Geocoding User Queries

Dipl.-Inf. Konstantin Clemens
ORCID: 0000-0003-1892-9574

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Dr. Manfred Hauswirth
Gutachter:
   Prof. Dr. Axel Küpper
   Prof. Dr.-Ing. Jochen Schiller
   Prof. Dr.-Ing. David Bermbach

Tag der wissenschaftlichen Aussprache: 10. Dezember 2019

Berlin 2020
Geocoding User Queries

Author: Konstantin Clemens

Supervisors:
Prof. Dr. Axel Küpper
Prof. Dr.-Ing. Jochen Schiller
Prof. Dr.-Ing. David Bermbach
Declaration of Authorship

I, Konstantin Clemens, declare that this thesis titled, “Geocoding User Queries” and the work presented in it are my own. I confirm that:

• This work was done wholly or mainly while in candidature for a research degree at this University.

• Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.

• Where I have consulted the published work of others, this is always clearly attributed.

• Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

• I have acknowledged all main sources of help.

• Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:  

Date:
Abstract

While human users refer to locations using toponyms and addresses, computers rely on latitude and longitude coordinates, or similar, numerical encoding of location information. Geocoding is the process of resolving such toponyms and addresses into machine-friendly numerical encoding. Thus, wherever human users need to specify a location to a computer, geocoding is involved. The process, thereby, is mainly dependent on two aspects. Firstly, the underlying address data needs to be complete, up-to-date, and accurate as otherwise, the geocoding result will not be precise. Secondly, the algorithms that process the user input query, browse candidates within the data and select the right result to the query need to be capable of handling queries of human users.

In this thesis, the specifics of human input addresses are tackled. Various kinds of modifications occur to an address when humans are involved. For example, address formats are not adhered to, superfluous tokens are specified, while required parts of an address are left out of the query. Also, humans abbreviate, escape, or misspell address element names often. The goal of this thesis is to investigate the algorithmic aspect of geocoding systems. Throughout the thesis, ways to process, organize, index, and rank address data are investigated independently of the underlying data. As a result, a method for creating a geocoding system with good precision and recall metrics even in the face of the human factor is developed.

A sequence of experiments is executed that gradually build on top of each other. First, the use of a generic document search engine as the index of a geocoding system is validated. Next, a suitable address model for indexing and searching is selected. Then, using log data from a live geocoding system, a statistical model is created that can be used to generate user-like requests. A similar approach based on the log data is evaluated to create geocoding systems that perform measurably better than geocoding systems relying on common methods to handle user queries. Finally, a process is developed that continuously improves a geocoding system using user queries issued against it.

The measurements prove that the path chosen in this thesis is a viable method to create geocoding systems that can handle user queries better than geocoding systems relying on common techniques.
Zusammenfassung


Messergebnisse belegen, dass die in dieser Dissertation vorgeschlagene Vorgehensweise eine plausible Methode ist, um Geocoding Systeme aufzusetzen, die Anfragen von Menschen besser verarbeiten können als Geocoding Systeme mit herkömmlichen Ansätzen.
Related Publications

Many of ideas, methods, and results collected in this thesis have been published in scientific papers and journals and presented at international conferences. To some extent, therefore, parts of this thesis are contained in the following publications:


Acknowledgments

First of all, I would like to thank my supervisor Prof. Dr. Axel Küpper for providing me with trust and space at times and pushing me to deliver when slowed down. As an external doctorate student, I was able to only spend parts of my time with the research group Service-centric Networking. Even though, I received a lot of support from my fellow research colleagues too, who spent a lot of time discussing with me and reviewing my publications. Dear SNET colleagues, thank you very much for your input.

I also understand the time and effort needed to read a body of work as this thesis, provide feedback and grade it. Therefore, my thank also goes to the committee of censors, including Prof. Dr. Axel Küpper, Prof. Dr.-Ing. Jochen Schiller, as well as Prof. Dr.-Ing. David Bermbach.

Another big source of support was my former employer HERE Technologies and the many supervisors that I had the opportunity to work with while working at this thesis. They supported me along with my terrific colleagues who always had my back while I was doing the research for this thesis.

A special thank you goes to my wife Anna, who supported me patiently, allowing me to complete this work.

Last but not least, I want to express my deepest gratitude to my family and friends. They all directed, pushed, and motivated me to start and complete this work.
Contents

Declaration of Authorship iii
Abstract v
Zusammenfassung vii
Related Publications ix
Acknowledgments xi

1 Introduction 1
1.1 Geocoding Use Cases ........................................... 1
1.2 Set of Issues .................................................. 2
1.3 Central Question and Contribution of this Thesis ............ 4
1.4 Organization of this Thesis ..................................... 6

2 Related Work 7
2.1 Geocoding Systems ............................................. 7
2.2 Alternative Addressing Schemes ............................... 9
2.3 Specific Geocoding Use Cases ................................. 15
2.4 Comparison of Geocoding Systems .......................... 20
2.5 Data used for Geocoding ...................................... 23
2.6 Matching of Address Element Names ....................... 25
2.7 Approaches to Geocoding .................................... 27

3 Background 33
3.1 Geographic Coordinate System ............................... 34
3.2 Postal Addresses ............................................... 35
3.3 OpenStreetMap ............................................... 38
3.4 Nominatim .................................................... 42
3.5 Elasticsearch .................................................. 47

4 Indexing Complete Addresses 51
4.1 Hypothesis .................................................... 52
4.2 Data for Evaluation ........................................... 53
4.2.1 Sampled Addresses from the Index ..................... 53
4.2.2 Scrapped Addresses from the Web ..................... 54
4.3 Implementation ................................................ 54
4.4 Results ....................................................... 57
4.5 Conclusions .................................................. 59
1 Introduction

Location is a ubiquitous property of everything existing in the real-world. Cars, ships, airplanes, buildings, stores, trees, colleagues, family members, meeting places as well as most other things have the location property. Like most other data in the age of information, location data is persisted, kept up-to-date, indexed and retrieved using computers. Computers are used to trace paths of airplanes\(^1\), track the whereabouts of trees\(^2\), or discuss and agree on meeting places\(^3\). For humans, addresses are the most common way to describe a location. Evolving, addresses usually reflect political and organizational boundaries and use named or numbered streets, districts, and houses as building blocks to describe a specific location within them. In computer science, representing this human-friendly description of a location is a challenge. The lack of clear rules makes modeling an address vague and complex. In addition to that, names of address elements - the building blocks of addresses - are reused often. Also, the sheer amount of addresses on the globe add a performance challenge to the stack of problems too.

In this thesis, approaches to storing and retrieving addresses in a computer are examined. Specifically, various endeavors to create a geocoding system are made. The focus lies on understanding how much existing and newly proposed mechanisms contribute to enabling a computer to comprehend addresses specified by human users. For that, several geocoding systems are set up with the same data and evaluated for their performance in the course of this thesis.

1.1 Geocoding Use Cases

Clearly, there is a multitude of reasons to geocode addresses. From straightforward understanding where to deliver a package or a passenger, over data analysis for commercial, medical, or police use, to plain exploratory visualizations: All use cases stem in the need to pass location information between a human and a computer.

Some less obvious but common use cases for geocoding are, for instance, the creating of a heat map of product sales, deriving the location of a sale from user input addresses, or the verification of a list of user input addresses. Table 1.1 lists various possible inputs to a geocoding service and examples of applications.

\(^1\)https://www.flightradar24.com (accessed: November 2019) provides a real-time view of airplanes all over the world
\(^2\)http://glh.unitar.org/ (accessed: November 2019) shows a map of trees that survived the bombing of Hiroshima with a nuclear weapon in 1945, as well as seeds of these trees planted across the world
\(^3\)Most modern messengers as WhatsApp, the Facebook Messenger, or Skype support sharing locations as part of a conversation
Table 1.1: Example use cases of geocoding services with various inputs.

<table>
<thead>
<tr>
<th>input data</th>
<th>application examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>single full address</td>
<td>Display location of a taxi ride destination, compute travel time or distance between addresses, plan the shortest route for delivery of multiple parcels, or verify the address against a geo-fence.</td>
</tr>
<tr>
<td>list of full addresses</td>
<td>Identify geographic patterns in customer or clients database for warehouse placement, advertisement campaign planning, or distribution of medical reserve. Validate addresses on the list.</td>
</tr>
<tr>
<td>incomplete address or addresses</td>
<td>Display geographic boundaries of a named entity, specify geo-fence, assess the maximum distance of two locations.</td>
</tr>
<tr>
<td>text mentioning locations</td>
<td>Identify geographies referred to in the text document.</td>
</tr>
<tr>
<td>latitude and longitude</td>
<td>Reverse geocode human comprehensible location or area name.</td>
</tr>
</tbody>
</table>

Geocoding can also be entertaining too. For example, the website *Johnny Cash Has Been Everywhere*\(^4\) plays the song *I've been everywhere* by the famous country music icon. In the song, the singer lists a huge number of cities and towns that, supposedly, he has visited. Along with the music, as each place is mentioned, Google’s geocoding API is used to display the location of the place on a map. Additionally, the website accumulates the distance between the places as the distance that the singer traveled.

Many researchers use geocoding during their work. It is applied to analyze medical, environmental, or criminal records as well as to generally deduce observations on population. Therefore, there is plenty of research around various geocoding use cases. Some rely on geocoding of, for example, addresses of patients to get statistical observations on the geographic spread. Others research the effect of the geocoding quality on research.

### 1.2 Set of Issues

For all applications of geocoding, similar obstacles have to be overcome: The data representing the real-world needs to be collected first and kept up-to-date. This task requires tremendous effort, as the real-world changes constantly while data modeling location can get arbitrarily huge. For example, on May 1st, 2019, the plain OpenStreetMap data takes over a terabyte when uncompressed, but not yet indexed for searching\(^5\). Obviously, to work with location data in any way, indexes need to be created that require additional space.

To provide a geocoding service specifically, address data needs to be indexed. That is a challenging task by itself. Address data can be modeled in a variety of ways, and it may become necessary to find additional data models that allow for a quick lookup. For instance, the street name can be modeled as a single string, or as a string pair of name, and street type. Various street types might be represented by constants that can be translated into different languages, or as opaque strings. Further, various options exist to assemble the full street name from name and type strings. The street type can come before or after the street name and might be separated from the street name with space. Depending on the language, the street type might be spelled with different casing. This variety of possibilities to model the street name extends to other address entities and illustrates the magnitude of possibilities. Having the data at hand and modeled appropriately, it can be indexed. Various indices can query data in different ways. Depending on the functionality of the used index, different types of queries can be supported by the geocoding system. A relational database allows, for example, to filter for geocoding results in a specific bounding box, if the necessary tables are populated with data. A NoSQL store might only be capable to retrieve address documents using their identifier.

Overall, geocoding can be considered to consist of two phases. First, a given query is parsed, processed, and understood. Then, the data is fetched from the index to be returned as a result. There is no reason to not query auxiliary indices during the first phase already. However, the fewer queries are executed against internal indices, the quicker the system performs. In any case, the query intent needs to be understood during the first phase. That might include understanding which address element various parts of the query refer to. After the query intent has been understood by the system, the second phase constitutes a mere data lookup. The result, or possibly, the set of result candidates are retrieved from the address index and returned to the user. If the system is set up to return multiple results, the likelihood of a candidate to be correct is established during the first phase already. Thus, while the first phase is depending on indexing methodologies, data processing algorithms, and ranking formulas, the second phase is subject to the performance of query execution and address data assembly. Thereby, with a less structured query format, the first phase becomes increasingly harder.

The least structured query interface is free text. Most of the geocoding services offered to the end-user, including geocoding services by Bing, Google, HERE, and OpenStreetMap provide this interface via a simple search box on their websites. It allows specifying arbitrary queries that the respective systems try to process. Clearly, that enables a magnitude of possible input. Users can specify arbitrary address elements in arbitrary order this way. Additionally, users might specify additional information useful to a human being. That extra input may obfuscate the query intent, especially if it contains information that does not fit to the chosen address model. For example, specifying the color of a building might help a human to find an address, but will only improve the geocoding accuracy if building colors are indexed. In addition to that, human users like timesaving methods. Thus, user input might not specify all of the
multiple tokens an address element consists of. In the same way, humans might abbreviate address element names or types to more or less common standards. Also, if the input of an address element name requires extra effort, humans are likely to skip that. Typing the German ß, for instance, is especially hard with an ASCII keyboard. The sharp s and most similar diacritics might be replaced by ASCII characters in a user query, therefore. Finally, humans make mistakes. They might specify a wrong number, a misspelled address element name, or incorrect and contradicting address elements in their query. All these issues sum up to an impressive task to be solved. Therefore, this thesis focuses on handling user queries exclusively. Data accuracy, clearly, has a strong effect on the correctness of the results of a geocoding system. For this thesis, however, various errors in user queries are analyzed together with the effectiveness of methods to encounter them.

Table 1.2 categorizes ways human users use to specify addresses into common classes and provides examples. Some of the listed issues may be solved by collecting the necessary data, while others need to be solved algorithmically. For example, translated address element names, or address elements that are not usually specified as part of addresses can be added to the data. This way, an index would contain these terms and return the correct results. That approach can also be followed for abbreviations as well as escaped diacritics. An exotic use case could also request historical address element names to be part of address data too. In reality, there is no data collection containing all translations, abbreviations, escaped variants and the like. Besides, other cases cannot be solved by extending the data. Indices are set up to function independent of the order of elements, for example, instead of listing all feasible orderings in the data. Similarly, there is currently no way to list all typing mistakes human users make.

Another important distinction is not clearly visible in Table 1.2. While the classes order of elements and presence of elements refer to entire queries, translation of elements, abbreviation of elements, replacement of diacritics, and plain typing mistakes refer to distinct address elements or even query tokens. Splitting a query into tokens that are separated by whitespace, obviously, might also split address elements names apart. That is specifically the case for address element names that consist of multiple tokens themselves. In either case, independent of a possible split of address element names, modifications of single tokens are described in the latter four classes. Thus, we can group the classes into two categories: Query format related issues include the absence of address elements and their order. Spelling variants of query tokens include all scenarios where a spelling different from the data is specified by the user. Obviously, nothing prevents a human from using a combination of error classes in a single query.

### 1.3 Central Question and Contribution of this Thesis

The goal of this work is to identify techniques that make up a good geocoding system. Throughout this thesis, various approaches will be discussed, measured, and evaluated in isolation. Some of the approaches might be in use by the various
1.3. Central Question and Contribution of this Thesis

Table 1.2: Example use cases of geocoding services with various inputs.

<table>
<thead>
<tr>
<th>class of user-made errors</th>
<th>query examples</th>
</tr>
</thead>
</table>
| order of elements: Address elements are specified in varying order. | 7, Ernst Reuter Platz, 10587, Berlin  
Ernst-Reuter-Platz 7, 10587-Berlin  
10587-Berlin, Ernst-Reuter-Platz 7 |
| presence of elements: Some address elements are specified though not usually part of a postal address. Other address elements are not specified though commonly used. | Ernst-Reuter-Platz 7, Berlin  
Ernst-Reuter 7, 10696 Berlin  
Ernst-Reuter-Platz 7, Charlottenburg |
| translation of elements: Address elements are specified in a different language. | Ernst-Reuter-Platz 7, 10587 Berlín  
Ernst-Reuter-Platz 7, 10587 Berlino  
Ernst-Reuter-Platz 7, 10587 Berlijn |
| abbreviation of elements: Address elements are abbreviated into more or less common terms. | Ernst-Reuter-Pl. 7, 10587 Berlin  
1 Beacon St, Boston, MA 02108  
405 East 42nd Street, NYC-NY, 10017 |
| replacement of diacritics: Special characters are replaced with their ASCII variants. | Struncovy sady 1 301 00 Plzen 3  
201 S Market St, San Jose, CA 95113  
2 Place de l’Etape, 45000 Orleans |
| plain typing mistakes: Address elements are specified incorrectly due to lack of knowledge or by mistake. | 1 Bacon Street, Boston, Masachussets 02108  
60 Providence Lane, Sprigfield, IL 62711  
Ernst-Ruter-Platz 7, 10587 Belin |

proprietary systems. With this thesis, the impact of the various techniques becomes apparent and can be put in relation to the cost of implementing them. Explicitly, supporting the non-interactive use case of serving a geocoding request of a human user is in focus. This is done for simplicity, as a good non-interactive service can be extended into an interactive solution while that is not the case vice versa. Thereby, for all implementations of geocoding services, only one single data set is used. This way, the specific change can be measured without interferences from differences in data.

In this thesis, first, a general address format suitable to contain arbitrary addresses from around the world is proposed. A geocoding service using that address format is set up and evaluated for its quality of service. Next, the address format is evolved into a more normalized one. The improved format aggregates addresses into documents thereby reducing disk space and index sizes. Additionally, the improved address format prepares the addresses for fuzzy search. It is validated to enable geocoding services to provide the high quality of service as with the general address format.

After that, the spelling variants of query tokens are tackled. A novel approach precomputes likely spelling variants of address elements names and augments
the index with these variants. This approach is evaluated against a classic method to enable fuzzy search of incorrect queries.

Lastly, the approach to precompute spelling variants is extended into a process that gradually and continuously improves a geocoding system with regards to the quality of service. Multiple iterations of this process are evaluated one at a time, measuring the system improvement gradually.

1.4 Organization of this Thesis

Next, in Chapter 2 the status quo of science and industry is presented. In Chapter 3 the main tools used in this thesis are presented. That Chapter also describes the general concept of geocoding, existing software that is utilized, as well as the data that the geocoding systems are built with. Thereafter, in chapters 4 through 9 various techniques are evaluated to achieve results that are more precise. Chapter 10 accumulates all findings, proposing a process that enables geocoding systems to continuously improve their performance over time. Queries issued against such a geocoding system are used as a basis for that process. Finally, in Chapter 10 conclusions are drawn and an outlook on further research is presented.
2 Related Work

In this chapter, related scientific work is presented. With the birds’ eye, it can be categorized into seven sections. First, an overview of existing geocoding systems is given. Most of the systems are proprietary and the way they function are corporate secrets. Also, they do not give away any details on the address data that is used to power the systems. Some services, however, rely on open data collected by the community. Further, some of the systems are open-source, revealing all implementation details on their functionality. Next, ideas to avoid the problem of geocoding are presented. For all cases, this means that not computers, but humans adapt to a newer, arguably better way of describing locations. Various such alternative addressing schemes are presented in that section and their strengths and weaknesses are discussed. Another section collects the large body of research around the quality of geocoding systems that measure the systems having a specific use case in mind. While all that work is valuable for those use cases, none of these studies care to look beyond the interface of a geocoding system. They treat the entire system as a black box and hence do not point to the spots where the respective systems can be improved. The next section of this chapter presents research around data used for geocoding. These publications focus on how to collect, measure, and improve data that is used to run a geocoding service. Thereafter, a section presents some interesting approaches to set up geocoding systems. Lastly, in the final section, research on how to match address element names is shown. Most of the work presented focuses on the address element names and does not look into all stages of a geocoding system. However, it relates to this thesis as matching a part of a query to the correct address element name is a crucial part of a geocoding system and is investigated here.

2.1 Geocoding Systems

In [8], the author defines geocoding components as follows: "A geocoding component generates geographic coordinate information, such as latitude and longitude values, for postal addresses." In other words, geocoding is the process of resolving addresses into a location. Thereby, the location can be represented in a magnitude of ways.

Nowadays, a variety of geocoding systems is available to the general public. Amongst the most commonly known ones are geocoding services that, besides an API also provide a web site that allows discovering areas and plan routes through
the means of a digital map. Google\textsuperscript{1,2}, Bing\textsuperscript{3,4}, HERE\textsuperscript{5,6}, and OpenStreetMap\textsuperscript{7,8} are providing such services. Out of these four, OpenStreetmap is an organization that collects data from the crowd and publishes it free of charge, while the others are proprietary services provided by companies. Because of that, the OpenStreetMap geocoding system Nominatim is open-source.

Among other well known geocoding service providers are Esri with their mapping and analytics platform ArcGIS\textsuperscript{9} or TomTom\textsuperscript{10}, Mapbox\textsuperscript{11}, or the US Census Bureau\textsuperscript{12} providing APIs to resolve address text into geographic coordinates.

While the services mentioned so far are relying on their own proprietary data, many companies chose a different route. Geobuffer\textsuperscript{13}, Geocode.xyz\textsuperscript{14}, Mapfit\textsuperscript{15}, or SmartyStreets\textsuperscript{16} provide their geocoding functionality relying on acquired or publicly available open data. Similarly, the QGIS\textsuperscript{17} plugins GeoCoding\textsuperscript{18} and mmqgis\textsuperscript{19} offer the capability to geocode using Google or Nominatims geocoding APIs. LocationIQ\textsuperscript{20} relies on OpenStreetMap data exclusively. The JavaScript framework Leaflet\textsuperscript{21} abstracts over geocoding APIs from arbitrary vendors.

All geocoding services depend on data. A variety of data sets exists. Proprietary data sets of Google, Bing, HERE, Esri or TomTom are closed data that is not available for others. Similarly, Zillow\textsuperscript{22} or Pitney Bowes\textsuperscript{23} offer APIs to access their proprietary location data sets. Alternatives to proprietary data are provided by governments. For example, besides offering a geocoding service, the US Census Bureau offers their Topologically Integrated Geographic Encoding & Referencing (TIGER) data to be downloaded from the internet\textsuperscript{24}. Similarly, the city of Berlin offers an address data set that can be downloaded through an API\textsuperscript{25}. Also, there are various initiatives to collect location data from the crowd. GeoNames\textsuperscript{26} aggregates data from end-users with data provided by various governmental agencies and offers it under the creative commons attribution

\textsuperscript{1}https://developers.google.com/maps/documentation/ (accessed: November 2019)
\textsuperscript{2}https://www.google.com/maps (accessed: November 2019)
\textsuperscript{3}https://docs.microsoft.com/en-us/bingmaps/rest-services/ (accessed: November 2019)
\textsuperscript{6}https://wego.here.com/ (accessed: November 2019)
\textsuperscript{7}https://nominatim.org/release-docs/develop/api/Overview/ (accessed: November 2019)
\textsuperscript{8}https://www.openstreetmap.org (accessed: November 2019)
\textsuperscript{11}https://docs.mapbox.com/api/search/ (accessed: November 2019)
\textsuperscript{12}https://geocoding.geo.census.gov/ (accessed: November 2019)
\textsuperscript{13}https://geobuffer.com/#api (accessed: November 2019)
\textsuperscript{14}https://geocode.xyz/api (accessed: November 2019)
\textsuperscript{15}https://www.mapfit.com/developers (accessed: November 2019)
\textsuperscript{17}https://www.qgis.org/en/site/ (accessed: November 2019)
\textsuperscript{18}https://plugins.qgis.org/plugins/GeoCoding/ (accessed: November 2019)
\textsuperscript{19}https://plugins.qgis.org/plugins/mmqgis/ (accessed: November 2019)
\textsuperscript{20}https://locationiq.com/ (accessed: November 2019)
\textsuperscript{21}https://leafletjs.com/ (accessed: November 2019)
\textsuperscript{22}https://www.zillow.com/howto/api/APIOverview.htm (accessed: November 2019)
\textsuperscript{25}https://daten.berlin.de/datensaetze/adressen-im-inspire-datenmodell (accessed: November 2019)
\textsuperscript{26}https://www.geonames.org/ (accessed: November 2019)
license\textsuperscript{27}. OpenAddress\textsuperscript{28} allows the users to provide data sources for data that, after aggregation and checks, are available grouped by licenses of the source data. Last but not least, OpenStreetMap\textsuperscript{29} is a project that allows users to contribute location data in a collaborative manner. OpenStreetMap data is available under the open data commons open database license\textsuperscript{30} and is used throughout this thesis.

In sum, there is a great number of companies providing geocoding services. Some of them are free of charge for private use. Some are open–source and can be deployed in any necessary environment. Most service providers, however, use proprietary implementations that are treated as corporate secrets. The details on which methods are employed to achieve a quality of service is not exposed to the general public. Also, while the bigger geocoding service providers are relying on proprietary data, many smaller companies build their services on top of a mix of data sources, including openly available data. This makes it hard to interpret or reproduce any measurement of quality of service. A service might employ the best algorithms and indexing techniques, but provide a poor service due to lack of high–quality data. Another service might perform greatly due to data, even though it lacks basic functionality. Additionally, data quality and necessary functionality might vary by region. This makes it even harder to evaluate the quality of a geocoding service.

2.2 Alternative Addressing Schemes

Approaches that try to mitigate the issues around human specified addresses by introducing alternative addressing schemes exist. Mainly, they try to specify an addressing scheme that is easily machine–readable and human user–friendly at the same time. Before going all the way to addressing schemes that only experts understand, some work on establishing coherent address models is presented. Afterward, research on alternative addressing schemes and how most of the approaches work is explained. Finally, an experiment for the usability of selected alternative addressing schemes is discussed.

Address formats are in constant change. As the needs of humans change, parts that constitute an address are modified, new parts are added, or existing parts extended. In [9], the authors evaluate the address model used in South Korea. It has limitations, even though it has been introduced as recent as 2014. The author suggests requirements that an address model has to fulfill and propose an extended address model that covers them. Finally, the author evaluates their address model by measuring the geocoding match rate of addresses in the new format. A similar problem is investigated in [10]. The authors analyze various competing address formats that are used in South Africa. They see the large variety of the address formats rooted in the lack of control of addresses by authorities. As a conclusion, the author calls for a single, standardized address format that is used in the entire country. In [11], the authors create such an

\textsuperscript{27}https://creativecommons.org/licenses/by/4.0/ (accessed: November 2019)
\textsuperscript{28}http://results.openaddresses.io/ (accessed: November 2019)
\textsuperscript{29}https://www.openstreetmap.org/export (accessed: November 2019)
\textsuperscript{30}https://opendatacommons.org/licenses/odbl/1.0/ (accessed: November 2019)
address model for Addis Ababa City in Ethiopia. Currently, there is no address model used at all. The study, therefore, suggests using a formal address model.

A broader perspective on address formats is taken in [12] and [13]. The former publication extends the latter. Both studies suggest criteria which can be fulfilled by addressing formats. The aim of the suggested criteria, however, is to evaluate alternative addressing schemes. The authors of [14] analyze and further expand these criteria. They conclude that user-oriented criteria should be weighted more heavily than the other ones.

Besides the evaluation of addressing schemes that are in use, many alternative addressing schemes rely on a Global Discrete Geodesic Grid [15][16] under the hood. These grids are colloquially referred to as geohashes. Figure 2.1 illustrates a simple way to compute the geohash\(^3\) of a location. For that, first, the entire globe is projected to a plane, for example, using a Mercator projection [17]. It is important that the resulting plane is rectangular. Then, the plane is split into four rectangles that fill the entire square and do not overlap. Each of the rectangles gets a distinct name from an arbitrarily chosen alphabet. In Figure 2.1, the binary numbers are used as the alphabet, resulting in the names 00, 10, 01, and 11. This step is repeated iteratively inside each of the rectangles resulting in rectangles of increasingly smaller sizes. The four rectangles inside a rectangle get the name of their parent extended by the same four names chosen at the very beginning. This way, rectangles with increasingly long names denote increasingly specific areas. While formally the names of the rectangular tiles are the geohashes, commonly the tiles themselves are referred to as geohashes as well. The area of the entire initial rectangle is addressable using these geohashes. Also, a shorter geohash addresses tiles that are larger and higher in the hierarchy. Longer geohashes drill further down and address smaller tiles. Usually, a binary alphabet is selected as it can be encoded into ASCII strings easily. Also, because

\(^3\)http://geohash.org/ (accessed: November 2019)
2.2. Alternative Addressing Schemes

Figure 2.2: Octahedron inside a sphere with each side split up into triangles of equal sizes [18].

the common Mercator projection results in a rectangle that has twice the width of its height, an initial vertical split in the center of the map is applied and encoded with an additional leading bit. This way, all geohashes address squares on the left or on the right side of the map. An alternative description\textsuperscript{32} of the same algorithm builds on that first step. It specifies that starting with a vertical split, each additional bit splits the remaining rectangle in half, alternating between horizontal and vertical splits. Thereby, each bit specifies if the encoded tile is above or below of the centerline, or left or right of the centerline respectively. Another simple modification is to split up the tiles into more than four pieces. As long as the bits a tile is split up in can be further split up recursively, the same algorithm can be applied.

Geohashes have very convenient properties that can be of use. For instance, when two geohashes have the same prefix, their maximum distance can be computed. Because of the hierarchical way to compose the geohashes, a longer common prefix of two geohashes implies that they are located closer to each other. That property, however, does not invert. Two geohashes can be right next to each other and have a short common prefix or no common prefix at all. That is the case when the two tiles are located in two neighboring but distinct tiles early in the hierarchy. Another benefit from geohashes over addresses is that they already cover the entire globe. Using geohashes, it is possible to address a tile in the middle of an ocean as precisely as in a city center. Addresses require a building and, likely, a street to be built as their resolution is not finer than this. Also, a bounding box of geographic coordinates can easily be computed

Chapter 2. Related Work

Table 2.1: Examples of various geohash values for addresses in three different cities.

<table>
<thead>
<tr>
<th>Address</th>
<th>Geohash in Base32 encoding</th>
<th>Geocode.xyz</th>
<th>plus code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ernst-Reuter-Platz 7, 10115 Berlin</td>
<td>u336xpegbqb</td>
<td>BERLIN-ZDMYE</td>
<td>G87C+62 Berlin, Germany</td>
</tr>
<tr>
<td>5 Avenue Anatole France, 75007 Paris</td>
<td>u09tunqu71mr</td>
<td>CERGY-WIENX</td>
<td>V75V+8R Paris, France</td>
</tr>
<tr>
<td>405 E 42nd St, New York, NY 10017</td>
<td>dr5ruf4t1z08</td>
<td>NEWYORKCITY-ELLUC</td>
<td>P2XM+M2 New York, NY, USA</td>
</tr>
</tbody>
</table>

along with its center point given a geohash and vice versa. This allows building indices on geohash values to quickly discover spacial collocations of entities or serve spacial queries. Because of the Mercator projection, however, different geohash tiles often have different sizes. Especially towards the poles, tiles tend to be compressed vertically due to the projection.

Distinct geohashes, therefore, specify different ways to encode locations close to the poles. Geocode.xyz\(^{33}\), for instance, defines regional geohashes that are prefixed with well-known city names. In these well-defined areas, the differences in tile sizes are small enough to be irrelevant. Plus codes\(^{34}\) work in a similar way. A lengthy geohash is computed from a latitude and longitude coordinate pair first. In a second step, the prefix can be cut-off and replaced by the country and region name, the geohash is located in. The region name, thereby, is selected by geography. It can be the city name of an urban area or a town in a rural area selected in a deterministic manner.

Alternatives to rectangular geohashes exist too. Octahedral Quaternary Triangular Mesh [19] avoids the compression of rectangular tiles through refraining from the Mercator projection of the globe onto a cylinder. Instead, the globe is projected on to an octahedron. These two pyramids with a square base form a shape that is much closer to the sphere than a cylinder. The eight triangular sides of the pyramid can easily be split up into four triangles of equal size. The triangles can be split up into smaller triangles iteratively, achieving the same properties that rectangular geohash tiles have. However, the triangular tiles are not prone to deformation towards the poles. Figure 2.2 presents an octahedron inside a sphere. Jokingly the service what3ducks\(^{35}\) implements a fully functional geohashing service that uses the Octahedral approach. Thereby, the service encodes the geohashes in words and is available in various languages. More swearwords allow more iterations of splitting the triangular tiles and hence, result in finer resolution. Similar paths are taken by the services

\(^{34}\)https://plus.codes/ (accessed: November 2019)
Table 2.2: Examples of various geohash values for addresses in three different cities.

<table>
<thead>
<tr>
<th>Germany</th>
<th>Paris</th>
<th>New York, NY 10017</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ernst-Reuter-Platz 7 10115 Berlin</td>
<td>5 Avenue Anatole France 75007 Paris</td>
<td>405 E 42nd St New York, NY 10017</td>
</tr>
<tr>
<td>(i) <strong>WGS84 latitude, longitude</strong></td>
<td>52.512937,13.320063</td>
<td>48.858312,2.294562</td>
</tr>
<tr>
<td>(ii) <strong>what3words</strong></td>
<td>blur.sofa.quantity</td>
<td>papers.fondest.gadgets</td>
</tr>
<tr>
<td>(iii) <strong>syllagloble</strong></td>
<td>lay uxre cer asne</td>
<td>lac abru haw uawa</td>
</tr>
<tr>
<td>(iv) <strong>geo-poet</strong></td>
<td>requesting emulation</td>
<td>propellant dissipation</td>
</tr>
<tr>
<td></td>
<td>accretion instigation</td>
<td>appended revocation</td>
</tr>
<tr>
<td>(v) <strong>Mapcode</strong></td>
<td>DEU 0LN.5T</td>
<td>FRA 4J.Q3</td>
</tr>
</tbody>
</table>

what3emojis\(^{36}\) and what3fucks\(^{37}\) that use emojis and swearwords to encode a triangular geohash. More similar solutions exist; they all are open-source and easy to modify or extend.

The service what3words\(^{38}\) is the one mocked by what3ducks. It uses a proprietary method to encode a square location tile in three words. The service is available in multiple languages. Unlike the previously mentioned services, what3words does not rely on a geohash mechanism under the hood. Therefore, tiles are referenced by exactly three words, have constant size, and are always rectangular. Also, due to the lack of common prefixes, similar words do not encode tiles that are located close to each other. More than that, the order of words is an important aspect of the code. Using the same three words in various orders results in different, fully unrelated locations encoded. Table 2.1 presents various geohashes of addresses in different cities.

These orthogonal approaches to define alternative addressing schemes point to the question: What makes a good addressing scheme? While all the mentioned systems fulfill the requirement to have unique and machine-readable codes, they all differ in two more aspects. For once, geohashing based systems allow variable specificity of the geohash tiles. Depending on the length of the geohash, tiles can be variously large or small. What3words, on the other hand, has a constant resolution. For most use cases, however, a variable resolution is not necessary. Thus, the one remaining aspect will have to break the tie. The various geohashes must be more or less user-friendly. While defined vaguely, it is clear that an alternative addressing scheme needs to be more convenient to the human user than the current addresses used. In [4], four addressing schemes are compared for their memorability. At least in part, memorability does impact the user experience and hence can be used as a distinguishing factor. In that paper, the comparison covered five addressing schemes. As

---

Figure 2.3: Memorability of various addressing schemes compared.

a simple baseline, (i) plain WGS84 coordinates were used. (ii) What3words was selected as a proprietary solution that should have a focus on usability. (iii) Syllagloble was set up specifically for the experiment. The geohash based system uses syllables to encode a location. As with all geohashes, a common prefix implies a collocation of the locations encoded. (iv) Geo-poet was created explicitly for this experiment too. Geo-poet is based on a geohash implementation that encodes a location using a quatrain. That is a word more than used by what3words, but the rhyme, so the hypothesis, might add to the memorability. Finally, (v) Mapcode was part of the comparison. Mapcode is a service generating geohashes that are extremely short and easy to remember by replacing a common prefix with a regional code similar to plus codes. Table 2.2 lists the five alternative addresses for the already known examples.

To measure the memorability of the five addressing schemes, users were offered to play a memory quiz game. Disguised as a quiz, the questionnaire generated eight questions for each addressing scheme. For each question, one location was picked at random. Each question had one correct answer and seven randomly generated incorrect ones. Of the seven incorrect results, one incorrect result was generated in a 50m radius of the correct result. All other results were further apart. For each question, the correct result was presented to the participant for five seconds. Afterward, a shuffled list of options was shown to the participant to chose from. Each set of eight questions for each addressing scheme was presented in a row. The quiz game was advertised on social media and had over 50 participants when the study was conducted. It is still available online.

The over 20,000 data points collected this way are presented in Figure 2.3. As anticipated, WGS84 geocoordinates are hard to remember. Only 82.9% of the questions were answered correctly. The addressing schemes computed by what3words and Syllagloble performed best and equally well with 96.5% and 96.9% correct responses respectively. These schemes are followed by Mapcode with 93.0% and Geo-poet with 91.0% cases remembered correctly.

One important difference between the three geohash based schemes Mapcode, Geo-poet, and Syllaglobe and what3words was clearly measured. The non-successful responses to what3words addresses are all not within 50m of the correct result. That is not surprising, as there was only a one in seven chance to pick the result that was nearby at random. However, the common prefix of the geohash schemes, as well as the common prefix of the WGS84 coordinates, allowed users to pick the nearby result disproportionally often. Thus, while Syllaglobe and what3words have a similar rate of correct picks, 43.7% of incorrect responses to the Syllaglobe addressing scheme are within 50m of the correct response. That might be an acceptable imprecision for many use cases and, hence, it is fair to conclude that the geohash based systems are easier to remember for the end-user compared to non-hierarchical approaches such as that from what3words.

An interesting extension of what3words is presented in [20]. The authors extend the mechanism of the system adding an additional word to address sub-cells for finer positioning. A further word is added to encode elevation. With only six words, the authors intend to encode time in future work too. Thus, using six instead of three words, a much more detailed cell, its elevation as well as time can be encoded.

In sum, what3words is a successful organization marketing and selling their product to many customers around the world. Other, open addressing schemes are working just as good and make up interesting projects to work with. Alternative addressing schemes, however, are not replacing common addresses for the human user as of now. From the experiment conducted in [4] it can be deduced that an easy to memorize addressing scheme should go down the geohash path. Probably, it would combine techniques proposed by Mapcode, plus codes, or Geocode.xyz replacing a long prefix of a geohash with a city or region well known to a human user. For now, however, we will focus on existing addresses and focus on improving the geocoding process for them.

### 2.3 Specific Geocoding Use Cases

Likely the biggest cluster of research is around geocoding in the medical field. In [21] the authors analyze how medical records can be enriched with additional data about the patient. For that, they start off with geocoding the patients’ addresses. The additional data about the patient is then pulled from statistical data about the patients home. The patient might be rich or poor, worker or academic, or live in an area that is known to be affected by specific diseases. The doctor can then incorporate this information and adapt conversations or the diagnosis accordingly. The authors focus thereby on the performance of geocoding services, geocoding addresses in real-time through different approaches.

Another focus of the publication is related to the patient's privacy. A comparison of three different geocoding services is undertaken in [22]. The authors use ArcGIS to geocode addresses of patients measuring the rate of successfully geocoded addresses. 73% of addresses geocoded out of the box,
while the remaining 37% were geocoded using an interactive approach. The authors were able to tune the software to geocode 59% of the initially failing addresses but received a poor accuracy this way. In [23] besides geocoding, various ways to cluster the geocoding results are evaluated. The goal is to get to a coherent map with denoted areas where newborns likely have low birth weight. A similar undertaking with various geocoding methods is undertaken in [24]. The authors use different interpolation techniques when geocoding addresses of cancer patients to a zip level. They conclude that that geocoding is a crucial step when clustering data. In their study, some areas had differences in the observed cancer rates of up to 400%, depending on which interpolation technique was used.

A deeper analysis of various health data geocoded to resolutions of zip code area, parcel, or street address was analyzed in [25]. While in their analysis the authors observed acceptable success rates with all three resolutions, their recommendation is to always use the highest possible resolution when conducting a geographic analysis of patients’ locations. A similar conclusion is made in [26]. The authors focus on analyzing the usability of zip code areas as a level of geocoding for medical research in the US. According to the authors, zip code areas have an average population size of 30,000 inhabitants and are therefore too heterogeneous to be used as the unit of geocoding accuracy. The authors conclude that more precise geocoding is needed. In [27], the specific effect of geocoding inaccuracy on the research of children affected by pollution is analyzed. In that study, the authors measure the exact location of the address using a satellite positioning system and compare their measured locations to the ones returned by a geocoding system. They achieve a median error of just 41m but conclude that this leads to an overestimation of the number of children exposed to high traffic roads.

In [28], the effect of the geocoding accuracy on health research is assessed in general. The authors observe that scientists need to have comparable methodology they use for geocoding for which the error rate is known. They call for open-access geocoding services and validation data sets, explicit geocoding error models and suggest to evaluate earlier studies for their wrongness due to geocoding. A different approach is taken in [29]. The authors focus on geocoding as many addresses as possible in their database of 14,804 patient addresses in the US. They use two independent geocoding services and reach out to the patients in case their address cannot be geocoded with either. Out of the 5% of addresses that could not be geocoded with either service, 49% of the households were successfully contacted. 97% of the updated addresses received this way could be geocoded in a separate step. While in this study only the success rate of the geocoding attempts was measured, it is fair to conclude that most of the addresses that failed to geocode were broken in transition by the human user.

The article [30] confirms the finding observed in [29]: One big source of geocoding errors remains in the inaccuracy of humans. That article is part of another whole realm of research around geocoding and geocoding use cases is the mapping of criminal incidents. In the article, addresses of investigations of gunshot locations are geocoded. As ground truth, the locations are used
that are computed by the acoustic system for gunshot detection in Wilmington, North Carolina. In the article an average positional accuracy of 20m is achieved, however, the article confirms that a big chunk of problems was caused by humans transcribing the address. The required geocoding success rate for geospatial crime data analysis is assessed by [31]. The authors compare various studies and conclude that a minimum rate of 85% is sufficient to gather insights. Another take on clustering geolocations is taken in [32]. Two address data sets – an older and a newer one – are compared as the base of geocoding. The authors do not observe severe differences between geocoding based on the different data sets. As a conclusion they recommend a kernel cell size and a cluster bandwidth to smoothen inaccuracies of both geocoding approaches. A different use case is experimented with in [33]. The authors try to use automated geocoding techniques to ensure that residencies of sex offenders are outside of limits to schools, kindergartens, and the like. The observation made is that the geocoding techniques used are not sufficiently accurate to automatically assess if residency restrictions of sex offenders are followed.

A variant of geocoding is geotagging. Like geocoding, in geotagging textual descriptions of locations are translated into a computer-readable code. Unlike geocoding, in geotagging a whole text document is provided as the input. Thus, besides understanding the textual description of a location itself, the geotagging software needs to identify which parts of the text refer to a location, and which parts do not. Researchers, too, are investigating the field of geotagging. Many focus on the microblogging service Twitter\(^{43}\). For example, [34] analyzes the locations specified on accounts of that service. As on Twitter the location may be specified optionally and is not checked for correctness or validity, a third of all specified data is made up or sarcastically describes non-existing locations or not locations at all. The authors propose a method to derive the country and state of users from their tweets. In [35] and [36] the authors try to actually geocode tweets. Because of the restriction on the length of a tweet, addresses – if specified – are not too blurred up with the remaining text. Therefore, [35] uses classic geocoding services, and relies on machine learning techniques to pick the best candidate location from a set of results. The machine learning model uses the tweet text and the result set, but also various metadata about the user, the tweet, and each result to pick a good candidate. For instance, it chooses a most likely result candidate based on the number of alternative names or the population size. On the contrary, [36] employs natural language processing methods to identify where a location is specified in the tweet. Additionally, the authors developed custom methods to deal with abbreviations or acronyms that are used in tweets specifically. Both studies use Yahoo! Placemaker\(^{44}\) as their benchmark. Both studies successfully beat their benchmark in their respective areas of application. A different view on geotagging is presented in [37]. The authors rely on Google Geocoder to find a location for a tweet. Thereafter, a set of rules evaluating the response parameters are used, to identify incorrect results. With this approach, the authors successfully identify a third of all false positives.

\(^{43}\)https://twitter.com/ (accessed: November 2019)  
\(^{44}\)A geocoding service that was shut down in 2012
Obviously, geotagging is not just practiced on tweets. The wide field of research focuses on geotagging also larger documents as, for example, web pages. For example, [38] experiments applying different methods to geotag tweets as well as Wikipedia articles. Thereby, the authors rely on techniques that are opaque to locations. They split the Earth into a grid, counting which words are used in documents from each grid cell. Afterward, they employ document similarity measures to find the most likely grid cell for a given document.

In [39], the authors chose a more classical approach. They run a sliding window with a fixed size of the document text. A machine learning model uses natural language processing techniques to identify whether a given window contains an address. That decision is taken based on previous windows contents. A request to a common geocoding service yields the geotag for a given window. Language–specific geotagging is assessed in [40] Greek web pages are geotagged using addresses, but also utilizing phone numbers and place names. The authors assemble a framework that geotags web pages but also allows measuring and improving the underlying methods. This is achieved by enabling the user to provide feedback on the correctness of results.

Similarly, [41] is focusing on geotagging Indian web pages. The geotags are deduced by checking if addresses are specified in specific predefined formats. The geotagged documents are then indexed, to provide a location–aware web search that ranks nearby web pages higher than far away ones. Given that specific use case of geotags, the authors also propose how to deal with multiple different geotags on the same document. Another language–aware geotagging approach is measured in [42]. In that study, documents in the Spanish language are translated into English prior to geotagging. Then, the Spanish and the translated English documents where geotagged using a quorum of geotagging systems that have various strengths and weaknesses. Some had better precision, while others had better recall. Some were great for street names, while others could deduce geotags from building names. The authors combined the geotagging systems achieving a much better match rate than the commercial product they used as the baseline. Interestingly, the documents that were translated into English had a higher match rate compared to the original documents in the Spanish language. The main learning, however, of the study is that toponyms are translatable, and geotagging systems should be language–aware therefore.

Language–aware geocoding is a problem that researchers focus on too. The Chinese language is especially hard to work with, as there are no delimiters between words in Chinese text. Chinese words can be one or multiple characters, depending on the characters themselves, as well as the context. That also holds for addresses that consist of the usual address elements. A Chinese address, therefore, is a combination of Chinese characters and numbers, without any separators. Complexity is added as there are various address formats and many exceptions to these in China. Research around geocoding Chinese addresses is ongoing, therefore. In [43], a common address format is proposed and used to geocode Chinese addresses. The authors rely on the natural hierarchy in

---

addresses to define the address model. Using exact querying of elements in that format, a match rate of just over 50% is achieved. A more flexible approach is undertaken in [44]. The authors build on the well-known address formats of geocoders for the US and Japanese markets. However, they account for the exceptions in the address formats by allowing addresses to be free text documents too. With this flexibility, the authors achieve a match rate of over 80% in the study. Another example of geocoding Chinese addresses is [45]. The authors rely on machine learning techniques to parse addresses. With that approach, 80% of the time, correct types are derived for address elements. Given that, it is fair to assume that the match rate would not be lower than that too. In [46], similarly, authors try to parse Chinese addresses using a combination of rules and a statistical model. The model was trained on 460,000 addresses in the Shenzhen Area. The rules were used to handle addresses that are formatted in an exceptional way. The authors achieve remarkable results parsing addresses, but focus on that exclusively and do not create a geocoding system.

An unorthodox use of geocoding is made by [47]. The goal of the research is to associate latitude and longitude information with collected genome sequences. That is only vaguely related to the common understanding of geocoding where addresses and named locations are mapped to latitude and longitude coordinates. As shown in the study, however, genome sequences that are geotagged can be geocoded with similar methods too. While some genome sequences in the data have latitude and longitude coordinates already, others only have textual location description or specify a specimen. Specimen, in turn, are stored in further databases that also contain coordinates or a textual description of the location that specimen resides attached. Resolving the textual descriptions, it becomes assign coordinates to genome sequences creating a map where various sequences exist.

Another twist to geocoding is executed in [48]. The goal of the study is to derive street addresses from post office boxes. That is a non-trivial task, as post office boxes are located in post offices. Instead of the post bringing mail to an address, owners of a post office box go to the post office to pick up their mail. The study is conducted on over 4,500 post office boxes in California. Only 37% of these addresses were leading to a valid street address. However, 81% of post office box addresses led to a misclassification of attributes that, in turn, can lead to a different medical outcome. The authors conclude that researchers in Health need to be trained in geographic information systems and understand the limitation and biases of such data.

Lastly, [49] introduces the temporal component to geocoding. In their work, the authors use multiple data sources to set up a geocoder that can put a location and time on a given query. The authors stress that doing so adds more ambiguity to the process. Also, historical data is much more prone to errors and uncertainties. However, the authors achieve high match rates using their geocoding system on Paris data from the 19th and 20th centuries.
2.4 Comparison of Geocoding Systems

A lot of attention is given to estimate the quality of geocoding systems, without investigating the inner workings of the respective systems. The rationale of these studies is valid. Researchers want to confirm that using geocoding services in specific regions do yield results accurate enough for their respective use cases. For that purpose, data sets of various sizes and sources are used. As a metric, all the works presented in this section compute the match rate. The match rate is the ratio of addresses for which the geocoder was able to produce a result. Further investigation checking whether the result is correct, are rarely made. However, when the ground truth locations of the geocoded addresses are available, the distance between the actual location and the geocoded result is computed. Also, when multiple geocoding systems are compared, the distances between their results are presented. This way, the studies indicate how often geocoding systems agree and how often they disagree on a given address set. A large body of work is available. Here, some select publications from the past two decades are presented in their chronological order.

In 2001, four commercial geocoding companies are commissioned to geocode 70 addresses in the area of Boston, Massachusetts [50]. Out of the 70 addresses, 50 address contain errors. The commercial companies receive the addresses via email. They are also evaluated for their quality of service. The match rate on the given addresses is, depending on the company, between 44% and 84%. The authors of [51] compare proprietary geocoding software using distinct data sets in 2003. One software uses TIGER\(^{46}\) shapefiles. The other uses parcel data from local authorities. They discover that the geocoding system using parcel centroids is more accurate than the one using TIGER shapefiles. The main discovery, however, is that geocoding in urban areas is more accurate than in rural areas for both systems. For example, the authors observed a match rate of over 80% in urban and just 62% in rural areas.

The study conducted in 2004 [52] assessed the geocoding accuracy of three geocoding systems. Similar to previous studies, the match rate was well above 80%. During their analysis, the authors also compared the results of the three systems for the same query. They observed that resulting addresses returned by the system differ from each other. Further analysis showed that resulting addresses also differed from the input address too. That is the reason why in this study, the success rate is used as one of the metrics instead of a match rate. Later, in 2005, in [53], researchers confirmed the findings of [51]. They geocoded addresses in Iowa using more recent TIGER data and compared their results with those provided by a commercial firm. In this study too, rural areas are harder to geocode than cities. The authors observed a correlation of a lower match rate and proximity to agricultural fields. In 2006, in [54] a similar study was conducted in Texas. On sampled addresses, positioning accuracy was validated with a satellite-based positioning system. They found that geocoding addresses

using the Centrus geocoding system\(^{47}\) yields a 10% higher match rate and is more accurate.

In 2007, a multi-stage geocoding system using multiple data sources was compared to a single source geocoding system in [55]. Addresses in Washington state had a match rate of 99% with the system using multiple data sources, and just 95% using a system with one single data source. The authors also measured the distance between the results for the cases that both geocoding systems matched on. They further looked for correlations with errors and confirm the findings of [53] and [51] seeing a higher error rate in rural areas. They also found a correlation between the observed geocoding error rate and the poverty of an area. A higher poverty rate implied a higher population density in that area and led to a smaller ratio of geocoding errors. Another comparison of different geocoding systems is undertaken in 2008 [56]. The authors use orthophotos to assess the actual rooftop locations of the addresses geocoded. They measure the positional accuracy of the two systems and observe that there are significant differences at times. According to the authors, conclusions drawn from geocoding results can vary based on the geocoding system used. A study published in 2010 comes to the same results comparing the five geocoding systems Geocoder.us, Google, MapPoint, MapQuest, and Yahoo [57]. They use addresses from all states of the US and observe significant differences between the results of these systems too.

A similar study compared a geocoding system that uses addresses from the Enhanced 9–1–1\(^{48}\) database with two proprietary geocoding systems. The Enhanced 9–1–1 database of addresses linked to telephone numbers. It is maintained by telephone companies and used to direct emergency services to the address the emergency call came from. In [58] the authors analyze geocoding performance using the Enhanced 9–1–1 addresses in rural areas. They geocode addresses with the proprietary geocoding system ArcView using proprietary data and observe that the positional accuracy and the match rate increase if addresses are transformed into the Enhanced 9–1–1 format prior to geocoding. In a subsequent study in 2011, the same geocoding system had a higher match rate and produced more accurate results with urban addresses [59]. This observation matches previous results. The authors compare two proprietary geocoding systems. They use 748 addresses from various cities in the US. The authors observe the two geocoding systems having a different match rate. However, where both systems yield a result, the distance between the positions is rather small. That is in line with the studies that observed a higher quality of geocoding services in urban areas.

In [60], also in 2011, the match rates of three geocoding systems is evaluated in relation to the housing types at the respective addresses. The authors distinguish between single-family residences in urban or rural areas, multifamily residences, and housing for commercial use. In line with previous observations stating that addresses in urban areas geocode with a higher match rate compared to addresses in rural areas, multifamily residences geocode better.


than the other proposed housing types. In 2011 too the Google geocoding system is evaluated in Belo Horizonte, Brazil [61]. A governmental address database is used as ground truth. The authors observe poor quality of the geocoding system. As sources of error, the authors identify the address token order and the lack of data completeness in the system. Naming ambiguity of address elements introduces additional errors. In contrast to findings made in [55], the authors observe that poorer areas are geocoded with a lower match rate and lower location precision.

The authors of [62] compare two proprietary geocoding systems using addresses from New York state in 2012. They observe different match rates and precision too. The authors suggest combining the methods of the proprietary systems compared to maximize the match rate. Another analysis of geocoding systems is published in 2013 [63]. The authors us addresses from crime locations in multiple areas of the US. They use geocoding systems using different open data sets as well as proprietary geocoding systems. They observe how match rates vary by vendor and data set used. They also observe how various types of crime correlate with higher or lower match rates. As some crime types happen to take place at actual addresses, while others take place on the streets, for some crime types only vague and incorrect addresses are specified. The authors conclude that for most geocoding systems the match rate can easily be increased by reducing the minimum score required to accept a possible candidate result as a match. They stress that this way the ratio of falsely geocoded results is likely to be increased. In 2014, a custom-crafted geocoding system is compared with a proprietary one [64]. Addresses from 14 Canadian cities are used to evaluate the performance. Variants of the custom geocoding system are set up with various data sources. The custom geocoding system has a lower match rate, but a higher positional precision.

Recent studies from 2018 continue to compare various geocoding systems. They, however, expanded to new areas and latest geocoding systems available. In [65], addresses from Zagreb, Croatia are used to evaluate multiple on-line geocoding services. The authors observe that for Zagreb, Google provides the best match rate and positional precision. Similarly, the Google geocoding service is compared with Nominatim in the Bhaktapur district in India [66]. While both systems perform poorly, the OpenStreetMap data based geocoding system Nominatim performs better than Google in that study. The quality of online geocoding services in sub-Saharan Africa is evaluated in [67]. The authors compare the geocoding systems with manual geocoding by human users. Due to the address formats used, or the data available, in that study manual placing of an address reached a higher match rate than any of the geocoding systems evaluated.

This thesis differentiates from the research mentioned in this section as it is not comparing geocoding systems as black boxes. It inspects the impact of a specific feature of a geocoding system in detail by comparing systems that differ in that feature only. Also, as some works pointed out, the match rate used in most of the research presented here only indicates the actual number of correctly geocoded addresses. An incorrect geocoding result due to the lack of
data, naming similarities, or a plain bug will not be observed using this metric. More suitable metrics are used to evaluate the impact of the suggested features.

### 2.5 Data used for Geocoding

Another body of research separates the data out of the geocoding process. The publications presented in this section are investigating the quality of data sets, abstracting from the algorithms that make use of it. Some studies compare authoritative data to other data sources. Others suggest metrics to continuously monitor the quality of data. Most research revolves around OpenStreetMap data. That dataset is especially interesting. While it can be consumed as other free and paid data sources, anyone can also contribute to it. That makes the dataset especially likely to be rich and up–to–date. That also raises the chance for mistakes in the data.

In [68], ground truth data is compared with the contents of OpenStreetMap in Germany. That data is also compared to the contents of the data that TomTom uses. Because TomTom offers services and does not expose the data itself, the view on the contents of the data is only possible through service APIs. Thus, in some cases, data might be available and accurate but not accessible through the APIs for some reasons. The authors observe that both data sets deviate from the ground truth. Also, the authors observe a heterogeneous distribution of errors in OpenStreetMap data resulting in clusters of high and low error rates. In Germany too, in North Rhine–Westphalia, [69] compares OpenStreetMap data with the geocoding service of Google. The same limitation as in the previous study applies. Actual data from Google is not assessed. Instead, data exposed through the APIs of geocoding services are assessed. In this study, data of the Google geocoding system is more complete and has a higher location accuracy. The authors conclude that the geocoding API from Google is superior to OpenStreetMap data. A similar study is made in [70]. Here, the authors assess the accuracy of the data in OpenStreetMap, Google, and Bing data sets. This study focuses on five locations in Ireland and does not find a clear winner. No dataset is most complete in all five locations selected.

In China, OpenStreetMap data is compared to Baidu geocoding system in [71]. In over than two–thirds of the cases, OpenStreetMap data is less detailed than that provided by Baidu. However, over two–thirds of the OpenStreetMap data available, is accurate. Overall both services have poor coverage according to the authors. Interestingly, economically poor areas, coverage is higher in OpenStreetMap data. In [72], addresses in London were checked in the OpenStreetMap dataset. The authors use governmental data as ground truth and observe fairly accurate results with only a few meters deviation and 80% overlap between the two sets. The accuracy of OpenStreetMap data in Teheran is assessed using a similar approach in [73]. The authors use governmental data as ground truth and propose a metric to quantify discrepancies. To do so, the authors suggest to gradually increase the buffer around entities in the canonical data set until the corresponding entity from the measured dataset falls within that buffer.

The larger the buffer had to be increased, the larger the observed error in the data. Using that metric, the authors conclude that there are neither areas with only high-quality data nor areas with very low-quality data in the OpenStreetMap dataset. A focus on building shapes in the OpenStreetMap dataset is taken in [74]. The authors compared data in Munich with governmental data and observed a high shape similarity. However, they discovered many architectural elements missing in the OpenStreetMap data set, concluding that it can be regarded as a simplified version of the actual data.

A different perspective on OpenStreetMap data is taken in [75] and [76]. Similar to [73], both publications propose generic metrics that can be used to quantify the quality repeatedly. This way, so the authors, quality of data can be observed continuously. In [76], data accuracy is assessed using ground truth data sets. That might be problematic, as even canonical data becomes stale at some point. Likely, what is assumed to be ground truth is reflecting the real-world at every point in time. In [75], besides a polygon similarity metric that requires ground truth data too, another metric is suggested. Given that shapes of lakes, forests, or city boundaries are usually complex, the authors propose to use the distance between points of a polygon to estimate its precision. While that is not always a clear metric, it is a good indicator of the quality of the data set in a given area. The authors state that using their metric, they observe that OpenStreetMap data is good for some use cases, and poor for others.

A third angle on the data quality of OpenStreetMap data is given in [77] and [78]. The authors of [77] analyze distinct contributors to OpenStreetMap data, assessing the quality of data for each of them. Using governmental data as ground truth, the authors can grade each contributor, extrapolating to a much more fine-grained quality score. The authors observe the common 90–9–1 rule that applies to many scenarios on the web in OpenStreetMap too. While most OpenStreetMap users do not contribute any content, only one percent contributes most of the content as well as the highest quality content in the data. The remaining nine percent contribute some amount of content with mediocre quality. In [78], the authors start off with monitoring distinct data values on frequently changed objects in the dataset. They observe that data changes back and forth often, with no obvious reason. These changes are also not correlated to the number of contributors to the data. Finally, the authors observe that most errors in the OpenStreetMap dataset stem from manual typos when data is fed in. They call for a cleaner ontology that can be automatically validated, therefore.

Unlike the publications presented in this section, this thesis abstracts from the data used. Using one and the same data source for all variants of geocoding systems set up, the effect of various features of the algorithms is measured abstracting from the data. This way, the impact of each feature becomes apparent and cannot be overseen because the data between the two systems differ. A scenario where one geocoding system is algorithmically strong, but has gaps in the data and performs, therefore, worse than a system that is average in its data and algorithmic methods, is avoided this way.

2.6 Matching of Address Element Names

The matching of address element names, also often referred to as toponym matching, is a crucial part of a geocoding system. Address element names mentioned in a query need to be matched to address element names in the data. For that, a similarity score is computed. If the similarity of two address element names is above a certain threshold, the two names can be considered equal. Thus, both names can refer to an address element this way, even if they differ. There are various algorithms to compute the similarity of any two given strings. Few selected approaches are presented in this section.

One very intuitive way of computing the similarity of two strings is the Longest Common Substring method [79]. The method is parameterized with a minimum length the longest common substring should have. In an iterative fashion, the longest common substring from two strings is computed and removed. If there are multiple longest common substrings, one is chosen arbitrarily. This process is repeated until no common substring is longer than the given parameter or no common substring exist. The similarity score can be computed by dividing the total length of the removed substrings by the average length of the two input strings. Variants exist dividing the length of the removed substrings by the shorter or the longer string. The former results in higher similarity scores, increasing the probability to pick up abbreviations. The latter is a stricter approach that generally results in lower similarity scores. The method has been developed to be applied to personal names, however, can be used on address element names too.

A whole class of measures can be computed with the n-gram technique. An n-gram is a substring of a string of length n. For instance, be, er, rl, li, and in are bi-grams of Berlin. Having chosen an n – the length of the n-grams to work with, the Dice Coefficient [80] of two strings can be computed. For that, the count of n-grams in both strings \( n_{ga} \) is multiplied by two and divided by the sum of the counts of n-grams either string \( n_{ga} \) and \( n_{gb} \). The Dice Coefficient \( d \) between two strings \( a \) and \( b \), thus, is:

\[
d(a, b) = \frac{2n_{ga}n_{gb}}{n_{ga} + n_{gb}}
\]

The Jaccard Index [81] is a variant of the Dice Coefficient. It is computed by dividing the cardinality of the intersection set of all n-grams of both words by the cardinality of the union set of same n-grams. The first set contains all n-grams that are present in both, while the second set contains all n-grams with no condition. The Jaccard Index \( j \) of two strings \( a \) and \( b \) with \( n_{ga} \) containing the n-grams of string \( a \) and \( n_{gb} \) containing the n-grams of string \( b \) is then computed as follows:

\[
j(a, b) = \frac{|n_{ga} \cap n_{gb}|}{|n_{ga} \cup n_{gb}|}
\]

Bag Distance [82], a very simple heuristic for the similarity of two strings uses single-character-grams and the Jaccard formula to compute the similarity measure. Besides having the option of the length, more sophisticated variants of this technique exist. For example, the skip-gram is an n-gram with a gap [83].
Also referred to as open n-grams, characters at a specific index of the substring are ignored. Especially when calculating the similarity of terms across language, the skip-grams have proven to be efficient. Similarly, multi-token text can be compared this way. A precondition for that is the clear notion of where to split the text into distinct tokens. Punctuation or white space characters are a good start for western languages.

Another string text similarity measure is proposed in [84]. The authors specify a generic field matching algorithm and evaluate it on three versions of departments of the University of California, San Diego. The algorithm is explicitly tailored to detect four kinds of common abbreviations. For that, the algorithm tries to split the text such that equal elements are left, or abbreviated variants are left together with their spelled-out counterparts. The algorithm is then applied recursively to each split; various splits are tried out. In the end, each split that cannot be split up further gets a score. The score is set to one if the texts are exactly equal or one is the abbreviation of the other, or zero otherwise. The score of joined splits is the mean of the maximum scores of the sub-splits.

Another class of string measures is commonly referred to as the Jaro-Winkler methods. Originally, the algorithm to compute string similarity was proposed in [85]. The algorithm counts the necessary transpositions $t$ – characters that need to be replaced, inserted, removed, or swapped – to transform one string into the other. Additionally, the string lengths $l_1$ and $l_2$, as well as the number of common characters $c$, are computed. The Jaro similarity $j$ between two strings $s_1, s_2$ is then computed as follows:

$$j(s_1, s_2) = \frac{1}{3} \left( \frac{c}{l_1} + \frac{c}{l_2} + \frac{t}{c} \right)$$

In [86], the author extended the algorithm to have a higher weight on characters at the beginning of the strings. This suggestion is based on other research that observed little typing errors at the beginning of words and most typing errors towards the end. Other modifications of the original algorithm involve predefined string pairs that are considered to have a similarity of one or a modification of how common characters are counted. A list of non-equal character pairs is proposed to be counted as a third of a common character. Among various other string similarity measures, however, the original Jaro algorithm with only the Winkler extension scored best in their class in [87]. Another extension accounted for multi-token text. It suggests to compute the Jaro-Winkler similarity on all permutations of token pairs and use the maximum average value.

Another well-known similarity measure is the Levenshtein measure, first published in [88]. Like the Jaro similarity measure, the Levenshtein algorithm is accounting for characters that need to be replaced, inserted, removed, or character pairs that need to be swapped. The publication specifies an algorithm to compute the minimal number of such edits that is necessary to transform one string into the other. This required amount of edits is referred to as the Levenshtein distance. Dividing the Levenshtein distance of two strings by the length of the longer string gives the similarity score of the two strings. Many variants and extensions of the Levenshtein algorithm exist. Specifically,
2.7 Approaches to Geocoding

Naturally, geocoding itself is part of research as well. In this section, selected publications that open the black box of a geocoding system are presented. Some create entire geocoding systems from scratch using novel ways to store, query and retrieve data. Some others investigate and suggest improvements to very specific aspects of geocoding systems. And others modify the interfaces allowing more control to the human user.

One group of research goes beyond evaluating geocoding systems as black boxes. They look inside the systems and identify inner workings and possible sources of error. Using these, they suggest approaches to match more queries to more accurate results. In [89], the authors do exactly that. Describing every step of the geocoding process the authors list various error scenarios. Their basic geocoding system normalizes the query text prior to querying the data index. The geocoding system iteratively relaxes the matching constraints until at least one result is found, or the maximum relaxation has been reached. The same authors extend their error scenarios into a metric in [90]. They suggest a metric that is not qualitative, but distributed in space and accounts for the various geocoding stages. Thus, among other factors, it incorporates the data accuracy of the result area, the normalization uncertainty, the matching certainty, and the interpolation accuracy. The resulting metric can be used by the client to decide how to consume a result, if at all. In the same year, the same authors further expand their metric. In [91], they propose a formula to compute the certainty of results coming from multiple data sources. Using the certainty of matches in their respective data sources, as well as the topological relationships between those matches, a more accurate score, and hence a better ranking of results is computed. This way, a more accurate geocoding system with a higher match rate and a lower deviation from the ground truth position is possible.

A different set of studies focuses on fixing the data. Every geocoding system can only be as good as the data it uses. The main purpose of the geocoding system is to retrieve the correct entity from the data. No geocoding system is capable of correcting the location if the geocoordinates of addresses in the data are all off. In [92], for example, the authors analyze different interpolation algorithms of three different geocoding systems. Their goal is to identify which algorithm produces results with the best positional accuracy. The conclusion of the study, however, reveals all three interpolation algorithms yield similar error rates. As

different weights and different edit operations have been experimented with. In fact, the class of Jaro–Winkler methods described in the previous paragraph is a direct successor of the Levenshtein algorithm.

In this thesis, the Levenshtein similarity measure is used to decide when two non–equal strings are to be considered equal. A specific modification is proposed here so that abbreviated terms can be matched to their spelled–out counterparts. In the end, however, the solution proposed in this thesis is not bound to any specific string similarity algorithm.
the underlying data sets are not accurate, an evaluation the algorithms is not feasible.

In [93], the authors continue the analysis of sources of error of a geocoding system. They review empirical studies to confirm that substantial bias may be introduced by results of erroneous geocoding. The authors focus on the underlying data and suggest refinements to the address data models as the main area for future research. In [94], the authors try to enrich the data of a geocoding system by additionally using data from OpenStreetMap and Wikimapia\(^{51}\). They experiment with using support vector machines [95] as the mechanism to rank results from multiple sources. In the city of Chennai, India, the authors identify over 88% of data stemming from the auxiliary source. This makes the approach to enrich data a promising start, though the positional accuracy of the used data sources is poor, according to the authors. A very similar approach is taken in [96]. The authors set up a geocoding system that can integrate arbitrary data sources. They use a gradient tree boosting model to rank the results of multiple data sources for a given query [97]. As a result, they present an open-source geocoding system that can rely on multiple data sources and be trained to rank the best result first.

Another batch of research focuses on the matching of names of address elements. The studies [98] and [99] evaluate string similarity measures discussed in Section 2.6 for matching address element names. Address element name matching scores using various measures and address pairs are computed in [98]. In [99], in addition to the measures discussed in Section 2.6, a new measure is proposed. While [98] comes to the conclusion that there is no measure that works best overall, as address element names vary by language, [99] finds that their proposed measure has the best recall in various countries. The measure is a combination of techniques; a maximum score out of three computed variants is assumed to be the measured similarity. While not for the sake of geocoding, addresses from multiple databases are matched to each other in [100]. The study is conducted to estimate the average energy consumption of households, depending on their sizes and room counts. The authors claim that matching addresses field-wise, similar to a qualified geocoding request, enabled them to join two independent databases into one.

A similar task is solved using machine learning techniques in [101]. Not having qualified addresses at hand, the authors use conditional random fields [102] to first label address elements with their types. Next, the word2vec [103] approach is used to compute field similarity between the addresses. Finally, logistic regression [104], random forest [105], and gradient tree boosting methods are applied to the per-field similarity scores to decide whether two addresses are the same or not. The authors give an overview of the precision and recall of the various techniques applied. While it is not feasible to use these approaches to quickly retrieve data from an index, they can be employed to filter out falsely returned candidates.

Research on the core functionality of geocoding systems is ongoing for several decades too. As early as in 1974, the authors summarize the early state of

2.7. Approaches to Geocoding

They observe that geocoding is performed by highly skilled subject matter experts only supported by computers. They state that geocoding is highly manual and highly sophisticated work. The authors are proved right in their assumption that the use cases for geocoding systems will set the direction of development. Nowadays, with the general-purpose use case of geocoding systems, they have become more accessible and require little training to be used. In 2003, the authors propose a complex but complete address model for Brazil addresses in [107]. They set up a geocoding system using that address capable of handling point addresses, address ranges, as well as entire regions. In their work, however, the authors do not take care of queries from arbitrary human users. The authors of [108] set up a geocoding system using governmental data in 2004. They employ a Hidden Markov Model [109] trained to understand address element types of a query. After query terms are tagged with address element types, the geocoder executes a qualified search for a fitting address.

In 2006, in [110], the authors pick another approach. Prior to geocoding the entire query, query terms are looked up in the data first. The authors use metadata of address entities to derive which one is referred to with a query term in the case of multiple options for a single name. Then, the query is enriched with alternative names of the address elements mentioned. Alternatively, the authors experiment with expanding the address element names in the documents as opposed to the query. It is their finding, that expanding the query or the documents improves the match rate of a geocoding system. In [111], the authors compare two geocoding systems in depth. One is geocoding to address entities in the data, the other capable to interpolate house number points from street segments. The authors evaluate the performance of both geocoding systems. They conclude that neither approach is absolutely superior to the other one. Depending on the use case one or the other approach, and hence, one or the other geocoding system perform better.

In the study [112] in 2007, authors look for sources of bias in the process of geocoding addresses taken from a crime database. They explicitly focus on the difference between gaps in the data and a non-matching query. They describe in detail, how the geocoding system they use functions. It first normalizes the query, using looking up query terms in a dictionary and replacing them with a normalized value. Then, it queries the data falling back to fetching entities with names that are spelled differently, but phonetically equivalent. Finally, the retrieved candidates are scored based on their similarity to the queried address, ranking the highest scored result first. This classic set up of the geocoding process and the thorough analysis of the study reveals multiple spots where specific addresses are fetched more likely than others. This creates a bias that might affect conclusions drawn from statistics that are based on geocoding results.

In 2010, in [113], the study performed in [111] is extended. Using five general-purpose geocoding systems, they analyze the impact of using interpolated results compared to point addresses. Unlike in the previous study, in [113] the authors observe that accepting interpolated results enables a higher
match rate, at the cost of positional accuracy. A different angle on geocoding is taken in 2011 [114]. The authors set up a system of independent agents. For each address element type, an agent analyzes the query and retrieves candidates that the query may be referring to. Communicating with other agents, scores of the candidates are adapted until a certain overall maximum is reached. For example, the score of an address element of type city grows if the score of an address element of type district grows and that district is part of the respective city. The geocoding system is encapsulating the independent agents, returning the candidate address that achieved the maximum score as the result. In [115], published in 2016, the qualified approach to geocoding is described in detail. The geocoding system described first tries to derive the address element types that are specified in a query. In a second phase, repeatedly, qualified queries are sent against the data, while relaxing the search criteria until either result is found, or the most relaxed search criteria did not yield any result. An iterative geocoding system is suggested in 2018 [116]. The authors try to increase the match rate of their geocoding system. To do so, they set up a geocoding system that executes in three steps. First, the query is geocoded as is. Next, the geocoding result is used for a reversed-geocoding query. In that step, the latitude and longitude coordinates of the result from the first step are used to find the geographically closest entity in the data. In a final step, the original query is enhanced with address elements that are part of the result from reverse geocoding. In addition to that, the authors implement various text pre-processing steps, such as expanding abbreviations, prior to executing step one. In the study, multiple ways to decide how to select the right result from the reverse-geocoding step are discussed. Overall, so the authors, this approach does increase the match rate of their geocoder.

In 2018, an overview of query normalization techniques is given together with approaches to query parsing [117]. Thereby, the authors focus on addresses from the US. Like previously described studies, the authors call for a unique address format that ought to be adopted by all users. As a novel idea, the authors suggest a common set of normalization rules to be used for all geocoding systems. That set can be collaboratively maintained by all developers of geocoding systems. The most recent study presented here is [118], published in 2019. The authors pick up the work from [56], where orthophotos were used to compute the positional accuracy of geocoding results. Instead of using orthophotos as validation data, the authors employ deep learning methods [119] to extract rooftop locations from them. These locations are then used as locations of results that otherwise would have been interpolated. The authors achieve an equally good match rate, but a much greater positional accuracy this way.

A different branch of geocoding research tries to bring the human user into the loop. In [120], the authors present a mobile online game, where humans are asked to describe their location. The study categorizes how locations are described by humans, counting what type of description, for example, country, city, street name, store name, house number, floor, or piece of furniture, are used. The study identifies many classes of location elements with varying precision. The most used location identifiers, as discovered by the study, are
2.7. Approaches to Geocoding

street name, house number, and building name. In [121], the authors establish an interactive geocoding system. They combine two classical geocoding systems with a user interface that allows the users to manually improve the geocoding result. The user interface comes to use whenever the two underlying geocoding systems disagree. The user can fix the data or select the appropriate result in that case. Making geocoding more user-friendly is clearly the right direction of development.

As the last batch of research, three publications will be presented here that are most closely related to this thesis. As in this thesis, in [122], [123], and [124] the general purpose document search engine Elasticsearch is used to index address entity names. In [124], in 2018, OpenStreetMap addresses are indexed in Elasticsearch together with auxiliary data. The author uses a very tiny set of hand-selected addresses to evaluate the performance of the geocoding system. Published in 2017, the author indexes Geonames data in Elasticsearch [123]. This work focuses on the programming language, the software stack, the deployment process, and the interface of the geocoding system. However, an interesting approach to derive the country of the location queried for is chosen. A Word2vec machine learning model is trained to infer the country from the query terms.

In 2015, the author extends the OpenStreetMap geocoder Nominatim with Elasticsearch [122]. The main purpose of this extension is to enable the suggest-as-you-type feature that Elasticsearch provides out of the box. With this feature, the system suggests to the user the address entity name, that the user is likely to type before the user has completed typing. Similar to the auto-complete feature of mobile phones address entity names are suggested. This is of great help, as it suggests the correct spelling of the entity name to the user. Typing errors are much less likely this way. In addition to this feature, Elasticsearch is used to re-rank results from Nominatim, increasing the performance metrics of the geocoding system. The system was evaluated on Swedish addresses in two ways. First, the self match rate – the match rate of addresses taken from the data of the system was computed. Additionally, a relatively small set of addresses with generated typing errors was evaluated. Both approaches for evaluation are also used in this thesis, though the number of test cases is multiple orders of magnitude larger. It is worth mentioning that besides scientific interest, there also is the project Pelias. Pelias is an open-source development project that intends to develop a geocoding system that uses Elasticsearch as the data index. Like in this thesis, OpenStreetMap data can be imported into the Pelias geocoding system.

Overall, this thesis differentiates itself from the other publications mentioned in this section. Using exactly the same set of data, all analysis is focused on algorithmic changes exclusively. Also, as success metrics, the match rate is not used. As discussed, the match rate is not accounting for cases that matched to the wrong result. The much more suitable metrics success rate, precision, and recall are used throughout this thesis. Finally, unlike most of the work

presented here, addresses specified by human users are in the focus of this thesis. The geocoding systems proposed do not require an interactive use case, where the user is prompted support while typing in the query. Neither is the user presented with candidates from which the correct result should be chosen. The most minimalist interface of a geocoding system possible is used in this thesis. A query, specified by the human user and therefore possibly containing typing errors, and, at most, one single result to that query. This simple interface makes it possible to not get caught up in candidate results and edge-case scenarios and allows measuring the algorithmic performance of the system.
Postal addresses are hard to process for computers because there is no canonical format that is always used. Different, contradicting formats are used to specify postal addresses, depending on the region. Hence, to know which format is used knowledge about the region is necessary. To deduce, which region the address is describing, it needs to be geocoded. This vicious circle is hard to break. To make things worse, human users do not adhere to any postal address format at all. It’s the nature of addresses that they do not need to be understood entirely by everyone. Those who are aware of address entities and their names in a given area comprehend addresses without any format. And those who do not understand an address in its entirety are satisfied with a vague location of a known high-level address entity as city or region usually.

In the following chapters, a series of experiments is conducted, trying to create a better geocoding system with each next step. There are various ways to define the success of such a system; they all depend on the specific use case at hand. For this thesis, the non-interactive use case is chosen. The geocoding systems under test are expected to return one result at most. This suits the needs of automated geocoding processes. Computers having one address as the input have no means to choose a correct result from a list. Thus, even if a list is returned by a geocoding system, for such a use case a machine would opt for the top result as it is presented as the most likely one to be correct. The non-interactive use case suites the needs of a human user too, however. While, actually, a human can interact with a geocoding system, specifying more precise requests and picking the correct result, a system with good quality of service does not require that interaction. Ideally, any machine understands immediately the needs of the users and presents the correct results. Thus, while capable to interact, human users do prefer the non-interactive use case too. It is worth pointing out that most of the geocoding systems available for use are returning result lists with candidates. The lists are ordered by different algorithms, trying to ensure that the top result is the correct response to the query. Thus, these services can easily be adapted to serve the non-interactive use case too: Simply returning only the first candidate from the result list, or no result at all if no candidate is available to ensure the respective geocoding system serves the non-interactive use case.

While the experiments will be presented starting with the next chapter, in this chapter the tools, concepts, and data the experiments rely on are described in detail.
3.1 Geographic Coordinate System

One common way of encoding locations in a computer-readable way is WGS84 latitude and longitude coordinates [126]. Representing points on the surface of a globe and, specifically, the Earth with latitude and longitude coordinates is generally credited to Eratosthenes of Cyrene who lived during the second century BC. Two poles on two opposite sides of the globes are selected. From these two planes are chosen. One plane crosses the two poles and, hence, also the axis between the two poles. It splits the globe into two equal halves. An angle to that plane is defined as the longitude. The other plane is perpendicular to the first plane. It too splits the globe into two equal halves on the equator line. The equator is the line most distant from both globes. Each half has the pole on their top, therefore. An angle to that plane is defined as the latitude. Figure 3.1 illustrates this. Both planes cut through the sphere; the vertical plane cuts through the poles while the horizontal one cuts through the equator between the poles. The latitude, denoted with $\varphi$ is the angle to the horizontal plane. It spans from $-90^\circ$ to $90^\circ$ and specifies the location between the poles. The longitude, denoted with $\lambda$ is the angle to the vertical plane and spans between $-180^\circ$ and $180^\circ$. The longitude specifies the location on horizontal circles. Together, the
two angles specify a vector that starts in the center of the globe. The location on
the surface of the sphere that the vector touches is the location specified by the
coordinates. To rely on this approach, however, the center of the globe needs
to be unambiguous, which is not the case as the Earth is not an exact sphere.
Various centers and poles can be assumed as a basis for using the latitude and
longitude coordinates. Depending on where the center of the Earth is set, a
cordinate system is more accurate in some regions while less accurate in other
regions of the world. WGS84 is specifying one center of the Earth and is the most
common geographic coordinate system used.

3.2 Postal Addresses

It is important to comprehend postal addresses before jumping to the process
of geocoding. Postal addresses specifically, as well as addresses in general,
are the main method humans use to describe locations. Obviously, when two
persons communicate about a location in an area that is well-known to both,
more detailed descriptions of a location can be used. However, "the big tree" and
"the yellow bridge" are not good location identifiers for persons from different
cities, for example. Addresses, on the other hand, serve that purpose quite well.
Amongst important attributes of an address, there are (i) a variable precision that
ranges from a sole country name, over region, state, city, and district to street,
building, or even apartment, (ii) a coherent hierarchy amongst the listed address
elements, as well as (iii) the option of adding additional arbitrary information
when necessary to reduce ambiguity or increase precision. This way, addresses
can be used to convey information in a very flexible way. An address can be as
precise as necessary to describe a location. At the same time, an address can
be consumed by a person that is not knowledgeable in a given area. Because
address elements of higher hierarchy levels are mentioned, a vague localization
is possible right away. When navigating to an address or looking it up on a map,
the same hierarchy can be resolved step by step to get to the encoded location.
Additional information can be consumed during this process to encounter any
possible difficulties in resolving.

Historically, addresses grew together with administrative or political
boundaries. Address elements in top levels of the address hierarchy, therefore,
specify the country, the geographical region, the state, or the city. Formally,
addresses are assembled from address elements that, most often, contain each
other. Thus, in an address, the house number is contained in a street, which, in
turn, is contained in a district, which is part of a city, and so on all the way up to
the address element representing the country. Because of the historical context,
most of the address elements describing these levels are language-aware too.
However, although administrative boundaries change over time, some times,
postal address elements remain in place. This way, named areas such as SoHo¹,
Greenwich Village², or Little Italy³ in Manhattan, New York, effectively span as

much as humans communicating about that area think they do. These areas, even if no longer administrative, can be used as parts addresses too. Because of the historical development of address, and the lack of a formal oversight on that, address element names are commonly reused. Besides naming cities in honor of common heroes or other cities, oftentimes address elements from different levels share the same name. For instance, there are dozens of cities named Alexandria after the great conqueror Alexander III of Macedon. While a series of these cities were established during various campaigns, many more cities with the same name have been founded without the direct influence of Alexander III. Similarly, the capital of the USA shares the name Washington with a state, both named after George Washington the first president of that country. In Europe, roads are named after the city they lead to. Thus, Potsdam and Berlin are two cities that both have streets named after their neighbor. This overlap in names introduces a level of ambiguity to addresses. Sometimes, especially when not all address elements are specified, an address might refer to multiple locations equally.

Postal addresses have the purpose of delivering postal mail or packages to a specific person or organization. Thus, next to the specified location, postal addresses specify a person or organization to be found at that location. This adds a new dimension to the address, as, in part, the location can be specified by the addressee. While the north pole is not a precise location as its location has been updated multiple times, the postal address "Santa Clause, North Pole" has a clear destination. In general, a recipient might be specified by name or role, or, in the case of an organization, not be specified at all. Alternatively, modern postal services can deliver the mail to a postbox that might be specified by one single number. These postboxes, essentially, are abstractions from classical postal addresses. Multiple persons from various locations might be able to access post boxes. In the context of location, and therefore throughout this thesis, postboxes are not taken into account.

A different aspect introduced for the purpose of delivering mail, however, encodes location information. With the growing population in cities and the flexible and at times ambiguous organization of addresses delivering post became a complex task. Various towns with similar or equal names being part of the address, for example, might confuse the resolving of an address. Additionally, common street names made it likely that mail got delivered to the wrong address. Already since the middle of the 19th century, big cities as London or Liverpool were divided into postal areas. In the early 20th century this concept got extended to the entire country first in the Ukrainian Soviet Socialist Republic. Since then, most countries identify areas using postal codes. Due to the nature of their purpose, however, postal codes do not fit well into the hierarchy of addresses. Because postal codes were created to deliver postal mail, their size is chosen so that they cover a roughly equal number of recipients. Thus, while urban areas with a dense population are split into many smaller postal code areas, rural areas are split into fewer larger chunks. As a consequence,
some cities are the hierarchical parents of multiple postal codes, while in other cases, single postal codes contain multiple cities. Postal codes are elements of addresses that not strictly contain or are contained in specific address elements, therefore. To be precise, such breaches of the strict hierarchy have always existed in addresses prior to the introduction postal code areas. Streets are often times longer and might pass more than one administrative area. The informal route of the Panamericana makes it pass through many countries. For such scenarios, instead of expecting a strict hierarchy in addresses, humans implicitly switched to expecting an overlap of the listed address elements. This way, postal codes fit perfectly well into addresses, reducing the ambiguity of some addresses that are hard to keep apart otherwise.

The Universal Postal Union\(^6\) (UPU) is a specialized agency of the United Nations\(^7\) organization coordinating the work of postal services. There are 192 member countries in the union. Amongst many other tasks, the UPU manages the official ways to specify postal addresses in the various countries. Table 3.1 lists example addresses in the formats specified by the UPU. It is easy to spot that all examples specify a postal code. That is not surprising, as the UPU providing these address examples is explicitly representing the use case of postal mail. Another interesting insight is that, indeed, even without local knowledge, some rough understanding of the locations on the globe stem from cities, or at least country names. Also clearly visible: Not only are address elements specified in a different order in various countries. Even within a single country, the ordering of the elements is not sequential. Instead of listing the elements from large to small or vice versa, pairs of elements seem to be grouped arbitrarily and the ordering differs from country to country even within groups. For instance, street name and house number usually appear next to each other. However, while this group is specified at the beginning of a German or French address, it shows up after the subdistrict in Turkey. Also within this group, the house number is specified before the street name in France, but after the street name in Germany. In Japan, where subdistrict numbers are used instead of street names, the subdistrict number is always followed by the house number but may be prefixed by a zone number that is otherwise specified together next to the district of the address. To make things more complex, additional information such as a district name or a building name can appear on predefined spaces, while at the same time address elements as street names or house numbers might be missing in rural areas.

In sum, a human might not comprehend the meaning of each address element of an address in an unknown area. A human can, however, identify known entity names such as country, city, or district name to roughly understand which location that address encodes. Yet, to fully resolve an address by following through its hierarchy, the specific address format needs to be known upfront. In the context of geocoding services that cover the entire world, this is especially hard as address formats are flexible and differ in so many ways. [1] analyzes various address formats and how they can be processed in more depth.

### Table 3.1: Example addresses from various countries in formats specified by the Universal Postal Union.

<table>
<thead>
<tr>
<th>Country</th>
<th>Address</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Germany</strong></td>
<td>Ernst-Reuter-Platz 7, 10587 Berlin</td>
<td>Street name, house number, postal code and city. The most common address format used.</td>
</tr>
<tr>
<td></td>
<td>Ortsteil Neukoelln, Alt-Britz 73, 12359 Berlin</td>
<td>According to the UPU, a district can be specified prior to the street name.</td>
</tr>
<tr>
<td></td>
<td>Rhondorfer Str. 666, Apartment 47, 50939 Köln</td>
<td>Similarly, an apartment number can be specified after the house number.</td>
</tr>
<tr>
<td><strong>United States of America</strong></td>
<td>1600 Pennsylvania Avenue NW, Washington DC, 20500</td>
<td>House number, street name, city and postal code. The sole address used format in USA.</td>
</tr>
<tr>
<td><strong>United Kingdom of Great Britain and Northern Ireland</strong></td>
<td>10 Downing St, Westminster, London SW1A 2AA</td>
<td>Typical British address containing house number, street, district, city, and postal code.</td>
</tr>
<tr>
<td></td>
<td>1A Seastone Cottages, Station Road, Weybourne, Holt, NR25 7HG</td>
<td>Dependent backstreet may be specified additionally after the house number.</td>
</tr>
<tr>
<td></td>
<td>1 Upper Littleton, Winford, Bristol, BS18 8HF</td>
<td>Some house numbers are valid in entire villages without street names.</td>
</tr>
<tr>
<td></td>
<td>Appleford, Abingdon, OX14 4PG</td>
<td>Sometimes, no house number is present on an address.</td>
</tr>
<tr>
<td></td>
<td>Victoria House, 15 The Street, Hurn, Christchurch, BH23 6AA</td>
<td>The house name may be specified next to the house number.</td>
</tr>
<tr>
<td><strong>France</strong></td>
<td>25 Rue des Fleurs, 33500 Libourne</td>
<td>House number, street name, postal code, and city make the most common address format.</td>
</tr>
<tr>
<td></td>
<td>Entrée A Bâtiment Jonquille, 25 Rue de l’Eglise, Caudos, 33380 Mios</td>
<td>Additional geographic information can be specified before the house number.</td>
</tr>
<tr>
<td></td>
<td>Le Village, 82500 Auterive</td>
<td>In small hamlets, house numbers are not used.</td>
</tr>
<tr>
<td><strong>Turkey</strong></td>
<td>Doğanbey Mahallesi, Şehitt Teğmen Kalmaz Caddesi 28/A, 06101 Ulus/Ankara</td>
<td>Subdistrict, street name, house number, postal code, district, and city specify this address.</td>
</tr>
<tr>
<td><strong>Japan</strong></td>
<td>10–23, Mitsugi 1–Chome, Musashi–Murayama–shi, Tokyo, 208–0032</td>
<td>Subdistrict number, house number, district name, zone, town, city, and postal code of an address.</td>
</tr>
<tr>
<td></td>
<td>2–17–10, Aicicho, Naka–ku, Yokohama, 231–0012</td>
<td>Zone, subdistrict number and house number can be grouped in the beginning of an address. The address can be specified in almost complete reverse order too. Only the pairs district name and zone, and subdistrict number and house number are not allowed to be reordered.</td>
</tr>
<tr>
<td></td>
<td>112–0001 Tokyo, Bunkyo–Ku, Hakusan 4–Chome, 3–2</td>
<td></td>
</tr>
</tbody>
</table>

### 3.3 OpenStreetMap

OpenStreetMap is a project to crowdsource map data in the style of a wiki. Everybody is able to register and upload map fragments. Everybody is able to download and consume the map data for free. The OpenStreetMap data is shared
under the Open Database License ODbL. Essentially the data is free to use for everyone, however, if the data is augmented in any way, the augmented data needs to be published under the same license too.

The approach of OpenStreetMap is very generalist. The data is collected and made available without any specific application in mind. Therefore, the format chosen is very raw. Listing 3.1 presents an example of the format the OpenStreetMap is available in. Besides the presented XML format the data can be downloaded in the Protocolbuffer Binary Format PBF. While the XML format is—somewhat—human-readable, the PBF format is much more space-efficient. The internal representation of the data, however, is the same in both formats.

Three element types node, way, and relation are used to model the entire map. All three element types come with common attributes id, user, uid, visible, version, changeset, and timestamp. The identifier is used to keep notes, ways, and relations apart from each other. User and uid attributes contain the user name and, as that can change over time, the identifier for that user who contributed the specific element. The visible flag is set to false for historic entities that should not be used for map rendering or any other use case. This way, if a user

---

Listing 3.1: Snippets of the OpenStreetMap XML format [127].

```xml
<osm version="0.6" generator="CGI-map 0.0.2">
  <bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900" maxlon="12.2524800"/>
  ...
  <node id="298884269" lat="54.0901447" lon="12.2565153" user="SvenHR0" uid="46882" visible="true" version="1" changeset="6947637" timestamp="2011-01-27T14:28:49Z"/>
  ...
  <way id="261726866" lat="54.0906309" lon="12.2441924" user="PikoWinter" uid="36744" visible="true" version="1" changeset="9396786" timestamp="2008-06-03T13:39:23Z"/>
  ...
  <relation id="56688" lat="54.0898880" lon="12.2491680" user="Masch" uid="65988" visible="true" version="6" changeset="4142606" timestamp="2010-03-16T11:47:08Z"/>
  ...
  <member type="node" ref="292704400" role=""/>
  ...
  <member type="way" ref="4579143" role=""/>
  ...
  <member type="node" ref="249673494" role=""/>
  ...
  <tag k="name" v="Küstenbus Linie 123"/>
  ...
  <tag k="operator" v=""/>
  ...
  <tag k="route" v="bus"/>
  ...
</relation>
</osm>
```
is identified to provide invalid data, all entities provided by that user can be set to not visible. For updates of existing entities, the version attribute is used. Identifiers of uploads of batches of data are stored in the changeset attribute. Finally, the timestamp attribute stores the last modified time in ISO-8601 UTC format.

Nodes have the additional attributes lat and lon for WGS84 coordinates. Hence, nodes are the most fine-grained element in OpenStreetMap data representing specific points on the globe that the map is made of. Besides the attributes for latitude and longitude coordinate, nodes can be tagged with key-value pairs. For instance, a tag with a corresponding key can be used to store the elevation of a point over sea level. Arbitrary data can be put into tags, and each key is allowed only once on an element. Because keys cannot be reused, keys are assembled in a hierarchical manner. For example, a node that is representing a monument clearly should have a name tag. Adding additional, for instance, translated, transliterated, or outdated names to that node is possible by specifying the type of the name in the key. The node could have an additional tag with the key name:alt for an alternative name of that point on the globe. Ways and relations can be tagged too. The structured keys of the tags allow attaching very detailed information to the various entities. The OpenStreetMap wiki describes how tags should be used. It defines a set of known keys for tags, and what their meaning is. Similarly, the acceptable values for each key that the community has agreed upon are listed in the OpenStreetMap values. The general approach of the community of OpenStreetMap users is that once agreed, the way to model the data about the real-world is stored in the wiki.

The next bigger elements in OpenStreetMap data are ways. Ways do not have coordinates themselves but contain links to an arbitrary amount of nodes. With this approach, paths and polygons are modeled. A way, for example, can contain the footprint of a building or the extent of a lake. To disambiguate between these two, ways are augmented with tags describing their name, category, or any other attribute required.

Relations are the largest elements in the OpenStreetMap data model. They can reference nodes, ways, and other relationships. Like all entities, relations are tagged with key-value pairs to specify their meaning. This way, one relation can model a complex building footprint as, for example, a polygon with a hole in the middle. For that latter use case, the member element not only specifies the type and identifier of the referenced element but also its role. For example, the way describing an inner polygon of a building will have the role attribute set to inner. Another relation can reference the modeled building, as well as other buildings describing, for example, a university campus. Country borders are modeled as relations that reference smaller relations to model administrative areas. A relation can also reference nodes to, for instance, specify where the capital of a country is, or where to put a name tag when rendering a map.

When extracting a subset of the OpenStreetMap data, an additional bounds element is added as the first child of the root element osm. The bounds element

---

3.3. **OpenStreetMap**

Overall, there are over five million users registered to contribute data to the OpenStreetMap project. One million of active contributors provide on average three million changesets every day. There are over five billion nodes in the database describing the real-world entities with 575 million ways and six and a half million relations. Figure 3.2 shows the development of counts for node, street, and relationship elements over time.

According to the tags, the relation on line 30 in Listing 3.1 models the route of the bus number 123 of the VVW lines in Rostock and Warnemünde. Hence, the ways it references describe the route of the bus, while the nodes specify the various bus stops. The tags with the keys `type` and `route` give away that it is a bus route. The tag `ref` specifies the bus number 123. Additional tags specify more details such as the bus name, operator, and network. Using this data, an application could render the bus route and the stops on a digital map. The bus stop locations are separate from the route itself and can be displayed on the street sides.

Line 21 of the example XML presents a way that refers to nodes by their identifier in the `nd` tag. It also is tagged with two tags. One specifies the name of this way. The other specifies that the way is a section of a highway that is not part of any highway class. The node on line 11 has tags with the keys `name` and `traffic_sign`. One, obviously, specifies the name of the real-world thing that is modeled by this node. The other tag specifies the type: At the latitude and longitude coordinates of this node, a traffic sign, specifically, a city limit sign is located.

It is important to measure any system on a variety of nuances. At the same
time, development work and experiments are slowed down immensely, when a data set is used that is too large. Hence, a medium–large area containing a multitude of countries has been selected to be used for this thesis. Geofabrik\textsuperscript{12}, a German company offering products and services around OpenStreetMap data splits the large data set in regional chunks that are made available for download. OpenStreetMap data for Europe was downloaded from the Geofabrik servers and used here.

In sum, the data format of OpenStreetMap data is very flexible and hence versatile. That flexibility comes at a cost. Because the data format is not designed with a specific application in mind, for most applications the data need to be preprocessed before use. For instance, ways represent road segments, however, as ways only reference nodes, there is no explicit linkage from one segment to another. In OpenStreetMap data, two segments are connected as soon as they share the same nodes. Thus, no routing algorithm can run on top of a road network stored in the OpenStreetMap format. The graph of streets needs to be derived from the data in a dedicated preprocessing step. Similarly, for a geocoding application, all we need are nodes that are tagged with the various address elements. In Listing 3.1 on line 28, a name tag specifies the street name of the street segment described by the way entity. However, no higher-level address elements are present. Similarly, most nodes representing house numbers, only have tags specifying that. Thus, addresses need to be assembled out of OpenStreetMap entities first, prior to being indexed in a geocoding system.

Another unfortunate aspect of a flexible data format is that it does not enforce correctness. That is especially problematic when data is gathered by enthusiasts as opposed to professionals. There are many inconsistencies in the OpenStreetMap data, and cleaning and polishing it is a large task for itself. Luckily, there are open-source geocoding services using OpenStreetMap data already. They implement the cleansing and assembling mechanisms required to create a usable system. Nominatim is the de-facto standard for geocoding with OpenStreetMap data. It will be introduced in the next section.

\section{3.4 Nominatim}

Nominatim is an open-source tool to search through OpenStreetMap data. It supports searching by name and by address providing results that contain geocoordinates. Nominatim also allows reverse lookup for the closest entity at a given location. With this feature set, Nominatim is the de-facto standard geocoder using OpenStreetMap data. According to the documentation\textsuperscript{13} of Nominatim, three main modules can be discussed separately. The Nominatim repository contains a script wiring the three modules together so that executing one single command sets up the entire service. The three modules are discussed in depth in the next passage.

\textsuperscript{12}https://www.geofabrik.de/ (accessed: November 2019)
\textsuperscript{13}http://nominatim.org/release-docs/latest/ (accessed: November 2019)
Before anything else, the data is imported into a PostGIS\textsuperscript{14} enabled PostgreSQL\textsuperscript{15} database. To render OpenStreetMap data, as well as for various analysis purposes, it is common to load it into a PostgreSQL database first. The tool osm2pgsql\textsuperscript{16} does that. It creates tables for three entity types node, way, and relation as well as some auxiliary tables and populates them with data. Figure 3.3 presents a simplified view of the created database schema. As one can see, some data is stored in an optimized way. For example, tags are stored in simple text arrays. Thereby, every element with an even index is the tag key, followed by an element with the tag value. Similarly, members of relations are persisted in a text array with the member identifier followed by the member role. To differentiate between the element types of relation members, the identifier is prefixed with an \textit{n}, \textit{w}, or \textit{r} for node, way or relation. That requires rows in the \textit{members} column to be of type string. The can no longer be used in join conditions, therefore. Thus, an additional array containing the unchanged identifiers of the members of a relation is stored in the \textit{parts} column of the \textit{relation} table. As ways only ever reference nodes, the schema of \textit{way} table is simpler. An array of identifiers of nodes is stored in the \textit{nodes} column. Note that \textit{lat} and \textit{lon} columns in the \textit{node} table are of type integer. That is counterintuitive as latitude and longitude coordinates stored in them are floating-point numbers. Storing the coordinates as integers is an efficient optimization. The documentation specifies that the coordinates are rounded to seven decimal figures. Also, we know that latitude and longitude only require up to three additional digits as they range between $[-90, +90]$ and $[-180, +180]$ respectively. Thus, to store the coordinates only ten digits are required in total. As that fits into signed integers in PostgreSQL, we can easily transform latitude and longitude floating-point values avoiding any floating-point precision issues.

After data has been loaded into PostgreSQL using osm2pgsql, Nominatim-specific tasks get executed during the address computation stage. Creating over one thousand auxiliary tables, Nominatim harvests addresses and points of interest from the OpenStreetMap entities. It also indexes their names and ranks them by importance. One crucial problem with this approach is that this importance ranking is also used to filter results, independent of a query. Nominatim computes two ranks that are combined when ordering results: One rank is for points of interests and is solely based on the category the respective OpenStreetMap entity is tagged with. The other rank specifies where in the address hierarchy an entity is located. Thus, while the query is only used to

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{node} & \textbf{way} & \textbf{relation} \\
\hline
id bigint not null & id bigint not null & id bigint not null \\
lon integer not null & nodes bigint[] not null & parts bigint[] \\
l lat integer not null & tags text[] & - \\
tags text[] & - & members text[] \\
- & - & tags text[] \\
\hline
\end{tabular}
\caption{The simplified database schema for OpenStreetMap data.}
\end{figure}

select the right results, filtering out wrong ones, it has no influence on which of the selected results is returned on top. As we show in this thesis, that leads to suboptimal results.

At the end of this process, besides several tables containing various meta-information with statistics and for efficient lookup, the three main tables `placex`, `word`, and `search_name` are populated. Figure 3.4 presents the simplified schema of the tables. The table `search_name` contains all necessary information to represent the indexed elements. It contains the identifier of the element in the `place_id` column, and the name vector - the array of identifiers of tokens of the name in the `name_vector` column. Additionally, it contains columns to specify the rank of each indexed element. As discussed, these columns are populated by Nominatim at indexing time and are not computed dynamically for each given query. The name tokens, as well as token n-grams, are stored in the `word` table. A name token is a part of the name. Usually, names are broken up into tokens by separating them at the whitespace character. However, oftentimes non-alphanumeric characters are used to split the name on. The `word` table assigns all tokens of all names a unique identifier. Using this identifier Nominatim can look up which names a token is part of in the `name_vector` column of the `search_name` table. Besides single tokens, the `word` table assigns identifiers to n-grams of tokens. N-grams are a series of tokens with a specific order. Usually, n-grams are indexed to detect if the same tokens in the same order are part of a query. Nominatim stores entire names that consist of multiple tokens as a multigram. For statistical reasons, the `word` table also stores how often that token is part of a `name_vector` of a `search_name` in the `search_name_count` column. This way, tokens that appear in most names can be detected as stop words and dropped. The `placex` table contains, finally, contains the elements themselves. The `place_id` column in the `search_name` table references the column with the same name in the `placex` table. Besides the identifier, the table contains the columns `osm_type` and `osm_id` that provide the necessary information on the source of that element. The type is one of `n`, `w`, or `r` for node, way, or relation. The `osm_id` column contains the identifier to retrieve the source in the respective table populated by osm2pgsql in the previous phase. The columns `class`
and type are categories that are extracted from tags on the source elements and are returned together with a result. The name column of type hstore contains a dictionary of names. Thereby, similarly to the organization of tags on the source data, the name types are used as dictionary keys, while as the names are stored as the dictionary values. The columns house_number, street, postcode, and country_code of the placex table are self-explanatory. Finally, in the column geometry, the polygon, line, or point of each element is stored. In addition to that, column centroid contains the designated centroid for each element to be returned as part of the query. Entire geometries, on the other hand, are only part of the result if explicitly requested in the query. Besides all this information, the placex table contains an identifier to the parent place in the column parent_place_id.

As you might have noticed, there is no entry with a complete address on any element yet. All three tables contain additional columns for various features that are not further discussed here. Given a place_id, the address can be assembled using placex table. Specifically, the parent_place_id of each row specifies the identifier of the row for the next higher-level address element in the database. Thus, following that value will get all address elements from house number to country. Listing 3.2 shows a version of the function doing
so, augmented with comments. The stored procedure uses the two additional procedures `get_addressdata` and `get_name_by_language`. The former of the two, `get_addressdata` is the one that iterates over the `parent_place_id` references. On the path, it decides using various auxiliary functions and tables which of the many address elements on a fully hierarchical path to keep as part of an address, and which ones to drop. The latter function, `get_name_by_language` tries to find a name in the dictionary of names, that has the desired language. If no matching names are found, a default name type to fall back to is picked from the available name types. Using these stored procedures, the addresses are assembled at query time prior to being returned as results. Listing 3.3 shows the query that calls the method assembling addresses and extracts them together with their latitude and longitude coordinates. Additionally, part of the projection is the identifier of the address itself, the identifier of the street segment the address is located at, as well as the house number. These metadata can be used to group addresses assembled by Nominatim in various ways. The array of name types defined in line five is the same that Nominatim uses to assemble query results.

The final module to mention is the Nominatim front-end. It is written in PHP and, in the default setup, served by the Apache HTTP server. The front-end implements the Nominatim API accepting queries from users, querying the database for results, and returning them in the response. For instance, the front-end supports serializing responses in XML or JSON formats. As a first step to serve a geocoding query, the front-end module fires multiple queries against the database, trying to As part of the query mechanism, Nominatim tries to make sense of the query tokens. For instance, it checks if tokens contain numbers, or match a certain pattern to do so. The module also queries the database for types of tokens that were observed while importing. Having tagged query tokens with types, the front-end assembles search queries that are sent to the database. Thus while the front-end does not strictly assume specific address formats for queries, it does make use of patterns in tokens and the query to anticipate the possible meaning of an unqualified query. This kind of qualified search is restrictive as multi-token names of address entities might not be completely specified by the user. The tokens specified might occur in the context of various address types. That, in turn, results in multiple possibilities of how a query can be interpreted and yield incorrect results. To mitigate this, however, the users have to specify the exact name of each address element they

---

query for so that all tokens of the name get the correct type assigned.

Nominatim is used as a baseline geocoding system in this thesis. Throughout various experiments, the performance of Nominatim is presented next to suggested alternative approaches. It is worth pointing out that Nominatim is a mature product and therefore offers a feature set richer than the systems proposed throughout this thesis. For example, Nominatim supports the search of points of interest by their names, as well as the geocoding of entities at higher administrative levels as districts or cities. As the focus of this thesis, however, is to establish a geocoding approach for full addresses containing a house number, the comparison with Nominatim is not entirely fair.

### 3.5 Elasticsearch

Multiple geocoding systems are set up and evaluated in this thesis. Instead of a PostgreSQL database, the document search engine Elasticsearch is used for these geocoding systems. It is a general-purpose open-source document search engine that offers features to be operated conveniently encapsulating a Lucene index. As the main feature, Elasticsearch provides an HTTP interface to create, read, update, and delete entries in the underlying Lucene index. Indexed or retrieved documents, queries of any complexity, and even partial document updates are thereby encoded in the JSON format and sent or received as defined in the HTTP protocol. Besides that, Elasticsearch offers features as search clusters with automated fail-over, data sharding with backups in case of failing nodes, or cross-cluster replication. One very convenient feature of Elasticsearch is that it does not require to specify a document schema upfront. Unlike similar solutions, Elasticsearch adapts the data type of the document fields dynamically depending on what the fields contain. It is a mature product that is used to search through logs, documents, or addresses in many scenarios.

Lucene is the underlying index that persists data to disk in a way that allows it to search for documents. Like Elasticsearch, Lucene is an open-source project. Under the hood, it builds an inverted index with document tokens pointing to documents they appear in. With the need to tokenize documents in various ways, it comes with a rich choice of various tokenizers, for example, suitable for different languages. After tokenizing a document, Lucene keeps track of token counts globally, as well as per document. With these statistics, the index can quickly compute the relevance of a document to any given query as follows: (i) First, the tokenizer that has been used to break up the document field queried for into tokens is applied to the query text. (ii) Next, all documents containing tokens from the query text are scored. The score of a document for each query token is computed. Thereby, that score is growing, if the token appears in the document more often. Similarly, the score of a document for a token is shrinking, if the token appears in many other documents. These two metrics are exactly the statistics that Lucene keeps track of. The ratio between these two is referred to

---


as term–frequency inverse document–frequency or TFIDF. The score of a document for the entire query is simply the sum of the scores of that document for all query tokens. (iii) Finally, the \( n \) best–scored documents are returned. The formula for the TFIDF score is very intuitive, as it reflects exactly the relevance of a document to a query. In its most simple form, the formula is defined as follows:

\[
\text{tf}(\text{token, document}) = \text{# of occurrences of token in document}
\]

\[
\text{df}(\text{token, corpus}) = \text{# of documents token occurs in at least once}
\]

\[
\text{tfidf}(\text{token, document, corpus}) = \frac{\text{tf}(\text{token, document})}{\text{df}(\text{token, corpus})}
\]

\[
\text{tfidf}(\text{query, document, corpus}) = \sum_{\text{token in query}} \frac{\text{tf}(\text{token, document})}{\text{df}(\text{token, corpus})}
\]

Note how the token–specific score defaults to zero if that token is not contained in a document. Also, as addition is commutative, the order of the tokens in the query does not influence the TFIDF score.

There are many variants of TFIDF that allow a more accurate scoring of documents for a query. For instance, the importance of a document should not grow linearly and indefinitely with the number of occurrences of a token in that document. Also, the occurrence of a token in a short document should have a different impact on the score compared to the same token occurring in a long document. Okapi BM25 [130] and BM25f [131][132] are two variant of TFIDF that accounts for these items. The impact of a token occurring in a document is reduced with every occurrence. Also, the document length is used to normalize the score. Additionally, BM25f accounts for the occurrence of tokens in various fields of the document. Specifically, while it computes the term frequency for each field independently, the document frequency of a term is computed over all fields. Because common tokens might appear often in some fields, but only appear rarely in others, this approach yields the best scoring of documents for a given query. While Elasticsearch is sufficiently flexible to allow a custom scoring with every query, BM25f is used by default.

Listing 3.4 shows how the API of Elasticsearch can be used. First, a simple HTTP PUT method is used to initialize an empty index with the name `greetings`. Next, in lines 3 and 7, two documents are indexed. Looking at the URL, note that after the index, the document type `short_greetings` is specified. Note how

---

3.5. Elasticsearch

different HTTP methods are used to index the documents. In both cases, the
document data is sent in the body of the request to Elasticsearch. In line 3,
however, only the index name and the document type are specified in the URL.
No document identifier is given requesting Elasticsearch to generate one. This
is done using the POST method. In line 7, on the other hand, the document
identifier is part of the URL. Hence, the PUT method is used to index the
document. This behavior is in line with the definition of the HTTP commands
and makes Elasticsearch a RESTful [133][134] service, therefore.

Listing 3.5 presents how the index with the two documents can be queried.
The _search part of the URL indicates that the query defines a search on the
documents. The index name greetings and document type short_greetings
are specified in the URL too. The search query itself is specified in the URL parameter
q. It sets Elasticsearch up to look for documents with the field text matching the
query text hey. As this example query is specified in the URL, the request body
is left empty. More complex queries, however, can be specified as part of the
request body in the JSON format. Independent of where the query is specified,
the HTTP method GET is used to query the index. That too is in line with the
definition of HTTP commands.

As expected, the result list contains both documents as both of them match
the query. The query consists of one token only. Both documents contain that
query in their text field. The JSON document returned by Elasticsearch, however,
contains various metadata. Besides some statistics on the involved shards,

Listing 3.5: Search query sent to Elasticsearch using CURL
command. jq25 is used to pretty-print the result.

```bash
curl localhost:9200/greetings/documents/_search?q=text:hey
--request GET
--silent | jq
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 2,
    "max_score": 0.39566286,
    "hits": [
      {
        "_index": "greetings",
        "_type": "short_greetings",
        "_id": "2hey",
        "_score": 0.39566286,
        "_source": {
          "text": "hey, hey, hey!"
        }
      },
      {
        "_index": "greetings",
        "_type": "short_greetings",
        "_id": "7there!
        "_score": 0.2876821,
        "_source": {
          "text": "hey there!"
        }
      }
    ]
  }
}
```
execution time, or the number of documents matching, for each document of
the result list the index name, the document type, the identifier, as well as the
score are given. The document contents are available under the JSON key _source.
Besides the index name and the document type set to values specified during
indexing, we can see the generated and the specified document identifiers in
lines 20 and 29 of the listing. Also, note how the document with the identifier
2hays has a higher score compared to the other one. That is how the TFIDF–like
approach scores the documents to the query. The document frequency of the
query text hey is equal for both documents. However, the term frequency of the
first result is higher compared to the term frequency of the second one: The
token hey appears two times in the first document and only one time in the
second. Therefore, the results are scored and ordered accordingly.

Throughout this thesis, Elasticsearch will be used to create geocoding
services. Similarly to Elasticsearch, the geocoding services will expose a RESTful
API. The aim is to find document structures that model addresses in a way
best–s suited for geocoding. Additionally, the data contained in those documents
will be augmented such that user–specified queries are handled as accurately as
possible.
This chapter extends the research, already published in [2]. With a few simple queries against Nominatim, one can easily demonstrate inconsistent results. It sometimes yields Points of Interest (POI) instead of addresses; in other cases, it returns addresses on streets not queried for, even if the address queried for clearly exists in the data and is visible on the plotted map. Table 4.1 illustrates cases of a strange behavior of Nominatim. Of the first two queries in Chicago, the first one clearly requests West Randolph Street. However, additional results for Upper and Lower East Randolph Street are returned. These two additional results for Upper and the Lower East Randolph Streets are valid results for the second query, but neither of them is returned. The next two rows present queries for non-existing house numbers in Berlin. For historic reasons, Linienstraße starts with house number 13. One would expect the system indicates that yielding the entire street, or possibly falling back to the closest existing house number. Nominatim, however, yields an entirely unrelated street with the queried house number instead. Querying for another non-existing house number exposes different behavior yielding no results at all. The last two queries in London show how the order of the elements of a query affects the result. Specifying the house number after the street name yields a POI on the specified address before the address. That the ranking changes depending on the input token order is unusual. Putting the house number before the street name adds a result for the entire square first, presenting the POI and the address in rank two and three. A change of results in the response presented to the user is unexpected too.

To some extent, such behavior can be rooted in the underlying data. Also, the data model used by Nominatim can contribute to this behavior. Finally, ranking POI results higher than plain address results is a choice made by the developers of Nominatim. The system might simply not be optimized to be used for address search.

The goal of the effort described in this chapter is to establish a good baseline for an address search system using a generic document search engine. Ideally, it would handle queries like those in Table 4.1 in a more natural way. As discussed earlier, address elements have implicit hierarchical relations between them. For a simple approach, however, entire addresses can be treated as opaque documents with no specific meaning of the various terms. Such address documents can be indexed and queried for using the document search engine Elasticsearch.
Table 4.1: Results yielded by the Nominatim web service¹.

<table>
<thead>
<tr>
<th>query</th>
<th>results and notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>425 west randolph, chicago</td>
<td>yields the correct result and two more results for 425 Upper East Randolph Street and 425 Lower East Randolph Street in Chicago.</td>
</tr>
<tr>
<td>425 east randolph, chicago</td>
<td>yields no results</td>
</tr>
<tr>
<td>linienstraße 10, berlin</td>
<td>yields one result for Mollstraße 10, Berlin – a street nearby, but not directly at Linienstraße. Note that there is no house number 10 on Linienstraße.</td>
</tr>
<tr>
<td>linienstraße 1, berlin</td>
<td>yields no results. Note that there is no house number 1 on Linienstraße.</td>
</tr>
<tr>
<td>london, belgrave square 34</td>
<td>yields a POI – the British–German association as the first result. The result for the address comes second.</td>
</tr>
<tr>
<td>london, 34 belgrave square</td>
<td>yields the Belgrave Square first, followed by the POI and then the address.</td>
</tr>
</tbody>
</table>

4.1 Hypothesis

As discussed, Elasticsearch is a general-purpose document search engine that uses BM25F to score documents with regards to a given query. The formula computes a weighted score for every term of the query. As we will query for addresses, our queries will have multiple terms. The scores of these terms are simply added up by Elasticsearch. Because addition is commutative, there is no dependency on the order of the query terms.

Elasticsearch will return all documents containing query terms in their score order. As address queries do contain generic terms such as cities or postal codes, many documents will match every query. Commonly, results below a certain score are cut-off from the list, so that documents matching only a few or only generic terms are no longer considered as results. Only documents matching the query well enough to produce the minimum score are looked at. In this chapter, for the baseline geocoder, we will cut off all results after the first one, independent of their score. This way, we receive at most one result – the addresses with the best score – for every query. While that seems very drastic for an interactive use case, it is a feasible approach when a geocoding service is consumed by an automated process. Also, it makes the metric of a geocoding system very straightforward. A response contains at most one result. That result may be correct or wrong. For the baseline geocoding system, we want to measure the ratio of successfully served queries. Thus, we want to count the ratio of correct results.

In contrast to the proposed geocoding system, Nominatim scores all addresses upfront and independent of queries. The expectation is that Elasticsearch will make a superior geocoding service, therefore.

¹https://nominatim.openstreetmap.org (accessed: November 2019)
4.2 Data for Evaluation

The evaluation data used is one of the most important aspects of every experiment. As this chapter is about establishing a good baseline, there is no need for a very sophisticated set of queries. Two simple address data sets of addresses are used here; for evaluation, a set of test queries is generated from these addresses. For once, entire addresses are used as queries. Additionally, address tokens are shuffled randomly. Also increasingly more and more tokens from the address are dropped. Table 4.2 shows example queries generated from the data sets.

<table>
<thead>
<tr>
<th>data set sample query</th>
<th>sample query</th>
</tr>
</thead>
<tbody>
<tr>
<td>sampled from index 5, Rue des Remparts, Vaux, Auxerre, Yonne, Bourgogne, France métropolitaine, 89000, France with 70% tokens shuffled 5, 89000, Rue des Remparts, Bourgogne, France métropolitaine, Vaux with 30% tokens shuffled Rue des Remparts 89000 5</td>
<td>Rue des Remparts 89000 5</td>
</tr>
<tr>
<td>scraped from the web Badstraße 34 13357 Berlin with 70% tokens shuffled 34 Berlin Badstraße</td>
<td>34 Berlin Badstraße</td>
</tr>
</tbody>
</table>

### 4.2.1 Sampled Addresses from the Index

For this set 2000 addresses that are present in the source data indexed are uniformly sampled. In accordance with the source data, the sampled set contains addresses from various countries in Europe. The benefit of using indexed data is that checking the responses for a match is a very simple task. Each indexed address has a unique identifier. When sampling addresses for the evaluation set, the identifier is kept. It is also stored when a query is generated from a sampled address. Thus, a result with the same identifier is the correct result of a query. As all but the top result are cut off, responses with the correct result in the first place are successful responses.

From the sampled addresses from the index, an entire collection of test query sets is generated. Increasingly, in steps of 5% more and more tokens are dropped from these addresses, resulting in queries with all 100%, 95%, 90% tokens all the way down to only 25% of address tokens being part of the test query for each address. Additionally, all the tokens of each query are shuffled. These test query sets with 2000 addresses each are meant to naively simulate user queries that tend to be incomplete or with address elements mixed up. For comparison, a test query set with all address tokens present and in their correct order is created too. This way, a collection of test query sets with increased complexity is made available. In each query set, every query keeps the identifier of the address it was generated from, making evaluating a response for success an easy task. Thus, a query is considered to be handled successfully, as soon the result returned has the same identifier as the original address that the query has been generated
from. The rate of successfully served queries is, thus, stricter than the match rate measured in many of other studies as it is checked whether the result is correct or not.

### 4.2.2 Scraped Addresses from the Web

Additionally, addresses, as used in the real-world, make up a good test set too. The official website of German pharmacies\(^2\) provides a list of pharmacies that offer emergency service, grouped by city. 1000 addresses of pharmacies, as presented on that website, are randomly sampled. Note that unlike with the previous address sets, there is no simple indicator to decide if a query was served successfully. However, the website provides the position of the listed pharmacies as GPS latitude and longitude coordinates. These coordinates can be used to assess the success of a response. Besides identifiers, geocoding responses contain GPS coordinates too. The distance between the expected and the received coordinates implies a successful or an incorrect response. For each of the 1000 sampled addresses, a manual check proves that it is present in the data indexed.

Similarly to the addresses sampled from the index, addresses scraped from the web are shuffled. Random address tokens are dropped too. These addresses have the common address format used in Germany: street name - house number - postal code - city. Oftentimes, city names, street names and house numbers are only a single token each; the German postal code is always just one single token. This way, many of the scraped addresses in this data set only have four address tokens in total. Therefore instead of increasing the ratio of dropped tokens gradually in 5% steps, 25% of tokens are dropped only once. For most of the addresses, this means that exactly one part of the address is missing. Thus, just two test query sets are generated from web-scraped addresses. One containing queries with full addresses as they are presented on the pharmacy website, and the other containing queries with 75% address tokens randomly shuffled.

### 4.3 Implementation

To set up a geocoding system based on the same data as Nominatim, first, the address data needs to be extracted out of the PostgreSQL database that Nominatim uses. As discussed, a table in the database contains all the required information of an address entity. Besides other fields, the data include the ID, the ID of the parent, the house number, the name, and the latitude and longitude coordinates of each entity. To get to a fully assembled address, Nominatim

---

\(^2\)www.aponet.de (accessed: November 2019)
4.3. Implementation

utilizes an SQL function that, given an ID, follows the parent ID pointers over the address hierarchy. Thereby, the function determines which address elements to specify as part of an address, and which ones to omit. As house numbers are specified in a dedicated column, a simple select statement for all entries with a value in that column produces the entries for house number addresses that we want to extract. The SQL query simply needs to project the results into the columns ID, latitude and longitude coordinates, and address text. The latter one, thereby, is the result of the SQL function Nominatim uses to assemble addresses. With this simple approach, all full house number addresses are easily extracted into a CSV file with the four fields. For indexing in Elasticsearch, each line of the CSV file is transferred into the JSON format that Elasticsearch uses. Listing 4.1 presents a document as it can be indexed in Elasticsearch. Also, to keep the index size at a minimum, Elasticsearch is set up to only index the contents of the address display_name field. Note that the fields in the document have the same names as the fields Nominatim uses in responses. Nominatim provides lots of additional information as the bounding box, the type, or if requested, the distinct names of the qualified address elements. However, display_text, place_id, lat, and lon are present in documents and Nominatim responses. Thus, if only the five fields present in both systems are consumed, documents out of Elasticsearch can be consumed with the same code as results from Nominatim. Note that mimicking the interface of Nominatim is the sole reason why the document fields for latitude and longitude are of type string. While using floats for these fields is much more appropriate, we gain simplicity when comparing this and the future geocoding systems with Nominatim.

One important aspect of the experiment undertaken is that the data in the two systems under test are identical. Addresses indexed as plain documents in Elasticsearch have been extracted from Nominatim. Thus, every address that is available in Nominatim is also indexed in Elasticsearch, and vice versa. This makes comparing the two systems straightforward. We can directly see how well each system performs with regards to any given test query set. Figure 4.1 illustrates the data flow from the raw OpenStreetMap file into Nominatim, then extracted into a file of Addresses that are imported into Elasticsearch.

Figure 4.2 outlines the experiment set up. The test runner is the central actor here. It reads test queries from query set files that contain either IDs or the geocoordinates next to the query text. The test runner then uses the
HTTP interface to query the system under test. Nominatim provides a convenient interface that allows querying the service with simple HTTP-GET requests. The address query is thereby encoded as a URL parameter. Elasticsearch provides an HTTP interface too, however, as it is a general-purpose document store, it is much more complex and exposes functionality not needed for this experiment. Therefore a simple wrapper is encapsulating Elasticsearch in the same way that Nominatim encapsulates the PostgreSQL database. The wrapper provides the same query interface that Nominatim provides. It then assembles and executes a query against Elasticsearch. Finally, it wraps the results. It only keeps the top result from Elasticsearch cutting off all subsequent ones. Listing 4.2 presents the Python implementation of the wrapper. A simple function, parametrized by a query, implements the Elasticsearch API transparently to the caller. The function returns at most one result extracting only the document with the same field names that Nominatim uses. Therefore, a similar wrapper for Nominatim has been implemented so that only the top result of both systems is considered. Encapsulating these wrappers in an HTTP interface is a straightforward task that is not shown here.

The responses are evaluated by the test runner. If the test query contains an identifier, the ID of the first result is checked for a match. Test queries with identifiers are categorized in successfully served and failed requests this way. Alternatively, if the test query did not contain an ID but specified expected latitude and longitude coordinates instead, the results are put into distance buckets. Identifying successful results by expecting exactly the same coordinates is not feasible. Because in reality addresses span an extent and are not a point on the globe, it is very unlikely that two data sets contain exactly the same coordinates for one and the same address. Addresses of pharmacies that were scraped from the web likely contained coordinates from a different source than OpenStreetMap. Thus, there is no reason to expect that the coordinates attached to the test queries will ever equal to coordinates in Nominatim or our geocoding system. Therefore, the distances between results and queries coordinates are considered. Distances between query and result are categorized into buckets of $0m - 100m$, $100m - 1000m$, and $1000m - \infty$. Additionally, a bucket for no reply
4.4 Results

is available too. While different, coordinates from different address data sets for the same address should not be 100m apart. That distance is chosen as an acceptable deviation that would not confuse users too much when arriving at an inexact location. Thus, the smallest bucket will contain all successful geocoding cases. The counts of cases in the other two distance buckets can provide insight into the severity of a false result. After executing all queries in a file and categorizing the result as successful or failed, or by computing the right bucket for it, the test runner stores the computed statistics into a file. For reference, Listing 4.3 presents the functionality of the test runner processing one single query. Calling the measure_single method in a simple loop while accumulating the returned results, entire test query sets can be evaluated easily.

4.4 Results

For the first two data sets of addresses taken from the index, Figure 4.3 displays the numbers of successfully served queries. Interestingly, neither Nominatim nor Elasticsearch managed to retrieve successfully all 2000 addresses even though they were taken from the indexed data itself. That is possibly a result of duplicated address data in OpenStreetMap. Even if multiple entries refer to the same address, each entry has a distinct identifier. When all entries are indexed in the geocoding system, there is no determinism on which of the multiple entries if returned if their common address is queried for. Hence, if a test query is generated from an entry that is duplicated, chances are that even if the response is technically correct it still will be categorized as a failure due to contradicting identifiers of query and result. While this problem will be tackled in a later chapter, it is clearly visible that Elasticsearch performs better than Nominatim even with unshuffled, complete addresses.

When address tokens are shuffled, Elasticsearch is still capable to find most of the addresses queried for. As one would expect, for Elasticsearch the ratio of successfully served queries drops proportionally to the ratio of address tokens dropped out of the query: The fewer address tokens are left in a query, the less likely it is that the tokens specifying the most–precise aspect of an address, for
def measure_single(query, expected_pid, expected_lat, expected_lon):
    results = dict()

    # iterate over available geocoding method pointers and names
    for method, method_name in ((es_wrapper, 'es'),
                                 (nominatim_wrapper, 'nominatim')):
        # execute query and get response document
        result = method(query)

        if expected_pid:
            # query came with identifier, check for success or fail
            if result and result['place_id'] == expected_pid:
                results[method_name] = 'success'
            else:
                results[method_name] = 'fail'
        else:
            # distribute result and no result into appropriate bucket
            if result:
                result_lat, result_lon = result['lat'], result['lon']
                dist = distance_on_earth(
                    expected_lat,
                    expected_lon,
                    result_lat,
                    result_lon)
                if dist < 100:
                    results[method_name] = '0m-100m'
                elif dist < 1000:
                    results[method_name] = '100m-1000m'
                else:
                    results[method_name] = '1000m-infinity'
            else:
                results[method_name] = 'no-result'

    return results

Listing 4.3: Method to compute spelling variants given a query.

e.g., the house number, are still part of the query. Thus, the less likely it is
to retrieve the correct result.

Nominatim, on the other hand, cannot deal with queries that have shuffled
address tokens at all. As discussed, Nominatim tries to guess the address format
and discover street, house number, etc. in the query before looking up data.
That approach doesn’t work with a random token order that a user might use.
Interestingly, more shuffled tokens are harder to handle for Nominatim than
fewer initially. The ratio of successfully served queries first grows unexpectedly,
reaching its sweet-spot at approximately 70% of address tokens left in the query.
Afterward, it degrades again. Generally, for every query generated out of an
address taken from the indexed data, Elasticsearch outperforms Nominatim.

Table 4.3 presents the results for the latter two test data sets of addresses
scraped from the web. Even with complete addresses, Nominatim failed to
geocode 5.4% of the queries that Elasticsearch managed to serve successfully.
With shuffled addresses that miss 25% of address tokens, the success rate of

<table>
<thead>
<tr>
<th>full addresses</th>
<th>0m-100m</th>
<th>100m-1000m</th>
<th>1000m-infinity</th>
<th>no result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominatim</td>
<td>946</td>
<td>0</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>75% mixed tokens</th>
<th>0m-100m</th>
<th>100m-1000m</th>
<th>1000m-infinity</th>
<th>no result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominatim</td>
<td>399</td>
<td>13</td>
<td>0</td>
<td>588</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>988</td>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.3:
Nominatim and Elasticsearch on web-scraped pharmacy addresses.
Nominatim drops to 39.9%. Elasticsearch still manages to serve 98.8% of queries successfully with shuffled and incomplete address tokens. Interestingly, in 58.8% of queries that confront Nominatim with that benchmark the service yields no result at all. Elasticsearch, on the opposite, provides a result for all the queries stated, the vast majority of which are within a kilometer of the expected result.

4.5 Conclusions

Overall, Elasticsearch with addresses indexed as plain documents outperforms Nominatim as an address geocoder. That is because Nominatim requires queries to adhere to the implemented addressing schemes, as discussed earlier. Also, that explains why the performance of Nominatim on queries with shuffled address tokens goes up at first: Fewer tokens can be mixed up in fewer ways. Therefore, the chance to hit one of the implemented and supported address schemes is higher. BM25F based ranking of documents to queries, on the other hand, does a decent job here. As expected, it handles shuffled address tokens correctly.

It is worth pointing out that Nominatim does have a richer feature set compared to the baseline geocoding system presented here. For example, only house number addresses have been indexed in Elasticsearch, but no higher-level entities as streets, districts, postal code areas, cities, etc. Also, translations of entity names or POIs have been left out although addresses of POIs - where present with house number - have been taken. However, out of the box, the established baseline is performing better than Nominatim for address queries at least. Also, in line with the measurement results, it does not demonstrate the same strange behavior as Nominatim. Table 4.4 presents results to queries stated at the beginning of this chapter. For the addresses in Europe no more
Table 4.4: Results yielded by the baseline geocoding service indexing addresses as documents.

<table>
<thead>
<tr>
<th>query</th>
<th>results and notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>425 west randolph, chicago</td>
<td>not tested as US addresses are not indexed in our system</td>
</tr>
<tr>
<td>425 east randolph, chicago</td>
<td>same as above</td>
</tr>
<tr>
<td>linienstraße 10, berlin</td>
<td>yields Linienstraße 135, in Berlin, as there is no house number 10 on Linienstraße the system falls back to a random result</td>
</tr>
<tr>
<td>linienstraße 1, berlin</td>
<td>same as above</td>
</tr>
<tr>
<td>london, belgrave square 34</td>
<td>yields the correct result: 34, Belgrave Square, St. James’s, Belgravia, City of Westminster, London, Greater London, England, SW1X 8NX, UK</td>
</tr>
<tr>
<td>london, 34 belgrave square</td>
<td>same as above</td>
</tr>
</tbody>
</table>

Unexpected behavior is present with the new system: Queries for non-existing house numbers fall back deterministically to one same result. Also, the order of the tokens does not affect the results anymore.

In sum, the baseline geocoding system is a decent and simple starting point for us to add functionality to. While geocoding systems have been set up based on Elasticsearch before, in the experiment undertaken the feasibility of this approach is measured and confirmed. The goal is to develop a feature-rich geocoding system handling user queries well over the course of this thesis.
5 Indexing Structured Addresses

In the previous chapter, we have established a simple baseline for a geocoding service. It is based on BM25F scoring of address documents with no structure. Addresses, as discussed earlier, have an implicit structure in them: House numbers are only unique within streets; street names repeat in different cities or districts; postal code areas usually cover geographies of whole countries. In sum, addresses are organized hierarchically. Usually, every next level entity of a postal address describes an area fully contained in the geography described by the previous one.

There are exceptions to this rule, of course. Postal codes areas have not grown organically with postal addresses. They have been introduced at a time when addresses were already established. Their purpose is to make delivery of mail simpler. Therefore, in urban areas, postal codes are more fine-grained. Cities and districts often contain multiple postcodes, therefore. In rural areas, on the other hand, postal codes can cover several villages in a large area. Multiple villages, hence, can share the same postal code. Thus, postal codes have no specific slot in the hierarchy of postal addresses. Some areas in neighboring countries are served by the postal organizations of the respective countries and have multiple postal codes, too. Other address entities are affected by such exceptions as well. Especially where regions are disputed, or where, historically, areas were joined or split over time postal addresses are less strictly hierarchical often.

As discussed, however, it is not the loose hierarchy that is problematic for processing postal addresses automatically. The bigger obstacle is the fact that there are little rules in composing a postal address. House numbers can go before or after the street names, depending on the region the address is referring to. Some regions even use area names instead of street names. Others use house names instead of house numbers. Elsewhere, house numbers are structured to describe one of the multiple buildings or even reference a specific floor of a house.

In this chapter, we are adapting the document format of our baseline geocoder to reflect some of the implicit hierarchy of postal addresses. Specifically, we are modeling the hierarchical aspect of addresses on house numbers only. Thus, address documents are referencing street segments, possibly as large as an entire street. All full addresses that are on a street segment are stored in the document that references the respective street segment. The full addresses are stored in

---

1For example, Büsingen am Hochrein is a part of Germany entirely surrounded by Switzerland. It is serviced by both postal offices and has the postal codes CH–8238, and D–78266 therefore.
Figure 5.1: The street Linienstraße is split up into segments that differ in postal codes. House number addresses with all their meta-data are stored in the segments that they belong to.

a map data structure, keyed by the house number. The values of the map are the same documents that are indexed in the baseline geocoding system. The findings in this chapter have already been published in [3].

Note that this way, house number interpolation be implemented. This is a common geocoding feature as it allows to algorithmically mitigate gaps in the data computing likely locations of house numbers that are missing in the data. Given the location of a preceding and a succeeding house number, the most simple version of the interpolation algorithm splits the space between these numbers evenly for all missing house numbers in between. The location is only estimated this way, but the accuracy usually suffices for common geocoding use cases as navigation or statistical analysis. In this thesis, we are not utilizing this feature, however, as it could hide a possible regression behind additional results geocoded this way. Figure 5.1 illustrates the desired structure of the data organization.

The address data in the PostgreSQL database that Nominatim sets up is structured. We, thus, can distinguish house numbers from the rest of the address easily. Additionally, the database is organized hierarchically already. Though not always accurately reflected, house numbers are part of street segments that are contained in districts, which are parts of cities, and so forth. Thus, documents for street segments with all the house number addresses on them can be assembled easily.

Besides the mere existence of hierarchy, there are more reasons to organize and index data this way. For once, eventually, we intend to support imprecise, erroneously formulated queries. Specifically, the system will have to be able to handle user queries with typos and abbreviations. While this topic will be treated with great detail in subsequent chapters, we already can establish that such flexibility is counterproductive for house numbers: Queries for addresses with erroneously specified house numbers are most certainly not going to be served correctly. Either the falsely specified house number exists in the world. Though resulting in a formally correct result, such a query does not return to the user what she was querying for. Alternatively, the incorrect house number does not exist in the data. The geocoding system might fall back to an address with an existing house number to return but has little chance in guessing how to undo the specific mistake made. Supporting non-exact house number search will, therefore, not lead to more correct results. Hence, it makes sense to separate
the house numbers from flexibly checked address element names into strictly checked keys.

One other reason to introduce this hierarchy in the indexed documents is normalization of the data. Given that all addresses are part of a hierarchical, tree-like structure, let us consider two leaves on the same branch. Speaking in address terms, we are looking at two house number addresses on the same street segment. These two addresses clearly differ in their identifiers and geographic coordinates, however, only the house number is different in their address text. Thus, by grouping the house numbers on the same street segment into one document, repeating the common address parts for every house number address is no longer necessary. The address text of the leaves can be assembled from the house number and the address text of the street segment. This way, the data indexed is reduced. To achieve that, rules need to be encoded in the documents to instruct the assembly of the full address text. The logic following the rules would have to go into the Elasticsearch wrapper and executed prior to returning a result. Because this approach does not affect the performance of the index in any way, and to retain simplicity, it is not implemented in this thesis.

However, another aspect of the proposed document organization is reducing the size of the index implicitly: Of the tree-like structure of addresses, every path from the root to the leaf need to be indexed as a separate document. In the previous chapter, these paths ended in the house number. Every full address was indexed as a separate document. With the proposed approach, the full addresses are stored inside the street segments. The entire set of leaves of the tree-like structure is cut-off this way. Documents indexed with the proposed approach are street segments containing several full addresses. The number of documents is reduced this way. As a side effect, the documents are also more distinct this way as there are no more documents where only the house number is different. Additionally, besides space consumption, normalization allows for easier editing of documents. In the case of, for example, a street name change fewer documents need to be updated.

Because of the successful utilization of Elasticsearch in the previous chapter, in this chapter, we are only modifying the documents that are indexed. We are adapting the wrapper function accordingly too. House numbers on the same street segment are aggregated into one document. We will refer to the geocoding system that is based on this data format as Elasticsearch with Aggregated House Numbers (EAHN).

<table>
<thead>
<tr>
<th>full address</th>
<th>address with $p = 50%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panoramastraße 1 10178 Berlin</td>
<td>Panoramastraße 10178</td>
</tr>
<tr>
<td>Schönhauser Allee 97 10439 Berlin</td>
<td>Allee 97 Berlin</td>
</tr>
<tr>
<td>Bahnhofstraße 22a 15517 Fürstenwalde Spree</td>
<td>22a Fürstenwalde Spree</td>
</tr>
<tr>
<td>Auf der Ruhr 86 50999 Köln</td>
<td>der Ruhr 50999</td>
</tr>
<tr>
<td>Juri–Gagarin–Straße 8 03046 Cottbus</td>
<td>8 Cottbus</td>
</tr>
</tbody>
</table>
5.1 Hypothesis

In this chapter, the data model is modified. The data indexed remains the same during that process. Even though address documents with no structure are functioning well already, the goal of the change is to get to a more natural representation of addresses. Hence, the expectation is that the results provided by a system set up in this way are on par with results from previous experiments. Besides that, the new organization of the data should result in a smaller space required to persist the data. That is a consequence of less repetition of address parts that are shared across house number addresses. Fewer data will be indexed and hence, the system will require a smaller memory footprint for storage and operative memory. This, potentially, will enable the system to operate quicker for both the indexing and the querying mode.

5.2 Data for Evaluation

To evaluate the geocoding system using this data model evaluation data sets similar to those in the previous chapter are used.

5.2.1 Sampled Addresses from the Index

Test query sets as described in Section 4.2.1 are generated in a similar way. As in the previous chapter, addresses are sampled uniformly from the index. Unlike previously, however, the resolution of the dropped address tokens is revised to be in 10% steps. Instead, additionally to the queries with shuffled tokens queries with tokens that remain in order are generated too. That is done to verify how much missing tokens and token order affect geocoding performance independently.

5.2.2 Scraped Addresses from the Web

Further test query sets as described in Section 4.2.2 are generated too. However, the process of dropping address tokens is enhanced: The intent is to get a more

<table>
<thead>
<tr>
<th>full address</th>
<th>address with p = 50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seefelder Straße 12c 86163 Augsburg Hochzoll</td>
<td>12c 86163 Augsburg</td>
</tr>
<tr>
<td>Attenkoferstraße 51 III Whg. 40 81369 München Obersendling</td>
<td>51 III Whg. München Obersendling</td>
</tr>
<tr>
<td>Rosa–Luxemburg–Straße 60 07973 Greiz (Kreis) Greiz</td>
<td>Rosa–Luxemburg–Straße 60 7973</td>
</tr>
<tr>
<td>Reichenberger Straße 21 45899 Gelsenkirchen Horst</td>
<td>Reichenberger 21 45899</td>
</tr>
<tr>
<td>Kuhgasse 5 06108 Halle (Saale) Altstadt</td>
<td>Kuhgasse 5 Halle</td>
</tr>
</tbody>
</table>
fine-grained understanding of how dropped address tokens of addresses as they are used in the real-world are affecting the quality of the service. Instead of dropping a part of address tokens that has one fixed size, the chance of being dropped is applied to each token individually: Every token in the address has a chance of \( p \) to be dropped from the address, and consequently, a chance of \( 1-p \) to remain in it. As in the test query set described in Section 5.2.1, \( p \) is increased in steps of 10%. Because decisions to retain or drop tokens are independent, an address has a chance to keep all the tokens it had, as well as to lose all of them, with this approach. Both extremes, however, are equally improbable. Hence, the decrease of quality that should be linear to the probability for a token to be dropped on average. As in Section 5.2.1, queries with shuffled address tokens are generated as well as queries with address tokens in their original order. Table 5.1 presents some examples of full addresses as well as their non-shuffled variant with \( p = 50\% \). Note that, as expected, not all those addresses are complete enough to be geocoded completely.

5.2.3 Additional Set of Scraped Addresses from the Web

To further increase the number of addresses used for evaluation, an additional address set is introduced to evaluate the geocoding system in this chapter. Besides German pharmacies, a German website with offerings for apartment rentals\(^2\) has been scraped too. Users creating offerings on that website click on a map to place their offering. The website uses Google’s reverse geocoder to derive an address for the clicked location. However, only the administrative area names of cities, districts, or postal area codes are taken from Google. Street and house number specifically are put in by the user. As in Section 4.2.2, these addresses have no identifiers of addresses in Nominatim attached. Thus, as with pharmacies, distance to the expected result will be taken into account. As in Section 5.2.1 and Section 4.2.2, test query sets have a chance for dropped tokens increasing in steps of 10%. In the same way, query sets with address token shuffled, and address tokens in their original order are generated. Table 5.2 lists some few example addresses from this data set. Compared to manually specified addresses of pharmacies in Table 5.1 these full addresses contain many more address elements. These additional address elements specify administrative areas, as these data are derived from a geocoding system. Human users would


<table>
<thead>
<tr>
<th>id</th>
<th>parent_id</th>
<th>house_number</th>
<th>text</th>
<th>lat,lon</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Deutschland</td>
<td>51.0834, 10.4234</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>-</td>
<td>Berlin</td>
<td>52.5075, 13.4251</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>-</td>
<td>Charlottenburg</td>
<td>52.5182, 13.3173</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>-</td>
<td>Ernst Reuter Platz</td>
<td>52.5127, 13.3218</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>7</td>
<td>-</td>
<td>52.5128, 13.3204</td>
</tr>
<tr>
<td>201</td>
<td>20</td>
<td>9</td>
<td>-</td>
<td>52.5129, 13.3203</td>
</tr>
</tbody>
</table>
not specify so many entities usually. Hence, even dropping address tokens at random produces queries that are closer to what a user might put into a query to a geocoding system.

### 5.3 Implementation

To set up EAHN, first, the data is prepared accordingly. The PostgreSQL database set up by Nominatim has two columns that are used for that: A dedicated *house_number* column enables us to select the addresses with house numbers explicitly by requiring non-null values in them; the column *parent_id* allows for deriving which street segment a house number is part of. Table 5.3 illustrates the database schema with example entries.

Thus, to aggregate house numbers on the same street segment, three steps are executed. First, a query is executed to dump house number addresses along as in the previous chapter. House number addresses are ordered by their *parent_id* column thereby. However, this time, the column specifying the ID of the parent is selected to be part of the projection too. Furthermore, the parent identifier is specified to be the ordering key of the data extracted. Secondly, the street segments are extracted too. Street segments can be recognized as they have identifiers referenced by *id* values of entries that specify a non-null house number. Note that with this approach even address formats that do not use street names are supported. Every address entity that is the parent of a house number is selected this way. Street segments are extracted with the same *id* fields as house number addresses. Their identifier, parent identifier, latitude and longitude coordinates, and address text are fetched. The address text is assembled by the stored procedure that is used by Nominatim to assemble a result address. Street segments, unlike full addresses, are fetched ordered by their own, instead of their parent identifier. As a result, two files with addresses are created. One file contains full addresses, while the other contains addresses of street segments. Addresses in both files are ordered by the identifiers of the
street segments. To aggregate full addresses attached to the same street segment into one document, both files are read in parallel: First, a line from the street segment file is read. Next, all the lines from the house number file are read that specify the parent_id to be that of the street segment. As soon as the next line from the house number file specifies a different parent_id, all the house numbers of the active street segment are in memory. All the read lines with full addresses as well as the street segment address are serialized into a JSON document. The process can continue with reading a line from the street segment file. Figure 5.2 demonstrates this data flow.

An important aspect of EAHN is that house numbers are used as keys in the house number map of each street segment. Therefore, it is important to normalize house numbers in the documents. Multiple ways to specify the same house number can function properly this way. The EAHN in this chapter uses the most simple normalization strategy: House numbers are simply lowercased prior to becoming keys in the document; all non-alphanumeric characters are removed. This way, house numbers with alphanumeric extensions can be queried with both casing variants, even if they contain hyphens or white spaces. For example, queries for house number 5a and 5-A result in the same response, therefore.

Listing 5.1 presents the actual document format used by EAHN. Compared to Listing 4.1, each document in EAHN contains a collection of full addresses in it. The number of documents is vastly reduced this way. For convenience, and because the focus of this thesis is the quality of service as opposed to runtime performance, lots of redundant information has been left in the documents. Specifically, only the display_name and the house_numbers fields of the root document contain data that are used. Also, the display_name fields of the house number documents contain lots of duplicated content.

Another interesting aspect of this new document schema is that some data inconsistencies are implicitly resolved this way. As discussed, OpenStreetMap data is not always fully consistent. Sometimes, multiple entries for the same house number address exist. By aggregating house numbers sharing the same
Chapter 5. Indexing Structured Addresses

Figure 5.3: The query flow of the EAHN wrapper. Each token of the query is assumed to be a house number. A resulting query is stated without each of the assumed house number tokens. Matching street segments are retrieved and checked to contain the assumed house number. If a matching house number is found, it is returned.

With the new data format, the wrapper implemented for the baseline geocoding system is no longer sufficient. A document is a valid response of the geocoding system introduced in Chapter 4. With EAHN, documents are street segments that contain multiple possible results. A different wrapper layer is required to extract the desired house number from the street segment document. That wrapper also has to decide, which token of the query to use as the house number.

The EAHN wrapper logic is visualized in Figure 5.3. First, the query string is tokenized into tokens. The wrapper assumes that house numbers are exactly one token large. Hence, one token is selected to be the house number token. It is normalized in the same way house numbers are normalized during aggregation: The token is lowercased and all non–alphanumeric characters are removed.

Next, the remaining tokens are assembled into a query string that is used to query the Elasticsearch index. As discussed, Elasticsearch returns street segment documents, ordered their BM25F match score. One by one, the wrapper checks for the presence of a matching house number key in those documents. If the normalized value of the assumed house number is also a key in the house number map, the value of that map is returned. Because the fields of the full address are exactly the fields of the documents in the baseline geocoding system, returning a
value of the map is sufficient to implement the same API. The consumer sees no
difference between a result from EAHN and a result from the baseline
gecoding system, therefore. If no document has contained such key, the next token is
assumed to be the house number. Another query is sent against the Elasticsearch
index and the returned documents are checked for a matching house number.
The loop is repeated until every token of the query has been evaluated as house
number. Only if no token happens to be a key in the house number maps of the
documents returned, no result is found.

One obvious question to answer before implementing the wrapper is how
many street segment documents need to be checked to find the one containing
the required house number key? Several iterations revealed, street segments
named by the function Nominatim created oftentimes do not have distinct
names. In other cases, street segments names only differ in postal codes. Postal
codes, however, might not be part of a query at all. At first, it seems to be a
better approach to look at as many documents as possible, therefore. On the
other hand, checking many documents comes at a cost. Clearly, there is the
time needed to parse and traverse every single one of them. That, however, can
be tackled by processing the documents in parallel. Even the multiple queries
can be executed concurrently, though it has not been implemented for the work
presented here. However, with an increased amount of documents checked, the
risk to detect the wrong house number key increases too. While with a low
score, Elasticsearch does present only partially matching documents to a query.
In the case of EAHN, for example, the system would check for house numbers

Listing 5.2: Wrapper function to encapsulate the usage of
Elasticsearch and handle EAHN documents.
on street segments of a differently named street or city. Iteratively repeating measurements with different values in a manual manner, we established a rule of thumb: Checking 200 documents for a matching house number key turned out to produce fine results for the un-cleansed OpenStreetMap data.

Listing 5.2 presents the implementation of the EAHN wrapper used. Compared to the wrapper presented in Listing 4.2, queries are stated in a loop in line 15. The query text is split up into tokens, each of which is picked to be the house number. With the remaining tokens, in line 20, a new query text is assembled. Elasticsearch is queried the same way it is queried in the previous chapter, with the exception in line 26: The query specifies the desired response to contain 200 results. In an inner loop in line 31, each of the results is checked to contain the matching house number. If a match is found, the value of the house number map is simply returned. If no query produced a result that would match the respective house number, no result is returned at all. As discussed, the documents in the house number map have the same format as the documents of the baseline geocoding system. Also, the wrapper returns one single result or no results at all. Because that is the same signature as the signature of the es_wrapper presented in the previous chapter, the test runner from the previous chapter needs no adaptation. Simply adding EAHN to the method list enables us to get statistics for all three geocoding systems in one go.

5.4 Results

Before looking at the quality of the new geocoding service implementation, we will examine the space consumption of the data and the index. Table 5.4 presents these data. Obviously, less space is required to store and index the addresses. The document count is reduced to 56.6%, and the index size to just 14.9% of the original values. Production systems can be optimized in a variety of ways; this exercise has not been done for this thesis. No measurements on runtime performance have been undertaken for the two geocoding systems, therefore. Still, an order of magnitude difference in index sizes indicates that the smaller index is faster to operate than the larger one. The data size of EAHN documents is increased at first. That is not surprising, as EAHN documents contain all the same data that address documents contain, grouped in some extra fields of the respective street segment. The data on disk is simply organized: The file contains one JSON document per line. Interestingly, compressing the files with Lempel-Ziv coding changes the result. The size of the compressed file of EAHN documents is smaller than that of plain address documents. Apparently,

<table>
<thead>
<tr>
<th></th>
<th>document count</th>
<th>size compressed</th>
<th>size on disk</th>
<th>size of index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominatim baseline</td>
<td>50641001</td>
<td>2.9GB</td>
<td>9.5GB</td>
<td>18.1GB</td>
</tr>
<tr>
<td>EAHN</td>
<td>2868115</td>
<td>1.3GB</td>
<td>11GB</td>
<td>2.7GB</td>
</tr>
</tbody>
</table>

Table 5.4: Data and index sizes of the documents of the baseline geocoding system and EAHN
while the organization of data in the EAHN format requires more bytes, storing similar display names next to each other allows for a better compression rate. The data for Nominatim presented in Table 5.4 cannot be compared with the other two geocoding systems. Nominatim uses an entirely different system – PostgreSQL opposed to Elasticsearch – to store and index the data. The database set up by Nominatim contains artifacts produced during indexing. These artifacts are not used for geocoding, but to gain insights in the process of Nominatim and further improve it. Also, there is no data file containing address data that Nominatim uses. The PostgreSQL database Nominatim uses is populated with OpenStreetMap Nodes, Ways and Relations. Hence, there are no documents in Nominatim to count.

Figure 5.4 presents the performance of geocoding addresses sampled from the indexed data. The top chart illustrates the performance on queries with unshuffled tokens, while below that the results of queries with shuffled tokens are shown. For each geocoding system Nominatim, Elasticsearch, and EAHN results of queries with increasingly more dropped tokens are presented. As observed in the previous chapter, Nominatim fails to handle shuffled address tokens.
Chapter 5. Indexing Structured Addresses

Figure 5.5: Ratios of buckets of geocoding responses to requests with web scraped pharmacy addresses and increasingly less address tokens with address tokens in their original order (top) and shuffled (bottom) for the three geocoding systems [3].

tokens. Both Elasticsearch based geocoding systems can cope with that and outperform Nominatim. Also, while the baseline geocoding system performs slightly better, generally, both systems perform similarly well.

In Figure 5.5, the results of geocoding web scraped pharmacy addresses are presented. Unlike in Chapter 4, the new approach to drop address tokens is employed to generate evaluation address sets. Therefore, it is possible to observe the change in the behavior of the systems with gradually increased rates of dropped tokens. Similarly to Figure 5.4, the performance for queries with tokens in order is presented in the top three charts, while the performance of queries with shuffled tokens is presented in the bottom. Because for pharmacy data no boolean decision for a correct or wrong result can be made, results are put into buckets depending on their distance to the expected location. Buckets of $0m - 100m$, $100m - 1000m$, and $1000m - \infty$ are stacked on top of each other and sum up to 100%. Similarly to the evaluation of addresses sampled from the indexed data, Nominatim struggles to handle shuffled address tokens, both Elasticsearch based systems perform better than Nominatim, and the baseline system with plain address documents indexed performs the best. In these charts, however, another interesting insight can be observed: While with more address tokens
dropped and fewer address tokens in the query Elasticsearch based systems yield fewer results overall, the ratio of incorrect results over 1000m away from the expected location yielded by Nominatim increases.

Figure 5.6 very much confirms the findings of the previous observations. Table 5.5 lists the rates of correctly geocoded queries of the three geocoding systems for queries with full addresses and queries with 50% of address tokens dropped. The rates of the best-performing system are presented in bold.

### 5.5 Conclusions

Overall, this experiment confirmed that it is beneficial to organize full addresses into documents containing the house numbers located on the same street segment. Both, the baseline geocoding system that is a plain Elasticsearch with address documents, and EAHN perform similarly well on the selected evaluation data sets. The slight performance hit of EAHN showed to be rooted in data problems. As discussed in the previous chapter, sometimes, addresses are duplicated in OSM data. The document structure of EAHN implicitly drops
duplicated house numbers in a deterministic pseudo-random manner. This way, some of the addresses sampled for evaluation are dropped out of the index. In the baseline geocoding system, the same problem exists, but all full addresses, including the duplicates, are indexed. Duplicates returned correctly are not detected as correct results by the test runner. However, while the baseline geocoding system has the chance to return the correct result, EAHN might not even contain them at all.

Similarly, distinct street segments with equal names are present in OpenStreetMap data. Because of these, the wrapper layer needs to check more results to contain the right house number. This leads to an additional performance decrease. In some cases, too many results were checked and false house number matches were retrieved. In others, too few results were requested so that the street segment with the respective house number was retrieved at all. While both of these symptoms will be tackled in the next chapters, concluding that EAHN is a suitable setup for a geocoding system is fair.

Independent of the two symptoms described, multiple results will have to be checked for a matching house number key in any EAHN setup. The value of 200 turned out to work in these experiments; finer tuning could further improve the performance of the system. Though this will be not done in the scope of this thesis, the number of documents inspected by the EAHN wrapper layer could be adapted dynamically. For example, more documents can be checked, if more documents with similar BM25F scores are returned. Additionally, this threshold could be adapted to the geography of the results.

Another improvement opportunity is the logic for normalizing house numbers. Smarter normalization could increase the number of correct results. House numbers that span multiple tokens could be handled too. In the database set up by Nominatim, house numbers are easily extractable.

Independent of the normalization, an interesting learning is that, apparently, non-house number tokens don’t match house numbers too often. While that allows EAHN to perform better, that might change if addresses from another region are indexed. Lots of streets are numbered in the USA, for example. Hence, EAHN might perform significantly worse there. Careful testing needs to be executed prior to using a system in a real production environment.

Table 5.5: Select rates of successfully served geocoding requests (ordered - shuffled) for the three evaluation data sets

<table>
<thead>
<tr>
<th></th>
<th>indexed</th>
<th>pharmacies</th>
<th>apartments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>full addresses</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nominatim</td>
<td>66.7% - 3.8%</td>
<td>71.6% - 16.2%</td>
<td>35.8% - 6.4%</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>92.3% - 92.1%</td>
<td>75.9% - 75.6%</td>
<td>59.2% - 58.6%</td>
</tr>
<tr>
<td>EAHN</td>
<td><strong>94.5% - 94.5%</strong></td>
<td>70.5% - 70.5%</td>
<td>51.5% - 51.5%</td>
</tr>
<tr>
<td><strong>50% tokens</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nominatim</td>
<td>18.4% - 4.7%</td>
<td>17.2% - 12.3%</td>
<td>17.35% - 8.5%</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td><strong>30.8% - 28.7%</strong></td>
<td><strong>23.7% - 21.9%</strong></td>
<td><strong>22.8% - 22.3%</strong></td>
</tr>
<tr>
<td>EAHN</td>
<td>26.0% - 23.2%</td>
<td>20.3% - 18.9%</td>
<td>20.9% - 19.6%</td>
</tr>
</tbody>
</table>
Creating a User Query Model

In previous chapters, we have established two sensible ways to set up a geocoding system. Both approaches presented are robust against shuffled token order and handle missing address tokens well. Neither of them behaves in an unexpected manner providing different results for seemingly no valid reason. However, the addresses we used to validate this behavior, so far, have only contained correctly spelled tokens. Human users, as we all know, tend to make mistakes. Many do not know how many S and T are in the State of Massachusetts. Even knowingly, some times, user tend to not spell out a name correctly. Similarly, users like to take shortcuts to reduce their effort. Therefore, av and st are often spelled instead of avenue or street.

Such spelling variants of real entity names need to be taken special care of. Our baseline geocoding system, as well as EAHN, can only cope with spelling variants to some extent already. As discussed, addresses are made of address entities that all are part of a hierarchical structure. For computers, however, that structure is only needed to disambiguate entities towards the bottom of the tree. Besides house numbers, street, district, city, region, and even country names are reused across the world. Table 6.1 lists some examples of address elements that are reused across the world. For such cases, the higher-level entity is specifying which of the multiple entities is meant. For other cases, where a lower level entity name is truly unique, the higher-level entities are not actually required. Thus, queries with misspelled or abbreviated city names will still be handled properly by both geocoding systems for addresses with, for example, unique street names. That, of course, only holds if the spelling variant is not adding extra ambiguity by matching to a different, unrelated address term. Table 6.2 presents example queries with and without spelling variants and how the current systems are handling them.

Clearly, the fewer tokens spelled-out correctly are part of a query, the less precise that query becomes. Besides the sheer presence of spelling variants, however, their specific spelling needs to be considered too. Not all spelling variants are equally probable. Findings presented in this chapter have been

<table>
<thead>
<tr>
<th>capital of Germany</th>
<th>Berlin</th>
<th>town in New Jersey</th>
</tr>
</thead>
<tbody>
<tr>
<td>square in Berlin</td>
<td>Ernst-Reuter</td>
<td>street in Munich</td>
</tr>
<tr>
<td>street in Berlin</td>
<td>Linienstraße</td>
<td>street in Dortmund</td>
</tr>
<tr>
<td>country in Asia</td>
<td>Armenia</td>
<td>city in Colombia</td>
</tr>
<tr>
<td>country in Europe</td>
<td>Macedonia</td>
<td>region in Greece</td>
</tr>
</tbody>
</table>
Table 6.2: Example queries with spelling variants in tokens and current baseline system behavior. Spelling variants are highlighted in bold.

<table>
<thead>
<tr>
<th>query</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ernst reuter platz 7</td>
<td>finds Ernst–Reuter–Platz 7 in Berlin, as there are no spelling variants in the query.</td>
</tr>
<tr>
<td>ernst reeter platz 7</td>
<td>finds the same address, because the tokens Ernst and Platz with the house number 7 are unique enough to find the right square.</td>
</tr>
<tr>
<td>linienstraße 27 dortmund</td>
<td>finds Linienstraße 27 in Dortmund, as there are no spelling variants in the query.</td>
</tr>
<tr>
<td>linienstraße 27 dotmund</td>
<td>finds a different Linienstraße 27 in Oberhaching. The misspelled city makes the query ambiguous and leads to the wrong result.</td>
</tr>
<tr>
<td>linienstraße 27 dotmund 44147</td>
<td>finds the correct result with the support of the postal code that disambiguates the query.</td>
</tr>
</tbody>
</table>

published earlier in [5] and [6].

6.1 Common Approaches

Because the problem of misspelled or abbreviated tokens is not new, established tools to mitigate them exist. The two most common ones – normalization [135] and edit distance [136] – are introduced in the next sections.

6.1.1 Normalization

One tool commonly used to tackle abbreviated but correctly spelled query tokens is normalization. The idea is to abbreviate tokens from indexed data and from queries in the same deterministic way. With this approach, the index contains only normalized tokens that are all abbreviated by the same rules. Before a query with non-abbreviated terms is executed, its tokens are normalized too. The same rules are applied to the tokens deterministically resulting in the same tokens that were indexed. This way, abbreviated and non-abbreviated token version match to their indexed counterpart, independent of which version was specified by the user.

The same technique can be used to handle diacritics too. The German umlaut, for example, might be omitted by a user having only an ASCII keyboard at hand. Normalizing the umlaut in the same way at indexing and query time, for example, to the same character without the umlaut, resolves this discrepancy.

Figure 6.1 illustrates the normalization workflow. Normalization rules are applied to data at indexing time. One rule transforms all tokens that are Street to St. Similarly, another rule could transform every ß to ss. At query time, the same
rules are applied to query tokens again. Every unnormalized token of the query gets transformed in the same way the data tokens got transformed at indexing time. Tokens that are normalized already are not modified in any way. This way, the index is only queried with normalized tokens that it contains. Therefore, both, the query with normalized tokens and the query with unnormalized tokens retrieve the matching results.

Note that normalization can be used to mitigate misspellings too. However, the approach is not very flexible due to the way how normalization rules are generated. Because the rules are defined manually and prior to indexing, only typos that are made very often can be handled by normalization. To create a rule, the history of user queries is analyzed. Specifically, common patterns of non-matching tokens of queries and correct results for these queries are examined. Such observations could reveal that, for example, user queries often contain "St" instead of "Street", or "Massachusets" instead of "Massachusetts". From these observations, normalization rules are derived. To get those insights, however, queries need to be linked to correct results first. This is a costly requirement, which oftentimes leads to an inverse approach: A normalization rule is only evaluated in a data-driven manner, but not derived from the data itself. Thus, a normalization rule is implemented and applied to a sampled subset of queries and data. Differences in results before and after application of the normalization rule are checked. If the system yields more correct results after normalization, the normalization rule is accepted and propagated to the production system. Because of the data-driven approach of normalization rules, and because it is such a manual task, only a limited amount of normalization rules are set up usually. Thus, for abbreviations or typos that are not appearing often enough, no normalization rules are derived.
6.1.2 Edit Distance

To tackle randomly misspelled tokens, oftentimes, edit distances or Levenshtein Distances [88] are employed. As discussed, the edit distance between two tokens is defined as the minimal number of edits that applied to one token transforms it into the other. Edits, thereby, are character inserts, character deletes, or replaced characters. In some implementation variants, swapping two consecutive characters with each other is considered as a single edit too. For example, the token `dotmund` is one edit distance away from the token `dortmund`, because one single edit on `dotmund` – inserting an `r` at index 2 – will make it equal to `dortmund`.

To employ this technique, the index needs to be capable of querying terms with an allowed edit distance. This way, it considers the token `dotmund` as a match to the token `dortmund`. The matching tokens are scored as if they have no difference between them, and the correct results are returned.

The downside of this approach is that allowing edit distances drastically increases the ambiguity of tokens. With an edit distance of one, not only `dotmund` match to `dortmund`. Also, `dortfund`, `ortmund`, or `odortmund` are only one edit away and match as if there were no spelling differences. In fact, the number of tokens that are just one edit away from a given token can be calculated based on the following observations: (i) There are \( n - 1 \) pairs of neighboring characters that can be swapped with each other. (ii) Each of the \( n \) characters can be deleted. (iii) Any of the letters of the alphabet can be inserted before each character, as well as after the last one. (iv) Each of the \( n \) characters can be replaced with any other letter of the alphabet. The numbers of the possible edits of each kind need to be summed up to compute the total number of possibilities of one single edit:

\[
\#\text{swaps} = n - 1 \\
\#\text{deletes} = n \\
\#\text{inserts} = (n + 1) \times 26 \\
\#\text{replacements} = n \times 25 \\
\#\text{edits} = \#\text{inserts} + \#\text{replacements} + \#\text{deletes} + \#\text{swaps} \\
= (n + 1) \times 26 + n \times 25 + n + (n - 1) \\
= 53n + 25
\]

That means that for our example token `dortmund`, there are 449 tokens that are just one single edit away. Allowing an edit distance of one when querying the index, makes them all match to the correctly spelled token as if they are spelled correctly too. Obviously, further increasing the number of allowed edit distances, further expands the space of possible matches. For instance, increasing the edit distance to two allows over 200,000 tokens to match to `dortmund`. One can easily imagine how this may lead to different, distinctly named entities to suddenly match to each other.

On the plus side, edit distances help to handle abbreviations too. When the spelled-out tokens are short, the edit distance to the abbreviated token is often small too. This is especially the case when edit distances are combined with normalization. Different abbreviation styles of the same token can match this
way, even though they are not normalized to the same value. Let’s assume, for example, a normalization rule normalizes all tokens *Avenue* to *Ave* at indexing time and allows an edit distance of one for matching the tokens. Abbreviating *Avenue* to *Av* in a query still matches to the normalized token *Ave* in the index, as these two are only one edit distance apart. This way, non-standard abbreviations can be picked up too.

The document search engine Elasticsearch we used as a core component of the geocoding system, supports fuzzy querying for terms with an edit distance of up to two edits. To encounter the problem of too many distinct tokens matching to each other, Elasticsearch reduces the space of matching tokens a bit. Not all discussed edit distances are supported. Swaps, inserts before the first character, as well as replacements of the first character of a token, are not allowed. The rationale for these limitations is that humans usually know the sound of what they are about to spell, such edits only appear in rare cases. Massachusetts, for example, is more likely to be spelled with a missing *t* than with a leading *N*. Similarly, most swaps mess up the term so much that a user would recognize and fix the typo. For the other edits, Elasticsearch allows the user to specify the allowed distance for each token separately. Matching tokens with only one or two edits can be enabled. Also, Elasticsearch supports an automatic fuzziness mode, where the number of allowed edits is derived from the token length. Longer tokens allow two edits, while very short ones are required to be correctly spelled.

In this chapter, we will prepare test data to evaluate how the baseline geocoding system and EAHN are able to handle spelling variants as typos and abbreviations.

### 6.2 Data for Evaluation

One simple approach to get evaluation data is to take a geocoding system exposed to users and collect the query logs. The queries – typed by human users – contain exactly the typos and abbreviations that are made by the users of the system. When sampled correctly, the classes of spelling variants will be distributed exactly as they are distributed naturally. Common spelling variants will appear in the sample more often, while rarely used spelling variants will only appear rarely. The drawback with this approach is that while this is an easy way to collect queries, it does not provide correct responses to those queries too. One would need to manually review the queries and select the correct responses out of the data where available. That is a tedious process that would vastly reduce the amount of available data. While queries can be simply sampled from the log, every correct response needs to be gathered by hand.

An alternative to manually linking queries to correct results is to employ some kind of signal that provides the linkage. If users, for example, are clicking on results they want information about, these clicks can be logged too. This way, besides the query itself, the result that the user clicked on is available in the logs too. Such a signal is never perfect. Users might click on results by accident, or because it is an interesting result that is independent of the query. However, in most of the cases, this signal encodes exactly what we are looking for. While
interacting with a geocoding system, users select the correct result to their query themselves allow implicit linkage of query and result. HERE Technologies\(^1\) is a company offering consumer-facing geocoding services. The company was kind enough to provide a year worth of query log data along with the structured address data that their log contained queries for.

With query and result pairs at hand, we have an evaluation data set ready. The clicked results provided by HERE, however, are not pointing to OpenStreetMap data. Thus, the plan is to generalize from the observation in the HERE logs. Instead of using the data set directly, we want to learn to generate user-like queries for any data set in the index. With a method to do so, we can generate user-like queries from data indexed in Nominatm, the baseline geocoding system, and EAHN. As indexed data comes with identifiers, deciding if a query has been served correctly is an easy task. Also, we can be sure that the reason for a query to not be served successfully was not missing data, but rather the algorithms employed. Thus, in this chapter, we will establish a method to create a user query model from query and result pairs. That model enables us to generate user-like queries from any data.

6.3 Implementation

To create a model from a set of query and result pairs that can generate user-like spelling variants, we need to identify the used spelling variants first. For that, we assign query tokens to result tokens and compute the set of edits between them. In a subsequent step, we group and count edits by observing the context they appeared in. To generate a spelling variant for a token, we verify what contexts it provides and pick an applicable edit.

In a similar way, query formats are derived too. Results that users click on are fully qualified. That means the type of each address element of the result is known. Having query tokens and result tokens paired up, we know the type of each query token from the type of the result token it is paired up with. Query formats too are grouped and counted. Given a fully qualified address, a query in an applicable format can be generated.

6.3.1 Assigning Tokens

Given a query and result pair, it is possible to identify query tokens that are misspelled. Naively removing tokens that are equal to address tokens is not sufficient, unfortunately: As observed in previous chapters, oftentimes fewer terms are used to assemble a query than there are terms that fully specify a complete address. Hence, even if we remove all terms from the query and address that are equal, a misspelled query term is likely to be left with multiple address terms. The fact that more than one query term might be misspelled further increases the problem.

Therefore, a more complex approach is employed generating token pairs of assigned query and result tokens from a query and result pair. First, the

\(^1\)www.here.com (accessed: November 2019)
Levenshtein distance between every query token and every result token is computed. To make sure that the token length is accounted for, and to allow more edits in longer tokens, the distance is normalized by the length of the longer token. Thus, instead of counting distinct edits, a token similarity value is computed and used. The similarity $s_{t_1,t_2}$ between the two tokens $t_1,t_2$ is computed using this formula:

$$s_{t_1,t_2} = 1 - \frac{\text{distance}(t_1,t_2)}{\max(\text{length}(t_1),\text{length}(t_2))}$$

Normalizing with the length of the longer token makes sure that the similarity does not ever result in a value greater than one. Even when computing the distance between two fully different tokens, the minimal number of edits required is exactly the number of characters of the longer token.

That step implicitly detects tokens pairs that are not misspelled and would be easy to assign with the naive approach already. Token pairs with a similarity of one are equal and can be considered as assigned right away.

In the second step, the Hungarian Method [137] is used to assign the non-equal token pairs with a similarity value of less than one. The output of the Hungarian Method is an assignment that results in the lowest cost globally. In our specific use case, the sum of the token similarities of all assigned token pairs is maximized.

Sometimes, when constructing a query, human users specify tokens that are not helpful. For example, "city center" might be added to a query for an address in the center of a city. Such tokens should not be assigned to any address tokens, as they are not part of the address. To avoid assigning them the Hungarian
Method needs to be extended to require a minimum similarity prior to assigning
two tokens to each other. While this way unrelated tokens do not get assigned,
we also prevent abbreviations from being assigned to their spelled-out version
too. That is because abbreviations usually shorten a very long term to a very
short one. The similarity of these very different terms drops below the threshold
and they are not assigned anymore. For example, the token *Street* is four edits
away from its abbreviation *St*. According to the similarity formula, these two
tokens have a similarity value of 0.33. The two fully unrelated tokens *Berlin* and
*Milan* are four edits apart too and have the same similarity value.

To encounter this, the Hungarian Method is implemented to use a modified
Levenshtein algorithm. From the perspective of the Levenshtein algorithm,
abbreviations are runs of multiple character deletes. The implementation is able
to detect consecutive edits of the same type and compute a dynamic price for

```
def match_query_result_tokens(query_tokens, address_token_to_type_dict):
    token_match = TokenMatch(list(address_token_to_type_dict), query_tokens)
    formats = list()
    edits = list()

    for address_token, query_token, levenshtein in token_match:
        if levenshtein:
            token_match = TokenMatch(list(address_token_to_type_dict), query_tokens)
            formats.append(lemenshtein.address_token_to_type_dict[address_token])

            # store edits from levenshtein with context
            for i, step in enumerate(levenshtein.steps):
                if step.a != - step.b:
                    # replace - just store pattern
                    edits.append((step.a, step.b))
                elif step.a:
                    # insert - store within context of prev and next chars
                    # compute context - previous and subsequent char
                    # '!' indicates begin or end of token
                    if i > 0:
                        prev_char = levenshtein.steps[i - 1].a[-1]
                    else:
                        prev_char = ':
                    if i + 1 < len(levenshtein.steps):
                        next_char = levenshtein.steps[i+1].a[0]
                    else:
                        next_char = ':
                    # embed insert in context
                    edited_version = prev_char + step.a + next_char
                    original_version = prev_char + step.b + next_char
                    # store insert
                    edits.append((edited_version, original_version))
                elif step.b:
                    # delete - store within context of prev and next chars
                    # compute context - previous and subsequent char
                    # '!' indicates begin or end of token
                    if i > 0:
                        prev_char = levenshtein.steps[i - 1].b[-1]
                    else:
                        prev_char = ':
                    if i + 1 < len(levenshtein.steps):
                        next_char = levenshtein.steps[i+1].b[0]
                    else:
                        next_char = ':
                    # embed delete in context
                    edited_version = prev_char + step.b + next_char
                    original_version = prev_char + step.b + next_char
                    # store delete
                    edits.append((edited_version, original_version))

    return formats, edits
```

Listing 6.1: Method added to the model allowing it to generate top
\(N\) spelling variants for a given token.
them. Specifically, the implemented version considers only the first insert or delete to count as a full edit. Each subsequent insert or delete counts as a quarter of an edit only. Thus, the cost of an edit that inserts or deletes \( n \) characters is specified to be \( 1 + \frac{n}{4} \). This way, edit distances leave the space of natural numbers and become real. That is not a problem as the similarity score is consumed instead of an edit count. This way, abbreviated terms are still considered similar enough to their spelled-out version and are correctly assigned. While Berlin and Milan keep their low similarity of 0.33, the similarity of Street and St grows with \((1 + \frac{3}{4})/6\) to 0.71.

Figure 6.2 illustrates a possible token assignment result. With this, more sophisticated approach, a set of token pairs is computed for each query and result pair. Refer to Appendix B to see the implementation details of the token assignment algorithm. The Levenshtein computation used by the algorithm is presented in Appendix A. The method utilizing the code in the appendix is presented in Listing 6.1. The token assignments are computed at first in line 2. The instance of TokenMatch behaves like a list of triplets each containing the assigned tokens and their Levenshtein instance. Then, iterating over the assigned token pairs, two lists are populated.

### 6.3.2 Query Formats

One list populated by the `match_query_result_tokens` method contains the query format. Each address token has a defined type, derived from the address entity that token is part of. Query tokens, assigned to address tokens inherit their type. This way, a list of query token types, ordered in the same way the tokens are ordered in the query specifies the query format. It exactly lists the types of the address entities of the query. To handle address entity names that consist of multiple tokens, a query token type is only appended to the list if it differs from the previously stored token type. Note that the method in Listing 6.1 does not handle cases where different address entities share the same token. The token assignment is not sufficient in that case to deterministically derive a query format.
format. The method presented here has been extended to handle repeated tokens in addresses by specifying a preferred order of address elements, therefore.

Figure 6.3 illustrates the derivation of a query format for an example query. By this means, we can derive the address format for every query in the query log.

### 6.3.3 Typos

The second list contains differences extracted from assigned tokens that do not equal exactly. For that, the steps from computed instances of the Levenshtein class are inspected. Internally, the Levenshtein implementation proposed maintains a list of steps that were taken to compute the edit distance. Steps encode edits themselves, as well as matches of tokens that do not require to be edited. Each step contains the specific substrings of both tokens together with the indexes of the substrings in the respective token. If these substrings are equal, no edit is necessary for these regions of the tokens. If these substrings differ, the necessary edit to get from one token to the other is encoded. One substring needs to be replaced by the other substring at the specified index. For more details on the implementation of the Levenshtein class, refer to Appendix A. Steps for equal, swapped, replaced, inserted, or deleted substrings are implemented. The custom implementations allow for multiple consecutive edits to be detected and grouped into a single one.

Before we can consume an edit, we need to differentiate between two general types of edits: Swaps and replacements have substrings for both tokens. They can be consumed in the way they are observed already. For inserts and deletes, only the inserted or deleted string is available in the Levenshtein step. To avoid random insertions of deletes of characters, the context of these edits is stored in addition to the edit. Specifically, besides the string to be inserted or deleted, the preceding and the subsequent characters are stored as part of the edit. The method in Listing 6.1 implements that in lines 20 and 37. The two characters surrounding the insert or the delete are appended to both substrings of a step. In this way, inserts and deletes get substrings of both tokens and can be treated as replacements: Two characters are replaced by a longer string that is surrounded
6.3. Implementation

by the same two characters. In the case of deletes, a longer string is replaced by the first and last character of the same string. Because inserts and deletes are possible at the end and beginning of a token, the end and the begin of a token are marked with the colon character in the \texttt{match_query_result_tokens} method. Figure 6.4 shows an example edit derived together with the context it appeared in.

Note that tokens specified multiple times in one query are all processed in the method in Listing 6.1. In the measurement, the method presented here has been extended to handle repeated tokens in queries for the user model creation as follows: All query tokens are assigned as often as they appear in the query. If multiple equal query tokens have been assigned to multiple result tokens, spelling variants are only derived from the most similar token pair.

6.3.4 Creating a User Query Model

To actually create the user query model, data provided by HERE Technologies\textsuperscript{2} was used. Besides the historic queries themselves, the provided log contained identifiers of addresses that were returned as search results. Additionally, a list of identifiers revealed which search results a user clicked. Users of HERE products click on results to interact with them. They can get a route computed to a result, save the result into their list of favorites, or, if the result is a POI, get additional information about opening hours or telephone numbers. Zero, one or multiple clicks were logged for each query, therefore. The structured address data was made available too, stored in a feature-rich model: Addresses were composed of address elements, most of which were available in multiple languages. Also, alternative names that are not official, but are in use are part of the provided address data.

For building the model, only queries are selected that have one single clicked result. We rely on the click as a signal to indicate the correct result for a user query. With multiple different clicks, it is not obvious which result the user referred to when stating the query. Even with one single click, a user might have looked into the details of a very wrong, and therefore very surprising result.

\textsuperscript{2}\url{www.here.com} (accessed: November 2019)
import random

def generate_format(address_type_to_name_dict, format_and_count):
    # collect formats that can be generated for given address
    # and their total count for random selection
    applicable_count = 0
    applicable_formats_and_counts = list()
    for format, count in format_and_count:
        if not (set(format) - set(address_type_to_name_dict)):
            applicable_formats_and_counts.append((format, count))
            applicable_count += count

    # randomize order and get "index" in list
    index = random.randint(0, applicable_count)
    applicable_formats_and_counts = random.shuffle(applicable_formats_and_counts)

    # drop formats up to "index"
    for format, count in applicable_formats_and_counts:
        index -= count
        if index <= 0:
            return format

Listing 6.2: Method choosing a query format suitable for a given address from observed formats and their counts. For each of the suitable formats the probability to be picked is that of its occurrences by the sum of occurrences of all suitable formats.

While these cases are exceptional for a single click, ignoring cases of multiple clicks helps to reduce the noise in our model. As discussed, edits in their contexts and query formats are extracted from queries and its clicked result by assigning query tokens to tokens of entity names of the query result using the Levenshtein algorithm and the Hungarian Method. From the query token order and the entity type of assigned tokens, a query format is constructed. Edits and edit contexts are extracted from assigned but not equal token pairs.

It is worth noting that no filtering has been done to make the model language-specific. While this is conceptually possible to only execute the above steps for queries and results in a specific language, the model constructed here was a model of all languages and users. The expectation is that queries might come in different languages, but address data is available in different languages too. Thus the query token Rome is assigned to the English city name, while the query token Rom to the German, and Roma to the Italian one. That functionality has been observed working well in the token assignment logic, though, typos made by users in one specific language could get assigned to existing translations in another. This way, some spelling variants made by users get away unobserved.

The model does not differentiate between typos and abbreviations in any way. The occurrence of every edit, within its context, is simply counted. Similarly, the occurrence of every query format is counted as well. Thus, the resulting statistical model simply keeps track of the various observed edits and query formats, as well as how often they do occur in the data.

Figure 6.5 presents how the model groups observations from query tokens and result tokens in their contexts. The same counting happens for query formats. For example, we may observe that the format street – house – city has been used \( n \) times, while the format street – house – city – district has been used \( m \) times.

To reduce the noise in such a model, extremely rare observations are dropped
Listing 6.3: Method choosing a spelling variant given a token and observed edits with their counts.

6.4 Generating User-Like Queries

From the distributions of spelling variants and query formats, queries and spelling variants can be generated easily. Starting with the address formats, let us assume that we have the distribution of the observed formats, as well as a fully qualified address at hand. First, we focus only on counts of those query formats that use address element types that the qualified address actually has

in the last step. Address formats or edits in the long tail of the observations are simply removed from the collected counts. With the counts of occurrences of the various query formats, and edits, we have a statistical model of the distribution of the edits, and query formats that the users have stated.
available. Counts of occurrences of query formats specifying a district are of no use if the qualified address does not have a district available. Out of the remaining formats, each one can be used to construct a query that a user would state for the given qualified address. To select one at random, weighted or stratified sampling [138] is used. With this approach, the probability to chose one of the available query formats is exactly equal to the count of its occurrence by the sum of the counts of the occurrences of all fitting query formats. Thus, if we pick multiple formats for the given address, the distribution of the query formats will be the same as that of the observed queries collected observation on. The model allows generating queries in user-like formats, therefore. Query formats that are used more often in the logs are observed more often too. This, in turn, raises the probability to generate queries in the same format as well. Query formats that only appear rarely, will get generated rarely too. As we are considering queries for full addresses only, query formats that did not specify a house number were filtered from the list. Listing 6.2 presents the method to generate query formats given a qualified address and a list of query formats observed in logs with their counts.

The same approach allows generating spelling variants for a given token too. It is presented in Listing 6.3. First, all available edits fitting that token are collected. Because checking if an edit is applicable takes as much time as generating the variant for it, spelling variants are generated right away. Generated spelling variants are collected together with the count of occurrences of the edit that produced them. One interesting aspect is that sometimes one token provides multiple locations an edit can be applied to. The possibility to produce different variants of the same token given one edit is handled in the code by the two loops in lines 16 and 37. In a second step, similar to address formats, a variant is selected using weighted sampling. Again, the probability of a variant to be selected is the number of times it occurred within the chosen context, divided by the sum of the number of times all edits occurred in the same
Table 6.3: Examples of generated queries.

<table>
<thead>
<tr>
<th>number of spelling variants</th>
<th>generated query</th>
</tr>
</thead>
<tbody>
<tr>
<td>22, Plettenberger Pfad, Tegel, Reinickendorf, Berlin, 13507, Deutschland</td>
<td></td>
</tr>
<tr>
<td>0 plettenberger pfad 22</td>
<td></td>
</tr>
<tr>
<td>1 plettenberger pfad 22 tigel</td>
<td></td>
</tr>
<tr>
<td>2 plettenberger pfad n22 tege</td>
<td></td>
</tr>
<tr>
<td>3 plettenperger pfed n22</td>
<td></td>
</tr>
<tr>
<td>4 tege plettenbetger pfyd n22</td>
<td></td>
</tr>
<tr>
<td>5 plettanberger pfed n22 tenel</td>
<td></td>
</tr>
<tr>
<td>2, Rue Maryse Bastié, Gare, 13e, Paris, Île-de-France, 75013, France</td>
<td></td>
</tr>
<tr>
<td>0 rue maryse bastié 2 paris</td>
<td></td>
</tr>
<tr>
<td>1 rue maryse bastié n2</td>
<td></td>
</tr>
<tr>
<td>2 rue maryve bastié 2 paris france</td>
<td></td>
</tr>
<tr>
<td>3 true moryse bastié 2 75013 paris france</td>
<td></td>
</tr>
<tr>
<td>4 rue14 mailyse baßtié n2</td>
<td></td>
</tr>
<tr>
<td>5 rues muryse bastué n2</td>
<td></td>
</tr>
<tr>
<td>62, Heißgasse, Sauberg, KG Mauer, Liesing, Wien, 1230, Österreich</td>
<td></td>
</tr>
<tr>
<td>0 heißgasse 62 wien</td>
<td></td>
</tr>
<tr>
<td>1 wien heißgasse 62</td>
<td></td>
</tr>
<tr>
<td>2 wien heisgasse n62</td>
<td></td>
</tr>
<tr>
<td>3 heißgisse n62</td>
<td></td>
</tr>
<tr>
<td>4 heeßgasse 62</td>
<td></td>
</tr>
<tr>
<td>5 heisgasse n62 wian östærreich</td>
<td></td>
</tr>
</tbody>
</table>

context. Thus, if executed repeatedly, a variant from a rarely observed edit will rarely be generated by the method. At the same time, spelling variants produced by edits that are observed often will be generated often. Figure 6.6 illustrates the process of generating spelling variants given a token and a model.

With such a statistical model and the proposed methodology, there is a two-step process to generate user-like queries with a controlled number of spelling variants in it. Given a qualified address, first, use the model to generate a query format. Next, assemble the query text from the qualified address in accordance with the specified format. Then, repeating as often as needed, pick one query token at random and get a spelling variant of it generated by the model. Finally, assemble the complete query keeping tokens in their order to evaluate a geocoding system. Generating queries from qualified addresses taken from data indexed into a geocoding system allows preserving identifiers of the qualified addresses attached to the queries. Evaluating if a query is served successfully is a straightforward task, therefore. Thus, using a user query model allows to observe query-result pairs from one data set, abstract user behavior from these observations, and generate evaluation data sets for geocoding services with a different one. In this thesis, we will use statistical models generated on data provided by HERE Technologies to evaluate the performance of geocoding systems indexing OpenStreetMap data.

One thing to keep in mind on that matter is the dependency of the proposed
approach on the query log and the clicks within it. Users are only able to click on results that the original system produced. Hence, if a typing error was big enough to not produce the correct result, there will be no click on the correct result in the log. The model will not be able to observe the context and occurrence of such a typo, therefore. In the model generated from logs provided by HERE Technologies, only user patterns can be observed that are already served by the HERE geocoding system. While the approach allows to generalize from observed queries and create evaluation sets that are applicable to different data, it does not remove the need to properly link user queries and correct results. In the next chapters, we will suggest how to mitigate this problem too.

6.5 Conclusions

Table 6.3 lists example queries generated for three random addresses in Berlin, Paris, and Vienna. For each addresses five queries with zero, one, two, three, four, or six spelling variants are displayed. As discussed, often queries have fewer than the expected amount of spelling variants in them, as for some tokens no spelling variants can be generated at all. Also visible, the most common query format consists of a street and a house number only. That is the most common query format used by the users in the query log. Queries with a different format, including city or postal code are rarer because users prefer to avoid typing things. With regards to the spelling variants, it is visible that the house number token is subject to be replaced with a generated variant too. Apparently, users often prepend numbers with a leading $n$ as many house numbers are modified this way.

Using query logs from real user queries, we created a user query model that can be used to create user-like queries from given data. Query formats, as well as spelling variants, are generated with the distributions that were observed in the real user queries. The user-like queries come with identifiers because they are generated from the same data that is indexed. This makes evaluating a geocoding system simple. Using the model an arbitrary amount of queries can be generated. The number of spelling variants in them can be manually controlled.
To get a general understanding of how the systems introduced in earlier chapters perform on user-like queries, five evaluation sets have been generated with this proposed approach. Each data set contains 50,000 queries and one, two, three, four, or five spelling variants. Figure 6.7 presents the measured performance. As anticipated, both systems handle user-like queries with spelling variants poorly. Interestingly, unlike with the previous data sets, Nominatim performs better than EAHN on every data set. The reason for that is the missing normalization mechanism in EAHN that is present in Nominatim. As discussed, some of the spelling variants observed and, hence, generated are abbreviations. As a mature geocoding system, Nominatim normalizes data indexed and query tokens, just as described in this chapter. Thus, some of the generated queries can be handled by Nominatim therefore. EAHN is not equipped with this functionality and performs worse, therefore. In the next chapter, we will extend the functionality of EAHN to be able to handle user-like queries with spelling variants.

Besides the work in EAHN, the model itself could be further tuned to generate more-accurate user queries. For example, only the context of inserts and deletes has been stored. Also, context has only been stored on edits that are inserts or deletes. The concept of the context can be extended further, to produce more accurate spelling variants. Additionally, two thresholds have been specified arbitrarily in this chapter: The minimum similarity required for two tokens to be assigned can be tuned finer. Also, the cost of subsequent inserts or deletes can be established in a data-driven way.
7 Supporting User Queries Through Edit Distances

In the previous chapter, we have created a very sophisticated evaluation data set mimicking human input. The statistical model allows us to generate queries in a format that a user would state. Also, full control on the number of spelling variants – typos or abbreviations – is available. In this chapter, we want to evaluate how the commonly used approach to tackling spelling variants allows our two systems to perform. The measurements described in this chapter have already been published in [5] and [6]. For that, first, we evaluate how our baseline geocoding system and EAHN perform on queries with spelling variants.

Table 7.1 shows the set up geocoding systems handle queries with spelling variants. Both systems: Elasticsearch equipped with address documents and EAHN with documents of street segments behave exactly the same for the example queries in the table. Interestingly, in some cases, addresses are retrieved correctly despite a spelling variant in the query. That is the case when sufficient correctly spelled address tokens specify the address in question unambiguously. In other cases, when important tokens that distinguish the address queried for from other addresses are spelled differently than indexed, the baseline geocoding system falls back to other addresses that fit the correctly spelled tokens.

Elasticsearch supports fuzzy search out of the box. That means Elasticsearch has the necessary functionality to vaguely match queries to documents as opposed to enforcing exactly the same tokens. There is room for interpretation of what vaguely and fuzzy mean; this is how Elasticsearch implements this feature: A parameter can be specified at query time to enable an edit distance for a specific query token, or for all tokens of the query at once. Edit distances of one or two edits are possible. Additionally, Elasticsearch allows specifying an AUTO flag that selects the allowed edit distance for each query token depending on the token length. In no case does Elasticsearch support an edit distance larger than two. Also, edits on the first character of a token are not allowed to match. With this feature, Elasticsearch enables us to use one of the two mechanisms for handling spelling variants introduced in the previous chapter. This functionality can be exposed through the wrappers around Elasticsearch in the baseline geocoding system and EAHN.

One thought experiment can be made prior to executing any measurement already: While the baseline geocoding system is indexing entire addresses as documents, EAHN documents are street segments. Thus, enabling the fuzziness functionality in Elasticsearch will also enable fuzzy search for house numbers.
Table 7.1: Examples of queries with spelling variants and how they are handled by EAHN and the baseline geocoding system.

<table>
<thead>
<tr>
<th>query</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>linienstraße 13, berlin</td>
<td>retrieve the correct result because the query tokens are all spelled correctly.</td>
</tr>
<tr>
<td>linienstr 13, berlin</td>
<td>retrieve a random street in Berlin with house number 13. The abbreviated street name is not matching the indexed street.</td>
</tr>
<tr>
<td>london, belgrave square 34</td>
<td>retrieve the correct result because the query tokens are all spelled correctly.</td>
</tr>
<tr>
<td>london, bellgrave square 34</td>
<td>retrieve a random square in London with house number 34. The misspelled square name is not matching the indexed, correctly spelled version.</td>
</tr>
<tr>
<td>ernst reuter platz 7 berlin</td>
<td>retrieve the correct result because the query tokens are all spelled correctly.</td>
</tr>
<tr>
<td>ernst reeter platz 7 berlin</td>
<td>retrieve the correct result still. There are sufficient other tokens disambiguating the query.</td>
</tr>
</tbody>
</table>

That might lead to problems where house numbers are similar to the names of other address entities. For example, a five-digit house number might be an edit away from a postal code. Allowing these two to match, therefore, makes the query more ambiguous and might lead to more wrong results than correct ones. EAHN does not put the house numbers into the document text, and hence, is less prone to this problem. House numbers are keys in a map and are treated as codes. No edit distance is applied to them.

Besides extending the functionality of the two geocoding systems, we also are extending the measurement methodology in this chapter. So far, we have only measured the ratio of successful geocoding requests for a geocoding system. Success, so far, is defined as having the correct result in the first place of a response with possibly multiple results. This approach is sufficient so far as only queries of addresses existing and indexed in the geocoding systems are stated. With the new set of user-like queries that is no longer the case: While the starting point for each query still is an address that exists and is indexed, with the statistical model generating query formats and spelling variants, a generated query might become one that looks like a query for an entirely different address, even for a human. Hence, besides measuring the ratio of correct responses, we need to look at the ratio of incorrect responses that the geocoding systems yield.

Commonly, for such cases, the metrics precision and recall are observed. The precision of a system is defined as the ratio of correct results the system yielded in the response list, divided by all the results yielded. Similarly, the recall of a system is the ratio of queries for which the system yielded the correct result in the response list over all the queries. The two formulas for
precision and recall, therefore, look as follows:

\[
\text{precision} = \frac{\# \text{ correct results}}{\# \text{ results}}
\]

\[
\text{recall} = \frac{\# \text{ queries producing correct results}}{\# \text{ queries}}
\]

The minor difference between precision and recall allows to comprehend two things: First, how accurately are the results matched to the query? And second, how often are queries served correctly? To consider the two extremes, a system could, for example, return all the indexed entries to every query. That would drive the recall metric up, as every query would contain the correct result in the response list. On the other hand, however, the system would get a terrible precision value, because the vast majority of the results returned would be wrong. Similarly, a system could only serve one specific query, returning the manually hard-coded, correct result for that. This gives an excellent precision value, though, the recall of such a system would be close to zero: The vast majority of queries would retrieve no result. Thus, the two metrics measure to what extent the system under test is correct, and to what extent it is wrong.

Hence, our geocoding systems need to optimize for both metrics – precision and recall. Also, we are trying to optimize for a non-interactive use-case: The systems should return only the correct result right away, instead of boring the user with result lists to select from. Assuming that there is only one correct result to every query, we will adjust the metrics to only consider the first result of every response list. As a consequence, the number of queries that produce responses with the correct results in it becomes equal to the number of correctly retrieved results. Also, the number of results retrieved overall is equal to the number of queries that produced a non-empty response list. We hence can define the two metrics \(\text{precision@1}\) and \(\text{recall@1}\) as follows: \(\text{precision@1}\) is the fraction of queries that produced a correct result in the first place by the queries that produced a non-empty response list. \(\text{Recal@1}\), similarly, is the fraction of queries that produced a correct result by the overall number of queries. The two formulas for \(\text{precision@1}\) and \(\text{recall@1}\), therefore, look as follows:

\[
\text{precision@1} = \frac{\# \text{ queries producing correct result}}{\# \text{ queries producing any result}}
\]

\[
\text{recall@1} = \frac{\# \text{ queries producing correct result}}{\# \text{ queries}}
\]

One of the two metrics, actually, is not new. \(\text{recall@1}\) is exactly the metric we have used so far to evaluate the geocoding systems. As discussed, so far, this was the only metric of interest: Because every query stated was for an address that existed in the index, it did not matter whether the result was wrong or empty. Now, by tracking \(\text{precision@1}\), we can observe if one system eagerly returns wrong results instead of returning no results at all. Therefore, the two metrics will be measured in all following chapters from now on. For the sake of simplicity they will be referred to as precision and recall, even though, continuously, \(\text{precision@1}\) and \(\text{recall@1}\) will be measured. Figure 7.1 presents the performance of the Nominatim geocoder with regards to the new metrics.
Unsurprisingly, both metrics drop further, with more spelling variants present in a query set.

### 7.1 Hypothesis

Considering that Nominatim uses SQL queries, and that fuzzy text search in SQL requires a full table scan, Nominatim is not likely to be able to handle queries with spelling variants. On the plus side, Nominatim contains normalization rules that will pick up some of the generated spelling variants.

The baseline geocoding system with complete addresses indexed as documents in Elasticsearch implements a fuzzy text search out of the box. Extending this functionality is likely to compensate for many spelling variants in the generated queries.

EAHN, finally, is likely to handle spelling variants in a way similar to the baseline geocoding system. However, as EAHN is not indexing house numbers, and hence, not allowing them to contain edit distances, it is likely to yield the best results. The expectation is that EAHN gets a greater precision@1 value while remaining on par with Elasticsearch with regards to recall@1.

### 7.2 Data for Evaluation

To evaluate how the geocoding systems perform with user-like queries that contain spelling variants, the query sets generated in the previous chapters are used. Six sets with 50,000 queries each were created. Each set contained zero, one, two, three, four, or five spelling variants in every query. It is worth pointing out that the number of spelling variants in the query sets is just an upper bound: It is the number of attempts that were made to replace a token with its spelling variant. As discussed, to generate a spelling variant, first, the set of contexts available in a given token is selected. For very short tokens, however, only a few contexts can be applied. In extreme cases, no context with observed edits was present in a token. That could happen because either no edits on contexts...
Table 7.2: Examples of queries generated with various amounts of spelling variants from the same addresses.

<table>
<thead>
<tr>
<th>Spelling Variants in Query</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Berliner Straße 98, 04910 Elsterwerda, Deutschland</strong></td>
<td>0 berliner straße 98</td>
</tr>
<tr>
<td>1 berliner strasse 98 04910 elsterwerda deutschland</td>
<td></td>
</tr>
<tr>
<td>2 berliner str 98 04930 elsterwerda deutschland</td>
<td></td>
</tr>
<tr>
<td>3 berlener straße 98</td>
<td></td>
</tr>
<tr>
<td>4 berlinerstrasse strase n98</td>
<td></td>
</tr>
<tr>
<td>5 elsterwarda berliemr straße 98</td>
<td></td>
</tr>
<tr>
<td><strong>25 Giesbach Rd, London, N19 3DA, UK</strong></td>
<td>0 giesbach road 25 london</td>
</tr>
<tr>
<td>1 giesbach road n25</td>
<td></td>
</tr>
<tr>
<td>2 giosbach road n25 london</td>
<td></td>
</tr>
<tr>
<td>3 giesbach read 25 londun uks</td>
<td></td>
</tr>
<tr>
<td>4 glegbach roed n25</td>
<td></td>
</tr>
<tr>
<td>5 gcesbach roud n25 losdon</td>
<td></td>
</tr>
<tr>
<td><strong>15 Rue de Dunkerque, 31200 Toulouse, France</strong></td>
<td>0 rue de dunkerque 15</td>
</tr>
<tr>
<td>1 rue de dunkerque n15 toulouse france</td>
<td></td>
</tr>
<tr>
<td>2 ruel de dunkerque n15 31200 toulouse france</td>
<td></td>
</tr>
<tr>
<td>3 rue des dunkerque 15 31000 toulouse france</td>
<td></td>
</tr>
<tr>
<td>4 2rue des donkerque n15</td>
<td></td>
</tr>
<tr>
<td>5 toulouses rues de4 dunkelque n15</td>
<td></td>
</tr>
</tbody>
</table>

of this token have been observed at all. Also, if edits were observed very rarely, the cleansing mechanism might have considered the rare observations as noise and dropped them from the final model hence. The token to be replaced by a spelling variant is selected at random, and independent of the edit contexts it provides. In some cases, therefore, the selected token is left unchanged and put back into the query. Thus, for some queries, the number of actual spelling variants is lower than expected based on the query set they are in. Table 7.2 lists some of the generated queries, along with the full address the queries have been generated from. Looking at the address formats first, the generated queries seem fair: All queries specify street and house number, sometimes augmented with city, country, and postal code. Also, the generated spelling variants look all right: We see dropped diacritics, abbreviations, typing mistakes. Sometimes digits of postal code numbers are replaced. Apparently, our model learned that numbers are prefixed with the character n often as visible on the house numbers of the examples. That is because users often prefix a house number with the abbreviation of the word number.
7.3 Implementation

To expose the fuzzy text search functionality of Elasticsearch to the geocoding systems, the Elasticsearch wrappers needed to be extended. Listing 7.1 shows the extended code wrapping the Elasticsearch index of complete address documents. The fuzziness parameter can now optionally be set when calling the search method. The parameter is propagated to Elasticsearch with the request body in Line 12. Similarly, in Listing 7.2, the code of the EAHN wrapper is extended in Line 25. The remaining code is unchanged from the wrappers introduced in Chapter 4.

Besides the changes to the wrappers, data in both geocoding systems was re-indexed, after a cleansing step has been executed. As discussed in earlier chapters, OSM data is inconsistent sometimes. Specifically, as discussed, oftentimes streets are broken into segments that have equal names. Such street address documents with equal names are not exact duplicates, because they contain different house numbers in their map. However, these additional documents are a problem for EAHN: They require EAHN to check a large number of results for a house number match. That number might be too little if there are many tiny street segments: A matching house number might not be discovered this way. That number might be too high if there are only a few segments: A house number matching from a wrong segment might be discovered this way. Therefore, prior to indexing EAHN data, street segments with equal names have been merged. Their house number maps have been joined together into one larger map. Also, their identifiers have been accumulated into a list for easier debugging.

Also, there are cases in OSM data where multiple entities with the same address are present in the data. When evaluating the performance of a geocoder, these cases can break the measurements. In the data sets used, each query comes with an identifier of the address it has been generated from. If the address a query originates from is has a duplicate, each of the duplicates is a correct result for that query. Our systems, however, serving the non-interactive use case are returning one result at most. Even if a result happens to be of the duplicate
7.4. Results

Listing 7.2: Extended wrapper of EAHN allowing to specify a fuzziness value.

address, and hence, correct, it will have a different identifier. The test-runner assumes such a result as wrong. To mitigate that, identifiers of house number addresses are accumulated. The place_id fields of the documents are transformed to be lists of identifiers of duplicates, therefore. If a document has no duplicate, its list of identifiers contains only one entry. Thus, if multiple entities for the same address are present in the data, only one document is generated from them. All the identifiers of all the entities are present in that document, however. Because of that, the test runner needs to be adapted too. It needs to check if the expected identifier is in the set of identifiers of the returned document to decide if the result is correct or not. To maintain a common interface, the wrapper of Nominatim needs to be adapted to wrap the value of the place_id field into a list too. Table 7.3 lists the counts of entities cleaned up this way.

7.4 Results

Figure 7.2 illustrates the performance of the two geocoding systems on the generated data sets with regards to the newly defined metrics. The top chart

<table>
<thead>
<tr>
<th>entity</th>
<th>before cleansing</th>
<th>after cleansing</th>
<th>reduced to</th>
</tr>
</thead>
<tbody>
<tr>
<td>house number address</td>
<td>52592499</td>
<td>50641001</td>
<td>96.3%</td>
</tr>
<tr>
<td>street segment</td>
<td>4388712</td>
<td>2868115</td>
<td>65.3%</td>
</tr>
</tbody>
</table>

Table 7.3: Numbers of duplicates reduced by the cleansing step.
Figure 7.2: Precision and recall ratios for the two geocoding systems and their various options for allowing edit distances, on the six generated, user-like query sets with various numbers of spelling variants each.

plots the precision and recall metrics of the baseline geocoding system, while the same metrics of EAHN are flipped on the bottom side. Six blocks of bars are grouped representing the performance of the systems on each of the six data generated data sets of user-like queries. In each block, precision and recall bars are plotted next to each other. Four pairs of such bars are representing the four possible setups of the geocoding systems: Allowed edit distances of zero, one, and two, as well as the automated selection of the allowed edit distance, are represented with their own precision and recall results.

Unsurprisingly, both systems perform well on the query set with no generated spelling variants. More than that, with the new deduplicated dataset, and the proper handling of identifiers of duplicates, both systems reach about 80% precision and recall with their basic set up. Also unsurprisingly, both the systems show a greater decrease in precision and recall, the more spelling variants are part of a query. That is especially the case for no edit distances allowed. While enabling edit distances mitigates the impact, naturally, the more spelling variants there are in a query set, the lower precision and recall can be achieved. Two interesting observations can be made here: For once, the specific configuration of edit distances per token is not influencing the result a lot. That is the case because while an extra allowed edit distance enables more
### Table 7.4: Examples of queries with spelling variants and how they are handled by EAHN and the baseline geocoding system.

<table>
<thead>
<tr>
<th>Query</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>linienstraße 13, berlin</td>
<td>retrieve the correct result because the query tokens are all spelled correctly.</td>
</tr>
<tr>
<td>linienstr 13, berlin</td>
<td>both systems still retrieve a wrong result. The abbreviated street name is more than one edit away from the indexed one.</td>
</tr>
<tr>
<td>London, Belgrave Square 34</td>
<td>retrieve the correct result because the query tokens are all spelled correctly.</td>
</tr>
<tr>
<td>London, Bellgrave Square 34</td>
<td>both systems discover the correct result now because the variant is just one single edit away.</td>
</tr>
<tr>
<td>Ernst Reuter Platz 7 Berlin</td>
<td>retrieve the correct result because the query tokens are all spelled correctly.</td>
</tr>
<tr>
<td>Ernst Reeter Platz 7 Berlin</td>
<td>both systems still return the correct result.</td>
</tr>
</tbody>
</table>

Correct matches, it also adds to the ambiguity and allows more false matches as well. That is also why, consistently, the results of the systems allowing an edit distance of two are worse than the results of the systems allowing an edit distance of only one. Similarly, selecting the allowed edit distance based on the token length performs better too. The other interesting takeaway here is that as soon as at least one spelling variant is present, EAHN yields a much higher precision compared to the baseline geocoding system. The reason for that has been discussed in Section 7.1 already: Not indexing the house number disallows edit distances to be applied to that. That, in turn, leads to less ambiguous queries. Together with the filtering on a house number match of the street segments these properties result in a significantly larger precision. The price for this high precision is a slight, but measurable, reduction in the recall of EAHN.

Table 7.4 presents the same searches executed at the beginning of this chapter once again, this time with allowing an edit distance of one. As expected, the addresses with spelling variants that are just one edit away are picked up fine now. Putting more spelling variants in a distinguishing token prevents it from matching. Wrong results are retrieved then.

### 7.5 Conclusions

Allowing edit distances for token matches is a meaningful way to handle user-made spelling variants. This approach does not work for tokens with multiple typing errors or abbreviations. EAHN reaches higher precision at the cost of recall: It is more selective before returning a result, compared to the current baseline geocoding system. Because of the much smaller index size, and because cleaning OSM data requires to assemble the EAHN data format even when
plain address documents are used, EAHN will be considered as the new baseline geocoding system from now on.

In line with the previous measurements, Nominatim does not handle spelling variants well. It is outperformed by the new baseline geocoding system with any number of spelling variants in the query.

In the next chapter, we will evaluate a novel, alternative approach for handling user-made spelling variants. The idea is to reduce the ambiguity of queries by only allowing specific but not all edits to tokens.
8 Supporting User Queries Through Enhanced Index

In the previous chapter, we have evaluated how enabling fuzzy token matches can handle erroneous user queries. Typos in tokens – if not too many edit distances apart – can be still recognized as matches and correct results can be returned. Also, we have learned how abbreviations still require normalization to be handled properly with this approach. Additionally, we have observed how allowing larger edit distances makes the token matching too fuzzy. More similar but incorrect tokens match so that more incorrect results are returned. In this chapter, we will evaluate an alternative idea. Based on the user query model from Chapter 6, the plan is to generate specific spelling variants of each token. These experiments have been published earlier in [5] and [6].

To recap: First, user queries and user clicks are extracted from the logs. We assume that a click indicates which correct result a user was querying for. Thus, for each query, we have a result with the fully qualified address and address element translations. Using the Levenshtein Algorithm and the Hungarian Method, both slightly modified to our needs, we are able to extract token pairs. Each pair contains a properly spelled token as well as the spelling variant of the token made by a human user. From these, we derive the specific edits that modify the original token into the spelling variant. The context of each edit is derived too and edits are counted within their contexts. This model can generate user-like spelling variants. Given a token, it can observe which contexts for edits are available in it, and thus, which edits have been observed to be used by users. Edits can now be selected randomly, weighted with the counts of their occurrences. This way, spelling variants that are made often by users will be generated most often, while rarely made spelling variants will only be generated rarely. In the same way, the $N$ most common spelling variants of a token can be computed.

Similar to allowed edit distances, such a model generates variants of tokens that are allowed to match with each other. Unlike edit distances, the model does not generate arbitrary variants with equal probability. Instead, it lists specific modifications that are allowed, based on spelling variants that were used by humans. The probability each allowed spelling variant is proportional to the number of observed occurrences of that variant.

The experiment undertaken in this chapter will, therefore, evaluate, how these two approaches to fuzziness perform compared to each other. For that, our baseline geocoding system EAHN will be enhanced with spelling variants generated by a user query model. An important aspect, thereby, is the question of
how many generated spelling variants to accept. Is there a minimum threshold, such that allowing variants with a probability below that no longer improves the geocoding service? Does, as with edit distances, the ambiguity of queries grow further with spelling variants at some point?

In the experiment spelling variants generated by the model will be indexed in the documents along with the correctly spelled tokens. To answer the two questions above, various amounts of most common spelling variants per token will be indexed. This will give an indication on whether there is a threshold for the number of indexed spelling variants that leads to peak performance. We will refer to the EAHN geocoding system that is enhanced with generated spelling variants as Enhanced EAHN (EEAHN). Both systems are configurable now. EAHN can be configured to propagate edit distances of one or two to Elasticsearch. The Elasticsearch index of EEAHN can be enhanced with a variable amount of spelling variants per token. These configuration parameters will be specified after a dash for the respective geocoding systems. This way, EAHN–2 is an EAHN geocoding system that queries the Elasticsearch index allowing an edit distance of two, and EEAHN–40 is an EEAHN geocoding system with an Elasticsearch index containing up to 40 spelling variants for each correctly spelled token in the street segment documents. Note that, as EAHN and EEAHN only differ in the contents of their index, EAHN–0 and EEAHN–0 are one and the same geocoding system.

8.1 Hypothesis

The experiment will compare the performance of a series of EEAHN geocoding systems, with an increasing amount of indexed spelling variants. Similar to increasing the allowed edit distance of matching tokens, there might be a set up after which further increase of the number of indexed spelling variants decreases the performance. Given, however, that the spelling variants are likely typing mistakes or abbreviations made by the user, one possible outcome could be that more is always better. We would observe that more indexed spelling variants always further increase the precision and recall metrics.

In any case, the expectation is that allowing specific, generated spelling variants to match will reduce the ambiguity of queries, compared to allowing edit distances. Even if there is a number of spelling variants after which the ambiguity starts to increase, that point is controlled by the number of indexed spelling variants. Hence, there must be a set up of EEAHN that yields better results than EAHN with an allowed edit distance of one.

8.2 Data for Evaluation

For the evaluation of the approach, multiple datasets were generated as described in the previous chapter. A model is used to generate user queries with spelling variants. Because in this experiment, a similar model is used to generate spelling variants to be indexed, two independent models are used. The dataset provided by HERE Technologies was split into two non-overlapping sets. Two
8.3 Implementation

As discussed, the number of variants the model can generate for each token is limited. Figure 8.2 visualizes this. It presents the average number of tokens in the indexed documents on the vertical axis. The horizontal axis denotes the number of spelling variants per token. As one can see, at first, the number of generated tokens is a constant factor to the document size. Documents with circa ten tokens and no spelling variant grow linearly to 200 tokens with 20 variants per token. The growth starts to flatten, however. With 80 variants per token, the average document size is below 700 tokens. The effect is bringing the growth of document sizes to a quick stop. Only a few additional spelling variants can be

Figure 8.2: Average number of tokens in documents with various amounts of spelling variant per token generated [5].
def generate_top_variants(token, edits_and_counts, typos_count):
    # result list
    variants = list()
    # surround token with begin and end markers
    token = ' : ' + token + ' : '
    # order observed edits by their counts from high to low
    edits_and_counts = sorted(edits_and_counts, key=lambda i: -i[1])
    # try to apply every variant in the order of its likelyhood
    for (token_part, variant_part), count in edits_and_counts:
        if len(token_part) + len(variant_part) == 2:
            # single char change
            index = token.find(token_part, 2)
            while 0 < index < len(token) - 2 and len(variants) < typos_count:
                variant = token[:index] + variant_part + token[index + 1:-1]
                variants.append((variant, count))
                index = token.find(token_part, index + 1)
        elif token_part == token[:len(token_part)]:
            # insert or delete at begin of token
            variant = variant_part[1:] + token[len(token_part):-1]
            variants.append(variant)
        elif token_part == token[-len(token_part):]
            # insert or delete at end of token
            variant = token[1:-len(token_part)] + variant_part[:-1]
            variants.append(variant)
        else:
            # multi char insert or delete occur in whole token
            index = token.find(token_part, 2)
            while 0 < index < len(token) - len(token_part) - 1 and len(variants) < typos_count:
                variant = token[:index] + \ 
                variant_part + \ 
                token[index + len(token_part):-1]
                variants.append(variant)
                index = token.find(token_part, index + 1)
        # stop when sufficient variants found
        if len(variants) == typos_count:
            break
    return variants

Listing 8.1: Method added to the model allowing it to generate top \( N \) spelling variants for a given token.

generated between 320 and 640 variants per token. The average document size does not seem to change at all anymore.

Listing 8.1 shows the added code allowing the model to generate the top \( N \) spelling variants. To recap, the model is a list of tuples containing edits, along with their occurrence counts. The edits include the original and the modified token parts that were observed to have changed, as well as context the change was observed in. The occurrence counts, thus, are the likelihood of an edit to occur. Hence, iterating over the edits in the order of their occurrence counts ensures that most likely edits are applied first. The method in Listing 8.1 does exactly that. For each edit, it checks if it can be applied. If so, a spelling variant is generated and appended to the result list. As soon as the result list contains the requested amount of spelling variants, the list is returned. Note how edits that may be applied in multiple parts of a term are handled. After each computed spelling variant the code checks if sufficient variants have been computed yet in lines 16 and 34. Alternatively, the method might run out of tuples with edits to apply. In that case, the list of variants computed so far is returned, even though it contains fewer spelling variants than requested.

The method has been used to generate documents with spelling variants for
8.3. Implementation

Listing 8.2: Example EEAHN–5 document in actual format as indexed. The display_name field in the root document contains all the correct address tokens, each followed by five spelling variants.

```json
{
  "place_id": ["41816262"],
  "lat": "52.5127",
  "lon": "13.3218",
  "display_name": "linienstrasse linienstr linienstrasse linienstrasse lysienstrasse scheunenviertel scheunenviertel scheunenviertel scheunenviertel scheunenviertel scheunenviertel linie nstrasse sc
  eueunenenvierte nite nitte nite nitte nitte nitte alt plazzeit salt.at art mailzeitalt ber
  lin berlyns betlin berlyn berlin berlyns betlin berlyns berlyn 10178 1
  0278 10078 11178 10168 12178 deutschland deutschland deutschland deutschland deutschland deuts
  chland",
  "house_numbers": {
    "13": {
      "place_id": ["9263130"],
      "lat": "52.5267883",
      "lon": "13.4148849",
      "display_name": "13, Linienstrasse, Schuennenviertel, Mitte, Alt-Berlin, Berlin
      10178, Deutschland",
      "14": {
        "place_id": ["9263134"],
        "lat": "52.5268646",
        "lon": "13.4146334",
        "display_name": "14, Linienstrasse, Schuennenviertel, Mitte, Alt-Berlin, Berlin
        10178, Deutschland",
      }
    }
  }
}
```

indexing as follows: First, (i) the address of an EAHN document is split into tokens. Note that EAHN documents do not contain house numbers as part of an address. Only street names address elements of higher layers are indexed in EAHN. Next, (ii) spelling variants of each token are computed. To identify whether a number of generated spelling variants exist, multiple indices with different amounts of indexed spelling variants have been set up. Starting with 10, the number of spelling variants generated was doubled continuously over 20, 40, 80, 160, and 320, up until 640 spelling variants generated for every token. Finally, (iii) the original, correctly spelled address is indexed together with the generated spelling variants into EEAHN. Listing 8.2 presents an EEAHN–5 document. Compared with Listing 5.1, the document fields are equal. The only difference between the documents of EAHN and EEAHN is in the contents of the root display_name field. The EEAHN–5 document contains the same address tokens that EAHN would have. However, for EEAHN–5, each correctly spelled address token is followed by five most common spelling variants generated for it. As discussed, that grows the document significantly only at first. While documents of EEAHN–5 are five times the size of documents of EAHN–0, there barely is any difference in document sizes of EEAHN–320 and EEAHN–640.

The redundant format comes in handy now: Each value of the house number map is a still unaffected document with a beautiful display_name field and can still be returned as a result. Note that as a consequence of the deduplication effort from the previous chapter, identifiers are lists now. Interestingly, this example also demonstrates how diacritics and abbreviations are detected as common spelling variants of the street name. Besides the contents of the display_text field, EAHN and EEAHN do not differ from each other. The document schemas are equal, and the same wrapper is used to query the documents in exactly the same way. Thus, EEAHN inherits the functionality to enable an edit distance
Figure 8.3: Detailed overview of precision (top) and recall (bottom) of EEAHN (left) and EAHN (right) geocoding systems.

at query time from EAHN. The edit distance, however, will be applied to the spelling variants too. This allows us a unique opportunity to observe if the two approaches can be utilized together.

8.4 Results

Figure 8.3 presents the detailed precision and recall metrics of EAHN and EEAHN. The evaluation sets with the various number of spelling variants in the queries are plotted on the horizontal axis, while the precision and recall metrics are presented on the vertical axis. Performance metrics of EEAHN with 0, 10, 20, 40, 80, 160, 320, and 640 spelling variants are in the two charts on the left-hand side. Right next to them, for simple comparison, the performance metrics of EAHN with allowed edit distances of 0, 1, and 2. As expected, a greater number of spelling variants in the query reduces both metrics for EEAHN and EAHN, independent of the number of spelling variants enhancing the index, or the allowed edit distance. Also, as already discovered, increasing the maximum edit distance from one to two does not measurably affect precision or recall of EAHN. The charts for EAHN allowing an edit distance of 1 and EAHN allowing an edit distance of 2 are almost identical. Another noteworthy thing is how EEAHN with no spelling variants indexed behaves exactly as EAHN with no allowed edit distances. These two systems are absolutely identical to each other. Similarly, as anticipated from the average token counts of documents in EEAHN, doubling the number of indexed spelling variants is only showing a minor effect at first. There is a small but visible increase in the metrics between 10, 20, or 40 spelling variants. The charts for precision and recall of EEAHN with 160, 320, or 620 indexed spelling variants seem to be exactly on top of each other.

Generally visible in the detailed charts already: The precision and recall of EEAHN are greater than the same metrics of EAHN. Figure 8.4 gives a better overview of the precision and recall of EAHN allowing an edit distance of 1,
and EEAHN with 320 spelling variants per token enhancing the index. For comparison, Nominatim and EAHN with no edit distances are plotted as well. As discussed, EAHN with no allowed edit distances fully equals to EEAHN with no spelling variants indexed. As already seen, EAHN with no edit distance allowed is slightly outperformed by Nominatim. Both systems handle spelling variants poorly. Nominatim, however, implements normalization techniques that EAHN does not have. The more interesting charts, however, are in EEAHN with 320 spelling variants indexed in yellow and EAHN allowing an edit distance of one in green. For both metrics precision and recall, the new approach with precomputed and indexed spelling variants outperforms allowing edit distances continuously. Looking closer at Figure 8.3, one can conclude that already starting with 40 to 80 indexed spelling variants, EEAHN is on par with EAHN. Additionally, indexed spelling variants continuously add but do not degrade the performance metrics.

Another interesting chart to look at is Figure 8.5. It visualizes for various
Chapter 8. Supporting User Queries Through Enhanced Index

Table 8.1: Configurations yielding best precision.

<table>
<thead>
<tr>
<th>variants in query</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>variants indexed</td>
<td>0</td>
<td>160</td>
<td>640</td>
<td>160</td>
<td>640</td>
<td>160</td>
</tr>
<tr>
<td>edit distance</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>precision</td>
<td>87.88%</td>
<td>86.06%</td>
<td>83.41%</td>
<td>73.63%</td>
<td>56.62%</td>
<td>36.34%</td>
</tr>
<tr>
<td>recall</td>
<td>72.90%</td>
<td>49.37%</td>
<td>28.36%</td>
<td>14.37%</td>
<td>7.45%</td>
<td>3.33%</td>
</tr>
</tbody>
</table>

EEAHN setups the ratio of responses that had any result – correct or wrong. The Blue chart for EEAHN with no spelling variants indexed expectedly produces little results for queries with spelling variants. The erroneous tokens queried for are not indexed. Thus, the system cannot retrieve matching results. The ratio of responses with results grows when spelling variants are indexed. Also, the more spelling variants are indexed in EEAHN, the higher the rate of responses with results. The most interesting observation, however, is that the growth of the rate of responses with results is not linearly proportional to the number of indexed spelling variants. Looking at the query set with one single spelling variant, the rate of responses with results is at about 2% with no spelling variants indexed, and at over 40% when EEAHN contains 20 spelling variants. Quadrupling the number of spelling variants in EEAHN to 80 only increases the rate of responses with results to about 56%. Quadrupling the number of spelling variants once again still leaves the rate of responses containing results at below 60%. This development is in line with the observations made in Figure 8.2.

Besides providing insights on the performance of the two approaches – enhancing the index with spelling variants and allowing edit distances – the experiment set up also provided the possibility to evaluate the combination of the two. Interestingly, oftentimes the combination of the two approaches resulted in better precision or recall than relying on one approach alone. Tables 8.1 and 8.2 present the configurations leading to the best precision and recall metrics observed. Clearly, increasing the edit distances, and enhancing the index with more spelling variants lead to better recall. Another interesting observation is that the differences in the metrics are not big. Thus, optimizing the geocoding system for precision implicitly ensures a good recall and vice versa.

8.5 Conclusions

One clear conclusion to draw from this experiment is that enhancing the index with precomputed spelling variants, as assumed, performs better compared to
Table 8.3: Examples of queries with spelling variants and how they are handled by EEAHN

<table>
<thead>
<tr>
<th>query</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>linienstraße 13, berlin</td>
<td>retrieves the correct result, because the query tokens are all spelled correctly.</td>
</tr>
<tr>
<td>linienstr 13, berlin</td>
<td>retrieves the correct result, because the abbreviated street name &quot;linienstr&quot; is indexed as a spelling variant.</td>
</tr>
<tr>
<td>london, belgrave square 34</td>
<td>retrieves the correct result, because the query tokens are all spelled correctly.</td>
</tr>
<tr>
<td>london, bellgrave square 34</td>
<td>retrieves the correct result, because the index is enhanced with the variant &quot;bellgrave&quot;.</td>
</tr>
<tr>
<td>ernst reuter platz 7 berlin</td>
<td>retrieves the correct result, because the query tokens are all spelled correctly.</td>
</tr>
<tr>
<td>ernst reeter platz 7 berlin</td>
<td>retrieves the correct result as with the address in London, because the index is enhanced with the variant &quot;reeter&quot;</td>
</tr>
</tbody>
</table>

allowing edit distances to match spelling variants in user queries. Another interesting fact is that enhancing the index with more spelling variants is not degrading the system. That is contrary to EAHN with allowed edit distances, where allowing edit distances of 2 led to slight regressions. Also that the combination of the two approaches often yields even better results is an interesting fact that we will come back to in Chapter 10.

There exist opportunities to further improve the approach. For instance, we have observed that the index should be enhanced with as many spelling variants as possible. However, that might not always be necessary. Thus, the number of indexed spelling variants could vary regionally to result in a well-enhanced index that is also concise for best runtime performance. Further, in the implementation used, correctly-spelled tokens are as important as their spelling variants. That is not necessarily the best approach for scoring documents with spelling variants. Especially where spelling variants of one address overlap with different, correctly spelled addresses ambiguity could be reduced by counting matches differently. For example, correctly spelled tokens could weigh twice as much as spelling variants when computing the BM25F score of a query. Additionally, the likelihood of each spelling variant is known when the spelling variant is generated. This could be incorporated in the score too, reducing the importance of rare spelling variants, compared to the often used ones.

Another important question to be answered is: How long can a set of generated spelling variants be used for? In the experiment here, two models were trained on two consecutive time frames of six months each. One model was used to re-create user queries, while the other generated spelling variants for index enhancement. The lack of a time gap ensured that the spelling variants
in the queries were similar to the spelling variants indexed. It may be, however, that humans use different spelling variants over time. In that case, the index could be enhanced with the wrong spelling variants resulting in lower precision and recall. How quickly a model for generating spelling variants decays is well worth investigating. An experiment will be undertaken to try to answer this in the next chapter.

For comparison with manually executed examples in Chapter 7, Table 8.3 lists how EEAHN behaves when queried with the same addresses. All example queries are handled now, including the example of an abbreviated street name that could not be handled through edit distances. The reason this query is handled well now is that there is no longer a limit on edit distances. Instead of failing to match the abbreviated term because it is too different, the index is enhanced with the variant of the term. This way, the query matches the correct document and the correct result is retrieved.
9 Validating the User Query
Model Stability

In the previous chapter, a model was used to generate spelling variants to enhance an index of addresses. A geocoding system with an enhanced index is EEAHN. It has proven to handle user queries with spelling variants better than EAHN – a geocoding system that supports fuzziness by allowing edit distances. Amongst other interesting aspects to investigate further, the question of the relevance of the model over time. The model generating spelling variants is trained on pairs of user queries and clicks. Thus, it observes modifications that users make to address tokens to generate modifications that are distributed similarly. Users, however, could change the way they state queries over time. They could stop using specific spelling variants, and start using new ones. Therefore, the rate at which a model generating spelling variants deteriorates is important to investigate.

Two models were used in the previous chapter. One model was used to enhance the index of the geocoding system. The other model was generating user queries with spelling variants. The models were trained on distinct data sets. In this chapter, we will use each of the two models to do both: Enhance the index of the geocoding system and generate a query set. This way we get two instances of EEAHN, each enhanced by its own model. Also, we get two different test query sets generated by the models that were used to enhance the geocoding indices. This way we can compute four sets of precision and recall reports: We can measure the performance of both EEAHN on the test set generated by its own model, as well as on the test set generated by the model of the other EEAHN. This analysis has already been undertaken and published in [6]. Figure 9.1 illustrates how the test setup changed compared to the previous chapter.

While – as visible in Figure 9.1 – we are measuring four combinations of two geocoding systems and two sets of test data, it makes sense to group the combinations into two: The goal is to understand how much the two models that are trained using consecutive, but distinct data differ from each other. Hence, in our evaluation, we will only differentiate between the scenario where one and the same model is used to enhance the index and generate the test queries and the scenario where two distinct models are used for that. We will group the four precision and recall reports accordingly, therefore.

The models require real, proprietary data to be trained, and only a fixed amount of such data is at hand. The chosen approach to compare the models trained on a six months time period each is suboptimal already: Seasonal influences could lead to model differences that do not exist if more data is
available to train each model. Therefore, only one segmentation of data is evaluated in this experiment. It does not seem to make sense to train models from, for example, a month worth of data, just to learn that different things are queried for in different months of the year, and hence, different spelling variants are used. Also, unlike in the previous chapter, there is no need to evaluate different amounts of indexed spelling variants here. The experiment is therefore conducted with EEAHN instances that were enhanced with 320 spelling variants per token each.

9.1 Hypothesis

There are exactly two scenarios that the experiment can result in: Either the two models are sufficiently different after being trained on different data, or they are not. Consecutively, we either will observe a difference in the precision and recall metrics of the two flows we are measuring, or there will be no difference.

A reasonable result would be that EEAHN serving queries generated by the same model that was used to enhance it yields better precision and recall compared to EEAHN serving queries generated by a different model. That would imply that the underlying data, and hence the models changed over time sufficiently leading to a reduction in performance. Another result could be that there is no observable difference between the two flows. That would not imply that a model can be used forever without regression. It only means that data does not change sufficiently in a six-month time frame. Hence a model can be used safely for at least six months before it needs to be updated.
9.2 Data for Evaluation

Two models trained on distinct six months of data each were used to generate two test query sets. As in the previous chapters, each of the two query sets contained six batches of 50,000 queries. Each batch contained queries with zero, one, two, three, four, or five spelling variants respectively.

9.3 Implementation

No additional code is required to execute this experiment. The entire indexing and evaluating pipeline from the previous chapter can be reused here. Only a little manual effort is required to execute the various measurements of EEAHN and test set combinations. The aggregation of the four reports into two scenarios was executed manually too. As a result, two precision and recall reports, grouped by the number of spelling variants in the queries were available.

9.4 Results

Figure 9.2 presents the performance of the two EEAHNs in the two given scenarios grouped by the number of spelling variants in queries. For each number of spelling variants, four bars denote two precision and two recall values. Thus, the precision and recall values of EEAHN are presented in two scenarios. The performance of EEAHN that was enhanced by the same model that was used to generate the test query set is plotted next to the performance of EEAHN that was enhanced by a model that is different from the model generating the test query set. Because we grouped the four reports into two scenarios, each bar is computed of 100,000 test queried against the respective EEAHN.

As clearly visible, the values of the metrics in the two scenarios do not differ. For every number of spelling variants in the test query set, the system behaves exactly in line with the results from the previous chapter.
9.5 Conclusions

Clearly, there is no difference in the performance of EEAHN between serving queries generated with the same model that was used to enhance the index or serving queries that were generated with a model trained on data that is six months away. Hence, we can conclude that – with sufficient data at hand – a model does not degrade within a time frame of six months. Unfortunately, only data from a time span of twelve months was made available by HERE Technologies. Therefore, we cannot identify a point in time, after which a model should be re-trained. In the next chapter, however, we will try to avoid the problem of the model degradation by not relying on a model at all.
In the previous chapters, we created EEAHN, a geocoding system that performs better on queries with spelling variants. The secret sauce to the increased performance is an index of address terms that is enhanced with spelling variants. To compute spelling variants from properly spelled address tokens, a statistical model was used. We relied on query logs to create such a model capable of generating spelling variants that are distributed in the same way as spelling variants are used by humans. Thereby, the log used needs to contain both: the query of the human user, as well as the result the user clicked on. That is a costly requirement that is not available often. For the work in this thesis, we used a log of queries and clicks shared by HERE technologies for scientific purposes. Another uncertainty of this process is that we considered the click of a user as a result selection. We just assumed that the user clicks on the result that she queried for. For example, a result could be amusingly wrong and motivate the user to explore it. That would lead to clicks that are not selecting the correct result to a query. Additionally, relying on a log of queries and results of an existing system implies that only query and result pairs can be observed that the system is already capable to serve. If a geocoding system is not able to discover the correct result to a query, the respective user does not get the chance to select the correct result. Also, such a model could degrade over time, as investigated in the previous chapter. While we have proven that a model does not observably decay over a period of six months, it is to be expected that after some time frame, a model would generate spelling variants that are not used by human users and vice versa. In sum, while the approach itself is promising, the dependency on a statistical model trained of a query log is a big burden. Therefore, in this chapter, we will investigate an alternative approach that avoids that dependency. The findings presented here have already been published in [7].

The high-level plan for the approach is simple: We know that enhancing an index allows the geocoding system to handle queries with spelling variants fine, without introducing additional ambiguity. Thus, we want to keep doing that. However, to derive spelling variants, we require query and result pairs. For convenience, we used data collected by a different, more advanced geocoding system. We discussed that collecting these pairs manually is an option too, albeit a costly one. In this chapter, the plan is to observe spelling variants on the very same system that we intend to enhance.

The idea is to set up an EAHN system as we did in Chapter 5. No edit distances or spelling variants are available at first. As we measured already,
EAHN can handle queries with no spelling variants pretty well: It performs on par with an Elasticsearch that indexed entire addresses as documents. Also, due to the document schema, it requires significantly fewer space. Due to the schema too, EAHN implicitly excludes house numbers from being part of spelling variants generation or edit distance computation. To enhance EAHN with spelling variants, now, the same queries will be used that were issued against EAHN. To derive spelling variants, queries need to be paired with correct results. Therefore, in an off-line process, we will enable edit distances on EAHN and replay logged queries. EAHN-1 performs worse than EAHN on queries with no spelling variants but significantly better on queries that have spelling variants in them. Thus, the plan is to simply derive the spelling variants to enhance EAHN with from the same EAHN configured to allow tokens to match with an edit distance of one. This way, no preprocessed set of queries and their corresponding results is required to derive spelling variants.

One obvious concern is the question: Why not just use EAHN-1 right away? How can EAHN-1 perform better than itself when applied in an off-line process? The rationale for this approach is the fact that allowing an edit distance of one introduces a regression for queries with no spelling variant. EAHN-0 performs better than EAHN-1 in that case. Thus, by sticking to EAHN-0 as the geocoding system of choice and by enhancing it with spelling variants derived from results from EAHN-1, the goal is to get better precision and recall results. Additionally, because an off-line process is used, we get the chance to use a result-filter that allows us to only enhance spelling variants derived from results that we are most certain of.

Figure 10.1 illustrates the process used in this chapter. One and the same EAHN geocoding system is configured to run as EAHN-0 allowing no edit distances between the query and result tokens, and EAHN-1 allowing an edit distance of one. Only results from EAHN without edit distances are used to measure precision and recall performance. In an off-line process, the same queries are sent against the EAHN-1 that allows one edit between the query
and result tokens. The results obtained this way are sorted by their BM25F scores first. A result-filter drops results that are not sufficiently certain from the pipeline. Queries yielding in results that are not dropped are used together with their results to derive spelling variants. In the last step, EAHN is enhanced with the spelling variants derived this way.

As discussed, geocoding is an error-prone process. Results ordered by their BM25F score. That score is a sum of the scores of the query tokens. The rarer a token is in the data, the lower is its document frequency and hence the smaller denominator of the formula to compute the score. The more often a term appears in the document, the higher is its term frequency and hence the greater the numerator of the formula. In sum, a document gets greater score if it contains more rare tokens from the query. While a great heuristic, that approach is not always providing the right result with the highest score. Especially when introducing fuzziness, for example, by allowing edit distances, some portion of the results will be wrong. Therefore, the results are filtered according to specific criteria in the proposed process: First, only the top result can be correct. Any result with a lower BM25F score will not be considered for spelling variant derivation. Second, the top result needs to have a minimum score value. This way, only documents with sufficiently many tokens that are sufficiently rare in the data are candidate results to derive spelling variants. Lastly, the gap between the scores of the first and the second result needs to be great enough. Only this way, we can be sure that the query matched exclusively to the one and only correct result. Even with this filtering approach, it is clear that some portion of incorrect results will be used for spelling variant derivation. That leads to incorrect spelling variants being part of the wrong addresses and is something to avoid as much as possible. The hope is, however, that the strict filtering drives the number of incorrect results down far enough so that the overwhelming majority of correct derivations still can improve the systems precision and recall metrics.

Another aspect to discuss is the fact that the system being enhanced is the source of the enhancements itself. The derived spelling variants are fed into the EAHN that has generated them. Thus, it is fair to assume that after the first round of enhancements, additional different spelling variants will be derived. We will investigate in this chapter how much benefit there is in iteratively repeating the proposed enhancement process, therefore.

For clarity, we will refer to the geocoding system without any enhancement as EAHN. After each round of enhancements, an asterisk will be appended to EEAHN to differentiate from the EEAHN with fixed numbers of spelling variants from the previous chapter. Therefore, EEAHN* is the geocoding system we get after executing the enhancement process one time, EEAHN** after executing it twice, and so forth.

10.1 Hypothesis

It is obvious that the enhancement process will be able to derive spelling variants. It is clear also that some spelling variants will be derived from queries
and results that are incorrect. The Hypothesis is that the spelling variants derived this way will improve the precision and recall of the geocoding system enhanced. Additionally, the hypothesis states that the number and the effect of incorrectly derived spelling variants will be small enough so that the system will not regress. Specifically, the expectation is that EAHN enhanced with spelling variants derived through the proposed process will have better precision and recall metrics than EAHN-1 that allows an edit distance of one.

10.2 Data for Evaluation

Unlike in the previous chapters, for the experiment undertaken here, we can no longer differentiate between the various test query sets with various numbers of spelling variants in them. The test query set that we will measure precision and recall on, is the same query set that we use to derive spelling variants from. Measuring how a query set with, for example, five spelling variants per query can be used to enhance the index is not what we want: Likely, the query sets with few spelling variants allow deriving spelling variants best. After an index has been enhanced, it is more likely to handle queries with many spelling variants too. Hence, a mix of queries with different numbers of spelling variants will be used to evaluate and enhance EAHN. Specifically, to ensure the effect of the enhanced index on the queries with many spelling variants, the same underlying query is required to be present with various amounts of spelling variants in the enhancement set.

Therefore, first, 50,000 addresses were sampled from the indexed address set at random. For each of the addresses, four queries were generated. Queries had zero, one, two, and three spelling variants in them. The resulting set of 200,000 addresses was used to enhance the index. For performance reasons, precision and recall were measured on a smaller subset. The test subset was a random sample of 40,000 cases from this set. It is large enough to properly represent the distribution of spelling variants and query formats from the full set of 200,000 queries. However, it is much quicker to execute against the index.

10.3 Implementation

Listing 10.1 presents a simple method to enhance data in an index. Given the identifier of the document to enhance, the spelling variants to be added to the document, as well as the index itself, the method simply extends the indexed field text with the provided spelling variants. As in EEAHN from the previous

```python
def enhance_index(nid, variants, es, index_name):
    variants = ','.join(variants)
    result = es.get(index_name, 'doc', nid)
    result = result['_source']
    result['text'] += ' ' + variants
    es.index(index_name, 'doc', result, id=nid)
```

Listing 10.1: Method enhancing an indexed entry of EAHN.
def compute_variants(query_text, result_text):
    # assign query and result tokens
    edits = TokenMatch(Tokernizer(query_text),
                       Tokenizer(result_text))

    edits = filter(lambda e: e[0] and e[1] and e[0] != e[1], edits)

    # we only care about the query terms - drop other tuple members
    edits = map(lambda e: e[0].encode('utf8'), edits)

    return edits

Listing 10.2: Method to compute spelling variants given a query.

chapter, the spelling variants end up indexed next to the properly spelled tokens in one and the same field.

The mechanism to compute the spelling variants is presented in Listing 10.2. Given the query text and the result text, similar to computing the model in Chapter 6, the text is tokenized first. Then, query and result tokens are assigned into matching pairs. The list of token pairs is filtered to only contain pairs with tokens that do not match exactly but require edits to match. Taking the query tokens from that list leaves exactly the spelling variants that are not part of the result, and hence, not indexed yet. Note that if a query has no spelling variants, or if all the variants from the query were indexed already, this method might return an empty list.

Finally, Listing 10.3 connects the methods into the desired functionality. The method presented queries the EAHN similarly to the default EAHN address search described in Chapter 5. Thereby, it enables Elasticsearch to match tokens with an edit distance of one. Unlike in the EAHN search, the results are not returned. For each assumed house number, all 250 street segments that have a matching house number are accumulated in line 19. As these candidates are collected from several queries, they need to be sorted by their score once again. That takes place in line 22. Next, in line 31, the spelling variants of the query to the top result are computed. The conditions are checked in line 37. Note that the gap between the scores of the first result and the second result is computed in a relative manner: The delta between the scores of the top and the second result needs to be bigger than 4% of the top score. The minimum required top score, on the other hand, is an absolute value set to 22. Finally, in line 38, the enhance_index method is invoked to apply changes to the EAHN index.

The thresholds for the conditions were derived through an iterative manual process. As in the previous chapters, the queries to measure and enhance the index are generated from indexed data. As a consequence, for every generated query, the identifier of the address it was generated from is known. Therefore, the method presented in Listing 10.3 can easily be adapted to output flags indicating whether the index is enhanced with spelling variants at all and whether the spelling variants were derived from a correct result: Simply passing in the identifier of the query allows to compare it with the identifier of the top result. The information on spelling variants being derived at all is already present. Thus, with minor variations of the method, statistics on the ratio of enhanced addresses and the ratio of incorrectly enhanced addresses can easily
def query_and_enhance(query_text, es, index_name):
    results = list()
    tokens = Tokenizer(query_text)
    tokens = sorted(tokens, key=len)
    for i in range(len(tokens)):
        text = ".join(tokens[:i] + tokens[i + 1:])
        candidates = es.search(index_name, body={'query': {'match': {'text': {'fuzziness': 1}}}, size=250})
        candidates = candidates['hits']['hits']
        for candidate in candidates:
            if 'hn' in candidate['_source']['house_numbers']:
                result.append(candidate)
    results = sorted(results, key=lambda r: r['_score'], reverse=True)
    top_result = results[0] if len(results) > 0 else None
    top_score = top_result['_score'] if len(results) > 0 else 0
    top_delta = results[1]['_score'] / top_score if len(results) > 1 else 1
    if top_result:
        variants = compute_variants(query_text, top_result['_source']['text'])
        # check conditions:
        if (1) top_result has edits,
        and (2) top result has absolute score > 22, and
        (3) gap between top and second result is larger than 96% of top result score
        if variants and top_score > 22 and top_delta < .96:
            enhance_index(top_result['_source']['ids'][0], variants, es, index_name)

Listing 10.3: Method to query and, if feasible, enhance the EAHN index.

be output. To allow repeated execution without affecting the result, enhancing the index was disabled in the modified method. An independent set of 1,200 queries with an equal amount of zero, one, two, and three spelling variants in them was used. Iteratively, the above thresholds were established by hand, trying to maximize the ratio of queries for which spelling variants could be derived at all while trying to minimize the ratio of queries that led to incorrect spelling variants derivation.

10.4 Results

Figure 10.2 shows the precision and recall metrics of EEAHN evolving after one, two, and three iterations, compared to EAHN allowing edit distances of 0, 1, or 2. For reference, the performance of Nominatim is presented as well. As discussed, Nominatim does not handle queries with spelling variants well. In the test query set of queries with zero, one, two, and three spelling variants in them it is outperformed by EAHN with no edit distances too. As also observed in Chapter 8, allowing tokens with an edit distance of one to match, leads to a strong increase in recall. Many queries that were not served correctly with EAHN-0 can be handled with EAHN-1. Also, as expected, the metrics regress when the greater edit distance of two is allowed: EAHN-2 performs worse than EAHN-1 in precision and recall. The last three bars on Figure 10.2 present precision and recall of EEAHN*, EEAHN**, and EEAHN***. All EEAHN variants have greater
10.4. Results

Figure 10.2: Precision and recall of Nominatim, EAHN allowing zero, one, or two edits between matching tokens, and EEAHN after one, two, and three enhancement iterations [7].

Precision than the EAHN counterparts: In fact, the precision of EEAHN** is with 87.6% 5.8 percent points larger than the precision of EAHN-1 with 81.4%. On the other hand, with 43.3% the recall of EEAHN is lower compared to the recall of EAHN with 48.3%.

Another interesting observation is the fact that with each iteration, the performance of EEAHN improves: EEAHN** has better precision and recall metrics than EEAHN*, while EEAHN*** performs better than EEAHN*. While the performance gain is not huge, it seems that there is value in repeating the enhancement cycles. Figure 10.3 confirms this displaying the ratios of queries that spelling variants could be derived from for each of the three iterations. The figure also shows for each iteration, how many of the queries had a correct result for computing spelling variants and enhancing. While the ratio of queries that spelling variants could be derived from decreases with every subsequent iteration, the portion of queries that enhanced an incorrect result does not change. In the specific scenario of the undertaken experiment, it seems like undertaking at least the first two iterations is of value.

Figure 10.4 presents a similar overview of ratios of queries, grouped by the
Chapter 10. Continuously Enhancing Geocoding System

Figure 10.4: Ratios of queries that allow deriving spelling variants for index enhancement and their portions of correct and wrong results, by the number of spelling variants in queries.

The number of spelling variants in the queries. The ratios are computed over all three iterations. Obviously, with zero spelling variants in a query, almost no spelling variants can be derived for enhancing the index. While the amount of cases is neglectable, all of the few derived were derived from incorrect results. The figure gets more interesting when looking at the bars representing the ratios of queries with various amounts of spelling variants that spelling variants could be derived from. Like in Figure 10.3, the portion of queries that led to correct results for spelling variants derivation and enhancement decreases, the more spelling variants are present in the query. Unlike in Figure 10.3, the portion of queries that led to incorrect results being enhanced grows.

10.5 Conclusions

Table 10.1 presents the same queries with spelling variants that were used in Chapter 7 and Chapter 8. Because the method relies on edit distances to derive spelling variants, no spelling variant can be derived sometimes. Specifically, abbreviations do not match their unabbreviated tokens any more and cannot be derived this way. This, however, can be mitigated by enabling the term normalization technique described in Chapter 10. In other examples, the correct result is retrieved but doesn’t score well enough to pass through the enhancement filters. As presented in the table with the London address example, adding more address elements to the query leads to a better matching of the query terms to the document. It, hence, increases the scoring so that computation of spelling variants is possible and the corresponding document is enhanced. Both queries with spelling variants for London started being served correctly after the document has been enhanced. The third example for the address in Berlin got enhanced too. It, however, did serve the query with the spelling variant correctly even before the enhancement. The many additional correctly spelled tokens of the query led to the correct result out of the box.

Several further investigations can be undertaken for this approach: The time effect of the derived spelling variants needs to be investigated similarly to the
10.5. Conclusions

Table 10.1: Examples queries and how they affect the enhancement of the EAHN index.

<table>
<thead>
<tr>
<th>query</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>linienstraße 13, berlin</td>
<td>no spelling variants in the query, hence no enhancement.</td>
</tr>
<tr>
<td>linienstr 13, berlin</td>
<td>an allowed edit distance of one is not sufficient to match the abbreviation. The wrong document comes up as the top result but does not score well enough. No enhancement.</td>
</tr>
<tr>
<td>london, belgrave square 34</td>
<td>no spelling variants in the query, hence no enhancement.</td>
</tr>
<tr>
<td>london, bellgrave square 34</td>
<td>the correct document is retrieved but does not get a sufficiently high score. No enhancement.</td>
</tr>
<tr>
<td>london, westminster, bellgrave square 34</td>
<td>with the additional query term, correct document scores well enough to this query. It is enhanced with the spelling variant &quot;bellgrave&quot;</td>
</tr>
<tr>
<td>ernst reuter platz 7 berlin</td>
<td>no spelling variants in the query, hence no enhancement.</td>
</tr>
<tr>
<td>ernst reeter platz 7 berlin</td>
<td>the correct document scores well enough to this query. It is enhanced with the spelling variant &quot;reeter&quot;</td>
</tr>
</tbody>
</table>

Experiment undertaken in the previous chapter. If we assume that users change the way they search, and therefore use different spelling variants over time, we should investigate if outdated spelling variants prevent the geocoding system to perform better. Possibly, spelling variants need to be monitored in their occurrence counts over time, and removed from the document when they are no longer used. The filtering mechanism clearly contributed to the success of the proposed approach, but can be refined even finer. Instead of manually setting the thresholds, they can be derived automatically in a data-driven manner. Also, filter thresholds might vary by language, or query or result region. We have observed on the manual examples that abbreviated address elements are not likely to be picked up by the process automatically. It remains to be investigated, how much standardization will further improve the precision and recall metrics. Finally, in the proposed approach, only the document queried for was enhanced. There likely is the possibility to enhance further documents, if, for example, a spelling variant is derived for city or district names. Because address entity names are used across regions – street and city names, for example, are often reused – that might contradict to the approach of regionally specified filter thresholds. Hence, a middle ground needs to be found that, for example, allows deriving spelling variants for a specific city, but not the other cities having the same name.

Overall, however, we have observed that the proposed approach results
in greater performance compared to allowed edit distances of EAHN. The
filtering mechanism works, allowing to enhance correct documents. It provides
additional precision, at the cost of some recall, unfortunately. We have also
observed that additional iterations lead to additional spelling variants being
derived. In this experiment, the same query set was used to iteratively
enhance the index of EAHN. In a production setting, however, already enhanced
documents can be further enhanced by additional queries with different spelling
variants. Also, and unsurprisingly, we have confirmed once again that a
gecoding system can handle queries with fewer spelling variants better than
queries with many. It, thus, remains a task for the specific application area to
observe the type of queries that need to be handled, and to pick a proper approach
to enable the geocoding system to do so.
11 Conclusions

Geocoding is the process of resolving address text into geographic coordinates. It is used where humans and computers communicate about locations. In this thesis, various approaches to geocoding have been tested. Dedicated measurements have been conducted, showcasing the impact of the algorithmic changes on the performance of the geocoding system. The data that was used, thereby, was not changed. Hence, a data change has no influence on the measurements.

In sum, in this thesis, first, a generic document scheme suitable for indexing arbitrary address formats was developed. The document scheme was then further enhanced and evaluated for its usage in geocoding systems. Next, using a log of queries and results from an existing geocoding system the approach to specifying spelling variants that the system should handle was explored. That approach has proven to guarantee a greater quality of service compared to common methods for handling typos, abbreviations, and the like. Finally, a process to continuously improve the performance of a geocoding system based on a generic document search engine was evaluated. That process has proven to increase the precision and recall of geocoding systems over time.

The goal of this thesis was to identify techniques that make up a good geocoding system. TFIDF to rank address results that match best to a query, normalization of house numbers by grouping them in street segments, indexing spelling variants to support fuzzy search maintaining a high level of precision and recall, and the iterative process to continuously improve a geocoding system at hand are such techniques that were evaluated empirically and have proven to make up a good geocoding system.

Specifically, this thesis focuses on geocoding queries that are stated by human users. Thereby, it is not important if the queries are actually stated by a human. It suffices if a human was in the loop of conveying the address that needs to be geocoded. Changes to the address made along the way are a challenge to modern-day geocoding systems. Algorithmic improvements to deal with this challenge are investigated in this thesis. Six types of accidental and deliberate changes were identified:

1. **Order of address element names**: Address elements, as well as parts of address elements, might be specified in a shuffled order.

2. **Presence of address element names**: Address elements can be absent even though the common postal address format expects them to be specified in a complete address. Similarly, address elements that are usually not part of a postal address can be specified.
3. Alternative, for example, translated or historical address element names: Especially names of administrative areas as countries, regions, and cities differ in various languages. Oftentimes, street names can be translated too. Similarly, address elements have historic names that are no longer used, or that are spelled in a different way.

4. Abbreviation of address element names: Street types such as street, Allee, boulevard, or esquina are commonly abbreviated in all languages. Names of states, cities, and countries are oftentimes abbreviated too. Sometimes non-standard abbreviations are used as well.

5. Replacement of diacritics in address element names: Character diacritics as accents, umlauts, or the sharp s are often escaped into their ASCII counterparts. This can be rooted in a technical limitation such as a database encoding or a plain English keyboard layout. This also can be based on the reluctance of users to spell out rare and complex characters.

6. Typing mistakes of address element names: Finally, typing mistakes are made by human users too. That is especially often the case with address elements that have a complex spelling as Massachusetts, Nürnberg, or Bruay-la-Buissière. The technical means used to input the address element names might introduce a specific kind of typing mistakes too.

In a nutshell, the first two changes are regarding changes to the address format, while the remaining four are describing cases where spelling variants of original address element names are used. These spelling variants are commonly encountered in various ways, enabling a geocoding system to handle queries stated this way. In this thesis, standardizing indexed data and queries, or enabling fuzzy matching by allowing distinct tokens within a specified edit distance to be considered a match are discussed. Enriching data with translations and historical variants of address element names is a common option too, but is explicitly avoided throughout the course of this work. To understand the impact of the various algorithmic methods employed, data changes are deliberately excluded.

The core idea of the approach pursued here can be summed up as follows: (i) Index addresses as opaque documents to support non-qualified searching. That should help to tackle changes to addresses that relate to the address format. Specifically, the changes of type 1 and type 2 can be resolved this way. (ii) Identify, which spelling variants are used for which address element names by human users. (iii) Index spelling variants of address element names as part of the address. That should support the geocoding system in handling spelling variants that fit into the remaining four types of changes.

11.1 Summary

As the first thing, the hypothesis that generic document search engines can function as geocoding systems was tested in Chapter 4. For that, all addresses from an instance of Nominatim were extracted and indexed in Elasticsearch.
11.1. Summary

This way two geocoding systems were set up with the exact same data. While Nominatim, however, executes qualified searches for addresses, Elasticsearch ranks address documents by the score computed using the TFIDF variant BM25. A simple wrapper took care of querying Elasticsearch and cutting off everything except the top result as the geocoding response. To test the hypothesis, the self-match rate of randomly sampled addresses was computed for both systems. Additionally, different uses of address formats were simulated. For that, query tokens were shuffled randomly and a varying amount of tokens was dropped from the query. Already with complete addresses with tokens in their original order, the geocoding system based on Elasticsearch demonstrates a much higher self-match rate than Nominatim.

Also shuffling query tokens has a minor impact on the performance of Elasticsearch, vastly reducing the performance on Nominatim on the other hand. For queries with increasingly dropped tokens, the performance of the Elasticsearch based geocoding system evolved as expected. The more query tokens were excluded from the query, the higher the chance that the query no longer contains sufficient relevant information to be correctly geocoded, and hence, the lower the performance of the geocoding system. In other words, the self-match rate of the Elasticsearch based geocoding system dropped proportionally to the rate of dropped tokens. Nominatim, on the other hand, showed that it cannot deal well with shuffled and incomplete address queries. While staying constantly low, the match rate increases at first when query tokens are dropped, reaching its maximum around addresses with only 70% of their tokens left.

Additionally, a set of addresses for testing was scraped from the web. The data came with two fields, containing the address itself as well as the latitude and longitude coordinates of that address. Unlike with self-matches, addresses taken from an external source can be linked to addresses in the data in various ways. The distance between the geocoding result and the location attached to the address was chosen in this case. Specifically, a geocoding query was considered to be successfully served when the result was within 100m of the given location. Like with the self-match rate, addresses from the web were shuffled. Also, 75% of the address tokens were removed from the queries. Like with the self-match rate, Elasticsearch outperformed Nominatim in this test set too. This experiment proved that ignoring the address format and indexing addresses as opaque documents is a viable approach to geocoding. The independence of address formats enables supporting queries that are incomplete and stated out of order.

In a second step, some knowledge about address elements was utilized. With the goal to handle spelling variants in mind, fuzziness to address search needed to be introduced. Before doing that, though, house numbers needed to be handled differently than the rest of the address. House numbers are the most specific and the most reused part of each address. Hence, when searching for an address, no fuzzy logic should be applicable to them. Thus, a modified document format was created for indexing. In Chapter 4, entire addresses were indexed and just the latitude and longitude coordinates were used as the payload. In Chapter 5, on the other hand, a key-value map was the payload of an index.
that contained street segments. A street segment, thereby, was considered to be described by the entire address, except the house number. Thus, multiple house numbers were part of each indexed street segment. The house numbers were used as keys of the map. The geographic coordinates of each house numbers were stored as the value. In sum, house number addresses were grouped by their street segment. Each street segment was stored into one document along with the house numbers on that. To geocode an address, therefore, the geocoding system had to discover a street segment first, and then, look up the house number in the map.

As a side effect of this data reorganization, the number of documents decreased vastly. That, in turn, led to a decreased index size. Out of convenience, some amount of data was replicated in the new format. Yet, due to the reorganization, the compression ratio was higher so that the compressed data file required less disk space in the new format. Another consequence of the new data model was that it oftentimes is not clear which street segment is the right one prior to looking into each house number map. That is due to the fact that street segments oftentimes only differ in one single address entity. For example, a street that goes through two districts of the same city would be split into two street segments. When a query, however, does not contain the district name the only way to tell which of the two street segments is meant. Only looking into the house number map and discovering a matching house number does select the right one. That, inherently, also introduces the risk of a false positive. In edge-cases, two equally named streets in two districts of the same city could have the same house numbers on them. The results of the experiment conducted in Chapter 5 state: These edge-cases are rare enough to not affect the measurements. Like in the previous experiment, the self-match rate of Elasticsearch with Aggregated House Numbers (EAHN) was measured along with the performance of on the address set scraped from the web. A third address data set was scraped from the web too, to further confirm the findings. With all three sets, EAHN behaved closely to Elasticsearch with plain addresses. Thus, the proposed approach to organize addresses has proven valuable.

In Chapter 6, prior to jumping on implementing fuzzy address search, appropriate test data needed to be created. The company HERE Technologies kindly provided a log of a year of address searches towards their customer-facing services. The log contained queries that the users typed, as well as the results that were presented to the user. Additionally, the user interaction with the results was logged. Specifically, to get a more detailed view on one of the results, or to, for example, compute a route to it, the user had to click a result from the result list. These clicks were part of the log too. One obvious option was to use the queries stated by the users directly on any geocoding system and see how many queries it is capable to serve. That, however, left the question about how to judge the results of that geocoding system open. A linkage between the queries from the log and addresses in the geocoding systems under test was missing. However, only with that linkage, it would be possible to decide if a response is correct or not. A different path to create a test set was chosen. Using the query log provided by HERE Technologies, a statistical model was
11.1. Summary

created that was capable of generating spelling variants and queries that human users would state. For that, first, the click even in the log was used as the link between the query and the correct result. While that is an assumption, it is not too far-fetched. Clearly, there were clicked results in the log that were not the right results for their respective queries. Most of the clicks, however, were part of the designed use case. Users searched for an address and clicked the correct result to either see its location, get additional information, or compute a route. Having query and result pairs, a modified Levenshtein Algorithm and a modified Hungarian Algorithm were used to link each token from the query to the corresponding token in the result. The algorithms were, thereby, modified such that they did not allow false token matches while still supporting fuzzy differences between these tokens. For instance, special care was taken so that abbreviated tokens did match to their spelled-out counterparts. Next, all such token pairs were considered. Pairs with equal tokens were discarded, as they provided no additional information. In the case the tokens were not spelled equally, though, the Levenshtein Algorithm was employed once again identifying the specific differences between the result and the query token.

These differences were counted by the statistical model in their respective contexts. A context, thereby, described the specifics, at which an edit happens. The statistical model then simply counted the number of specific edits in their context. For example, the context of deleting a character is specified by the preceding and the succeeding character of the delete. Assuming the characters \( e \) and \( t \) as a context, the statistical model would count how often the character \( e \) is deleted in between. Thus, if a query contains the token \( strewt \) and that token is paired with the address token \( street \), the counter of deletes of the character \( e \) between \( e \) and \( t \) is incremented in the model. After processing all token pairs that are not equal, the model contains counts of observed edits in their contexts.

A second model was created in a similar way. The token pairs from query and address texts were used as the starting point again. From the address token, the address element type was known. That type was propagated to the corresponding query token, resulting in a fully qualified query. The address element type for each token of the query was known. That, implicitly, defined the address format that the user used to specify the query. The second model simply counted the occurrences of each such query format used. These two models were used to generate user-like queries from fully qualified addresses. In the first step, the query format was chosen. For that, the available address elements of an address were used, to select available address formats. Formats that contained address element types that are not part of the qualified address were not used to generate queries. The counts of the applicable formats specified the strata for stratified sampling. High counts got a high probability of being sampled, while low counts got a low probability. This way, query formats that were used often were sampled often, while rarely used query formats were only sampled rarely.

Having sampled a query format at random, in accordance with the query format counts, the query text could be assembled from any qualified address. In subsequent steps, query tokens were replaced with spelling variants using the other model. For that, first, a query token to be replaced was selected at
random. Next, the contexts of that token were collected. Only edits that were observed in the contexts of the token were considered. The selection of the edit to apply was done in the same way as the selection query address. The higher the count of a specific edit, the more likely it was sampled to be applied. Once an edit is selected, it was applied to the token creating a spelling variant, a human user could have had spelled. The process was repeated to generate the necessary amount of spelling variants in the query.

Six test query sets were generated to have zero, one, two, three, four or five spelling variants per query respectively. It is important the stress that because of the stratified sampling, the distribution of the spelling variants and the query formats generated was as the distribution of spelling variants and query formats observed in the user log. Rare spelling variants and query formats were used rarely, while common spelling variants and query formats were used more often. The test query sets, therefore, contained user-like queries for addresses that were randomly sampled from the indexed data. That approach had two benefits that made evaluating geocoding systems simple. Firstly, it was absolutely certain that the addresses existed in the data. Thus, a data gap could not be made responsible for not serving a query. And secondly, the correct result to each query was known. The address from the data that was used to generate the query was the one and only correct result to that query. The queries could be used similarly to queries for assessing the self-match rate in the previous chapters. To test the test sets, their queries were issued against the geocoding systems Nominatim and EAHN. The more mature geocoding system Nominatim outperformed EAHN with all six test query sets. That is because Nominatim has features built in to deal with some common abbreviations. However, with one single spelling variant in the query already, the performance of Nominatim drops severely resulting in an unusable geocoding system.

Now, with test sets of user-like queries, traditional approaches to handling spelling variants through a fuzzy search were measured in Chapter 7. For that, the six generated user-like query test sets were replayed against the three geocoding systems Nominatim, Elasticsearch, and EAHN. All three systems were set up with the same data. Also, the test sets were generated from the same data. Hence, for each test query, exactly one single correct result existed. Also, that result was known at query time. Thus, the self-match rate could be computed easily. However, measuring the self-match rate was no longer sufficient given that queries were generated. Besides measuring the ratio of correct responses, the ratio of wrong responses needed to be measured too. Therefore, the precision and recall metrics of a geocoding system were introduced prior to measuring the performance of the systems.

The recall is the ratio of correct responses over the entire query set. Thus, in the context of generated test queries that are rooted in the data, recall is exactly equal to the self-match rate that we have been measuring thus far. The precision is exactly the additional metric indicating how many wrong responses are given by a system. Precision is defined as the ratio of correct responses over such queries that have produced any result. With these two commonly used metrics, we were able to observe (i) how often a system serves a query correctly, and
whenever a system was not able to yield the correct response, (ii) how often it provided a wrong response instead.

That was an important extension to the measurement methodology, as depending on the use case, wrong responses can be harmful and high precision is favored instead of a high recall. Looking at the precision and recall metrics of Nominatim first, there were no surprises. The recall of Nominatim has been measured in the previous chapter and was in an acceptable range for the query test set with no spelling variants. Precision behaved in the same way. Both metrics dropped as soon as queries with a single spelling variant were measured. The more spelling variants were present in the queries, the lower those metrics dropped. That, indeed, is expected behavior as with additional spelling variants the queries become harder to serve. Elasticsearch with address documents and EAHN behaved in the same way. Both geocoding systems are based on the document search engine Elasticsearch. Elasticsearch offers fuzzy search out of the box. When stating a query, the user thereby can specify to allow a query token to match to document tokens with an edit distance of one or two. Also, an edit distance of one or two can be set by Elasticsearch depending on the token length automatically.

Thus, for each query test set and each geocoding system, precision and recall were computed allowing an edit distance of zero, one, two, as well as allowing an automatically set edit distance. Again, the recall of EAHN that allows no edit distances was known from the previous chapter. Both systems behaved similarly for both metrics. Due to the nature of EAHN, it had a much higher precision at the cost of recall in all six test query sets. The baseline geocoding system based on Elasticsearch had much more similar precision and recall values for all query sets. Also, both systems preferred clearly better than Nominatim as soon as an edit distance of one was enabled. Allowing no edit distances, both systems had trouble serving queries with spelling variants. That, firstly, confirmed that allowing an edit distance is a valuable approach to handling spelling variants in geocoding queries. Secondly, it proved that the generated test set is close to how humans would state queries with spelling variants.

Next, in Chapter 8, a first experiment to improve the precision and recall metrics was executed. The hypothesis was that the common approach to handle spelling variants had clear flaws. Allowing edit distances between query and address tokens certainly did allow to handle some of them. However, edit distances are a very vague way to describe spelling variants. Clearly, some tokens in a given edit distance from a properly spelled address element name are spelling variants that are used a lot. Also clearly, some tokens are never used as spelling variants but still qualify for a match as they are within the given edit distance from a properly spelled address element name. Thus, a better way to handle spelling variants is to explicitly allow only those spelling variants that users actually make. Especially the precision of such a geocoding system was expected to be greater than the precision of a geocoding system relying on edit distances.

The experiment was set up as follows: First, the user query logs provided by HERE Technologies were split up into two equally large chunks. The first chunk
Chapter 11. Conclusions

contained log entries from the first six months while the second chunk contained the last six months of log data. From one chunk a spelling variant and a query format model were trained just as described in the previous chapter. As in the previous chapter too, these models were used to generate six test query sets with zero, one, two, three, four, and five spelling variants in each query respectively. The second chunk of log data was used to create another spelling variant model. This model was created on different data than the previous models and thus, is independent of them. The model was used to enhance the indexed address data with spelling variants of the correctly spelled address tokens. Multiple indices with different amounts of generated spelling variants per token were set up this way.

We expected that the size of the indexed documents would grow linearly, proportional to the number of generated spelling variants per address token. That, however, is not the case. As discussed, to generate a spelling variant, first the contexts of a token were considered. Shorter tokens do not provide many contexts, and hence, the number of spelling variants that can be generated for them is small. This observed limit of spelling variants that can be generated this way confirmed that the experiment was on track to confirm the hypothesis. Only such spelling variants that were actually observed in the logs were the ones that the index was supposed to contain. Generic tokens that are a specific number of edits away from the original token but never used by humans were excluded this way. Thus, the most common spelling variants were indexed for tokens for which many spelling variants could be generated. For some tokens, however, the number of spelling variants that could be generated was maxed out. The class of geocoding systems that contain data enhanced with spelling variants was labeled Extended EAHN (EEAHN). The performance of EEAHN was measured and compared to the performance of EAHN, EAHN allowing an edit distance of one, and Nominatim. EEAHN performed best with regards to precision and recall.

An additional experiment was conducted in Chapter 8. The performance of a hybrid geocoding system containing generated spelling variants of address element names and allowing an edit distance was measured. Various combinations of edit distances and amount of indexed spelling variants were tried out to observe what setup yields the best precision or recall. As a rule of thumb, as one would expect, more indexed spelling variants and a greater edit distance lead to a higher recall, while as for precision no edit distance should be allowed and only a limited number of most common spelling variants should be in the index.

In Chapter 9 an important test on the statistical models is executed. A statistical model that generates spelling variants was used in the previous chapter to enhance addresses with spelling variants. Indexing such enhanced addresses allowed to handle spelling variants in queries better than allowing arbitrary tokens within a specific edit distance to match. An important question, however, was: How long can such a statistical model be used? Clearly, people change over time. Likely, the spelling variants used by humans change over time too. To verify if such a change is observable within a year, the following experiment was conducted. As in the previous chapter, the user log data was
split up into two chunks. Each chunk had six months worth of data. Unlike in
the previous chapter, this time both models were used to generate test query
sets as well as to set up an EEAHN geocoding system. Each test query set was
run against every EEAHN instance resulting in four sets of precision and recall
numbers. The four sets were combined into two. One set of precision and recall
metrics was measuring the performance of EEAHN geocoding test query sets that
were generated with the same model that was also used to enhance that same
EEAHN. The other set of precision and recall metrics was measuring the opposite.
It showed the performance of EEAHN enhanced with one model geocoding query
test sets that were generated with the other model. The two precision and recall
sets were exactly on par with each other. From this experiment, we concluded
that even if with time spelling variants made by human users change, it does not
happen within a single year.

Lastly, in Chapter 10, all the insights from the previous experiments are
combined. A geocoding system is set up that can continuously improve itself
using the techniques explored in the previous chapters. Specifically, that
geocoding system – a variant of EEAHN – is mitigating the limitations of EEAHN
introduced in Chapter 8. There, a log of queries and clicked results was used to
create a statistical model that was capable of reproducing the spelling variants
observed in the log with their original distribution. That model was used to
precompute likely spelling variants for address tokens and index them along
with the properly spelled tokens, thereby explicitly defining the allowed spelling
variants that users might make.

While the approach did produce a geocoding system with higher precision
and recall metrics, the limitation was in the very beginning of the process. A
log from an existing geocoding system was used at the root of EEAHN. Thus,
only such query and result pairs that the original geocoding system could handle
can be found in the log. A different method was necessary to use the approach
for learning spelling variants that no geocoding system can handle. One such
method would be to link correct results to each query manually. That is a very
costly way to find correct results for user queries. Moreover, finding correct
results to user-entered addresses is exactly the purpose of a geocoding system.
Based on that, a process for continuously enhancing a geocoding system has
been developed. The core idea of that process was the EAHN geocoding system
serving user queries the common way. In a separate offline process, however,
the received user queries were geocoded allowing an edit distance of one, similar
to the experiment conducted in Chapter 7. The output of the offline process was
consumed by the modified Levenshtein Algorithm and the modified Hungarian
Algorithm linking non-equal query and result tokens. Now no statistical model
was necessary anymore.

The observed spelling variants could be used to directly enhance the result
from the result and query pair it has been observed on. Several aspects of
that method needed special care. Firstly, it was important to understand that
an EEAHN geocoding system with some enhanced addresses produces different
results than a geocoding system that was used to enhance it. That is due to
the redundancy of addresses. For example, an address specifying a rare street
name is likely to be handled correctly without any enhanced tokens even if the district name is misspelled in the query. However, having the spelling variant of the district available might break ties, if the request contains a street that exists in multiple districts, and misspells the district name specifying the exact spelling variant indexed. Thus, enhancing an EEAHN geocoding system with spelling variants observed using an EAHN geocoding system that allows an edit distance between the query and the result tokens is not equivalent with simply using that EAHN geocoding system. Secondly, it is clear that geocoding systems have an error rate. Therefore, a result-filter is necessary filtering out obviously incorrect results returned by the EAHN geocoding system that allows an edit distance between the query and the result tokens and is used to observe spelling variants. As discussed, even combining EAHN with the result-filter yields different results compared to enhancing EEAHN using that same combination of EAHN and result-filter. The result-filter used in Chapter 10 used scores of the top results yielded by Elasticsearch. It ensured that the absolute score of the top result was above a threshold as well as that the score delta between the first and the second result was large enough. Thirdly, it is important to comprehend that the process described had an iterative character. Thus, while some spelling variants may have been derived this way in one go, the hypothesis was that additional spelling variants will have been derived in subsequent iterations over the same query log.

The geocoding system was set up as described and in three iterations a query log has been used to enhance EEAHN. The queries thereby were generated as in Chapter 7. This time, one single query set with queries with zero, one, two, and three spelling variants mixed into it. However, no model was used to enhance the geocoding system. The hypothesis that the approach suits to improve the geocoding system and allows to serve additional queries was confirmed by measuring the precision and recall of the geocoding system after each iteration. Furthermore, as hypothesized, subsequent iterations allowed to observe additional spelling variants and further improve the metrics. As intended, the error rate introduced by the geocoding system used to observe spelling variants was low enough to allow the continuous improvement of the geocoding system.

11.2 Outlook

As common in science, understanding more leads to more questions. Several ideas used in this thesis should be further investigated.

Starting at the end, the method for continuous improvement of a geocoding system can be further improved. Specifically, the results showed that precision and recall grow with each additional iteration. However, even after three iterations, the recall of that geocoding system was below the recall of EAHN allowing a spelling variant of one. In part, that is due to the result-filter that has the purpose to reduce the incorrect results to be used to observe enhancements. The filter used in this thesis relied on two simple thresholds that were derived by hand. The TFIDF score of the top result had to be large enough and sufficiently
larger than the score of the second result. These two scores can be derived in a more thorough, data-driven fashion. They also might vary depending on the region or the language of the result. Further scores might be introduced to better tune the result-filter to let through more correct results thereby increasing the recall. Also, the recall can be increased by introducing standardization techniques into the geocoding system. EEAHN was handling abbreviations because these were generated by the statistical model after being observed in the user query log provided by HERE Technologies. The continuously improving geocoding system only relies on edit distances to match query tokens to address tokens. Especially abbreviations are oftentimes more edits apart and are not picked up this way. Introducing standardization techniques as, for example, they are present in Nominatim could further improve the metrics.

Various other algorithms can be finer tuned too. A modified Levenshtein Algorithm was used as the measure of similarity when aligning query and result tokens into token pairs. Different modifications or a completely different string similarity measure could yield better results. Also, more modern approaches to detect spelling variants such as character embedding could work well here. Recent findings in the field of artificial intelligence enable learning similar vectors for words with similar meaning from a large corpus of text. Similarly, from a corpus of logs, address element names that are spelled differently, but mean one and the same thing can be learned.

One specifically interesting aspect of learning spelling variants is the possibility to work around faults of various interfaces. Clearly, different spelling variants are used when specifying the query through a common keyboard, a virtual keyboard of a mobile device, a voice interface. It is fair to assume that all kind of spelling variants can be learned and handled with the proposed approaches. However, a thorough investigation of the various interfaces and how to approach them best would be interesting.

Another area leaving space for further research is the address format used for indexing. In this thesis, house numbers were treated in a special way as fuzzy text search should not apply to them. That, however, could be the case for postal area codes too. Investigating further relations between the elements of an address could lead to address models that are better suited for indexing.

The propagation of observed spelling variants can be improved too. Currently, the continuously improving geocoding system only indexes spelling variants next to the address that has been queried for using them. However, if the spelling variant applies to a city name, it seems to make sense to enhance all addresses in that city. An address format that persists the hierarchy and is suitable for geocoding can also help in doing so. Without such an address format, there are two obvious options. One way to propagate spelling variant of a city name to all addresses in that city is to keep the address hierarchy in a separate model that is not used for geocoding. Relying on that, all addresses in that city could be identified and enhanced. Another approach could be to enhance all addresses with that token, independent on whether this is the same city or maybe a different address element type that happens to have the same spelling. That would, for instance, propagate a spelling variant of the token Paris to the capital.
of France as much as to the capital of Texas. Investigating whether the same spelling variants are used for both cities and identifying the better approach is an interesting topic for further questions.

The time factor was also only partially in the scope of this thesis. An experiment was conducted showing that the spelling variants that humans use do not change within the span of a single year. However, it is fair to assume that spelling variants used do change over longer time spans. The process that continuously improves the EEAHN geocoding system does not have an overview over time. Currently, it simply accepts every spelling variant that is observed. This way, by accident, very rare spelling variants can be used as enhancement even though they provide little to no value. Thus, keeping track of how often spelling variants that are used to enhance addresses are observed could be useful. Rare spelling variants could be discarded this way. Common spelling variants that are no longer used could be removed from the address too. That would increase the precision of the geocoding system and, likely, increase the run-time performance as fewer spelling variants would be part of the index.

Lastly, the run-time performance of the geocoding systems was not investigated in this thesis at all. At its core, Elasticsearch uses Lucene, an inverted index from token to documents that contain that token. The tokens are stored in a data structure that allows looking up entries in logarithmic time. Thus, enhancing documents with spelling variants increases the number of entries in the inverted index and slows down the search for viable candidates. Searching for candidates allowing an edit distance is not impacted by that. The spelling variants are not part of the index. Instead, a finite automaton is used to search through the inverted index. It is not obvious what requires more computational effort: the search through a larger set of tokens, or the search with search terms expanded through finite automata. Also, likely, an automaton powered by a statistical model could bring increased precision and recall while reducing the overhead through indexing precomputed spelling variants.
A Implementation of a modified, extensible Levenshtein algorithm.

The inner classes specify various supported edits. Each possible edit is defined in a Step class. Given a matrix of computed paths, a Step class verifies if it can be applied, compute a value for the edit distance given it is part of the path and persists the specific edit it models. The best path of taken edits is persisted in the instance of the Levenshtein class. Initialized with two tokens, the instance provides the similarity score and the necessary edits this way.
class StepSkipTake(object):
    def __init__(self, matrix, avord, bword, a, b):
        self.can_apply = a >= 0
        if self.can_apply:
            self.b = None
            self.prev = Levenshtein._get_step(matrix, avord, bword, a - 1, b)
            if type(self.prev) == type(self):
                self.cost = self.prev.cost + .25
                self.a = self.prev.a + avord[a]
                self.prev = self.prev.prev
            else:
                self.cost = self.prev.cost + 1
                self.a = avord[a]

    def __repr__(self):
        return '{'.format(self.a.encode('utf8'))

    def __eq__(self, other):
        return type(self) == type(other) and (self.a, self.b) == (other.a, other.b)

    def __ne__(self, other):
        return type(self) != type(other) or (self.a, self.b) != (other.a, other.b)

class StepTakeMatch(object):
    def __init__(self, matrix, avord, bword, a, b):
        self.can_apply = a >= 0 and b >= 0
        self.can_apply &= avord[a] == bword[b]
        if self.can_apply:
            self.prev = Levenshtein._get_step(matrix, avord, bword, a - 1, b - 1)
            self.cost = self.prev.cost
            if type(self.prev) == type(self):
                self.a = self.prev.a + avord[a]
                self.b = self.prev.b + bword[b]
                self.prev = self.prev.prev
            else:
                self.a = avord[a]
                self.b = bword[b]

    def __repr__(self):
        return '{': '{}'.format(self.a.encode('utf8'), self.b.encode('utf8'))

    def __eq__(self, other):
        return type(self) == type(other) and (self.a, self.b) == (other.a, other.b)

    def __ne__(self, other):
        return type(self) != type(other) or (self.a, self.b) != (other.a, other.b)

class StepTakeMismatch(object):
    def __init__(self, matrix, avord, bword, a, b):
        self.can_apply = a >= 0 and b >= 0
        if self.can_apply:
            self.a = avord[a]
            self.b = bword[b]
            self.prev = Levenshtein._get_step(matrix, avord, bword, a - 1, b - 1)
            self.cost = self.prev.cost + 1
            if type(self.prev) == type(self):
                self.a = self.prev.a + avord.a + self.a
                self.b = self.prev.b + self.b
                self.prev = self.prev.prev
            else:
                self.a = avord[a]
                self.b = bword[b]

    def __repr__(self):
        return '{': '{}'.format(self.a.encode('utf8'), self.b.encode('utf8'))

    def __eq__(self, other):
        return type(self) == type(other) and (self.a, self.b) == (other.a, other.b)
 Appendix A. Implementation of a modified, extensible Levenshtein algorithm.

```python
return type(self) == type(other) and (self.a, self.b) == (other.a, other.b)

def __ne__(self, other):
    return type(self) != type(other) or (self.a, self.b) != (other.a, other.b)

class StepSwap(object):
def __init__(self, matrix, aword, bword, a, b):
    self.can_apply = a >= 1 and b >= 1 and \
        aword[a-1] == bword[b] and \
        aword[a] == bword[b-1]
    if self.can_apply:
        self.a = aword[a-1:a+1]
        self.b = bword[b-1:b+1]
    self.prev = Levenshtein._get_step(matrix, aword, bword, a - 2, b - 2)
    self.cost = self.prev.cost + 1

def __repr__(self):
    return '{},{}'.format(self.a.encode('utf8'), self.b.encode('utf8'))

def __eq__(self, other):
    return type(self) == type(other) and (self.a, self.b) == (other.a, other.b)

def __ne__(self, other):
    return type(self) != type(other) or (self.a, self.b) != (other.a, other.b)

@staticmethod
def _get_step(matrix, aword, bword, a, b):
    assert a >= -1 and b >= -1, 'indexes requested are far negative:' \
        '{},{}'.format(a, b)
    if a == -1 and b == -1:
        return Levenshtein.StepStart(matrix, aword, bword, a, b)
    elif a == -1 and b >= 0:
        return Levenshtein.StepSkipTakeB(matrix, aword, bword, a, b)
    elif a >= 0 and b == -1:
        return Levenshtein.StepSkipTakeA(matrix, aword, bword, a, b)
    else:
        return matrix[a][b]

@staticmethod
def _best_step(matrix, aword, bword, a, b):
    steps = (Levenshtein.StepSkipTakeB, \
        Levenshtein.StepSkipTakeA, \
        Levenshtein.StepTakeMismatch, \
        Levenshtein.StepSwap, \
        Levenshtein.StepTakeMatch)
    steps = map(lambda s: s(matrix, aword, bword, a, b), steps)
    steps = filter(lambda s: s.can_apply, steps)
    steps = min(steps, key=lambda s: s.cost)
    return step

def __init__(self, aword, bword):
    if type(aword) is not unicode:
        aword = aword.decode('utf8')
    self.aword = aword
    if type(bword) is not unicode:
        bword = bword.decode('utf8')
    self.bword = bword

matrix = list()
for a in range(len(self.aword)):
    matrix.append(list())
for b in range(len(self.bword)):
    matrix[a].append(Levenshtein._best_step(matrix, aword, bword, a, b))
self.steps = list([matrix[-1][-1]])
```
while type(self.steps[-1].prev) != Levenshtein.StepStart:
    self.steps.append(self.steps[-1].prev)
self.steps = list(reversed(self.steps))

difference = float(self.steps[-1].cost) / max(len(self.aword), len(self.bword))
similarity = 1 - difference

def __repr__(self):
    steps = ' '.join(map(str, self.steps))
    steps += ' (:%.0f%% similarity)' % format(self.similarity * 100)
    return steps

def __eq__(self, other):
    return type(other) == Levenshtein and \
    other.steps == self.steps

def __ne__(self, other):
    return type(other) != Levenshtein or \
    other.steps != self.steps

def __gt__(self, other):
    if type(other) != Levenshtein:
        return False
    elif other.similarity != self.similarity:
        return other.similarity < self.similarity
    else:
        otherlen = len(other.aword) + len(other.bword)
        selflen = len(self.aword) + len(self.bword)
        return otherlen < selflen

def __lt__(self, other):
    if type(other) != Levenshtein:
        return False
    elif other.similarity != self.similarity:
        return other.similarity > self.similarity
    else:
        otherlen = len(other.aword) + len(other.bword)
        selflen = len(self.aword) + len(self.bword)
        return otherlen > selflen

def __ge__(self, other):
    return self.__eq__(other) or self.__gt__(other)

def __le__(self, other):
    return self.__eq__(other) or self.__lt__(other)
B Implementation of a Token Assignment algorithm.

The inner class implements the Hungarian algorithm. The TokenMatch class takes care of four things: First, it arranges the token lists so that the first list of tokens atokens is not longer than the second list of tokens btokens. Next, it creates instances of the Levenshtein class for every token in atokens to every token in btokens. Then, it creates an instance of the HungarianAlgorithm class which, in turn, computes the token assignments. Finally, it splits all assignments that have a similarity value below the specified threshold. The assigned tokens are stored in tuples containing both token values as well as their computed Levenshtein instance.

```python
class TokenMatch(list):
    
class HungarianAlgorithm(object):

    def __repr__(self):
        return '\n'.join(map(str, self.assignments)) + '\n'

    def __init__(self, costs):
        assert len(costs) <= len(costs[0]), 'Cost matrix needs to be quadratic, or wider than higher! Consider flipping it.'

        self.assignments = list()

        is_to_skip = set()
        js_to_skip = set()

        def reduce():
            i_mins = dict()
            for i in filter(lambda i: i not in is_to_skip, range(len(costs))):
                for j in filter(lambda j: j not in js_to_skip, range(len(costs[i]))):
                    i_mins[i] = min(costs[i][j], i_mins.get(i, object()))

            for i in filter(lambda i: i not in is_to_skip, range(len(costs))):
                for j in filter(lambda j: j not in js_to_skip, range(len(costs[i]))):
                    costs[i][j] -= i_mins[i]

        def assign():
            def _take(i, j):
                is_to_skip.add(i)
                js_to_skip.add(j)
                self.assignments.append((i, j))

            i_to_j = dict()
            j_to_i = dict()
            i_sum = dict()
            j_sum = dict()

            for i in filter(lambda i: i not in is_to_skip, range(len(costs))):
                for j in filter(lambda j: j not in js_to_skip, range(len(costs[i]))):
                    if costs[i][j] == 0:
                        i_to_j.setdefault(i, list()).append(j)
```
j_to_is.setdefault(j, list()).append(i)
i_sum[i] = costs[i][j] + i_sum.get(i, 0)
j_sum[j] = costs[i][j] + j_sum.get(j, 0)

for i in j_to_is:
    if len(j_to_is[i]) == 1 and len(j_to_is[j_to_is[i][0]]) == 1:
        _take(i, j_to_is[i][0])
        break
    elif len(j_to_is[i]) > 1:
        j = max(filter(lambda j: j not in j_to_is[i], j_to_is[i]),
                key=lambda j: j_sum[j])
        _take(i, j)
        break
    elif len(j_to_is[i]) == 1:
        j = j_to_is[i][0]
        i = max(filter(lambda i: i not in i_to_is[j], i_to_is[i]),
                 key=lambda i: i_sum[i])
        _take(i, j)
        break

# reduce and assign as long as necessary
while len(is_to_skip) < len(costs):
    reduce()
    assign()

# assign the remains to nothing
for j, _ in enumerate(costs[0]):
    if j not in j_to_skip:
        self.assignments.append((None, j))

def __init__(self, atokens, btokens, min_similarity=.67):
    flip = len(atokens) > len(btokens)

    atokens, btokens = btokens, atokens

    rows = list()
    for atoken in atokens:
        row = list()
        for btoken in btokens:
            if flip:
                row.append(Levenshtein(btoken, atoken))
            else:
                row.append(Levenshtein(atoken, btoken))
        rows.append(row)

    hungarianAlgorithm = TokenMatch.HungarianAlgorithm(
        lambda row: map(lambda levenshtein: 1 - levenshtein.similarity, row),
                    rows)
    matches = list()

    for a_index, b_index in hungarianAlgorithm.assignments:
        if a_index is not None and b_index is not None:
            levenshtein = rows[a_index][b_index]
            if levenshtein.similarity > min_similarity:
                if flip:
                    matches.append((btokens[b_index], atokens[a_index], levenshtein))
                else:
                    matches.append((atokens[a_index], btokens[b_index], levenshtein))
            else:
                if flip:
                    matches.append((btokens[b_index], atokens[a_index], levenshtein))
                else:
                    matches.append((atokens[a_index], btokens[b_index], levenshtein))
        elif b_index is not None:
            if flip:
                matches.append((btokens[b_index], atokens[a_index], None))
            else:
                matches.append((atokens[a_index], btokens[b_index], None))
        elif a_index is not None:
            if flip:
                matches.append((btokens[b_index], None, None))
            else:
                matches.append((None, btokens[b_index], None))
            matches.append((atokens[a_index], btokens[b_index], None))

    self.assignments = matches
m = None

if a_index is None:
    matches.append((atokens[a_index], None, None))
else:
    matches.append((None, atokens[a_index], None))
else:
    raise Exception('something went terribly wrong for matching' + 
    '\n{} and{} !'.format(atokens, btokens))

super_matches.append((btokens[b_index], None, None))

if a_index is not None:
    if flip:
        matches.append((atokens[a_index], None, None))
    else:
        matches.append((None, atokens[a_index], None))
Bibliography


