dict_file:
            if key in token_dict:
                doc_count = token_dict[key][0] + 1
                occurrence_count = token_dict[key][1] + token_dict_file[key]
                token_dict[key] = (doc_count, occurrence_count)
            else:
                token_dict[key] = (1, token_dict_file[key])

        print('Number of tokens in dict: %s' % len(token_dict))
        print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
        delta_time = time.time() - Starttime
        print "time till now %.2f Minutes"%(delta_time / 60)
        print "time till end %.2f Minutes"%(((delta_time/60)/(upper))*(len(HTMLIDS)-(upper)))
        lower += 1000
        upper = lower + 1000

with io.open('./Vector/HTMLVectorSpace.pickle', 'wb') as fout:
    pickle.dump(token_dict, fout)

print 'parsed HTML Files %s' % parsedhtmls

Combined Vector Space

import io
import cPickle as pickle

with io.open('./Vector/WikiVectorSpace.pickle', 'rb') as fin:
    WikiVectorSpace = pickle.load(fin)

with io.open('./Vector/HTMLVectorSpace.pickle', 'rb') as fin:
    HTMLVectorSpace = pickle.load(fin)

CombinedVectorSpace = {}

for key in WikiVectorSpace:
    if key in HTMLVectorSpace:
        CombinedVectorSpace[key] = HTMLVectorSpace[key]
with io.open('./Vector/CombinedVectorSpace.pickle', 'wb') as fout:
pickle.dump(CombinedVectorSpace, fout)

HTML Tokenization

```python
def DBConnect():
    conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=#######")
    cur = conn.cursor()
    return conn, cur
def createColumn():
    conn, cur = DBConnect()
    cur.execute("ALTER TABLE HtmlUnique DROP COLUMN IF EXISTS VectorDICT;")
    conn.commit()
    cur.execute("ALTER TABLE HtmlUnique ADD COLUMN VectorDICT bytea;")
    conn.commit()
    cur.close()
    conn.close()
def ReadFromHTML(offset):
    conn, cur = DBConnect()
    cur.execute("SELECT id,stripped_html FROM ORDER BY id limit 1000 offset %s ;", (offset,))
    data = cur.fetchall()
    conn.close()
    return data
def Delete_stopwords(Tokens):
    return [token for token in Tokens if not token in nltk.corpus.stopwords.words('german')]
def UpdateHtmlUniquewithDict(Dict, ID):
    conn, cur = DBConnect()
    cur.execute("UPDATE HtmlUnique SET VectorDICT = %s WHERE id = %s", (psycopg2.Binary(Dict), ID))
    cur.close()
    conn.close()
def countrows():
    conn, cur = DBConnect()
    cur.execute("select count(id) from HtmlUnique;")
    data = cur.fetchone()
    conn.close()
    return data[0]
```
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  while htmls:
070      print(len(htmls))
071      Starttime2 = time.time()
072      for html in htmls:
073          HTMLdict = {}
074          id, HTMLtext = html
075          tokens = tokenizer.tokenize(HTMLtext)
076          for token in tokens:
077              stemmedtoken = GermanStemmer.stem(token)
078              if stemmedtoken in CombinedVectorSpace:
079                  if stemmedtoken in HTMLdict:
080                      HTMLdict[stemmedtoken] += 1
081                  else:
082                      HTMLdict[stemmedtoken] = 1
083          htmlDictpickled = pickle.dumps(HTMLdict)
084          UpdateHtmlUniquewithDict(htmlDictpickled, id)
085      offset += 1000
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))
090      htmls = 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  def DBConnect():
008      conn = psycopg2.connect("dbname=TEST_DB user=postgres password=######")
009      cur = conn.cursor()
010      return conn, cur
011 012  def countrows():
013      conn, cur = DBConnect()
014      cur.execute("SELECT count(*) FROM HtmlUnique;")
015      data = cur.fetchone()
016      cur.close()
017      conn.close()
018      return data[0]
019 020  def VectorDICTReader(offset):
021      conn, cur = DBConnect()
022      cur.execute("SELECT id,VectorDICT FROM HtmlUnique order by id limit 100 offset %s ;","(offset,)")
023      data = cur.fetchall()
024      cur.close()
025      conn.close()
026      return data
027 028  def UpdateHtmlUniquewithTFIDFlist(Values):
029      conn, cur = DBConnect()
030      cur.executemany("UPDATE HtmlUnique SET TFIDFVector = %s WHERE id = %s", Values)
031      conn.commit()
032      cur.close()
033      conn.close()
def createColumn():
    conn, cur = DBConnect()
    cur.execute("ALTER TABLE HtmlUnique DROP COLUMN IF EXISTS TFIDFVector;")
    conn.commit()
    cur.execute("ALTER TABLE HtmlUnique ADD COLUMN TFIDFVector bytea;")
    conn.commit()
    cur.close()
    conn.close()

offset = 0
length = countrows()
Starttime = time.time()
createColumn()

with io.open('./Vector/CombinedVectorSpaceIDFT.pickle', 'rb') as fin:
    CombinedVectorSpaceIDFT = pickle.load(fin)

with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'rb') as fin:
    CombinedVectorSpaceIDFTKeyList = pickle.load(fin)

while offset <= length:
    Starttime2 = time.time()
    arraylist = []
    for tuple in dicts:
        array = []
        id = tuple[0]
        dictionary = pickle.loads(str(tuple[1]))
        for key in CombinedVectorSpaceIDFTKeyList:
            if key in dictionary:
                array.append(numpy.multiply(CombinedVectorSpaceIDFT[key][2],
                                             dictionary[key]))
            else:
                array.append(0)
        array = numpy.array(array)
        array = numpy.divide(array, numpy.linalg.norm(array))
        array = zlib.compress(array)
        arraylist.append((psycopg2.Binary(array), id,))

UpdateHtmlUniqueWithTFIDFlist(arraylist)
offset += 100
print("Operation took %.2f Minutes" % ((time.time() - Starttime2) / 60))
delta_time = time.time() - Starttime
print("time till now %.2f Minutes" % (delta_time / 60))
print("time till end %.2f Minutes" % (((delta_time/60)/offset)*(length-offset))

import cPickle as pickle
import io
import nltk
import numpy
import zlib

with io.open('./Vector/CombinedVectorSpaceIDFT.pickle', 'rb') as fin:
    CombinedVectorSpaceIDFT = pickle.load(fin)

with io.open('./Vector/CombinedVectorSpaceIDFTKeyList.pickle', 'rb') as fin:
    CombinedVectorSpaceIDFTKeyList = pickle.load(fin)

with io.open('./Co-Occurrence.pickle', 'rb') as fin:
    CoOC = pickle.load(fin)

GermanStemmer = nltk.stem.SnowballStemmer('german', ignore_stopwords=True)
Cosine Similarity

```
import cPickle as pickle
import io
import psycopg2
import time
import zlib
import io
import cPickle

import sklearn.metrics.pairwise

def DBCConnect():
    conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=######")
    cur = conn.cursor()
    return conn, cur

def VectorDICReader(range, offset):
    conn, cur = DBCConnect()
    cur.execute("\nSELECT id,TFIDFVector FROM HtmlUnique order by id limit %s offset %s ;\n\n", (range, offset,))
    data = cur.fetchall()
    cur.close()
    conn.close()
    return data
```

```
import cPickle as pickle
import io
import psycopg2
import time
import zlib
import io
import cPickle

import sklearn.metrics.pairwise

def DBCConnect():
    conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=######")
    cur = conn.cursor()
    return conn, cur

def VectorDICReader(range, offset):
    conn, cur = DBCConnect()
    cur.execute("\nSELECT id,TFIDFVector FROM HtmlUnique order by id limit %s offset %s ;\n\n", (range, offset,))
    data = cur.fetchall()
    cur.close()
    conn.close()
    return data
```
def createColumn(column):
    conn, cur = DBConnect()
    sqlstring = "ALTER TABLE HtmlUnique DROP COLUMN IF EXISTS %s;" % column
    cur.execute(sqlstring)
    conn.commit()
    sqlstring = "ALTER TABLE HtmlUnique ADD COLUMN %s FLOAT;" % column
    cur.execute(sqlstring)
    conn.commit()
    cur.close()
    conn.close()

def Sqlstringconstructor(Columnlist):
    sqlstring = "UPDATE HtmlUnique SET 
    for key in Columnlist[:-1]:
        sqlstring += key + ' = %s, '
    sqlstring += Columnlist[-1] + ' = %s' 
    print(sqlstring)
    return(sqlstring)

def UpdateHtmlUniquewithCosinelist(sqlstring, Values):
    conn, cur = DBConnect()
    cur.executemany(sqlstring, Values)
    conn.commit()
    cur.close()
    conn.close()

countrows():
    conn, cur = DBConnect()
    cur.execute("SELECT count(*) FROM HtmlUnique;")
    data = cur.fetchone()
    conn.close()
    return(data[0])

with io.open('./Vector/TFIDF_CoOc.pickle', 'rb') as fin: 
    TFIDF_CoOc = pickle.load(fin)
    Columnlist = []
    SearchTermList = []
    TFIDFworkingdict = {}
    for searchterm in TFIDF_CoOc:
        SearchTermList.append(searchterm)
        Columnlist.append(TFIDF_CoOc[searchterm]+'_CoOc')
        Columnlist.append(TFIDF_CoOc[searchterm]+'_ST')
    for searchterm in SearchTermList:
        TFIDFworkingdict[searchterm] = 
pickle.loads(pickle.loads(TFIDF_CoOc[searchterm]['TFIDF_CoOc'])['TFIDF_ST'])
        for searchterm in Columnlist:
            createColumn(searchterm)
            sqlstring = Sqlstringconstructor(Columnlist)
    offset = 0
    range = 1000
    length = countrows()
    vectors = VectorDICTReader(range, offset)
    while vectors:
        Starttime = time.time()
        updateList = []
        for vectortup in vectors:
            pass
        print(Starttime2 - Starttime)
        break
Annex

cosinelist = []
id, vector = vectortup[0], pickle.loads(zlib.decompress(vectortup[1]))
for searchterm in SearchTermList:
    cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][0], vector), 8)
    cosinelist.append(cosine)
    cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][1], vector), 8)
    cosinelist.append(cosine)
    cosinelist.append(id)
update_list.append(cosinelist)
offset += range
print 'Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60)
delta_time = time.time() - Starttime
print "time till now %.2f Minutes"%(delta_time / 60)
print "time till end %.2f Minutes"*((delta_time/60)/offset)*(length-offset)
vector = VectorDICTReader(range, offset)

Address Classification

import cPickle as pickle
import io
import psycopg2
import time
import zlib
from sklearn.metrics.pairwise import cosine_similarity

def DBConnect():
    conn = psycopg2.connect("dbname=Master_DB_spatial2 user=postgres password=######")
    cur = conn.cursor()
    return conn, cur

def VectorDICTReader(range, offset):
    conn, cur = DBConnect()
    cur.execute("SELECT id,TFIDFVector FROM HtmlUnique order by id limit %s offset %s ;", (range, offset,))
    data = cur.fetchall()
    cur.close()
    conn.close()
    return data

def createColumn(column):
    conn, cur = DBConnect()
    sqlstring = "ALTER TABLE HtmlUnique DROP COLUMN IF EXISTS %s; " % column
    cur.execute(sqlstring)
    conn.commit()
    sqlstring = "ALTER TABLE HtmlUnique ADD COLUMN %s FLOAT; " % column
    cur.execute(sqlstring)
    conn.commit()
    cur.close()
    conn.close()

def SQLstringconstructor(Columnlist):
    sqlstring = "UPDATE HtmlUnique SET 
    for key in Columnlist[:-1]:
        sqlstring += key+' = %s', '  
    sqlstring += Columnlist[-1]+' = %s'
    sqlstring += ' WHERE id = %s'
    print sqlstring
    return sqlstring

def UpdateHtmlUniquewithCosinelist(sqlstring, Values):
    conn, cur = DBConnect()
    cur.executemany(sqlstring, Values)
    conn.commit()
cur.close()
close()

```
def countrows():
    conn, cur = DBConnect()
    cur.execute("SELECT count(*) FROM HtmlUnique;")
data = cur.fetchone()
close()
close()
return data[0]
```

```
with io.open('./Vector/TFIDF_CoOc.pickle', 'rb') as fin:
    TFIDF_CoOc = pickle.load(fin)

Columnlist = []
SearchTermList = []
TFIDFworkingdict = {}
for searchterm in TFIDF_CoOc:
    SearchTermList.append(searchterm)
    Columnlist.append(TFIDF_CoOc[searchterm]+'_CoOc')
    Columnlist.append(TFIDF_CoOc[searchterm]+'ST')
for searchterm in SearchTermList:
    pickle.loads(zlib.decompress(TFIDF_CoOc[searchterm]["TFIDF_CoOc"])),
    pickle.loads(zlib.decompress(TFIDF_CoOc[searchterm]["TFIDF_ST"]))
for searchterm in Columnlist:
    createColumn(searchterm)
sqlstring = Sqlstringconstructor(Columnlist)
offset = 0
range = 1000
length = countrows()
vectors = VectorDICTReader(range,offset)
Starttime = time.time()
while vectors:
    Starttime2 = time.time()
    updatelist = []
    for vectortup in vectors:
        cosinelist = []
        id, vector = vectortup[0], pickle.loads(zlib.decompress(vectortup[1]))
        for searchterm in SearchTermList:
            cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][0], vector),8)
            cosinelist.append(cosine)
            cosine = round(cosine_similarity(TFIDFworkingdict[searchterm][1], vector),8)
            cosinelist.append(cosine)
        cosinelist.append(id)
        updatelist.append(cosinelist)
    UpdateHtmlUniquewithCosinelist(sqlstring,updatelist)
    offset += range
    print('Operation took %.2f Minutes' % ((time.time() - Starttime2) / 60))
    delta_time = time.time() - Starttime
    print "time till now %.2f Minutes"%(delta_time / 60)
    print "time till end %.2f Minutes"%((delta_time/60)/(length-offset))
    vectors = VectorDICTReader(range,offset)
## Mapping Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Street</th>
<th>street_number</th>
<th>post_code</th>
<th>hotel</th>
<th>gaststätte</th>
<th>arbeiten</th>
<th>bildung</th>
<th>wohnen</th>
<th>dienstgebäude</th>
<th>kreditinstitut</th>
<th>einkaufen</th>
<th>kultur</th>
<th>other</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>Aegidigasse</td>
<td>7-11</td>
<td>1060</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>Albertgasse</td>
<td>1a</td>
<td>1080</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rechtsanwalt, Immobilien,</td>
<td>PR-Agentur, edv, Ärzte Flugambulanz</td>
</tr>
<tr>
<td>329</td>
<td>Alser Straße</td>
<td>71</td>
<td>1080</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pensionistenclub</td>
<td></td>
</tr>
<tr>
<td>440</td>
<td>Am Hof</td>
<td>13</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Gewürz geschafft, Kosmetik</td>
<td>Geschäft, Consulting, Medienproduktion</td>
</tr>
<tr>
<td>707</td>
<td>Argentinierstraße</td>
<td>14</td>
<td>1040</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Griechische Botschaft</td>
<td></td>
</tr>
<tr>
<td>724</td>
<td>Argentinierstraße</td>
<td>30</td>
<td>1040</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Funkhaus</td>
<td></td>
</tr>
<tr>
<td>725</td>
<td>Argentinierstraße</td>
<td>30A</td>
<td>1040</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Funkhaus, kaffee</td>
<td></td>
</tr>
<tr>
<td>732</td>
<td>Argentinierstraße</td>
<td>39</td>
<td>1040</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Rumänisches Kulturinstitut, Ingenieurbüro</td>
<td></td>
</tr>
<tr>
<td>834</td>
<td>Augasse</td>
<td>11</td>
<td>1090</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Supermarkt</td>
<td></td>
</tr>
<tr>
<td>849</td>
<td>Augustinerstraße</td>
<td>1</td>
<td>1010</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Museum, restaurant, museum geschäft</td>
<td></td>
</tr>
<tr>
<td>918</td>
<td>Bacherplatz</td>
<td>11</td>
<td>1050</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Film und Design verleih</td>
<td></td>
</tr>
<tr>
<td>933</td>
<td>Bäckerstraße</td>
<td>10</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Schmuckgeschäft</td>
<td></td>
</tr>
<tr>
<td>1245</td>
<td>Baumgasse</td>
<td>83</td>
<td>1030</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Hotel</td>
<td></td>
</tr>
<tr>
<td>1297</td>
<td>Bechardgasse</td>
<td>18</td>
<td>1030</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1372</td>
<td>Bennogasse</td>
<td>1</td>
<td>1080</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Bank, post</td>
<td></td>
</tr>
<tr>
<td>1373</td>
<td>Bennogasse</td>
<td>10</td>
<td>1080</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Arzt, edv</td>
<td></td>
</tr>
<tr>
<td>1440</td>
<td>Berggasse</td>
<td>32</td>
<td>1090</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Schuhservice, Vorbeugung von sexuellem Missbrauch von Kindern</td>
<td></td>
</tr>
<tr>
<td>1952</td>
<td>Bräuhausgasse</td>
<td>37</td>
<td>1050</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>Ingenieurbüro, Steuerberatung, Kindergarten</td>
<td></td>
</tr>
<tr>
<td>Nummer</td>
<td>Straße</td>
<td>Hausnummer</td>
<td>Postleitzahl</td>
<td>Wohnungsnummer</td>
<td>Geschäftsbetrieb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
<td>------------</td>
<td>--------------</td>
<td>----------------</td>
<td>--------------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2084</td>
<td>Brigittaplatz</td>
<td>18</td>
<td>1200</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Aquarien Fachgeschäft, Elektrik Fachgeschäft, psychotherapie, Heilpraktiker</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2091</td>
<td>Brigittaplatz</td>
<td>9</td>
<td>1200</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Küchenmöbel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2262</td>
<td>Burggasse</td>
<td>28-32</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Werbeagentur, Fotografie, Film, Kommunikation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2288</td>
<td>Burggasse</td>
<td>56</td>
<td>1070</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fotografie</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2312</td>
<td>Burggasse</td>
<td>81</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hochzeitsgeschäft, Schneider, Bäcker, Gemüse Händler, Café</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2576</td>
<td>Daffingerstraße</td>
<td>1</td>
<td>1030</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2677</td>
<td>Dannebergplatz</td>
<td>12</td>
<td>1030</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2816</td>
<td>Diehlgasse</td>
<td>28</td>
<td>1050</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3009</td>
<td>Dorotheergasse</td>
<td>2</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Restaurant, Architekt,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3036</td>
<td>Dr.-Karl-Renner-Ring</td>
<td>3</td>
<td>1010</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Parlament</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3114</td>
<td>Dresdnerstraße</td>
<td>68</td>
<td>1200</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Baugesellschaft, Pipelines, EDV, Arzt, Telefon, Ingenieur, Reisebüro</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3131</td>
<td>Dresdnerstraße</td>
<td>87</td>
<td>1200</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MA 42, österreichische Patentamt, hotel management, trafik</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3195</td>
<td>Ebendorferstrasse</td>
<td>7</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Österreichischer Austauschdienst, wohnraumverwaltung GmbH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3227</td>
<td>Einsiedlergasse</td>
<td>19</td>
<td>1050</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Appartementhaus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3296</td>
<td>Elisabethstrasse</td>
<td>13</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Arzt, Physiotherapie, rechtsanwalt, Fischerei Gesellschaft, Model Management, Immobilien, Kunden Forschung, Marketing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3709</td>
<td>Erzherzog-Johann-Platz</td>
<td>1</td>
<td>1040</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Universität</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3724</td>
<td>Eslargasse</td>
<td>11</td>
<td>1030</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nummer</td>
<td>Straße</td>
<td>Hausnummer</td>
<td>Postleitzahl</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>t4</td>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>----------------------</td>
<td>------------</td>
<td>--------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3774</td>
<td>Esteplatz</td>
<td>5</td>
<td>1030</td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3857</td>
<td>Fasangasse</td>
<td>35-37</td>
<td>1030</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3862</td>
<td>Fasangasse</td>
<td>4</td>
<td>1030</td>
<td>t</td>
<td></td>
<td></td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3941</td>
<td>Favoritenstraße</td>
<td>52</td>
<td>1040</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4122</td>
<td>Fischhof</td>
<td>3</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4145</td>
<td>Fleischmarkt</td>
<td>24</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4173</td>
<td>Florianigasse</td>
<td>16</td>
<td>1080</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4181</td>
<td>Florianigasse</td>
<td>24</td>
<td>1080</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4283</td>
<td>Frankenberggasse</td>
<td>11</td>
<td>1040</td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4411</td>
<td>Franzensgasse</td>
<td>4</td>
<td>1050</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4448</td>
<td>Freyung</td>
<td>4</td>
<td>1010</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4451</td>
<td>Freyung</td>
<td>6a</td>
<td>1010</td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4468</td>
<td>Friedrich-Engels-Platz</td>
<td>21</td>
<td>1200</td>
<td></td>
<td></td>
<td>t</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4490</td>
<td>Friedrichstraße</td>
<td>10</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5115</td>
<td>Gonzagagasse</td>
<td>1</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5169</td>
<td>Graben</td>
<td>7</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5395</td>
<td>Große Pfarrgasse</td>
<td>2</td>
<td>1020</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5503</td>
<td>Große Stadtgutgasse</td>
<td>31</td>
<td>1020</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5520</td>
<td>Grünangergasse</td>
<td>6</td>
<td>1010</td>
<td></td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5751</td>
<td>Gumpendorfer</td>
<td>9</td>
<td>1060</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Annex
<table>
<thead>
<tr>
<th>Straße</th>
<th>Hausnummer</th>
<th>Index</th>
<th>Betrieb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hahngasse</td>
<td>2</td>
<td>1090</td>
<td>t</td>
</tr>
<tr>
<td>Herrengasse</td>
<td>13</td>
<td>1010</td>
<td>t, t</td>
</tr>
<tr>
<td>Herrengasse</td>
<td>7</td>
<td>1010</td>
<td>t</td>
</tr>
<tr>
<td>Himmelpfortgasse</td>
<td>4</td>
<td>1010</td>
<td>t, t</td>
</tr>
<tr>
<td>Hofenedergasse</td>
<td>3</td>
<td>1020</td>
<td>t</td>
</tr>
<tr>
<td>Hollandstraße</td>
<td>15</td>
<td>1020</td>
<td>t, t</td>
</tr>
<tr>
<td>Hornbostelgasse</td>
<td>16-18</td>
<td>1060</td>
<td>t</td>
</tr>
<tr>
<td>Johann-Böhm-Platz</td>
<td>1</td>
<td>1020</td>
<td>t, t</td>
</tr>
<tr>
<td>Josefsplatz</td>
<td>5</td>
<td>1010</td>
<td>t, t, t</td>
</tr>
<tr>
<td>Josefstädter Straße</td>
<td>7</td>
<td>1080</td>
<td>t, t</td>
</tr>
<tr>
<td>Judenplatz</td>
<td>8</td>
<td>1010</td>
<td>t, t</td>
</tr>
<tr>
<td>Kaiserstraße</td>
<td>26</td>
<td>1070</td>
<td>t</td>
</tr>
<tr>
<td>Karlsplatz</td>
<td>5</td>
<td>1010</td>
<td>t</td>
</tr>
<tr>
<td>Kärntner Ring</td>
<td>42190</td>
<td>1010</td>
<td>t, t</td>
</tr>
<tr>
<td>Kärntner Straße</td>
<td>21-23</td>
<td>1010</td>
<td>t, t</td>
</tr>
<tr>
<td>Kirchengasse</td>
<td>1</td>
<td>1070</td>
<td>t, t</td>
</tr>
<tr>
<td>Kirchengasse</td>
<td>44</td>
<td>1070</td>
<td>t</td>
</tr>
</tbody>
</table>

Kulturmärkte, gemischte Geschäfte

Annex
<table>
<thead>
<tr>
<th>Annexe</th>
<th>Kohlmarkt</th>
<th>11</th>
<th>1010</th>
<th>t</th>
<th>t</th>
<th>t</th>
<th>Zahnarzt, rechtsanwalt, Atelier, Kleidungsgeschäft, Apotheke, Schmuckgeschäft, Parfümerie</th>
</tr>
</thead>
<tbody>
<tr>
<td>8702</td>
<td>Kollerstrasse</td>
<td>14</td>
<td>1030</td>
<td>t</td>
<td>t</td>
<td>Holzgroßhandel</td>
<td></td>
</tr>
<tr>
<td>9167</td>
<td>Lammgasse</td>
<td>8</td>
<td>1080</td>
<td>t</td>
<td>t</td>
<td>Universitäts Institut</td>
<td></td>
</tr>
<tr>
<td>9362</td>
<td>Landstraßer Hauptstraße</td>
<td>8</td>
<td>1030</td>
<td>t</td>
<td>t</td>
<td>Arzt, notar, Möbelgeschäft, Elektronikgeschäft</td>
<td></td>
</tr>
<tr>
<td>9397</td>
<td>Lange Gasse</td>
<td>25</td>
<td>1080</td>
<td>t</td>
<td>t</td>
<td>Schneiderei</td>
<td></td>
</tr>
<tr>
<td>9480</td>
<td>Lassallestraße</td>
<td>7b</td>
<td>1020</td>
<td>t</td>
<td>Parkettgeschäft, Physiotherapeut, Spiegelgeschäft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9514</td>
<td>Laudongasse</td>
<td>26</td>
<td>1080</td>
<td>t</td>
<td>t</td>
<td>Parkettgeschäft, Physiotherapeut, Spiegelgeschäft</td>
<td></td>
</tr>
<tr>
<td>9564</td>
<td>Laudongasse</td>
<td>8</td>
<td>1080</td>
<td>t</td>
<td>t</td>
<td>Hotel, Restaurant</td>
<td></td>
</tr>
<tr>
<td>9792</td>
<td>Leopold-Rister-Gasse</td>
<td>5</td>
<td>1050</td>
<td>t</td>
<td>Beauty, Consulting, Arzt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10411</td>
<td>Linke Wienzeile</td>
<td>102</td>
<td>1060</td>
<td>t</td>
<td>t</td>
<td>Restaurants, Beratungszentrum, Kulturzentrum</td>
<td></td>
</tr>
<tr>
<td>10548</td>
<td>Lothringerstraße</td>
<td>16</td>
<td>1030</td>
<td>t</td>
<td>t</td>
<td>Rechtsanwalt, Consulting, Finanzberatung</td>
<td></td>
</tr>
<tr>
<td>10608</td>
<td>Löwengasse</td>
<td>37</td>
<td>1030</td>
<td>t</td>
<td>t</td>
<td>Second hand Geschäft, Kleidungsgeschäft, Bäcker, Friseur</td>
<td></td>
</tr>
<tr>
<td>10988</td>
<td>Margaretenstraße</td>
<td>33</td>
<td>1040</td>
<td>t</td>
<td>Möbelgeschäft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11006</td>
<td>Margaretenstraße</td>
<td>52</td>
<td>1040</td>
<td>t</td>
<td>Architekt, Restaurant, Kleidungsgeschäft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11118</td>
<td>Mariahilfer Straße</td>
<td>111</td>
<td>1060</td>
<td>t</td>
<td>Kleidungsgeschäft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11123</td>
<td>Mariahilfer Straße</td>
<td>117</td>
<td>1060</td>
<td>t</td>
<td>Restaurant, Beauty, Kleidungsgeschäft, Arzt, Psychotherapie, Consulting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11146</td>
<td>Mariahilfer Straße</td>
<td>22</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td>Restaurant, Kleidungsgeschäft, Technikgeschäft, Bank, Militärische Schule, Militärisches Immobilien Management Zentrum</td>
<td></td>
</tr>
<tr>
<td>Hausnummer</td>
<td>Straße</td>
<td>Landkreis</td>
<td>Postleitzahl</td>
<td>Ort</td>
<td>Hausnummer</td>
<td>Straße</td>
<td>Landkreis</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------</td>
<td>-----------</td>
<td>--------------</td>
<td>-----</td>
<td>------------</td>
<td>------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>11160</td>
<td>Mariahilfer Straße</td>
<td>t</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td>Kleidungsgeschäft, Medien firma, cortical.io</td>
<td></td>
</tr>
<tr>
<td>11165</td>
<td>Mariahilfer Straße</td>
<td>t</td>
<td>1060</td>
<td>t</td>
<td>t</td>
<td>Kleidungsgeschäft, werkzeuggeschäft, Kosmetikgeschäft, Sexgeschäft</td>
<td></td>
</tr>
<tr>
<td>11193</td>
<td>Mariahilfer Straße</td>
<td>t</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td>Rechtsanwalt, Kleidungsgeschäft, orthopädie</td>
<td></td>
</tr>
<tr>
<td>11540</td>
<td>Messeplatz</td>
<td>t</td>
<td>1020</td>
<td>t</td>
<td>t</td>
<td>Messe Wien</td>
<td></td>
</tr>
<tr>
<td>11658</td>
<td>Minoritenplatz</td>
<td>t</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>Bildungsministerium</td>
<td></td>
</tr>
<tr>
<td>11692</td>
<td>Mittersteig</td>
<td>t</td>
<td>1040</td>
<td>t</td>
<td>t</td>
<td>Architekt, IT, Großküche, Sachverständiger</td>
<td></td>
</tr>
<tr>
<td>12125</td>
<td>Naglergasse</td>
<td>t</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>Stahlgrobhandel, Badutensilien geschäft, Filmproduktion</td>
<td></td>
</tr>
<tr>
<td>12283</td>
<td>Neubaugasse</td>
<td>t</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td>Theater, Tanzstudio, kaffeemaschinen Vertrieb</td>
<td></td>
</tr>
<tr>
<td>12289</td>
<td>Neubaugasse</td>
<td>t</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td>Fortbildung, Finanzberatung, zeitarbeit, Consulting, reisen, evd, rechtsanwalt</td>
<td></td>
</tr>
<tr>
<td>12391</td>
<td>Neulinggasse</td>
<td>t</td>
<td>1030</td>
<td>t</td>
<td>t</td>
<td>Botschaft Elfenbeinküste, Consulting, restaurant</td>
<td></td>
</tr>
<tr>
<td>12424</td>
<td>Neustiftgasse</td>
<td>t</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td>Kaffee, Reisebüro</td>
<td></td>
</tr>
<tr>
<td>12438</td>
<td>Neustiftgasse</td>
<td>t</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td>Verlag, PR Agentur, erneuerbare energien, Ingenieur</td>
<td></td>
</tr>
<tr>
<td>12472</td>
<td>Neustiftgasse</td>
<td>t</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td>Eisgeschäft, Kleidungsgeschäft, Tier Erziehung</td>
<td></td>
</tr>
<tr>
<td>12656</td>
<td>Nordbergstraße</td>
<td>t</td>
<td>1090</td>
<td>t</td>
<td>t</td>
<td>Physiotherapie, arzt, restaurant</td>
<td></td>
</tr>
<tr>
<td>12948</td>
<td>Obere Donaustraße</td>
<td>t</td>
<td>1020</td>
<td>t</td>
<td>t</td>
<td>Hundertwasser Geburtstags, friseur, gaststätte</td>
<td></td>
</tr>
<tr>
<td>13126</td>
<td>Opernring</td>
<td>t</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>Opa, restaurant, Bücher und musik geschäft</td>
<td></td>
</tr>
<tr>
<td>13233</td>
<td>Paniglgasse</td>
<td>t</td>
<td>1040</td>
<td>t</td>
<td>t</td>
<td>Restaurant</td>
<td></td>
</tr>
<tr>
<td>13247</td>
<td>Pannaschgasse</td>
<td>t</td>
<td>1050</td>
<td>t</td>
<td>t</td>
<td>Bibliothek</td>
<td></td>
</tr>
<tr>
<td>Nummer</td>
<td>Straße</td>
<td>Hausnummer</td>
<td>Postleitzahl</td>
<td>Ort</td>
<td>Sachen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
<td>------------</td>
<td>--------------</td>
<td>-------</td>
<td>---------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13495</td>
<td>Petersplatz</td>
<td>7</td>
<td>1010</td>
<td></td>
<td>Bank</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13681</td>
<td>Plößlgasse</td>
<td>4</td>
<td>1040</td>
<td></td>
<td>Sprachreisen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13777</td>
<td>Postgasse</td>
<td>19</td>
<td>1010</td>
<td></td>
<td>Arzt, Garage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13833</td>
<td>Praterstraße</td>
<td>14</td>
<td>1020</td>
<td></td>
<td>Kleidungsgeschäft, Büchergeschäft, eltern treff.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13939</td>
<td>Prinz-Eugen-Straße</td>
<td>16</td>
<td>1040</td>
<td></td>
<td>Botschaft San Marino, Steuerberatung ärzte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14004</td>
<td>Rabengasse</td>
<td>3</td>
<td>1030</td>
<td></td>
<td>Theater, spö</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14022</td>
<td>Radetzkyplatz</td>
<td>2</td>
<td>1030</td>
<td></td>
<td>Restaurant, Zahnarzt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14026</td>
<td>Radetzkystraße</td>
<td>1</td>
<td>1030</td>
<td></td>
<td>Berufsrettung</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14037</td>
<td>Radetzkystraße</td>
<td>2</td>
<td>1030</td>
<td></td>
<td>Gesundheitsministerium, am bäcker, friseur, restaurant, kindergarten</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14049</td>
<td>Radetzkystraße</td>
<td>3</td>
<td>1030</td>
<td></td>
<td>Gaststätte, bäckerei</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14247</td>
<td>Rathausstraße</td>
<td>11</td>
<td>1010</td>
<td></td>
<td>Restaurant, Cafe, rechtsanwalt, Consulting, Steuerberater, edv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14526</td>
<td>Reisnerstraße</td>
<td>40</td>
<td>1030</td>
<td></td>
<td>Rechtsanwälte, consulting, sport, Inkasso, mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14580</td>
<td>Rembrandtstraße</td>
<td>5</td>
<td>1020</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14610</td>
<td>Rennweg</td>
<td>16</td>
<td>1030</td>
<td></td>
<td>Hotel, restaurant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14644</td>
<td>Rennweg</td>
<td>51</td>
<td>1030</td>
<td></td>
<td>Hotel, restaurant, bar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14700</td>
<td>Resselgasse</td>
<td>4</td>
<td>1040</td>
<td></td>
<td>Universität</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14958</td>
<td>Rotenkreuzgasse</td>
<td>11</td>
<td>1020</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14977</td>
<td>Rotenlöwengasse</td>
<td>19</td>
<td>1090</td>
<td></td>
<td>Steuerberatung, consulting, medizinprodukte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15129</td>
<td>Rudolfsplatz</td>
<td>2</td>
<td>1010</td>
<td></td>
<td>Bar, sport, werbung, Immobilien,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15275</td>
<td>Salmgasse</td>
<td>8</td>
<td>1030</td>
<td></td>
<td>Ingenieur</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15575</td>
<td>Schiffamsgasse</td>
<td>11</td>
<td>1020</td>
<td></td>
<td>Kunstraum, beauty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15626</td>
<td>Schimmelgasse</td>
<td>42222</td>
<td>1030</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15825</td>
<td>Schnirchgasse</td>
<td>14</td>
<td>1030</td>
<td></td>
<td>Gesundheitsdienst stadt wien</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

xlix
<table>
<thead>
<tr>
<th>Nummer</th>
<th>Straße</th>
<th>Postleitzahl</th>
<th>Postleitzahl</th>
<th>t</th>
<th>t</th>
<th>t</th>
<th>t</th>
<th>Botschaft belgien</th>
</tr>
</thead>
<tbody>
<tr>
<td>16040</td>
<td>Schönburgstraße</td>
<td>42285</td>
<td>1040</td>
<td></td>
<td></td>
<td>t</td>
<td>t</td>
<td>Gaststätte, bar, arzt, immobilien, gebäudereinigung</td>
</tr>
<tr>
<td>16045</td>
<td>Schönlaterngasse</td>
<td>11</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>Kaffee, Kleidungsgeschäft, architekt, arzt, Consulting</td>
</tr>
<tr>
<td>16096</td>
<td>Schottenfeldgasse</td>
<td>42096</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>Business center</td>
</tr>
<tr>
<td>16106</td>
<td>Schottenring</td>
<td>17</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>Versicherung, eisenwarengeschäft, münzgeschäft, bank, Honorarkonsulat von rumänien, Consulting, Finanzberatung,</td>
</tr>
<tr>
<td>16233</td>
<td>Schreygasse</td>
<td>14</td>
<td>1020</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td>Beauty, Eventmanagement</td>
</tr>
<tr>
<td>16273</td>
<td>Schrotzbergstraße</td>
<td>6</td>
<td>1020</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td>Hotel, Bäcker, Arzt, Steuerberater, investment Beratung,</td>
</tr>
<tr>
<td>16304</td>
<td>Schubertring</td>
<td>11</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>Rechtsanwalt, französische OSZE, französische UN, Privat Bank, Arzt, super markt</td>
</tr>
<tr>
<td>16508</td>
<td>Schwarzenbergplatz</td>
<td>16</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>Albert-schweitzer-haus</td>
</tr>
<tr>
<td>16561</td>
<td>Schwindgasse</td>
<td>14</td>
<td>1040</td>
<td></td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>Kroatischer Kulturverein, Ingenieurbüro, Eventmanagement</td>
</tr>
<tr>
<td>16606</td>
<td>Sechskrügelgasse</td>
<td>2</td>
<td>1030</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>Growshop, Kleidungsgeschäft, kaffee, sportgeschäft, Pflanzen, Beauty, apotheke</td>
</tr>
<tr>
<td>16687</td>
<td>Seidengasse</td>
<td>27</td>
<td>1070</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td>Growshop, Kleidungsgeschäft, kaffee, sportgeschäft, Pflanzen, Beauty, apotheke</td>
</tr>
<tr>
<td>16709</td>
<td>Seidengasse</td>
<td>9</td>
<td>1070</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td>Verlag, Dachverband soziale Einrichtungen, Consulting, Krankenhaus bedarf, Logistik, customer management, hotel grosshandel, EDV</td>
</tr>
<tr>
<td>16768</td>
<td>Seilerstätte</td>
<td>11</td>
<td>1010</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td>Aluminium Großhandel, Verlag, arzt</td>
</tr>
<tr>
<td>16825</td>
<td>Sensengasse</td>
<td>3</td>
<td>1090</td>
<td>t</td>
<td>t</td>
<td>Ärzte Zentrum, Bibliothek</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17050</td>
<td>Sigmundsgasse</td>
<td>16</td>
<td>1070</td>
<td>t</td>
<td>t</td>
<td>Immobilien, Schmuckdesign</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17064</td>
<td>Simon-Denk-Gasse</td>
<td>42159</td>
<td>1090</td>
<td>t</td>
<td>t</td>
<td>Bibliothek</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17094</td>
<td>Singerstraße</td>
<td>27</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>Rechtsanwalt, Beauty, Consulting,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17193</td>
<td>Sonnenfelsgasse</td>
<td>19</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>Österreichische Akademie der Wissenschaft, Institut für Informationsverarbeitung, Publikums Forschung, Seelsorgezentrum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17952</td>
<td>Stubenring</td>
<td>1</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>Bundesministerium für Wissenschaft Forschung und Wirtschaft, Bundesministerium für Land und Forstwirtschaft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18020</td>
<td>Stumpergasse</td>
<td>65</td>
<td>1060</td>
<td>t</td>
<td>t</td>
<td>Bank</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18179</td>
<td>Taborstraße</td>
<td>81</td>
<td>1020</td>
<td>t</td>
<td>t</td>
<td>Personalvermittlung</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18304</td>
<td>Theobaldgasse</td>
<td>9</td>
<td>1060</td>
<td>t</td>
<td>t</td>
<td>Schmuckgeschäft, Kleidungsgeschäft, bar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18643</td>
<td>Turkenstraße</td>
<td>8</td>
<td>1090</td>
<td>t</td>
<td>t</td>
<td>Architekt, psychologe, Kommunikation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18959</td>
<td>Veithgasse</td>
<td>3</td>
<td>1030</td>
<td>t</td>
<td>t</td>
<td>Serbische orthodoxe Kirche</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18961</td>
<td>Veithgasse</td>
<td>5</td>
<td>1030</td>
<td>t</td>
<td>t</td>
<td>Australische Botschaft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19446</td>
<td>Währinger Straße</td>
<td>68</td>
<td>1090</td>
<td>t</td>
<td>t</td>
<td>Restaurant, elektrofachgeschäft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19478</td>
<td>Wallischgasse</td>
<td>13</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>Tüv Österreich</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19498</td>
<td>Wallensteinplatz</td>
<td>8</td>
<td>1200</td>
<td>t</td>
<td>t</td>
<td>Konzerte, bar, Bank, post</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19610</td>
<td>Wallnerstraße</td>
<td>3</td>
<td>1010</td>
<td>t</td>
<td>t</td>
<td>Kleidungsgeschäft, rechtsanwalt, Consulting, Steuerberater, friseur, vermögensberatung</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19696</td>
<td>Wassergasse</td>
<td>2</td>
<td>1030</td>
<td>t</td>
<td>t</td>
<td>Psychotherapie</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19765</td>
<td>Weggasse</td>
<td>43</td>
<td>1060</td>
<td>t</td>
<td>t</td>
<td>Kaffeegeschäft, arzt, sportstudio, Psychotherapie</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20126</td>
<td>Widerhofergasse</td>
<td>3</td>
<td>1090</td>
<td>t</td>
<td>t</td>
<td>Steuerberatung</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nr.</td>
<td>Straße</td>
<td>Hausnummer</td>
<td>Stadtteil</td>
<td>Telefon</td>
<td>Nutzung</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------</td>
<td>------------</td>
<td>-----------</td>
<td>---------</td>
<td>----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20139</td>
<td>Wiedner Gürtel</td>
<td>16</td>
<td>1040</td>
<td>t</td>
<td>Geschlossenes lokal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20253</td>
<td>Wiedner Hauptstraße</td>
<td>46</td>
<td>1040</td>
<td>t</td>
<td>Rechtsanwalt, Foto Büro</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20448</td>
<td>Wimmergasse</td>
<td>7</td>
<td>1050</td>
<td>t</td>
<td>Hausverwaltung</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20468</td>
<td>Windmühlgasse</td>
<td>15</td>
<td>1060</td>
<td>t</td>
<td>t t Kleidungsgeschäft, blumengeschäft, psychotherapie</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20481</td>
<td>Windmühlgasse</td>
<td>32</td>
<td>1060</td>
<td>t</td>
<td>t t Restaurant, bar, Ingenieur, Künstlermanagement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20838</td>
<td>Ziegelofengasse</td>
<td>37</td>
<td>1050</td>
<td>t</td>
<td>Restaurant, Filmemacher, Ausstellungsgutensilien</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20987</td>
<td>Zirkusgasse</td>
<td>28</td>
<td>1020</td>
<td>t</td>
<td>t Autowerkstatt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21021</td>
<td>Zollergasse</td>
<td>13</td>
<td>1070</td>
<td>t</td>
<td>t Sport, Design, Fotografie, neos</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21049</td>
<td>Zollergasse</td>
<td>5</td>
<td>1070</td>
<td>t</td>
<td>t t Café, Bar, Marketing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21132</td>
<td>Lindengasse</td>
<td>22</td>
<td>1070</td>
<td>t</td>
<td>t t Geschäft, Kleidungsgeschäft, Finanzberatung</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21193</td>
<td>Schönbrunner Straße</td>
<td>47</td>
<td>1050</td>
<td>t</td>
<td>t t Werbeagentur, Maler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21218</td>
<td>Webgasse</td>
<td>6</td>
<td>1060</td>
<td>t</td>
<td>t Beauty</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lebenslauf

Name: Alexander Czech B.Sc.
Geburtsort: Erfurt

Bildungsweg


Relevante Berufserfahrung

06/2012 - Institut für Energiesysteme und Elektrische Antriebe, Technische Universität Wien, Freier GIS-Analytiker / Consultant
10/2008 – 9/2011 Institut für Geographie, Universität Hamburg, Tutor
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