TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Department für Geodäsie und Geoinformation
Forschungsgruppe Geoinformation E120-2
Gußhausstraße 27-29
1040 Wien
Tel.: +43 (1) 58801 - 12711, Fax: +43 (1) 58801 – 12799

Diplomarbeit

Assessing the Effect of Currentness of Spatial Data on the Quality of Routing

(Beurteilung des Einflusses der Aktualität räumlicher Daten auf die Qualität des Routings)

verfasst von:
Martin Schmidl, 01425323
Masterstudium Geodäsie und Geoinformation
Technische Universität Wien

Betreuer: Privatdoz. Dipl.-Ing. Dr.techn. Gerhard Navratil

Wien, April 2021

Unterschrift Verfasser

Unterschrift Betreuer

Affidavit

I hereby declare that I am the sole author of this thesis. No assistance other than that which is permitted has been used. Ideas and quotes taken directly or indirectly from other sources are identified as such.

This written work is based on the paper by Schmidl et al. (2021) but has not been submitted in the same form to any other examiners as a form of examination.

Eidesstattliche Erklärung zur Diplomarbeit

Hiermit erkläre ich, dass die vorliegende Diplomarbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt wurde. Ich habe außerdem die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht.

Diese Arbeit basiert auf dem Paper von Schmidl et al. (2021) wurde aber in gleicher Form bei keiner anderen prüfungsrelevanten Stelle eingereicht.

Wien, April 2021

_____

Martin SCHMIDL

Acknowledgements

I would like to express my gratitude to my advisor, Privatdoz. Dipl.-Ing. Dr.techn. Gerhard Navratil for the support I received regarding the topic of this thesis and his advice in general, which helped me immensely in the compilation of this thesis. I also want to sincerely thank my father Peter and his wife Elisabeth for going out of their way for me to follow my path of studying at university. Finally, I want to thank my partner Kathrin for her everlasting support and understanding for my studies.

Danksagung

Ich möchte meinem Betreuer, Privatdoz. Dipl.-Ing. Dr.techn. Gerhard Navratil, meinen ausdrücklichen Dank für die Unterstützung auf diesem Gebiet und für die zahlreichen Ratschläge aussprechen. Beides hat mir bei der Verfassung und Durchführung dieser Arbeit immens geholfen. Auch möchte ich meinem Vater Peter und seiner Frau Elisabeth danken, die alles in ihrer Macht stehende getan haben, um mir mein Studium zu ermöglichen. Zuletzt danke ich meiner Lebensgefährtin Kathrin zutiefst für ihre unaufhörliche Unterstützung und ihr Verständnis für mein Studium.

Abstract

When making spatial decisions, the quality of the underlying data plays an important role. Especially in navigation these data are necessary to route the user to a desired location (usually going by the shortest or fastest route). Road networks are prone to changes, which are represented in the data and those changes might have an impact on the computed route. By using outdated street network data, these changes might not have been accounted for and thus lead to an extension travel time or, even worse, a route to the destination that is not legally allowed any more.

This thesis focuses on the mentioned temporal change. The freely available, route-able graph from OpenStreetMap can be downloaded with different timestamps. On each of these datasets the fastest route between a two randomly chosen points can be computed. Each of those fastest routes are reconstructed on the most recent dataset, where as it is also checked, if the same route is even possible. If that isn't the case, a new route from the incident point to the destination is computed.

This is done for 500 routes in Vienna, Austria. With this, the travel times can be compared on the most recent dataset and the extension of the travel time can be quantified. Ultimately, a first assessment of temporal quality based on the currentness of a dataset is given.

Kurzfassung

Sollen räumliche Entscheidungen getroffen werden, spielt die Qualität der zugrundeliegenden Daten eine wesentliche Rolle. Vor allem in der Navigation sind diese Daten von großer Bedeutung um ein vom Nutzer gegebenes Ziel zu finden (und das üblicherweise nach der kürzesten oder schnellsten Route). Straßennetzwerke sind ständigen Veränderungen ausgesetzt, was auch in den zugrundeliegenden Daten erfasst wird. Mit der Verwendung von veralteten Straßennetzwerkdaten gehen werden einige dieser Änderungen nicht erfasst und können dadurch zu einer Verlängerung in der Fahrzeit oder sogar zu einer nicht mehr fahrbaren Route führen.

Diese Diplomarbeit beschäftigt sich mit der Beobachtung dieser zeitlichen Veränderung. Der frei verfügbare Straßengraph der OpenStreetMap kann in verschieden alten Datensätzen heruntergeladen werden. Auf jedem dieser Datensätze kann dann die schnellste Route zwischen einem zufällig gewählten Start- und Endpunkt berechnet werden. Es wird versucht, jede dieser Routen auf dem aktuellsten Datensatz nachzufahren. Dabei wird auch ein Augenmerk darauf gelegt, ob diese Route überhaupt fahrbar ist. Ist dies nicht der Fall, wird ausgehend vom Punkt, an dem ein Problem vorliegt, eine neue Route zum Zielpunkt berechnet.

Dies wird für 500 Routen in Wien durchgeführt. Damit kann man die Fahrzeiten im aktuellsten Datensatz miteinander vergleichen und die Verschlechterung der Entscheidung quantifizieren. Zuletzt wird eine erste Beurteilung der zeitlichen Qualität aufgrund der Aktualität eines Datensatzes gegeben.

Table of contents

1 Introduction

In navigation, but especially in the vehicular navigation, the usage and provision of up-to-date street network data is inevitable. Changes in the street network lead to changes in the data, whereas an old dataset might not contain these changes. This can result in incorrect or, even worse, illegal manoeuvres and therefore frustrate the user. In particular, that could be a route going the wrong way through a one-way street, which has been changed in reality but not in the outdated dataset. The user then arrives there and finds out that the suggested route can not be followed and therefore has to take a detour, which results in an extension of travel time. This problem is not only limited to the private sector, it may also apply to business cases (e.g., an adoption of the travelling salesman problem on delivery services).

Navigation services originally used commercial data, which resulted in high costs due to the acquisition and update processes. The introduction of map-based Volunteered Geographic Information (VGI) (Goodchild, 2007) services like OpenStreetMap (OSM) in 2004 made it possible to share spatial information with everyone for free. As OSM data is published under a free license (Open Database License – OdbL) it is possible to use this data for navigation purposes, both privately and for commercial applications and therefore offer a great opportunity to eliminate the cost factor.

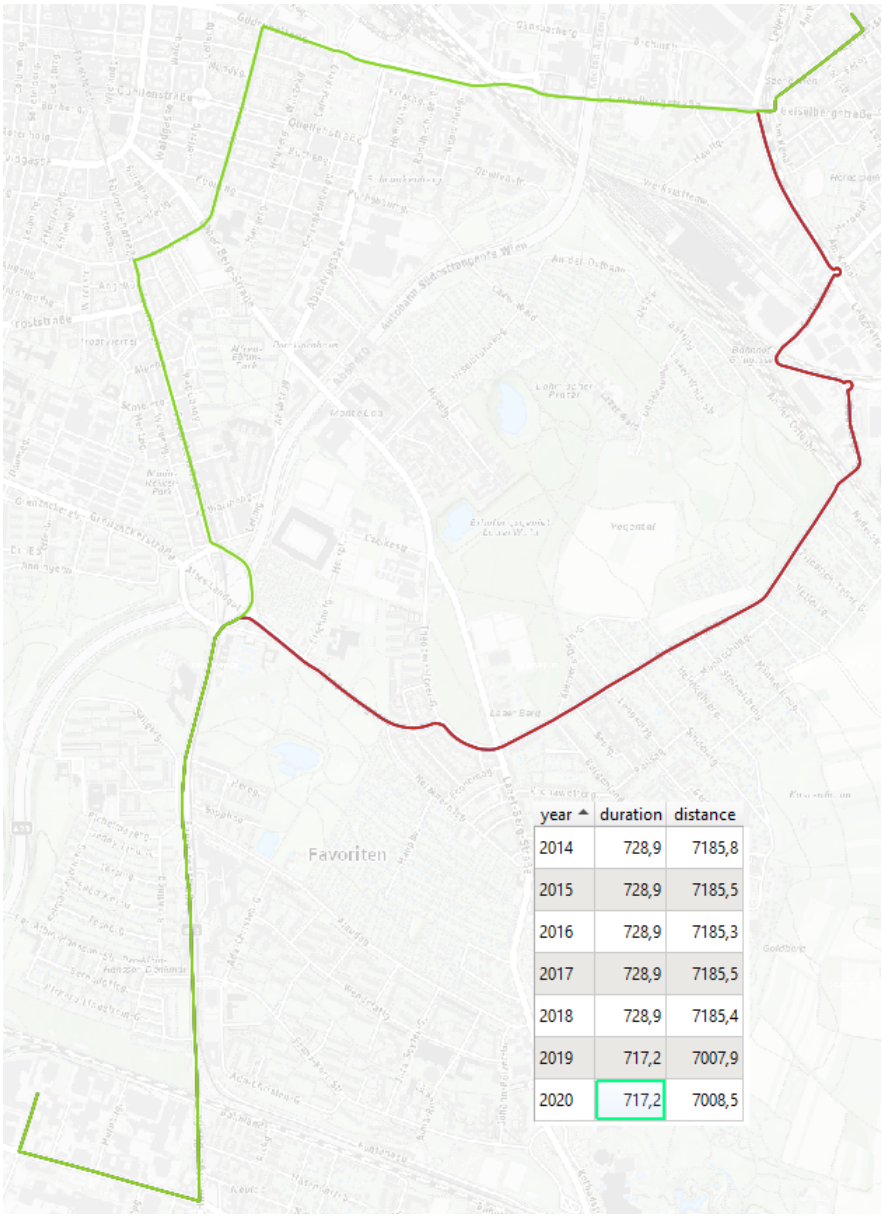
As technical hurdles, such as the import of data into an application or merging different datasets can be easily rectified, an important question can be raised: *Is it possible to quantify the effect of outdated road network data on routing quality?* To answer this question, it is important to know the degradation of routing results over time, caused by missing data updates. As Geofabrik offers annual snapshots of OSM data starting from 2014, the approach described in this work uses these snapshots and compares them against each other in order to trying assess the effect of outdated datasets on the routing.

Computing the fastest routes between two random point pairs on seven datasets from 2014 to 2020 might show variations as the underlying data (e.g., turn-restrictions, access restrictions or speed limits) change over the years. The routes derived from the 2014 to 2019 datasets can be mapped on the 2020 dataset, which simulates an user being routed on an outdated dataset. In the best case, the route will always be the same and therefore the travel time will not change. In other cases, the fastest route might change and the solution based on the older dataset might be slower due to changes in speed limits or various restrictions, or even rendering a route not feasible at all. By computing 500 routes with random start and end points (resulting in a total of 3500 routes in seven years) for the city of Vienna, this provides a first quantitative assessment of the deterioration of the routing decisions based on the age of the dataset.

Whenever spatial data is used, there should be a concern about the quality of the used data. The chapter “Data quality” explains assessment methods in general, but also for VGI in particular and which elements are relevant for the analysis. In the “Data and used software” chapter the used data and software packages are described, which includes OpenStreetMap, Project OSRM (Open Source Routing Machine) and Python scripts.

The analysis of the routes and its parameters over the years as well as the workflow and setup is described thoroughly in the “Implementation” chapter, as is the process for finding and removing a route that can not be used. Finally, the “Results” chapter focuses on giving a first assessment of the temporal change depending on the currentness of a dataset is given and explains reasons for possible route changes over the years.

An example for the extension of travel time of a single route is shown in Figure 1. The red route is the fastest in the years 2014 to 2018. The green route defines the fastest route in the 2019 and 2020 datasets respectively. If one reconstructs the fastest routes from 2014 to 2019 on the 2020 dataset, the extension of travel time is visible when using older datasets from 2014 to 2018.



year	duration	distance
2014	728,9	7185,8
2015	728,9	7185,5
2016	728,9	7185,3
2017	728,9	7185,5
2018	728,9	7185,4
2019	717,2	7007,9
2020	717,2	7008,5

Figure 1: Example for the analysis of fastest routes

2 Data quality

The introduction of the internet provided an easy method for communicating and the exchange of data. Thanks to open-data initiatives and VGI platforms, this is also valid for spatial data and over the time, a vast amount of data is now available (mostly for free). Information about the quality of this data is connected to its usage in several ways: The quality needs to be described first with parameters describing specific aspects. While the parameters need to be quantified for each dataset, the effect on the use also needs to be addressed.

2.1 Elements of data quality

The ISO (International Organization for Standardization[1]) has created a standard for the quality of spatial data (ISO 19157, 2013) which describes the standard for quality of geographic information based on work by the International Cartographic Association (Guptill and Morrison, 1995). It also aims to define parameters for the assessment of quality of a dataset. A recommended work-flow for this assessment is also given. Quality parameters should be described via Metadata, which are defined in another ISO standard (ISO 19115, 2014). In the mentioned ISO 19157 standard, every element of data quality describes a different aspect of the quality of the underlying data, as seen in Figure 2.

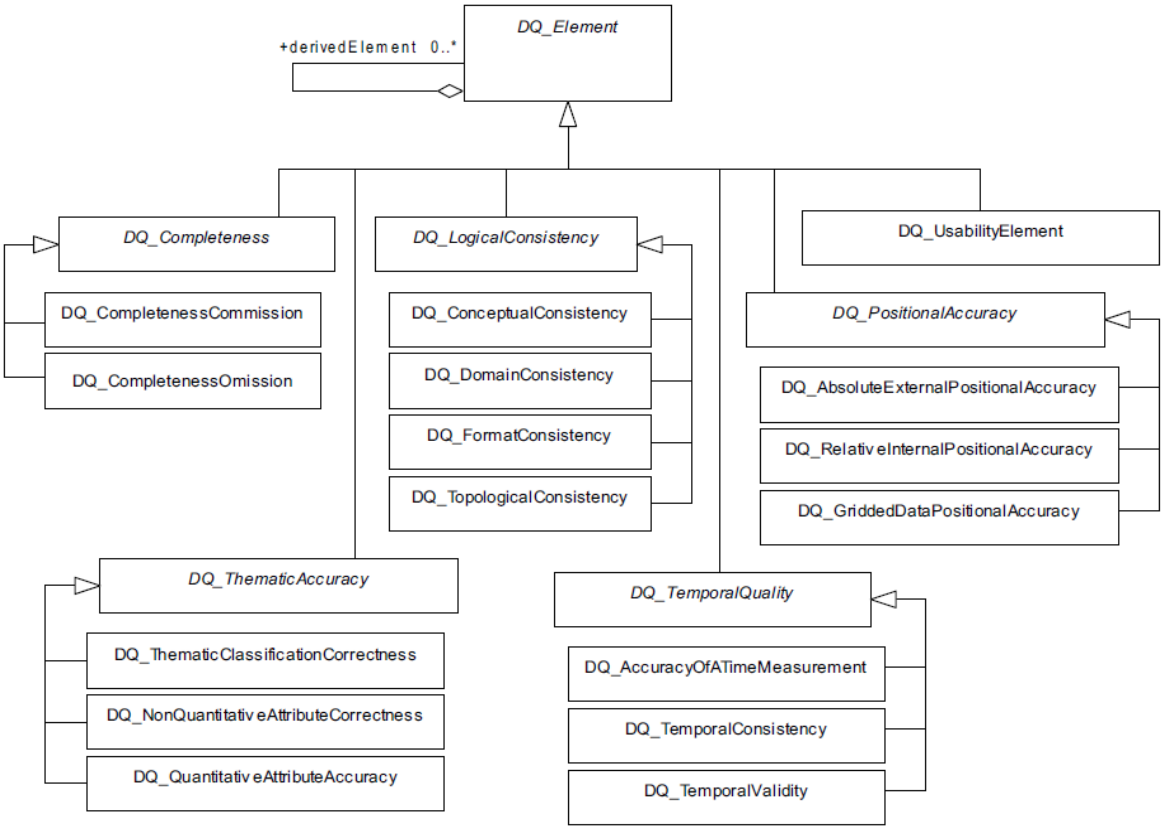


Figure 2: Overview of the data quality elements (ISO 19157, 2013)

1 https://www.iso.org/about-us.html

- **Completeness**

  Completeness refers to the amount of missing data (omission) or excess data (commission) of objects in a dataset comparing to the real world. An omission happens, for example, when a motorway was just constructed but is still missing from a road network dataset. A good example for commission would be a dataset of the road network of Vienna also containing road network data from the city of Graz, which in this case would unnecessarily result in a larger size of the dataset.

- **Logical consistency**

  Datasets can be checked for logical consistency by defining simple rules that should always be valid. An example might be, that the only road that leads to a summit is a one-way road only. The standard defines four different types of logical consistency (ISO 19157, 2013):

  - Conceptual consistency
    Adherence to the rules of the conceptual schema
  - Domain consistency
    Adherence of values to the value domains
  - Format consistency
    Degree to which data are stored in accordance with the physical structure of the dataset
  - Topological consistency
    Correctness of the explicitly encoded topological characteristics of a dataset

- **Positional accuracy**

  This element is defined as the accuracy of the position of features contained in a dataset within a spatial reference systems. It can be split into three parts (ISO 19157, 2013):

  - Absolute or external accuracy
    Closeness of reported coordinate values to values accepted as or being true
  - Relative or internal accuracy
    Closeness of the relative positions of features in a dataset to their respective relative positions accepted as or being true
  - Gridded data position accuracy
    Closeness of gridded data spatial position values to values accepted as or being true

Usually, the positional accuracy is defined through the accuracy of the devices and the methods used while gathering the data. That includes methods for the assessment of the accuracy like the propagation of error, RMSE (Root Mean Square Error), ellipsis of confidence, amongst others.

- **Thematic accuracy**

The element of the thematic accuracy is also split into three different aspects and is defined as the accuracy of quantitative and correctness of non-quantitative attributes (ISO 19157, 2013):

- Classification correctness
  Comparison of the classes assigned to features or their attributes to a universe of discourse (e.g. ground truth or reference data)
- Non-quantitative attribute correctness
  Measure of whether a non-quantitative attribute is correct or incorrect
- Quantitative attribute accuracy
  Closeness of the value of a quantitative attribute to a value accepted as or known to be true

- **Temporal quality**

Temporal quality refers to the quality of attributes and relationships, which have one or more temporal characteristics. In this case, time can be defined as a single point in time, but also as a period of time. It consists of three aspects (ISO 19157, 2013):

- Accuracy of a time measurement
  Closeness of reported time measurements to values accepted as or known to be true
- Temporal consistency
  Correctness of the order of events
- Temporal validity
  Validity of data with respect to time

For example it matters, how often a dataset is getting updated (currentness of dataset). Important changes should always be entered as soon as possible to keep the dataset up-to-date, if that is needed for the application to deliver accurate results based on the decisions made.

- **Usability**

  The usability elements are used to describe how fitting a dataset is for a certain purpose and are used independently of the quality elements mentioned so far. This fact can also be described as “fitness-for-use” or “fitness-for-purpose” (Chrisman, 1984).

  As a possible user defines the usability, it is the only element from the external quality category. All t he other mentioned quality elements can be grouped into the category of internal quality, as they are defined and assessed by the creator (or editor) of the data.

However, these concepts can be more difficult to assess than statistical parameters, as elements may use different approaches for the assessment of quality. Boin & Hunter (2007) tried to investigate the missing link between the quality of spatial data and the customers by asking them a set of questions about what they expect from the data and started inducing theories by the findings. They found out, that customers were more interested in finding out the contents of the data and how it would match with other sources than the internal quality. They therefore suggested to use “fitness-for-use” as a valuable parameter to be included in the description of a dataset. Another suggestion was the provision of other users opinions on using the dataset to serve as a basis for the decision of users with similar applications.

A different approach was used by Yu et al. (2014). They researched the trustworthiness of sensory data by using data from other sensors and looking at the correspondence of both and defined this correspondence as a quality indicator. Fogliaroni et al. (2018) had a look at the contributors themselves: They assessed the trustworthiness of information and the reputation of contributors by creating a model, which analyses their respective edits geometrically, qualitatively, and semantically on a full OpenStreetMap history dump. The data from the VGI dataset were compared with a ground truth dataset provided by authoritative sources. For each feature in the authoritative dataset, the corresponding feature in the VGI dataset was computed and by comparing the similarity of both, a trustworthiness score can be given.

Uncertainty in geographic data can also have an effect on decisions made based on assumptions derived from the data, which should not be neglected. Heuvelink (2002) suggested the usage of the Monte Carlo method to assess this uncertainty. While this method is computationally demanding, it can be used with a wide range of situations. Karssenberg and De Jong (2005) used the Monte Carlo method in a suitable system, while Heuvelink also mentions a number of problems regarding the description of data quality, with one being the spatial and eventually temporal variation of quality. This topic was also addressed by Tsutsumida and Comber (2015).

Another aspect of data quality was used by Krek (2002), where she investigated how incomplete information can affect a way-finding application.

2.2 Quality of Volunteered Geographic Information (VGI)

In the professional sector (e.g., national mapping agencies, public sector bodies etc.) the data collection and quality assessment processes are done following the criteria defined in several ISO standards. This is however is not as easy for Volunteered Geographic Information, as a wide range of collecting methods and no standard for the quality assessment of VGI data exist. Therefore, some proposals to assess the quality of these data have been made in the past in the literature.

Goodchild and Li (2012) propose an approach consisting of three parts in their paper:

- Crowd-sourcing approach
- Social approach
- Geographic approach

The first mentioned crowd-sourcing approach describes the approach to view and check the data by as many users as possible to minimise errors, such as misplaced locations for certain features. The higher number of people looking at the data makes it more likely to discover and correct errors in a reasonable amount of time. Based on the trust in the contributor itself, the social approach is based on ranking the user through his previous submissions and can build up a "trust score". With that, a rating or rank system can be implemented based on how this users contributions can make a service better. The geographic approach adopts logical rules valid for geographical data. This can be compared with the logical consistency element from the ISO 19157 standard. An algorithm could be created that checks all features for adhering to these logical rules, with the ones not adhering to those being automatically filtered out for further adjustments.

Meek et. al (2014) suggest to use a third quality model, which sits in between both the internal (quality elements defined in ISO 19157, except for the Usability) and external quality model (Usability). They call it the "Stakeholder" model, which consists of 6 different elements:

- Vagueness
  Inability to make a clear-cut choice (e.g., lack of classifying capability)
- Ambiguity
  Incompatibility of the choices or descriptions (e.g., lack of understanding)
- Judgement
  Accuracy of choice/decision in a relation to something known to be true (e.g., perception)
- Reliability
  Consistency in choices/decisions (e.g., testing against itself)
- Validity
  Coherence with other peoples choices (e.g., against other knowledge)

- Trust
  Confidence accumulated over other criterion concerning data captured previously (linked to Reliability, Validity and Reputability)

The methods proposed by Goodchild and Li (2012) and Meek et al. (2014) are not the only ones however. An thorough overview of methods to assess the quality of spatial data from VGI has been provided by Senaratne et al. (2017). They distinguished between literature dealing with map-based VGI, image-based VGI and text-based VGI, but also propose their own approach for map-based VGI, which is split into three parts:

- Quality measures
- Quality indicators
- Data mining

The quality measures part mostly resembles the internal quality aspects known from the ISO 19157 standard and can be split into seven parts:

- Positional accuracy
- Thematic accuracy
- Topological consistency
- Completeness
- Temporal accuracy
- Geometric accuracy
- Semantic accuracy

Quality indicators are defined by the external quality aspects (i.e., Usability). For this part of the approach, Senaratne et al. (2017) define ten of these quality indicators in their paper:

- Lineage
- Usage
- Credibility
- Trust worthiness
- Content quality
- Vagueness
- Local knowledge
- Experience
- Recognition
- Reputation

The additionally recommended approach of data mining consists of using Machine Learning and artificial intelligence (A.I.) to recognise geographical patterns and rules for an additional assessment of quality. This might detect patterns or rules that might not be noticed by a human or algorithm right away. The data mining approach can be used completely independent from the other quality aspects mentioned previously, if desired.

2.3 Relevant elements for this thesis

As the data for this thesis is coming from the OpenStreetMap project, some quality assessment methods suitable are mentioned here. Senaratne et al. (2017) describe a few approaches from the literature for all aspects of data quality. Will (2014) describes some approaches especially for street networks: He used the same type of analysis for assessing the Completeness as Haklay (2010) and Zielstra and Zipf (2010) did before by comparing OSM datasets of a certain area to a reference dataset created by authoritative sources.

Girres and Touya (2010) conducted a similar analysis with data from the French IGN (Institut Geographique National). Noteworthy in their paper is the currentness. To quantify this parameter, the authors looked at the total number of features in the dataset between June and October 2009. In this period, the number of features present in the OpenStreetMap went up quite significantly, as seen in Figure 3. This concurrently results in the improvement of the completeness aspect, highlighting the importance of the currentness of the OSM data extract.

Layer	Objects – june 09	Objects – oct 09	Evolution (objects)	Evolution (%)
France_poi	76653	111440	34787	45,4%
France_highway	674205	886680	212475	31,5%
France_coastline	1201	1200	-1	0%
France_administrative	42286	51413	9127	21,5%
France_water	10531	12507	1976	18,8%
France_natural	20958	24756	3798	18,1%
TOTAL	**825834**	**1087996**	**262162**	**31,7%**

Figure 3: Comparison of the features present in the OpenStreetMap (Girres and Touya, 2010)

Will also analysed the positional accuracy by looking at the absolute accuracy of the features in the OSM dataset, by calculating the average positional error between the OSM dataset and the dataset from authoritative sources. The assessment of thematic accuracy was limited to the analysis of road names by comparing road names from both datasets against each other.

No further work dealing with the data quality based on the currentness of a dataset has been found.

3 Data and used software

For this thesis, open-source software and data are used in the analysis and therefore those are thoroughly described in this chapter. This includes detailed information about how data is acquired and which features of a certain software is used.

3.1 OpenStreetMap (OSM)

OpenStreetMap[2] is an online project to provide a free of charge and freely useable, editable map of the world. Viewing the map is possible without the need to create an account, which is only needed for being able to edit any data on the map. Once an account is created, it is possible to edit any content right away. The project itself was started in 2004 by Steve Coast in the United Kingdom, and since then its user base grew rapidly. As of January 2020, nearly 7.5 million users are registered on the OSM site[3]. The data can be recorded with GPS receivers, smartphones etc. Data can also be digitalised based on satellite or aerial images. Changes and edits can either be done via HTML editors or special tools and software for a broad selection of hard- and software. As all data from the OpenStreetMap project is licensed under the ODbL (Open Database License[4]), it is possible to freely use the data for a wide range of private and professional applications, including geocoding and navigation for example.

Figure 4: OpenStreetMap centered on Vienna, Austria

2 https://www.openstreetmap.org/
3 https://www.openstreetmap.org/stats/data_stats.html
4 https://opendatacommons.org/licenses/odbl/

All the data that can be seen in Figure 4 is stored in a database and rendered individually in the web browser, depending on the current view box and zoom level. In the database itself the data can consist of four different core elements (or data primitives):

- Nodes
  Point features with a geographical coordinate (latitude and longitude). This is for example used at junctions or for points of interest (e.g., petrol stations). At least it has to feature an ID and a pair of coordinates.

- Ways
  A list of two to 2.000 nodes, to represent linear features (e.g., streets or rivers). A way can also be a closed loop to represent the boundary of features like buildings or forests.

- Relations
  This type documents a certain relationship between two or more elements. This can be used for describing bus routes along a street or turn restrictions from one street to another. This is mostly defined further by tags.

- Tags
  Tags describing the meaning of a data element mentioned above. It always contains of 2 text fields, namely a "key" and a "value". There are a lot of attributes to use in mapping, with the most common one being the ID field to properly identify an element in the data.

An example is given in Figure 5 below:

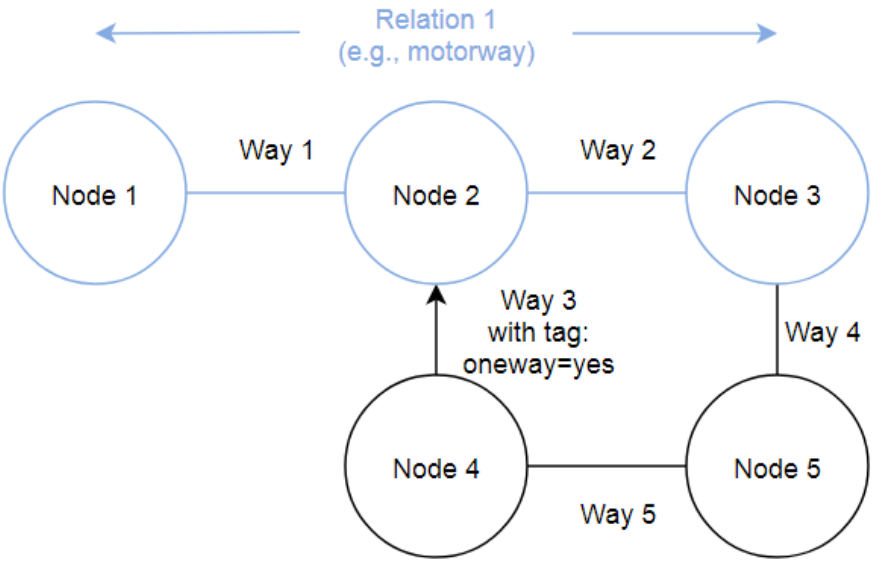


Figure 5: A very basic example for the data model used in OpenStreetMap

All the nodes in Figure 5 would consist of their IDs and their respective coordinates. Each of the ways would be defined as a list of 2 nodes, with Way 3 having a special tag called “oneway=yes”, to define it can be only used in one direction. If a motorway exists from Node 1 to Node 3 via Node 2, this is mapped with a relation. Every node, way or relation can have a variety of tags, as with Way 3 being a one-way street for example.

3.2 OSM snapshots

The OpenStreetMap data primitives are stored in databases, but there is also the possibility to use data offline or locally by downloading OSM dumps (snapshots) from the database. To list an example, planet.osm[5] is a complete, world-wide dump on a weekly basis. For local applications like this, a list of extract sources[6] is available. For this project, Geofabrik dumps for Austria have been used[7], as they are available from 2014 on. This results in 7 different datasets from 2014 to 2020. Statistical data about this dataset is listed in Table 1.

Year	Number of nodes	Number of ways	Number of relations	Total elements
2014	32,394,084	3,212,914	58,738	35,665,736
2015	37,666,139	3,825,089	72,543	41,563,771
2016	45,138,472	4,771,241	85,229	49,994,942
2017	51,618,302	5,598,078	102,405	57,318,785
2018	57,813,697	6,257,263	112,732	64,183,692
2019	62,658,551	6,822,308	121,774	69,602,633
2020	66,428,283	7,246,229	130,725	73,805,237

Table 1: Comparison of data primitives included in the Geofabrik Austria dumps

From Table 1 it is clearly visible, that the amount of data primitives has more than doubled during these seven years. For the full planet.osm dumps and regional extracts, two versions of file formats can be used:

- OSM XML Format (.osm)[8]
  These files are uncompressed XML-files, specially used for the OpenStreetMap data primitives. The advantage is that it is human-readable and useable for nearly any software package, but parsing takes a lot of time and files can be very big (for example well over 1 TB for the whole planet.osm dump). The official full planet.osm dump is packed in a bzip2-file.

5 https://planet.openstreetmap.org/
6 https://wiki.openstreetmap.org/wiki/Planet.osm#Worldwide_extract_sources
7 http://download.geofabrik.de/europe/austria.html
8 https://wiki.openstreetmap.org/wiki/OSM_XML

- Protocolbuffer Binary Format (.pbf)[9]
  The advantage with this compressed format is that it can be a lot smaller than an uncompressed OSM XML file and thus can be handled faster by a software package that supports it. The data is split into different blocks of files, where as every single of these can be decoded independently, eliminating the need to decode a whole file.

3.3 Osmconvert

The downloaded OSM snapshot contains much more data than needed. To save space and computing time, data has to be cut down to a certain extent, in this case from Austria to just Vienna. Osmconvert is a useful command line tool to edit snapshots easily. It is available for download through the OpenStreetMap Wiki[10]. It is mostly used for dropping features from extracts or cutting extracts down to a certain smaller extent. It is run by opening it in the command line and can be customised with a number of parameters.

```
C:\Users\Martin\Downloads>osmconvert64.exe -h

osmconvert 0.8.10  Parameter Overview
(Please use  --help  to get more information.)

<FILE>                        input file name
-                             read from standard input
-b=<x1>,<y1>,<x2>,<y2>        apply a border box
-B=<border_polygon>           apply a border polygon
--complete-ways               do not clip ways at the borders
--complete-multipolygons      do not clip multipolygons at the borders
--complete-boundaries         do not clip boundaries at the borders
--all-to-nodes                convert ways and relations to nodes
--add-bbox-tags               add bbox tags to ways and relations
--add-bboxarea-tags           add tags for estimated bbox areas
--add-bboxweight-tags         add tags for log2 of bbox areas
--add-bboxwidth-tags          add tags for estimated bbox widths
--add-bboxwidthweight-tags    add tags for log2 of bbox widths
--object-type-offset=<id>     offset for ways/relations if --all-to-nodes
--max-objects=<n>             space for --all-to-nodes, 1 obj. = 16 bytes
--drop-broken-refs            delete references to excluded nodes
--drop-author                 delete changeset and user information
--drop-version                same as before, but delete version as well
--drop-nodes                  delete all nodes
--drop-ways                   delete all ways
--drop-relations              delete all relations
```

Figure 6: Preview of the osmconvert tool in the Windows CMD

An overview of useful functions is given below:

- Cropping of input data
  This can be done via the `-b` and `-B` parameters. `-b` uses a simple rectangle bounding box, while `-B` uses .poly files, which define a closed polygon, where data will be kept inside. Those files can be easily created from every relation existing on OSM[11].

9 https://wiki.openstreetmap.org/wiki/PBF_Format
10 https://wiki.openstreetmap.org/wiki/Osmconvert
11 http://polygons.openstreetmap.fr/

- Complete features when cropped
  When cropping data, features might clip the borders (e.g., motorways leading in and out of a city). In some applications it can be useful to keep those features. This can be done with the following commands, depending on the feature type that should be kept:
    `--complete-ways`, to keep ways at the borders
    `--complete-multipolygons`, to keep multipolygons at the borders
    `--complete-boundaries`, to keep boundaries at the borders

- Drop certain features
  One can drop certain data from the extract with the drop commands:
    `--drop-broken-refs`, to drop broken references to nodes after clipping
    `--drop-author`, to drop user name, user id, changeset and timestamp informations
    `--drop-version`, to drop author information and version numbers
    `--drop-nodes`, to drop all nodes
    `--drop-ways`, to drop all ways
    `--drop-relations`, to drop all relations

- Statistical features
  Osmconvert also can give some basic statistic values by just stating the `--out-statistics` parameter. This is used for getting the values for Table 1 for example.

- Output file formats
  With the -o parameter a custom file format can be used for the output. Osmconvert supports following file formats:
    .osm
    .osc
    .osc.gz
    .osh
    .o5m
    .o5c
    .pbf
    .csv

- Miscellaneous other features
    `--all-to-nodes`, to convert ways and relations to nodes
    `--max_objects`, to limit the number of maximum objects from -all-to-nodes
    `--max_refs`, to limit the number of references per object
    `--fake` functions, to insert some dummy information into the data

For the data in this project, following command was used:

```
osmconvert64.exe at_year.pbf -B=Wien.poly -o=wien_year.pbf --complete-ways
```

Here, `at_year.pbf` represents the input file, which is the complete Austria extract downloaded from Geofabrik . `Wien.poly` is the polygon file created from the relation “Vienna” with the ID 109166, so it only keeps data from inside this defined polygon. `wien_year.pbf` is the corresponding file that the output is written in, this will be kept as PBF, as OSRM works with this file format and it saves disk space. `--complete-ways` makes sure, that ways along the border are kept, as they might be important for a correct routing.

3.4 QGIS

QGIS is a freely available software to view and edit all sorts of geospatial data[12]. It supports a very broad range of data types and connections to databases used in the Geographic Information System field.

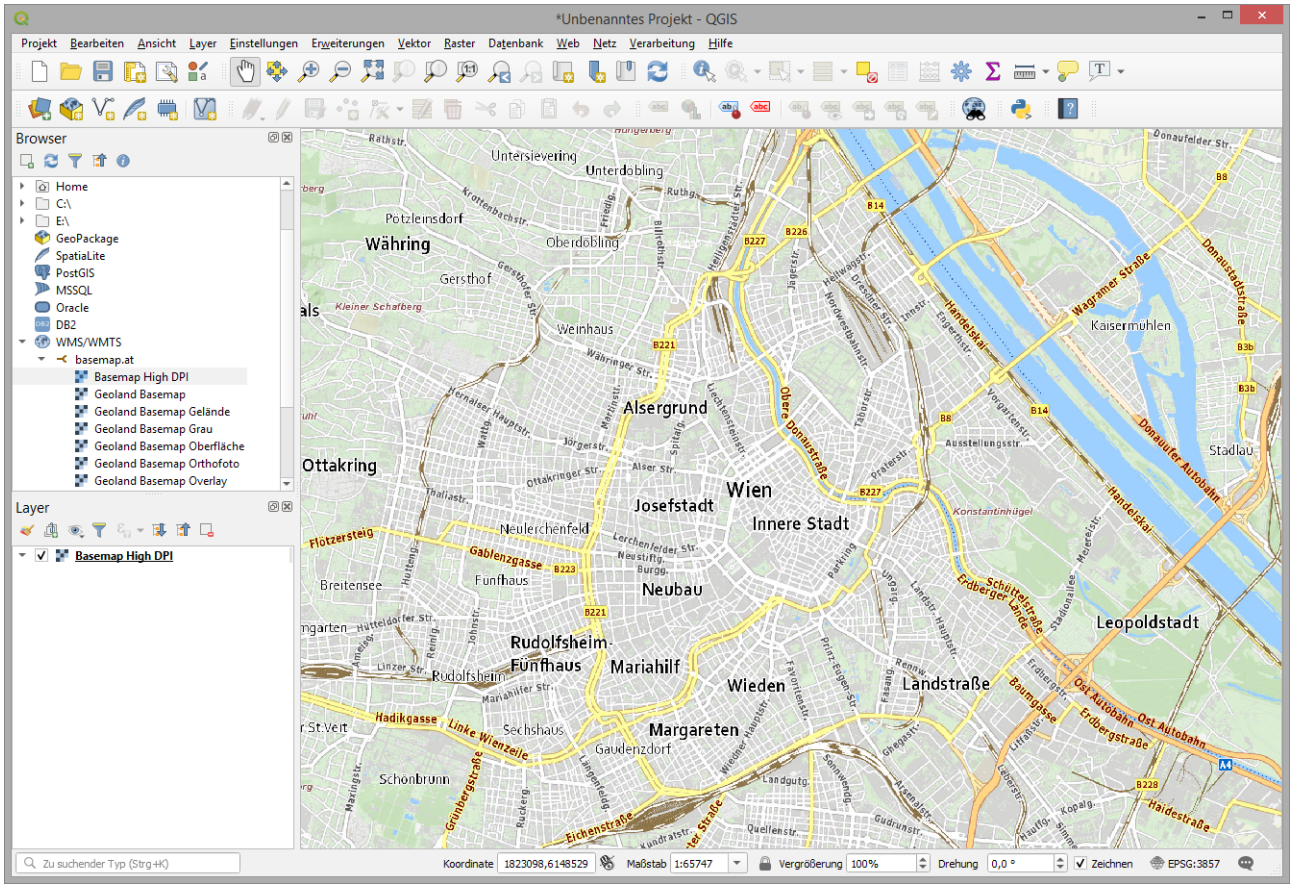


Figure 7: Main screen of QGIS

12 https://www.qgis.org/en/site/

QGIS also supports a vast amount of plug-ins in a separate plug-in manager and access to several coding languages. As seen in Figure 7, the main view consists of the layers, that can consist of map views or data that can be displayed on those map views. For this project, QGIS is used to display the computed routes and to check if those are valid, both by the map view and via the attribute table.

The sheer amount of features and connections QGIS offers would go beyond the scope of this thesis by a lot, therefore only a reference to the extensive QGIS documentation[13] is given here.

3.5 Open Source Routing Machine (OSRM)

The Project Open Source Routing Machine is an engine to compute the fastest route on OpenStreetMap-based datasets. It is written in C++ and supports different routing profiles (e.g., car, bike, foot), based on which the engine factors in turning restrictions, access restrictions and speed limits. This should always give a user the fastest, legal route to arrive at a desired destination, and that even with provided instructions. A demo page is provided[14] or the service can be used via the OpenStreetMap site.

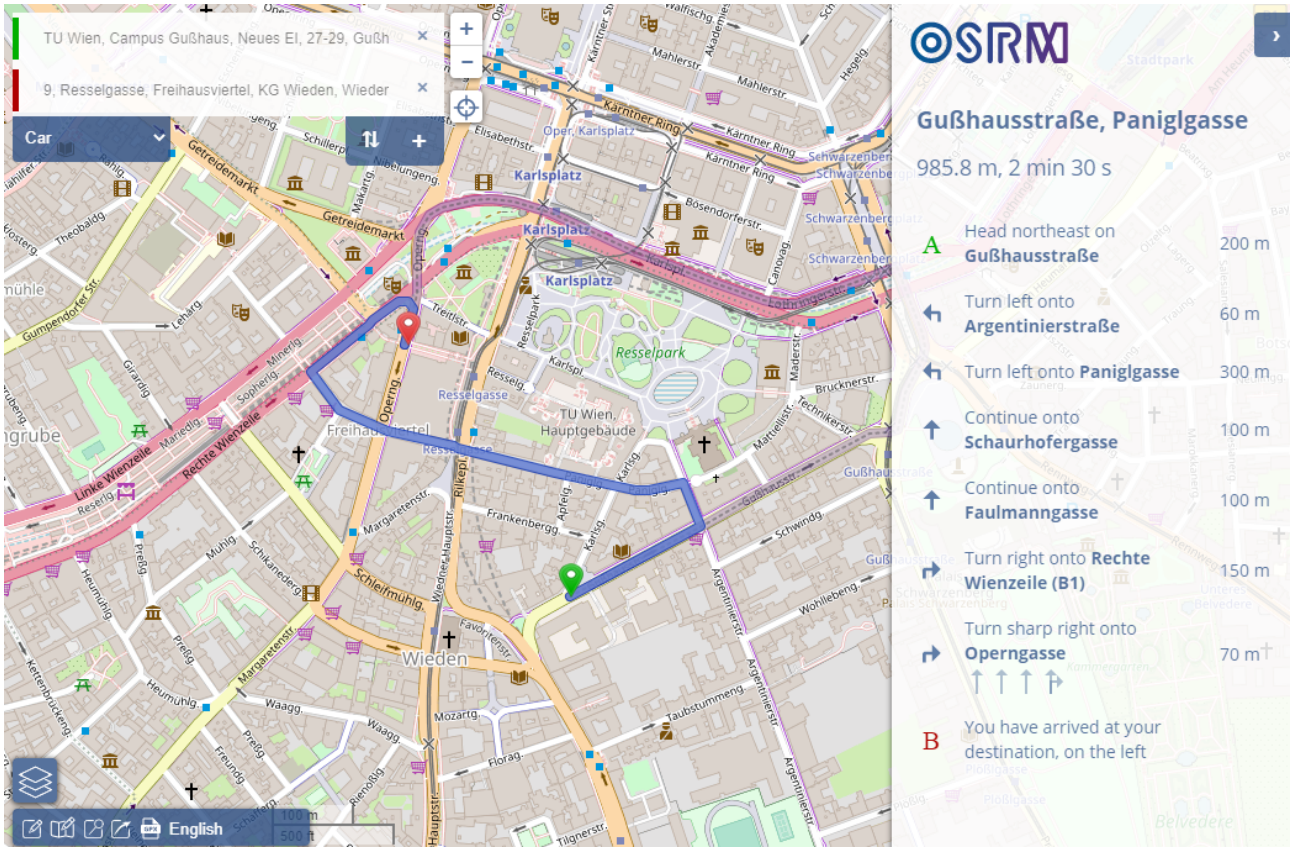


Figure 8: Example route from Project OSRM with on-screen instructions

13 https://www.qgis.org/en/docs/index.html
14 http://map.project-osrm.org/?hl=en

OSRM is a server-application, while an optional front-end can be installed on the same server. For easy installation and setup, both the back-end and the front-end have Docker images[15] to be installed and run from there. Requests to the server are done like browsing a webpage on the internet: A request is sent to the server via different parameters in the URL, like described in the documentation[16] with the parameters involved described below:

```
http://{server URL/address}/{service}/{version}/{profile}/{coordinates}[.{format}]?
option=value&option=value
```

The *server address* or *URL* defines the location that the OSRM server (and in some cases the port) is running on. As OSRM is not only capable to compute the fastest route, it can run a number of different *service*s:

- route – Computes the fastest route between given coordinates
- nearest – Snaps a given coordinate pair to the nearest point on the road network
- table – Computes a table with the fastest route between all coordinates given
- match – Matches a given list of coordinates to the road network in the most likely way
- trip – An approach to solve the Travelling Salesman Problem
- tile – Generates Mapbox Vector tiles for external examination of the road network graph

The *version* parameter is always "v1" for all OSRM versions 5. *Profile* is the routing profile used in the computation (car, bike, foot), which defaults to car in this analysis. The *coordinates* parameter is the main part of any request, it contains the list of coordinate pairs to be included in the computation. The *format* parameter can be neglected, as this only can be json for now and defaults to it. The last parameter are the optional *option* parameters. These differ from service to service. As in this analysis only the route, nearest and match services are used, only these 3 are described in detail below.

- Nearest
  The service to find the nearest point(s) to given coordinates only takes a single coordinate pair per single request. It only can take one extra option in addition to the general ones described before, the *number* parameter. It states how many of the closest segments should be given back.
  Output-wise it will return a *code*, which normally returns Ok when a request was computed successfully. If it was not, an error code will be given. A *waypoint* array will return the desired point(s) and their *location* with the *distance* to the given coordinate.

15 https://hub.docker.com/u/osrm
16 http://project-osrm.org/docs/v5.5.1/api/#general-options

- Route
  The service to find a fastest route between two or more points has several additional parameters. You can define *alternatives*, for alternative routes, *steps*, to return the instructions for every step on the route or *annotations* for additional metadata about the route. The *geometries* parameter decides how the geometry format will look like, while the *overview* parameter decides on how much the route will be simplified.
  As with the nearest service there are a number of output parameters: The *code* parameter shows Ok again when the request was successfully computed, and an error message otherwise. Two arrays are given in the result, *waypoints* and *routes*. While the first one contains waypoints and information about them, the full route and its parameters like duration and distance are part of the routes as attributes. Depending on some input parameters, the response might contain more or less data about the computed routes. For this analysis, full, uncompressed routes in the json format are used.

- Match
  The matching service is similar to the route service, with the original intent to match real-world GNSS data to the road graph. Therefore the parameters are a bit different: While *code*, *steps*, *geometries*, *annotations* and *overview* still exist, *timestamps* and *radiuses* are added for defining a time stamp and an accuracy (not used in this project).
  Regarding output parameters, the only difference to the routing service is that the 2 arrays are named slightly different, namely *tracepoints* and *matchings* while the *code* parameter stays the same. The former contains the points along the route that could be matched. The matchings array contains the matched route(s), including their metadata like distance and duration. The only added parameter is confidence, which gives an indication about how the machine is confident (between 0 and 1) that the matched route is correct. Again, full, uncompressed routes in the json format are used in the analysis.

3.6 Python

Python is a high-level programming language known for its code readability and use of white-spacing. It is used in all sizes of projects and according to the TIOBE index[17], it is the third-most popular programming language in the world (as of February 2021) and has been in the Top 10 of the index since 2003, while it dates back to the 1990s. In its standard library it already has a lot of functions included for every day use, but libraries for almost every application that one could think of are available for a later download. Nowadays a lot of applications and software packages use Python, some Linux distributions include it with their builds and even in some computer games it was featured to an extent.

17 https://www.tiobe.com/tiobe-index/

Popular services that implemented Python to some extent include:

- Anaconda[18] (popular data science platform, also used in this analysis)
- Dropbox (popular cloud-based storage service)
- Ubuntu Software Manager (graphical package manager in the Ubuntu OS)
- Flask (web framework used by Pinterest and LinkedIn for example)
- Panda3D (game engine for Python)
- Notepad++ (via a PythonScript plug-in)
- QGIS (mentioned before, for scripting inside the application)
- Instagram (Back-end)[19]
- Reddit (re-written in Python)
- Rosneft (for geoengineering applications, e.g. RN-GRID is written in Python)
- NASA (for multiple frameworks and applications)

As mentioned in previous list, for this analysis the Anaconda framework is used because of including both the "conda" and "pip" package managers for an easy install and management of different Python packages. Also it does not solely work for Python, but also for other programming languages like R and similar. The framework itself includes the graphical package manager called Anaconda Navigator but also tools like the Integrated Development Environment (IDE) of Spyder and the interactive Jupyter Notebooks. Also the command-line tools can be started from here, however with this project, all commands were used and started directly from and with command line instructions.

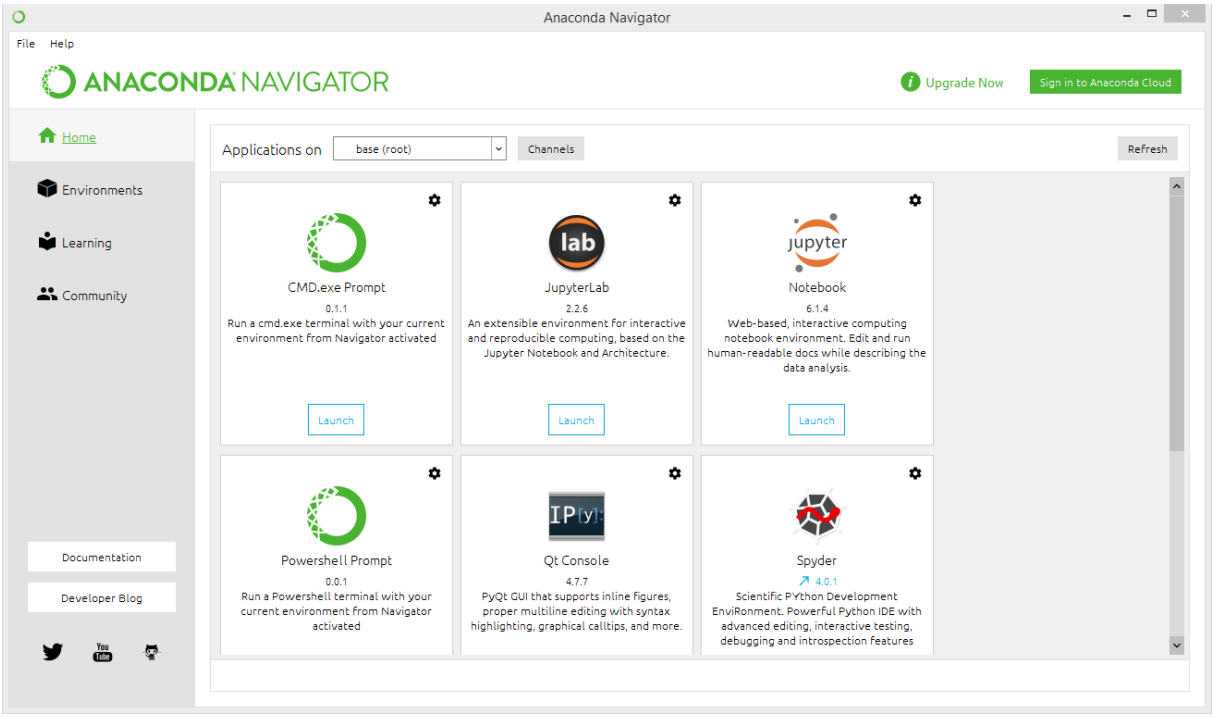


Figure 9: Overview of the Anaconda Navigator

18 https://www.anaconda.com/
19 https://www.fastcompany.com/3047642/do-the-simple-thing-first-the-engineering-behind-instagram

Apart from the pre-installed libraries that ship with the framework itself, only the "requests[20]" library for sending requests and getting results back from a server and the "geojson[21]" library for a simplified handling and creation of GeoJSON files had to be installed with the `pip install <package_name>` command inside the Anaconda command prompt. The python scripts itself (.py) in this case were created in the Notepad++ editor with syntax highlighting for Python enabled. To run the scripts following command was run in the Anaconda command prompt:

```
python script_name.py
```

Debugging the scripts and all output was directly done in the same command line tool via the default "print" commands included in Python 3.

3.7 GeoJSON

GeoJSON is a file format for the interchange of geospatial data introduced in 2007 (Butler et al., 2016). It is based on the Javascript Object Notation (JSON) with added attributes to represent simple spatial features such as points, lines and polygons. As both file formats are open standards, they are widely supported in editors and software tools. There is even a handy online editor for JSON files[22]. A GeoJSON file mostly contains of a Type "Feature Collection" containing one or multiple "Features", such as points or polygons. An overview of possible data types with examples is given below, starting with the simplest feature, a Point like a node would be represented in the OpenStreetMap:

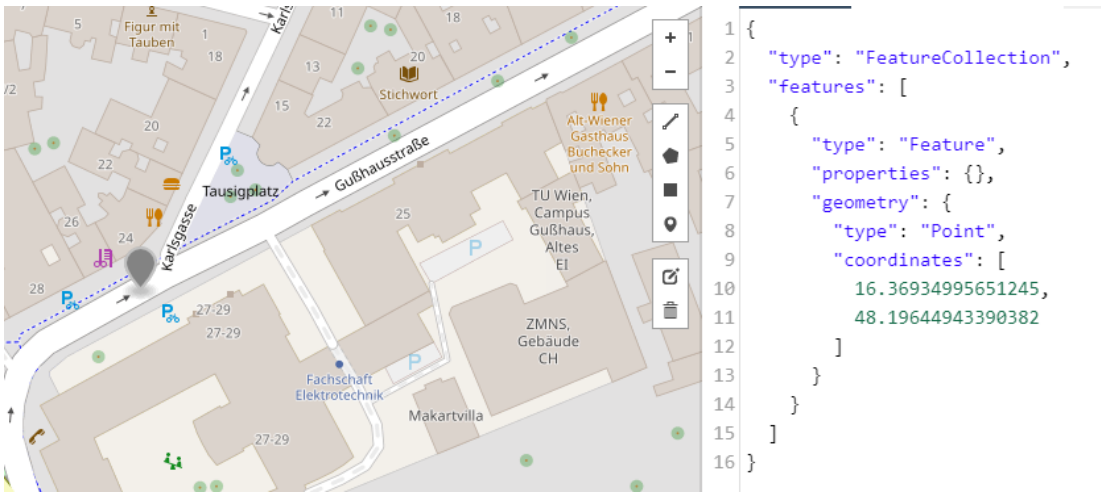


Figure 10: Example of a point in GeoJSON

20 https://pypi.org/project/requests/
21 https://pypi.org/project/geojson/
22 http://geojson.io/

LineStrings are defined by a number of points bigger than two (like routes are defined in the analysis).

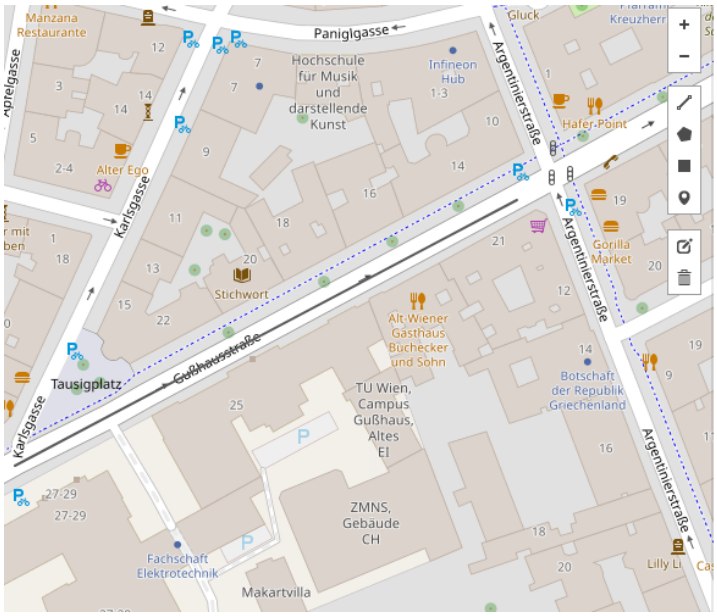


Figure 11: Example of a LineString in GeoJSON

Polygons can be made of two types, default closed ones and ones with representing a “ring” with an outer and inner polygon, where as the inner ring is an optional parameter.

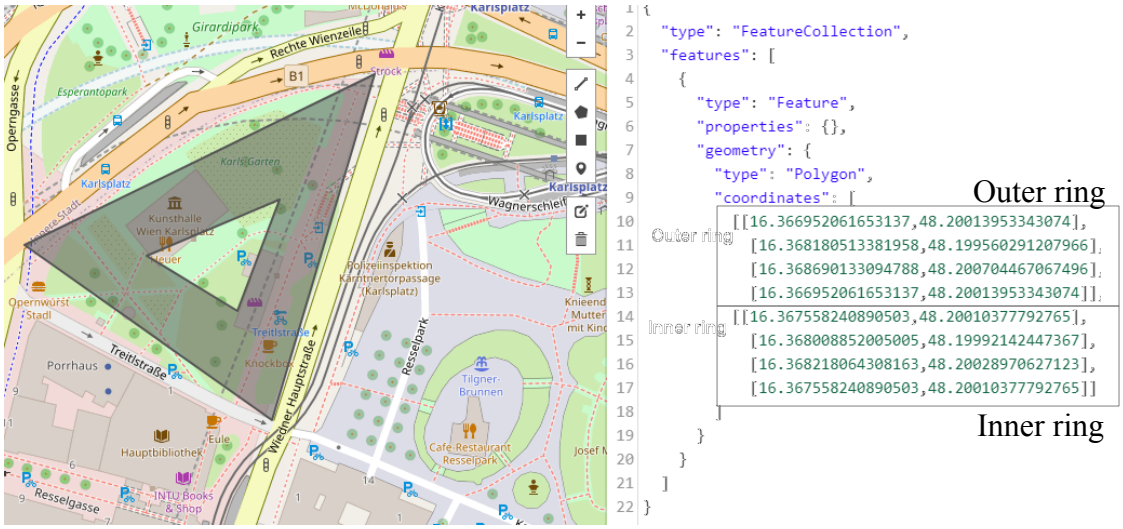


Figure 12: Example of a Polygon in GeoJSON

Theoretically one could use the “Multi” data types (namely MultiPoint, MultiLineString and MultiPolygon) to group multiple of the mentioned data types in one feature, only separated by a comma. As this is similar to the three figures above, these are not explained in detail. However there is a third useful way to group the three different data types in one single geometry, which is named “GeometryCollection”.

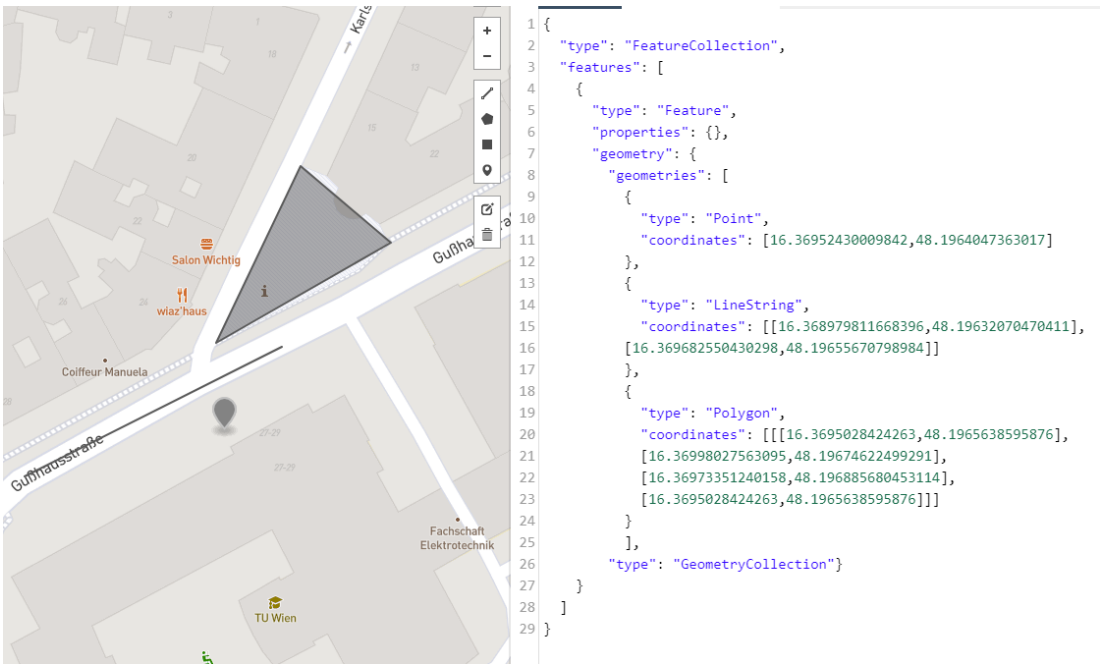


Figure 13: Example of a GeometryCollection in GeoJSON

The routes calculated in the analysis are saved as LineStrings with the years added as metadata to them. That way the routes can be analysed in other software packages and tools.

4 Implementation

Thanks to the freely available data and software tools explained in the previous chapter this analysis can be reproduced. The main chapter of this thesis is thoroughly describing every aspect of the analysis of fastest routes in all aspects, starting with the general ideas about navigating, giving some insights on the technical setup and the route creation and verification process. At the end the extraction process to get some statistical insight is explained.

4.1 Navigation problems

For the analysis it is necessary to define four cases that might occur while trying to match a route. The first case represents the best solution possible: A route will not change over the seven years, which results in a constant travel time in every dataset. This will not be the case with each route, as the road network changes constantly in real life and thus these changes will be pushed to the OpenStreetMap by users. A typical change might be a change in speed limits. This would represent the second case, in which a matching is indeed computed successfully (meaning that the route can be driven) but the matched route will be slower than the fastest (reference) route following a different path. In the data this will be visible by higher travel times in one or more years. This happens, for example, when a route allowed a higher speed in previous years, but the route in 2020 suffers from changes in the maximum allowed speed.

In case a route could not be matched successfully at all, this can result in two cases: The matching might be incomplete until an incidence point is reached, which is mostly represented by a change in one-way or access restrictions. In this case, the existing matching from the origin point up until the incident point is seen as the first part (matching part) of a route. To now simulate the user going to the original destination from the incident point, the fastest route from there to the original destination is computed on the 2020 dataset. This does not include the time the user needs to find that solution, but provides an optimal solution to estimate the time and distance needed to finally reach the destination. This part acts as the second part (routing part). Both parts will be added together by adding the geometries together and summing up their travel times and distances each. This assumes the user would have an alternative option to compute a route on current data.

If a more severe error occurred (e.g., a matched route had a travel time lower than the fastest route in 2020, while representing exactly the same route), this mostly rendered the route impossible to use in the analysis. The set of fastest route in this case was discarded then and would not be included in the total of 500 routes, meaning that all 3000 matched routes consists of routes, that are complete and are able to be followed from their respective start to their respective end point.

4.2 Workflow

The workflow of this project follows a clear, linear structure:

- Download and preparation of datasets from Geofabrik
  - Download of the Austria datasets (2014-2020)
  - Clipping of the datasets to Vienna with osmconvert

- Download of the software tools needed in the project
  - Download of the Anaconda Python distribution
  - Download of QGIS
  - Download of VirtualBox and Lubuntu OS for OSRM in a Docker environment

- Setup of the software tools
  - Download of the Python libraries not included in Anaconda
  - Setup of the Virtual Machine running Lubuntu with Docker
  - Preparation of the Vienna datasets for use with OSRM in a Docker environment
  - Setup Port Forwarding to be able to use OSRM outside the Lubuntu VM

- Compute a set of seven fastest routes
  - Define a random start and end-point
  - Compute fastest routes from 2014 to 2020
  - Check points and routes for completeness and correctness (compute a new set if an error occurs)
  - Save fastest routes in a GeoJSON file

- Match a set of fastest routes to the most-recent dataset
  - Read fastest routes from a GeoJSON file
  - Try to match the fastest routes from 2014 to 2019 on the 2020 dataset
  - Check all routes for completeness and correctness (compute a new set if an error occurs)
  - Save fastest and matched routes in a GeoJSON file

- Analyse routes (after 500 routes have been computed and checked successfully)
  - Extract statistical parameters automatically from all GeoJSON files
  - Do a statistical analysis in a spreadsheet software

The workflow of computing and matching routes is also described in the workflow diagram in Figure 14. Apart from computing and matching the routes with the OSRM routing engine, the process of checking the routes for completeness and correctness is done manually by hand.

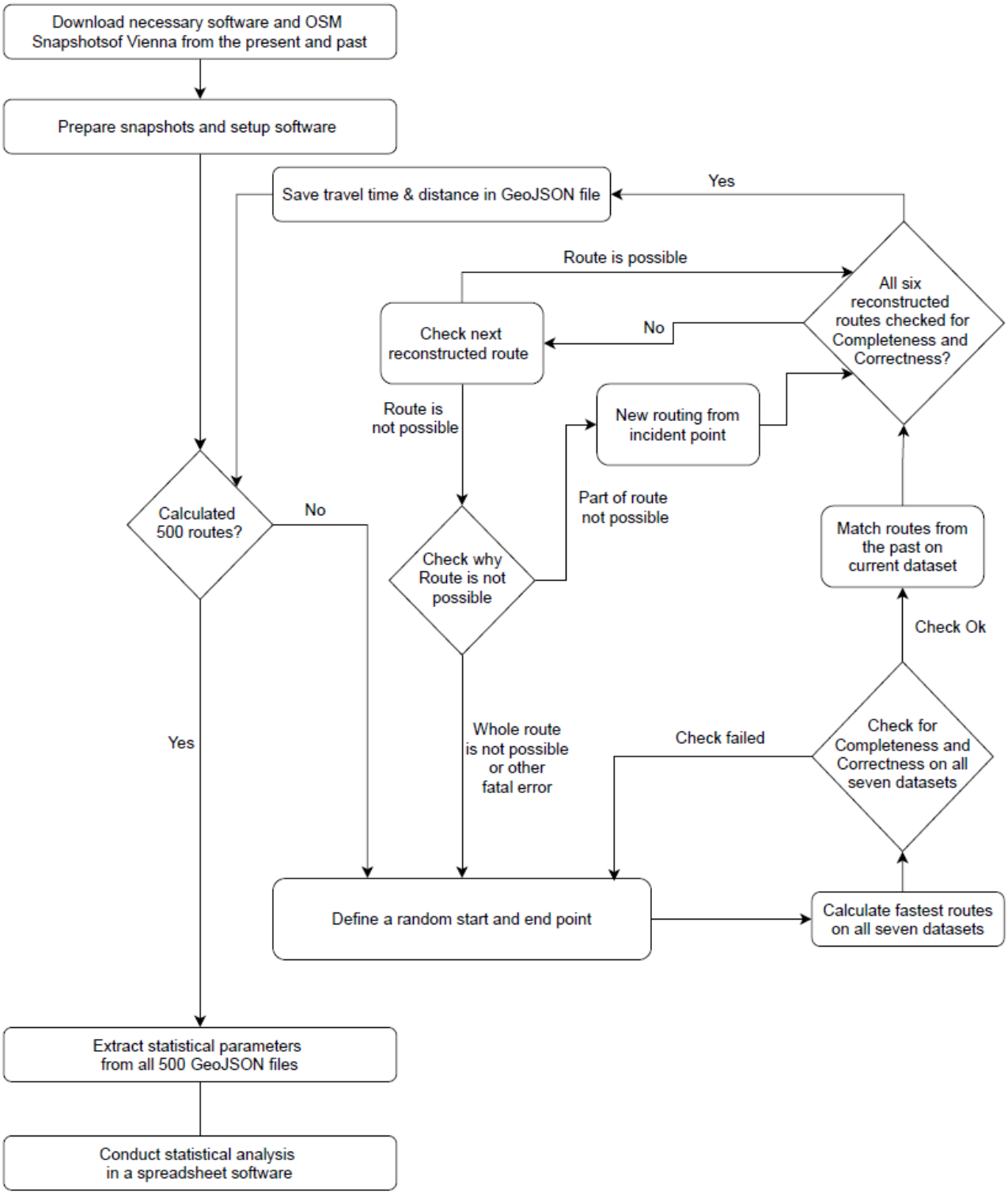


Figure 14: Workflow of the project

4.3 Environment setup

As the whole analysis is run on a system running Windows 8.1, all software tools have to be compatible with that operating system. Notepad++ is used as an editor for the Python scripts and was already installed on the machine. QGIS was downloaded and installed from their website, no further additional plug-ins or settings are needed, as it was the case with osmconvert.

Installing OSRM takes more prerequisites. Building it from scratch on Windows takes more storage than available on the host machine used for this thesis and the pre-compiled Windows binaries are out-dated by August 2020. The simplest solution would be running OSRM via a Docker image. As the Docker desktop software is only available for Windows 10, one solution would be to run Docker Machine, an older variant of Docker which runs Docker on a Virtual Machine. Instead of that, the decision was made to set up a small Linux distribution and running Docker images from there, as that is natively supported. Therefore VirtualBox[23] was downloaded and installed to run a Virtual Machine from there. The 64-bit version of Lubuntu[24], a light-weight variant of the popular Ubuntu Linux distribution was used as operating system for the virtual machine. Following parameters were used in the initial setup:

- Name: Lubuntu (can be set to the users preference)
- Type: Linux
- Version: Ubuntu (64-bit)
- Memory size (RAM): 2048 MB
- Virtual Hard Disk with 10 GB
- Hard Disk File Type: VDI (VirtualBox Disk Image)
- Storage on physical hard disk: Dynamically allocated
- File location and size: Set to user preference

On booting up the Virtual Machine, the image of the Lubuntu OS has to be inserted into the virtual disk drive to trigger an installation process. The installation process does not require any parameters. After installing Lubuntu, a desktop shows up, as pictured in Figure 15. Now that the virtual machine is set up, the Docker environment has to be installed. This can be done either via the Synaptic package manager (included in Lubuntu) or via the apt commands over the command-line, for which a tutorial by Docker is provided[25]. For Lubuntu it is important to run every command as an administrator with the sudo prefix (it will ask for the password provided for the active user entered in the setup process). Included in the Docker setup tutorial is a way to check if Docker is installed properly. The correctness of the installation can be checked on the command line by typing:

```
sudo docker run hello-world
```

23 https://www.virtualbox.org/
24 https://lubuntu.net/
25 https://docs.docker.com/engine/install/ubuntu/

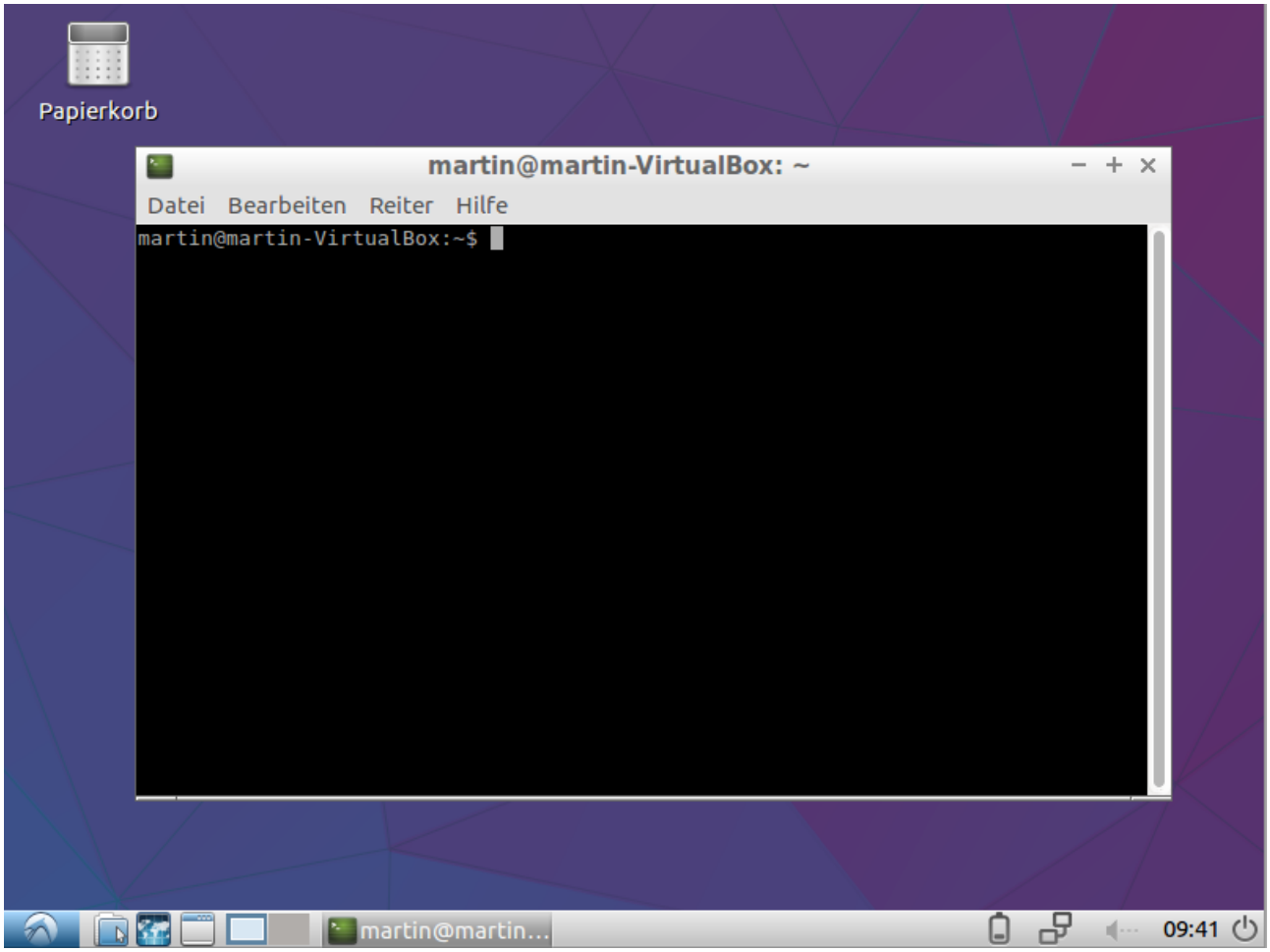


Figure 15: Overview of the Virtual Machine running the LXTerminal command-line tool

This runs the hello-world container included in the standard Docker installation. If Docker was installed successfully, it should show a message starting with following text:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

After the installation of Docker, OSRM back-end can be installed. Project OSRM offers two Docker images, one for the back-end (the routing itself) and a front-end (like the interface shown in the demonstration) which is optional in this application. For the back-end a quick start tutorial is available[26], the steps in this case are described thoroughly. The pre-processing pipeline has to be done for all seven datasets, however it is shown here for one dataset only.

First, the PBF files for Vienna have to be put in a directory, which acts as the working directory. In this case "osrm/data/" was used in the analysis, so with the first command the current working directory will be changed into. The second command is extracting routing information with the car profile from the PBF file.

26 https://hub.docker.com/r/osrm/osrm-backend/

1) `cd osrm/data`
2) `sudo docker run -t -v "${PWD}:/data" osrm/osrm-backend osrm-extract -p /opt/car.lua /data/wien14.pbf`

The `-v "${PWD}:/data"` block creates a /data directory inside the docker container for the dataset and makes the current working directory (available under `"${PWD}"`) available there. The `data/wien14.pbf` file then refers to the `"${PWD}"/wien14.pbf` on the host.

Now the data has to be partitioned and customised to be used in the routing engine itself (keep an eye on the different file extension of the dataset – it now is .osrm instead of .pbf):

3) `sudo docker run -t -v "${PWD}:/data" osrm/osrm-backend osrm-partition /data/wien14.osrm`
4) `sudo docker run -t -v "${PWD}:/data" osrm/osrm-backend osrm-customize /data/wien14.osrm`

To run the routing service the following command can be used:

5) `sudo docker run -t -i -p 5000:5000 -v "${PWD}:/data" osrm/osrm-backend osrm-routed --algorithm mld /data/wien14.pbf`

The routing service is now running on the localhost of the virtual machine (127.0.0.1) on the port 5000. It will also show any access to the routing machine in the LXTerminal window. To test the setup, open the Firefox Browser in the Lubuntu virtual machine and open following example request:

`http://127.0.0.1:5000/route/v1/driving/16.369473,48.196471;16.366711,48.199435`

It should show you the result of the example routing request as a JSON formatted code. The result can be read by a human user but it is more useful to another application to process this further.

As this project uses seven datasets, seven different ports should be used, one for each dataset. This way, computations can be done simultaneously without having to restart the Docker images. To achieve that, Docker has a possibility to assign custom ports. Also containers should be running in the background instead of starting them up when needed. Therefore the detach flag can be used, which allows to detach containers at start-up. The full Docker command used in the end is shown below, which requires to be run seven times to start a single service for each dataset:

```
sudo docker run -d -t -i -p 5014:5000 -v "${PWD}:/data" osrm/osrm-backend osrm-routed --algorithm mld /data/wien14.pbf --max-matching-size 10000
```

The first difference to the previous start-up command is the -d parameter, which allows running a Docker image detached (in the background). Instead of the default port 5000, the 2014 dataset runs on port 5014. Therefore the 2015 dataset runs on port 5015 etc., which means that a constant number of 3000 is always added to the year the dataset is based in. The last change is the max-matching-size parameter. This prevents a public server to overload because of requests with a high number of input parameters. In the case of the analysis this is needed though, and as the OSRM server is running on request and for this project only, it can be set to 10000 to allow the requests to be successful. To check if all seven instances have been started successfully (change ports and dataset names accordingly), the persistent services command can be used:

```
sudo docker ps
```

When using this it should list the Docker images running at the moment. Therefore it should list seven running services showing the IP addresses from 0.0.0.0:5014 to 0.0.0.0:5020. It can be tested by doing an example request in the Firefox browser again and changing the port number for the different datasets:

```
http://127.0.0.1:5014/route/v1/driving/16.369473,48.196471;16.366711,48.199435
```

The only setting left is being able to access the ports from outside of the Virtual Machine. VirtualBox supports port forwarding for that case. This can be set by accessing the settings of the Virtual Machine (it can still be running). Access the Network tab and the only selectable Tab in there should be Adapter 1. After extending the Advanced tab, a “Port Forwarding button can be clicked”. This should be done and a table comes up, where one can set up rules. In this case, the ports should be forwarded from the localhost of the OS of the virtual machine (Lubuntu) to the host OS (Windows). Therefore the Host IP is always 127.0.0.1 (localhost) and the Guest IP can be left at default (0.0.0.0). The port numbers are always identical in both (e.g., 5014 and 5014). The final setup can be seen in Figure 16.

To test the port forwarding setup, one can access the example URL above from a internet browser of choice on the Windows host. By manually changing the port number from 5014 up to 5020 in the URL, the different routes on the different datasets can be accessed. This means, one can now do a first assessment of how a route duration changes over the years, as the same 2 points are used to calculated the fastest route between those two. The travel time can be found in the “duration” parameter of the result of the query directly in the browser:

- 2014: 142.2 seconds
- 2015: 142.2 seconds
- 2016: 143.7 seconds
- 2017: 158.8 seconds
- 2018: 127.2 seconds
- 2019: 127.2 seconds
- 2020: 127.2 seconds

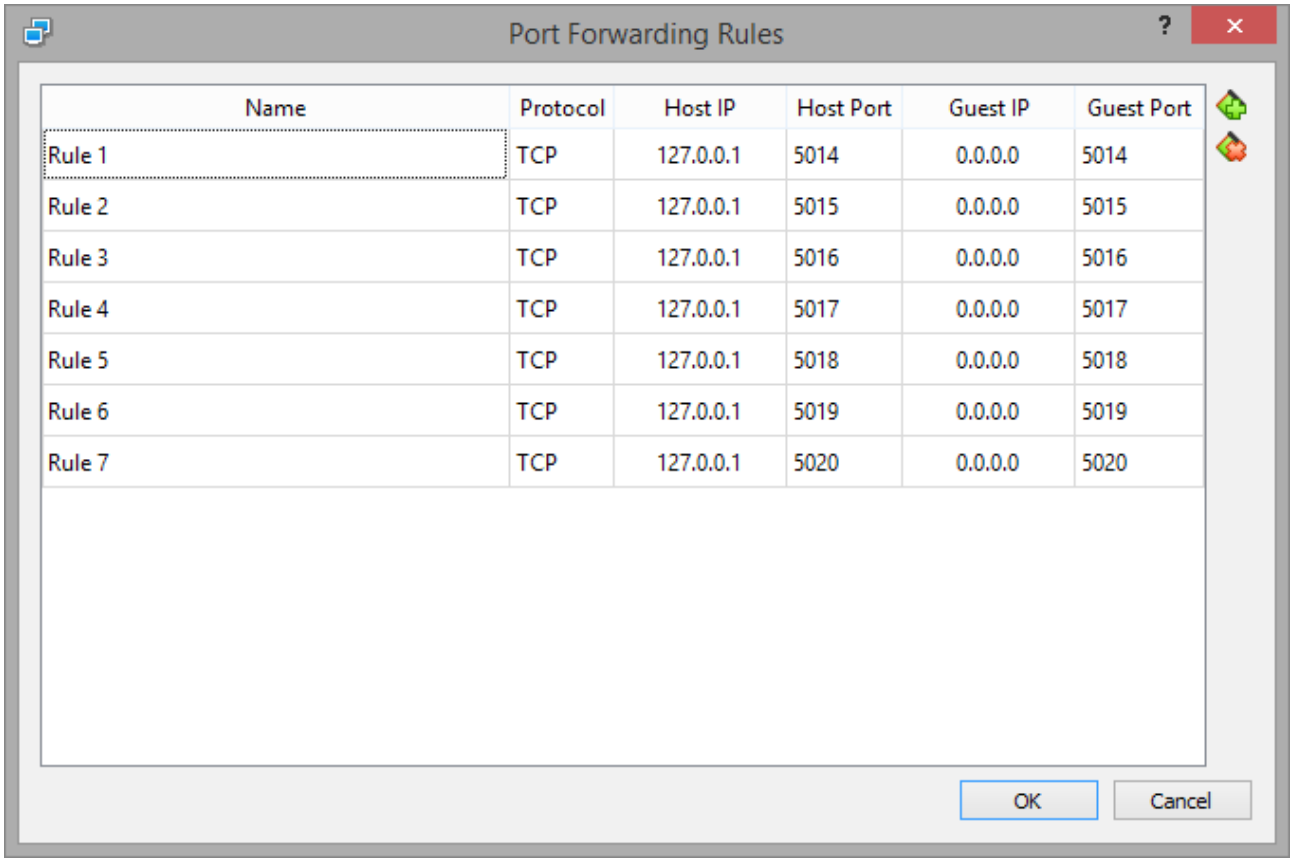


Figure 16: Port forwarding rules in the settings of the Virtual Machine

This is only a short example route (as pictured in Figure 8), but the changes of travel time over the years are quite significantly due to changes on the underlying OpenStreetMap data. Examples on why the travel time changes are thoroughly described later in this thesis. By correctly getting this output, the setup of the environment is done and the actual process of creating routes can be started.

4.4 Route creation

The creation of routes between two random points in Vienna is handled by the first Python script (*01_get_routes.py*). First, three imports have to be done at the beginning of the file:

- `import numpy as np`
  The Numpy package is used in this case for getting random numbers in a certain range of numbers for the latitude and longitude.

- `import requests`
  The request library is needed to send requests in form of URLs to the server and also to process responses from it.

- `from geojson import Point, LineString, Feature, FeatureCollection, dump`
  The listed functions of the GeoJSON library are needed to read and write features into GeoJSON files.

As two routines are used commonly, two functions have been defined to easily execute them repeatedly. One for getting the nearest point on the road network to a given coordinate in a year of choice, the other function is wrapped around the actual routing request. Both are explained below:

```
def nearest_road_point(lon, lat, year):
	loc = "{},{}".format(lon, lat)
	y = year + 3000
	url = "http://127.0.0.1:" + str(y) + "/nearest/v1/driving/"
	r = requests.get(url + loc)
	if r.status_code!= 200:
		return {}
	res = r.json()
	nearest_point = res['waypoints'][0]['location']
	return nearest_point
```

The function to find the nearest point on the road network to a given coordinate, one parameter for the longitude and one for the latitude. Also the `year` parameter specifies the dataset on which the nearest point should be computed, as seven datasets are available. The `loc` variable formats the coordinate string so that it can be used in the URL easily. A constant of 3000 is added to the `year`, to match the port number used for the OSRM routing server. The `url` variable forms the first part of the URL that is later used for the request. The `r` variable now requests the query from the server, combining the `url` and `loc` variables to form the full request. Should the server not return a status code of 200 for the successful computation of the request, an empty result is returned.

In case the request was successfully done, it is now possible to convert the result to a JSON format, which is done in the `res` variable and the `.json()` command. The nearest point then gets read from this JSON construct and gets returned to use it in the main part of the Python script. The second function to get the fastest route between two given points works in a similar way:

```
def get_route(start_lon, start_lat, end_lon, end_lat, year):
	loc = "{},{};{},{}".format(start_lon, start_lat, end_lon, end_lat)
	y = year + 3000
	url = "http://127.0.0.1:" + str(y) + "/route/v1/driving/"
	url2 = "?overview=full&geometries=geojson"
	r = requests.get(url + loc + url2)
	if r.status_code!= 200:
		return {}
	res = r.json()
	geometry = res['routes'][0]['geometry']
	coords = res['routes'][0]['geometry']['coordinates']
	distance = res['routes'][0]['distance']
	duration = res['routes'][0]['duration']
	start_point = [ res['waypoints'][0]['location'][1],
			res['waypoints'][0]['location'][0] ]
	end_point = [ res['waypoints'][1]['location'][1],
			res['waypoints'][1]['location'][0] ]
	out = {
		'geometry':geometry,
		'coords':coords,
		'distance':distance,
		'duration':duration,
		'start_point':start_point,
		'end_point':end_point
		}
	return out
```

As with the first function, the `loc` variable formats the input parameters so that they can be used to form the query URL later on. The difference to the function "`nearest_road_point`" is that this function now takes two values for latitude and longitude to be able to compute the fastest route between these two points. The request URL is formed by the `y`, `url`, `loc` and `url2` variables this time, with the `url2` variable containing extra parameters needed for further analysis later in this thesis. It is checked again if a valid result can be retrieved from the server, and if possible, the result gets converted into a JSON structure again. A number of values are retrieved from this JSON structure including:

- geometry
  This array contains the full GeoJSON geometry for the usage in software that supports the GeoJSON file format.

- coords
  Same as geometry, however it just contains the raw array of coordinate pairs that represents the fastest routes.

- distance
  Distance of the fastest route in meters.

- duration
  Travel time of the fastest route in seconds.

- start_point
  Origin of the route (latitude/longitude coordinate pair), snapped to the road graph.

- end_point
  Destination of the route (latitude/longitude coordinate pair).

By modifying the option parameters of a request and/or defining other output values, even more parameters from the request could be used, like the route instructions for every decision point or important waypoints and details about them, those however are not needed in this analysis and thus they are not featured. The main part of the first Python script consist of structured parts, starting with the command to get random coordinates for the start and end points of a route:

```
lons = np.random.uniform(16.22,16.53,2).round(6)
lats = np.random.uniform(48.14,48.28,2).round(6)
```

The random longitude and latitude values are computed with a uniform distribution, so that every number between two given values has an equal chance to appear. In this case, two random longitude values between 16.22°E and 16.53°E and two random latitude values between 48.14°N and 48.28°N. Thus, technically a rectangle bounding box is created, which the coordinates can not escape. This makes sure that no origin or destination points outside of Vienna are randomly generated. However, those two points are not snapped to the road network. The above function can be used to snap points to the road network:

```
start_point = nearest_road_point(lons[0], lats[0], 2020)
end_point = nearest_road_point(lons[1], lats[1], 2020)
```

Now the routing can be done for each year between 2014 and 2020. A loop is created to compute the seven fastest routes in one go. Before the start, an empty array has to be defined to store the fastest routes after their computation.

```
features2 = []
for year in range(2014,2021):
	req = get_route(start_point[0], start_point[1], end_point[0], end_point[1], year)
	route_json = req['geometry']
	route_time = req['duration']
	route_dist = req['distance']
	features2.append(Feature(geometry=route_json, properties={"year": year,
		"duration": route_time, "distance": route_dist})
```

The `req` variable manages to get the output from the actual request, which is done via the "`get_route`" function explained before. The start and end points are taken from the earlier function. Three parameters are taken from the result gained from the OSRM server: The actual geometry of the fastest routes, and their respective travel distance and travel time. The geometry then gets added to a GeoJSON compliant feature, with three properties: The year the fastest route was computed in, and their travel distance and travel time. After this loop was executed seven times, the features2 array now contains seven features. These have to be added to a FeatureCollection to get added to a single GeoJSON file, which is done via following prompts:

```
feature_collection2 = FeatureCollection(features2)
with open('id_fastest.geojson', 'w') as file:
	dump(feature_collection2, file)
```

A GeoJSON file is first opened, with the filename containing "`id`" which should represent the number of the route file later. The parameter "`w`" defines that this file should be possible to write into. Then the whole Feature collection of the seven fastest routes gets dumped into the file that was just opened. After the `with` command was executed and the Feature Collection was put into the respective file, the first Python script terminates. The created GeoJSON file can now be opened in QGIS and loaded into a new project. All routes are shown in the same colour and style, so at first, a new style schema has to be created to assign one colour by year each. This should make it possible to distinguish the fastest route for each year. Now it is checked manually if each route is complete and does not suffer from inconsistencies (e.g., each route should start and end at the same point). A second check should be done by using the attribute table: If there is a distance or duration that is much lower than one (or more) than those of the other routes, the matching is most likely incomplete. This has to be verified first and a decision has to be made if it can be fixed with a reasonable amount of work. A quick solution to this problem is to just create a new set of coordinates and calculate the fastest routes between those. An example of this whole route creation process and quality check should be given below:

At first, the Python script is run, which gives us a new “id_fastest.geojson” file. This file is loaded into a prepared QGIS project, which has a background map enabled for easier orientation. The routes are visually inspected at first to detect possible errors.

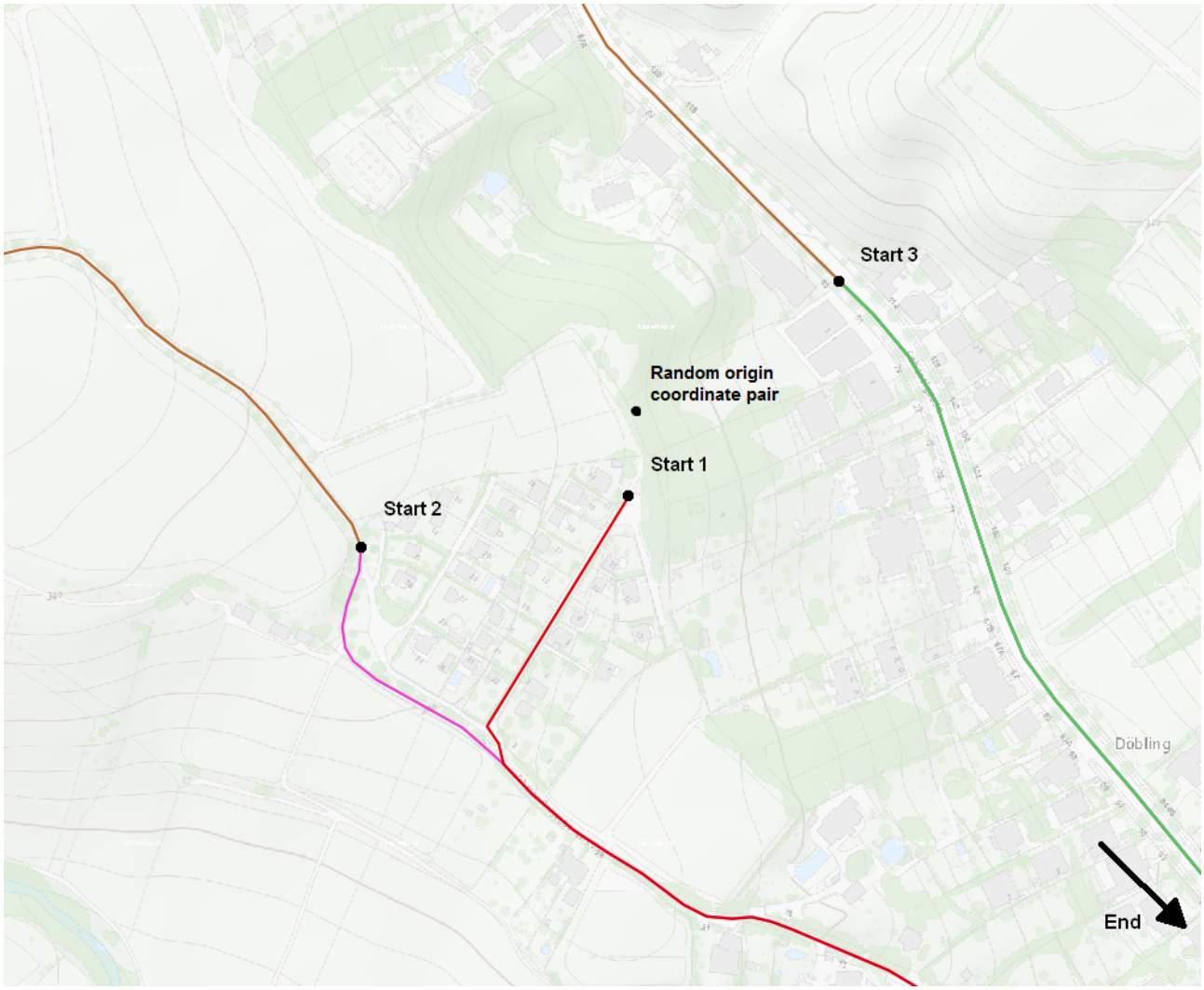


Figure 17: Multiple starting points for fastest routes

Figure 17 shows multiple starting points created from the same random coordinate pair around the Kahlenberg in Vienna. This happens due to the changes over the years made on OSM. Here, the red route is from the most recent dataset (2020). Random coordinates are always snapped to the nearest road point in the 2020 dataset. As the Start 1 point might have existed in the 2020 dataset, but not in those before, The Start 2 and Start 3 are chosen for older routes because they have been the closest to the random origin coordinate pair. In this case the inconsistent origin points are a problem, as they all seven fastest routes should have the same origin and destination. In this case, a new set of seven fastest routes was computed to be included in the analysis.

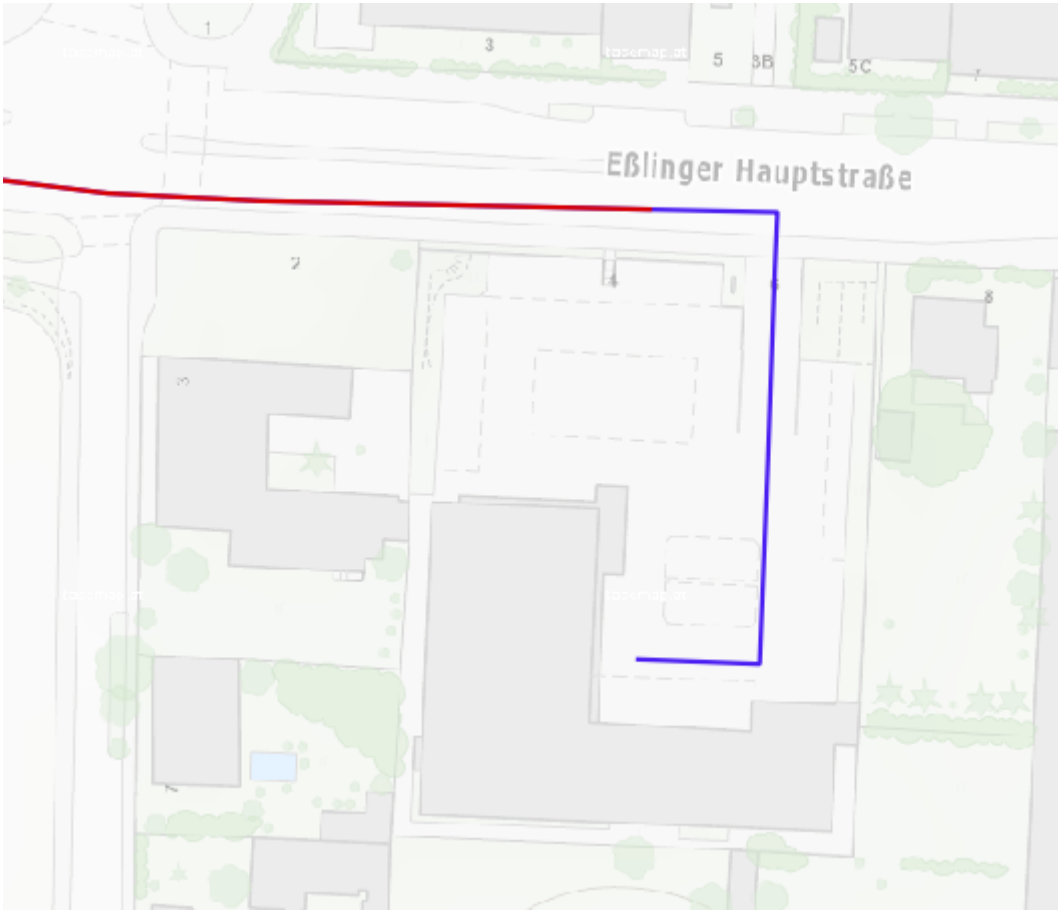


Figure 18: Incomplete route into a backyard

Another error that occurred often is shown in Figure 18. Sometimes, entries into backyards or parking spots were mapped as useable by car traffic (or in a way that the routing engine thinks that it is useable). The red route resembles a route from a year which is different to the blue route from another year, as the blue route uses paths into/out of the backyard of a building. Like in the example given before, this means a set of fastest routes has inconsistent start and/or end-points and therefore should be withdrawn and replaced by a new set of seven fastest routes.

Other types of errors include:

- Fastest route(s) could not be computed at all
- Fastest route(s) starts or ends at the adjacent side of the street (e.g., separated lanes for each direction of travel)
- Fastest route(s) use roads that they should not use (legal aspect)

Also every set of fastest routes is not only visually checked for errors. The attribute table in QGIS also is useful for giving an indication that one or more routes might contain an error of some sort, as the travel times and distances in every single year are visually checked for outliers in this window. If both of this checks are successfully done, the routes can now be matched to most-recent dataset.

4.5 Route matching

The matching of fastest routes to the most recent dataset is handled by the second Python script (*02_match_routes.py*). As with the first script, the imports have to be defined first:

- `import requests`
  The requests library is used for requests against the server and getting results from it.

- `from geojson import Point, LineString, Feature, FeatureCollection, dump, load`
  The listed imports are used for writing into a GeoJSON file. This time also several parts are read from an existing GeoJSON file with the load command.

In addition, two functions are defined that are required for the analysis. One function converts a route list into a string for an easier use with matching requests, the other function takes care of the actual matching process. The first function is explained below:

```
def route_list_to_string(ref_route_list):
	ref_route_string = ""
	for c in range(0, len(ref_route_list[:-1])):
		this_c = ref_route_list[c]
		next_c = ref_route_list[c+1]
		loc = "{},{};{},{}".format(this_c[0], this_c[1], next_c[0], next_c[1])
		ref_route_string = ref_route_string + loc + ";"
	return ref_route_string[:-1]
```

The function takes a single input parameter, a list of coordinate pairs, which in this case comes directly from the coordinate array in the GeoJSON file of fastest routes. Then an empty string is created to add coordinates for the output. A for loop iterates over the whole list of coordinates, where as a current point is defined by a coordinate pair and a second point defined by the next coordinate pair to route to. The `loc` variable again formats the string so that it can be used right away. With every iteration the reference route string adds two coordinate pairs to the output string, followed by a semicolon to end this pair of coordinates. At the end the reference route string is being given back, with removing the last semicolon at the very end of the request string because that is not needed any more and a request would throw an error. For example, a coordinate list of 3 coordinate pairs `[[16.369454,48.196492],[16.369454,48.196492],[16.369467,48.19655]]` would get converted to the following string to be used in a request URL:

```
16.369454,48.196492;16.369454,48.196492;16.369454,48.196492;16.369467,48.19655
```

The second function uses two input parameters. A coordinate string retrieved from the first function, and a `year`, which corresponds to the year the route should be matched to. In this analysis, the `year` variable will be always set to 2020, as every route should be recreated on the most-recent dataset to get an idea of possible extensions of the travel time and the changes in travel distance.

```
def match_route(coord_string, year):
	y = year + 3000
	url = "http://127.0.0.1:" + str(y) + "/match/v1/driving/" + coord_string + "?
		overview=full&geometries=geojson"
	r = requests.get(url)
	if r.status_code!= 200:
		return {}
	res = r.json()
	geometry = res['matchings'][0]['geometry']
	distance = res['matchings'][0]['distance']
	duration = res['matchings'][0]['duration']
	confidence = res['matchings'][0]['confidence']
	try:
		start_point = [res['tracepoints'][0]['location'][1],
				res['tracepoints'][0]['location'][0]]
	except:
		start_point = [0,0]
	try:
		end_point = [res['tracepoints'][-1]['location'][1],
				res['tracepoints'][-1]['location'][0]]
	except:
		end_point = [0,0]
	out = {
		'geometry':geometry,
		'distance':distance,
		'duration':duration,
		'confidence':confidence,
		'start_point':start_point,
		'end_point':end_point
		}
	return out
```

The first lines are similar to the routing function defined in the first Python script. A constant of 3000 is added to the `year` to match the port numbers in the routing requests. The `url` variable defines the request URL for the OSRM server and takes the year and coordinate string into account. The request to the OSRM server is then made, and if a successful request with the status code 200 is returned, the whole answer from the server is parsed into the JSON format again. Following parameters are then retrieved from the response (with the exceptions for the start- and end-point explained later on, as they could cause an error while running the code):

- geometry
  Array containing the full geometry in the GeoJSON format.

- distance
  Distance travelled in the matched route.

- duration
  Travel time of the matched route.

- confidence
  This parameter gives a number between 0 and 1, which states how confident the routing engine is with the matched route. The higher the number, the more confident the matching seems to be.

- start_point
  Start point of the matched route.

- end_point
  End point of the matched route.

Sometimes the matching process fails completely for a route, leaving the start_point and end_point as an empty array. Therefore no value can be read from them, crashing the Python script. This is not desired at all in the case of trying to match multiple fastest route files in one go. To avoid this, the try and except commands were used. Should those arrays be empty, the start and/or end point will get replaced by a coordinate pair of zeros. This makes it technically possible to complete a batch of matching processes, as the routing for the involved set(s) will fail. This will be recognised during the manual checking process done in QGIS, and a new set will be created later on.

The option to be able to match one or more routes is managed via the `number_array` variable:

```
number_array = [1001]
```

As the fastest routes have IDs ranging from 001 to 500 to distinguish between them, the variable was done in a way so that different fastest route files can be matched in one go. This was especially useful when first matching all 500 fastest route files or later on, when matching new sets replaced routes that were not suited for the analysis. The first digit will be ignored by the script later on and is just there to avoid that for e.g., 001 the leading zeros will not get removed. For a full batch operation the array could be just set to read [1001:1501], this will read all fastest routes from 001_fastest.geojson to 500_fastest.geojson. As the rest of the script is all included in a big for-loop, it is listed as one code block on the next page.

The first block converts the current ID to a string so it can be used to define the filename consisting of strings to open the next fastest route file in the GeoJSON format. Also a new empty features list is created to fill in the fastest routes and matched routes later. For the string also the first number is left out to represent the real IDs of the fastest routes. The GeoJSON file is then opened and a new empty list for the fastest routes is created.

The second block of the code runs through a loop seven times to read out the seven fastest routes and its parameters from 2014 to 2020: The full GeoJSON geometry, the travel time and the travelled distance go directly into the complete GeoJSON routes file. The coordinates of the fastest route however will be exported and converted into a single string to be put in the matching request URL. After these four parameters have been read from the “fastest” GeoJSON file, they are put into the “routes” GeoJSON file and the fastest_list variable for later matching respectively.

In the third block the year the data is matched from is defined as the integer part, the part after the comma will always be 2020, as all data is matched on the 2020 dataset. Technically the last zero gets ignored again, but it is added back later to a string. The following for loop is called 6 times, for matching the fastest routes from 2014 to 2019 to the 2020 dataset. The loop will be left when the year variable reaches 2020.2020, as the fastest route from 2020 can be directly used as a result. Via the req_year variable and the match_route function the actual matching request is done. From the result of the query, three parameters are extracted, just like in a normal routing request: The complete route in the GeoJSON format, the travel time and the travelled distance. All three parameters plus the year number gets added to the features list and the year number is raised by one for the next iteration.

Creating the final GeoJSON file is handled in the last block. First all features are added to a single FeatureCollection. Then the filename is defined as “ID_routes.geojson” where as the ID is the same as in the fastest route file used in the first code block. At last, the GeoJSON file is created and opened, and the whole FeatureCollection is dumped in there. In case of a batch operation an output of the finished files is given too, to give an idea of how far the matching has succeeded. This output will show as follows:

```
File 001 done!
File 002 done!
...
```

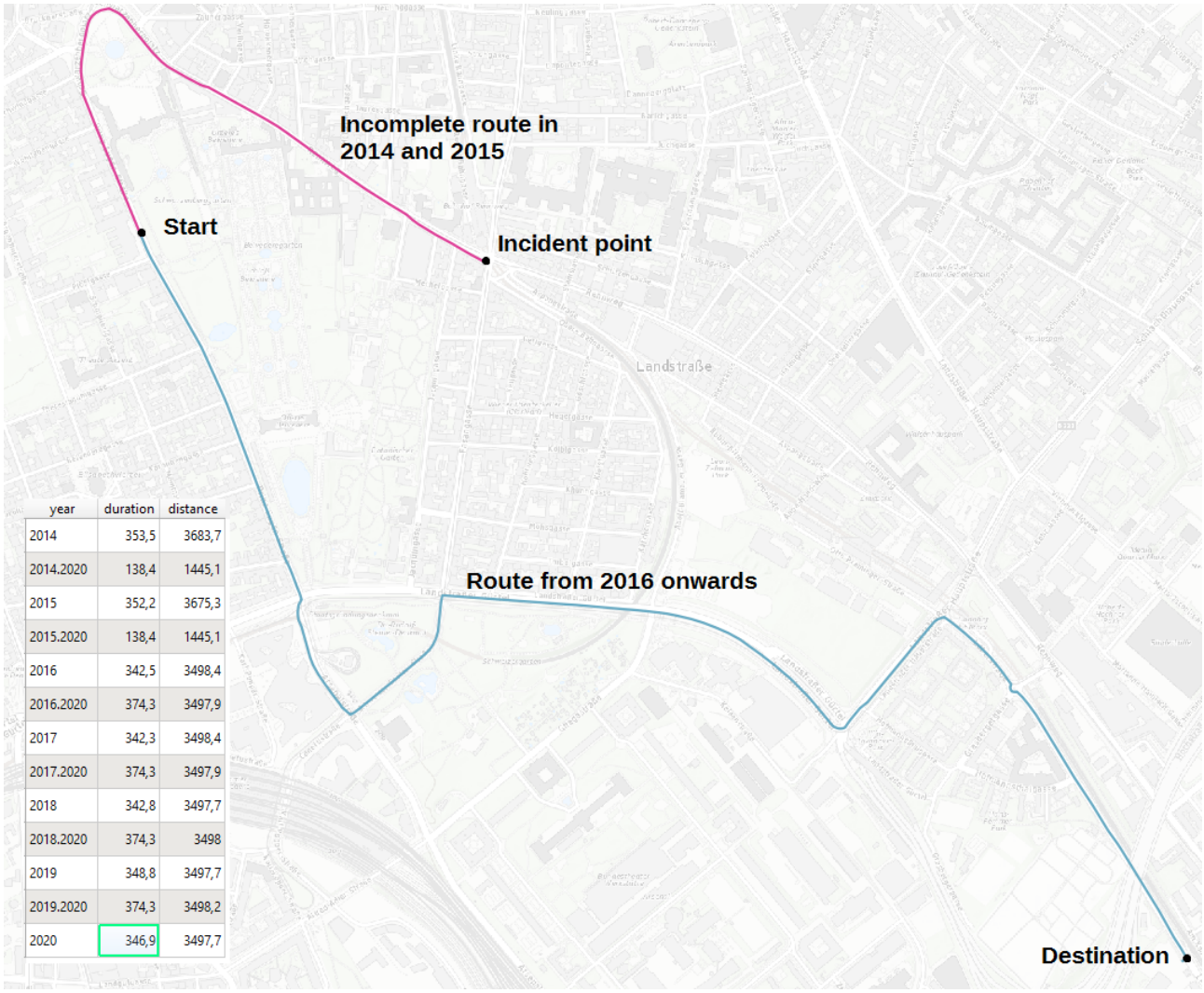
```
for number in number_array:
	number_str = str(number)
	file_string1 = number_str[1:] + '_fastest.geojson'
	features = []
	with open(file_string1) as read_file:
		gj1 = load(read_file)
	fastest_list = []

	for i in range(0,7):
		fastest_geometry = gj1['features'][i]['geometry']
		fastest_route = route_list_to_string(gj1['features'][i]['geometry']
			['coordinates'])
		fastest_time = gj1['features'][i]['properties']['duration']
		fastest_distance = gj1['features'][i]['properties']['distance']
	fastest_list.append(fastest_route)
	features.append(Feature(geometry=fastest_geometry,
		properties={"year": i+2014, "duration": fastest_time,
		"distance": fastest_distance}))

	year = 2014.2020
	for fastest in fastest_list:
		if year == 2020.2020:
			break
		req_year = match_route(fastest, 2020)
		year_route_json = req_year['geometry']
		year_route_time = req_year['duration']
		year_route_dist = req_year['distance']
		features.append(Feature(geometry=year_route_json, properties={"year":
			str(year)+"0", "duration": year_route_time, "distance":
			year_route_dist}))
		year = year + 1

	feature_collection = FeatureCollection(features)
	file_string2 = number_str[1:] + '_routes.geojson'
	with open(file_string2, 'w') as write_file:
		dump(feature_collection, write_file)
	print("File " + number_str[1:] + ' done!')
```

The created GeoJSON files can now be loaded into QGIS again, this is done in this case with a project having a map in the background for easier orientation and styling files so that every route is displayed in a different style and colour to be able to tell them apart. As in the routing process, every route file has now to be checked manually by hand in QGIS for completeness and errors. An example of this manual check is given below.



year	duration	distance
2014	353,5	3683,7
2014.2020	138,4	1445,1
2015	352,2	3675,3
2015.2020	138,4	1445,1
2016	342,5	3498,4
2016.2020	374,3	3497,9
2017	342,3	3498,4
2017.2020	374,3	3497,9
2018	342,8	3497,7
2018.2020	374,3	3498
2019	348,8	3497,7
2019.2020	374,3	3498,2
2020	346,9	3497,7

Figure 19: Example of a matching error

In Figure 19, a common matching error is pictured. The years with a suffix of ".2020" are the matched routes, while the routes with the sole year number are the fastest routes from that year. As in 2020 the area around the Landstraße was under several construction works, the routes from 2014 and 2015 could not be matched completely on the 2020 dataset. As the fastest route changed from 2016 onwards, the routes from 2016 to 2020 did not change and therefore have the same travel time (blue route). In this case, for the routes matched from 2014 and 2015 (pink route), the first part of the matched route was taken into account and from the last point of the incomplete route, a routing from that point to the original destination was computed in addition.

Then the travel times and distances of those two parts have been added together, as has been the geometry to represent a full route. Also this route had to be discarded, as the route in 2020 is the same as in the years before, but shorter in time. This is a perfect example in which the usage of an older dataset results in an extension of travel time and exactly what the analysis should contain. Also there can be changes in the distance of about one to two meters in the same routes, which stem from accuracy errors in the routing engine. These can be ignored, as those don't have an effect on the result in the end and are smaller than an average car length.

4.6 Extraction of quality elements

After all 500 files have been both successfully created and manually checked for completeness and correctness, the third Python script can be used (*03_write_data.py*). As with the two Python scripts before, the imports are getting handled first:

- `import geojson`
  The GeoJSON library to parse the parameters from the 500 route files.

- `import csv`
  The CSV library to create and write into a spreadsheet file for further analysis.

The difference in the third script is, that no defined functions have been used for the last script, as everything defined in the main part of the script. Because the script is mostly one big loop, the code is listed in as one code block on the next page but described on this page.

In the beginning, a new CSV file has to be opened with the parameter **"w"** to be able to write into this new file. As no special character is needed for a new line, an empty string is used. After that, the structure of the CSV file has to be defined, using a comma as delimiter and defining the columns. As with the matching before, a `number` variable is used as a counter, to read all 500 files in one go. This integer gets converted to a string at the beginning of the for loop for the the combination with other strings, which ultimately defines the file path for each route. This file path then is used to open the actual GeoJSON file and extract the parameters of the routes. Each file contains 13 routes, and as in Python an index starts with 0, following order is always valid:

- 0 – Fastest route in 2014
- 1 – Fastest route in 2015
- 2 – Fastest route in 2016
- 3 – Fastest route in 2017
- 4 – Fastest route in 2018
- 5 – Fastest route in 2019

- 6 – Fastest route in 2020
- 7 – Route from 2014 matched on to the 2020 dataset
- 8 – Route from 2015 matched on to the 2020 dataset
- 9 – Route from 2016 matched on to the 2020 dataset
- 10 – Route from 2017 matched on to the 2020 dataset
- 11 – Route from 2018 matched on to the 2020 dataset
- 12 – Route from 2019 matched on to the 2020 dataset

```
with open('route_data.csv', mode='w', newline='') as write_file:
	csv_writer = csv.writer(write_file, delimiter=',')
	csv_writer.writerow(['route_id', 'time20', 'time14', 'time15', 'time16',
		'time17', 'time18', 'time19', 'dist20', 'dist14', 'dist15', 'dist16',
		'dist17', 'dist18', 'dist19'])
	for number in range(1001,1501):
		number_str = str(number)
		file_string1 = './routes/' + number_str[1:] + '_routes.geojson'
		with open(file_string1) as read_file:
			gj1 = geojson.load(read_file)
		time20 = gj1['features'][6]['properties']['duration']
		time14 = gj1['features'][7]['properties']['duration']
		time15 = gj1['features'][8]['properties']['duration']
		time16 = gj1['features'][9]['properties']['duration']
		time17 = gj1['features'][10]['properties']['duration']
		time18 = gj1['features'][11]['properties']['duration']
		time19 = gj1['features'][12]['properties']['duration']
		dist20 = gj1['features'][6]['properties']['distance']
		dist14 = gj1['features'][7]['properties']['distance']
		dist15 = gj1['features'][8]['properties']['distance']
		dist16 = gj1['features'][9]['properties']['distance']
		dist17 = gj1['features'][10]['properties']['distance']
		dist18 = gj1['features'][11]['properties']['distance']
		dist19 = gj1['features'][12]['properties']['distance']
		print(number_str[1:] + " done")
		csv_writer.writerow([number_str[1:], time20, time14, time15, time16,
		time17, time18, time19, dist20, dist14, dist15, dist16, dist17, dist18,
		dist19])
```

As the fastest route from 2020 directly acts as reference route, all other directly computed fastest routes are only included for completeness reasons and can be ignored here. All other routes are accessed as shown and both the travel time and distance are saved in a variable. For batch operations a simple progress string shows how far the extraction process has advanced. The last thing left is to write the rows in the schema that has been defined at the beginning of the script, which is done with the last `writerow` command.

After this step, the actual analysis left to do is the computation of statistical parameters in the spreadsheet software. This was done by using the most-recent values (2020) as reference values and calculating the mean difference, both absolutely and in relation. The results of this analysis are listed later in the next chapter.

5 Results

After the successful conduction of the analysis, the results are listed here, including the lessons learned that were discovered during the whole process.

5.1 Evaluation of results

Before going into the details of the results, the distribution of start and end points around Vienna should be shown in a graphic to see if the distribution is indeed random. This fact is shown in Figure 20 below, where the start points are represented by the green dots while the red dots are representing the destination points.

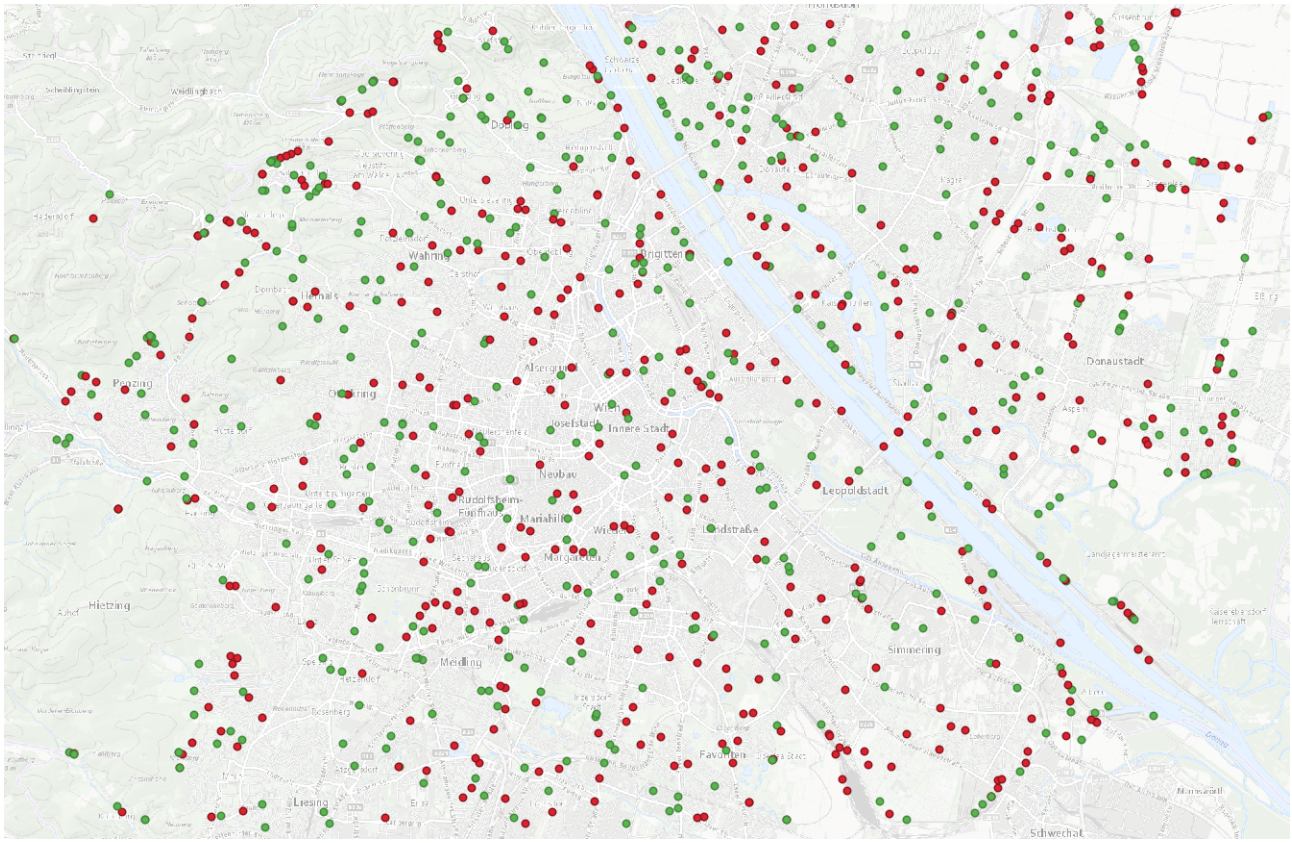


Figure 20: Distribution of start and endpoints around the survey area

As all of those 1000 points have been matched to the road network, some roads can even be reconstructed by connecting points together. Figure 20 also shows that 500 routes was a good number to represent some of the peculiarities in the road network of Vienna and to get an idea on how outdated road network data has an impact to the travel times and distances.

Adding up the changes and calculating the average extension of travel time by computing the mean, both absolute and relative to the mean travel time in 2020, using an out-dated dataset in 2020 results in following values:

Year	Absolute change [sec]	Relative change [%]
2014	+ 16.8	+ 1.47
2015	+ 14.9	+ 1.34
2016	+ 13.1	+ 1.09
2017	+ 8.6	+ 0.70
2018	+ 7.9	+ 0.65
2019	+ 4.3	+ 0.36

Table 2: Changes in travel time from using an out-dated dataset

This underlines the fact that the age of a dataset can have an effect on the travel time, which can be clearly observed. The extension in travel time is rather linear over time, as seen in Figure 21. That fact can be used to compute a polynomial fit of second degree to this data, resulting in a polynomial with the following coefficients:

$$f(x) = -0.008\,x^2 + 0.2911\,x + 0.0319$$

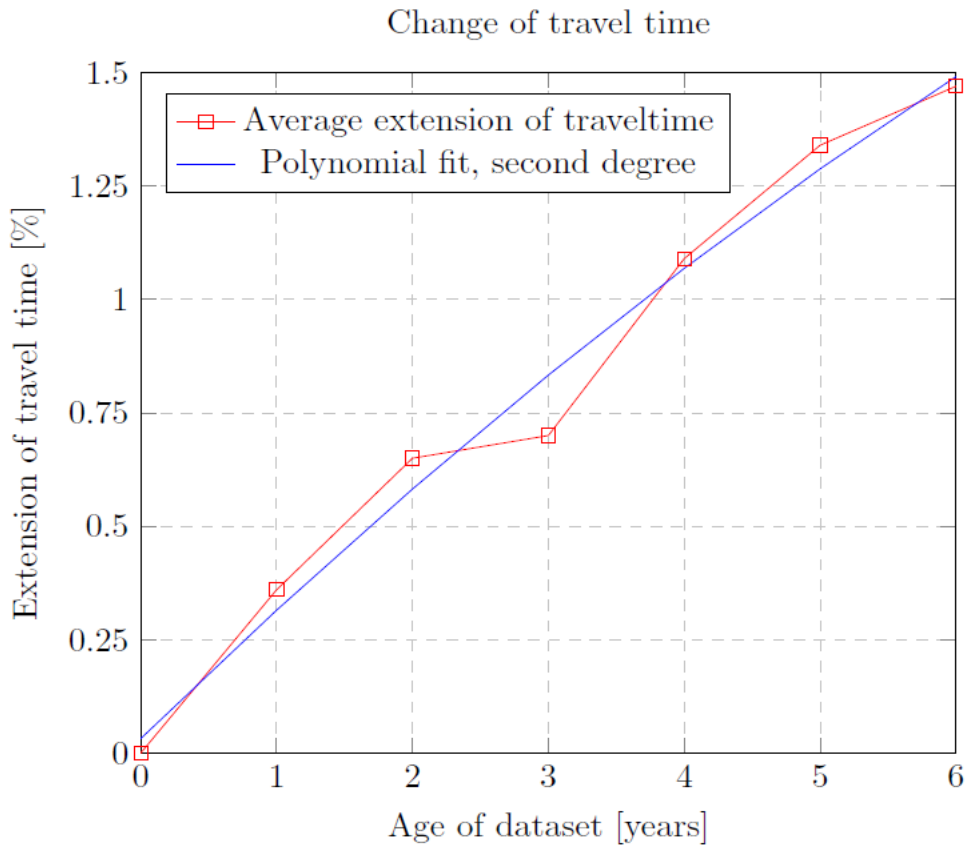


Figure 21: Change of travel time based on the age of the dataset

The same is done for the travel distance, resulting in following values:

Year	Absolute change [m]	Relative change [%]
2014	+ 3.9	+ 0.04
2015	- 39.0	- 0.15
2016	- 43.2	- 0.23
2017	- 37.0	- 0.25
2018	- 10.4	- 0.11
2019	+ 18.6	+ 0.14

Table 3: Changes in travel distance from using an out-dated dataset

This analysis shows that the mean change in travel distance is not as straight-forward as the extension of the travel time. The changes can also be negative, as a shorter way can suddenly become faster over the years, e.g., caused by changes in speed limits or turning restrictions. The change of distance might also heavily influenced by the road network of different cities or agglomerations and availability of different road types.

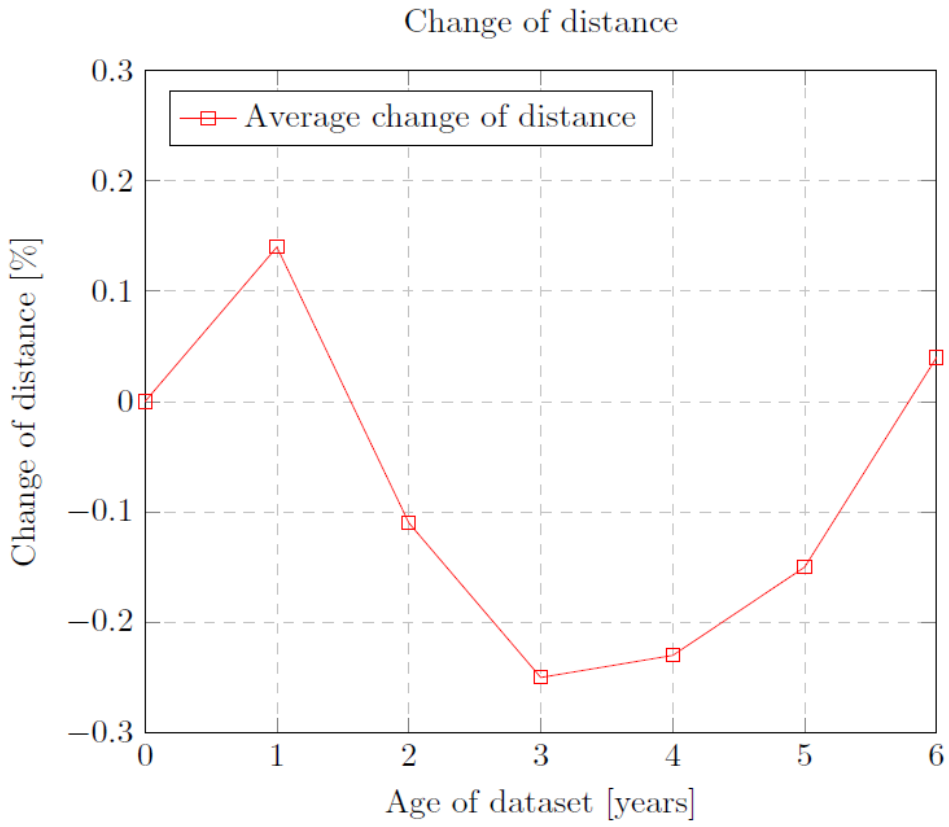


Figure 22: Change of travel distance based on the age of the dataset

5.2 Examples for route changes

During the analysis, a number of routes showed a significant change over the years, which either were automatically taken care of by the routing engine or had to be enhanced by manually calculated parts of the route. Some examples are listed here. The most common route change was based around the construction works on the motorway junction “St. Marx”. The road layout changed quite significantly over the years and that is why often no matching could be computed in several dataset and had to be revised manually in seven different route datasets, leading to noticeable changes in the travel time in some of the routes.

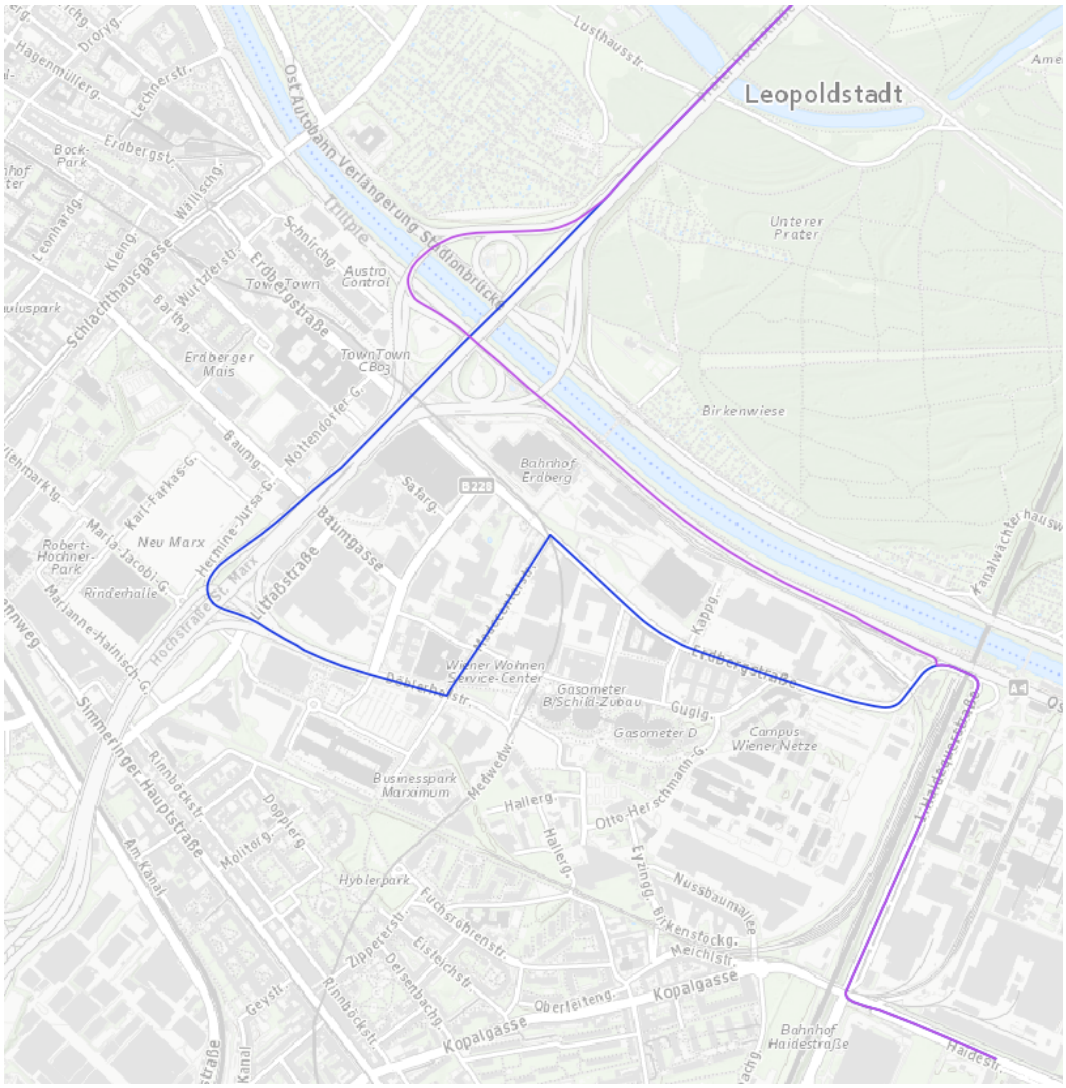


Figure 23: Example around St. Marx junction (route ID 486)

The purple route shows the fastest route after 2017, which was correctly matched to the 2020 dataset. As the used exit was further down the road in the years 2014 and 2015, this would result in a user missing the exit as the detour pictured in blue suggests. In 2016 the blue route was directly suggested as the fastest route. This results in a significant extension of travel time if the dataset from 2014, 2015 or 2016 was used in 2020.

Another example for a common incident point in this analysis was the area around the train station "Rennweg" and the adjacent road junction. In 2020 there were major construction works going on and thus some detours have been in place. This occurred in eight different route files. In the directly computed fastest route in 2020 this has been taken into account and the fastest route takes a very different route through the city. In the years 2014 from 2019 the fastest route lead through this junction. This means that the matching for these years ended there and had to be extended by a routing to the original destination. An example is shown below in Figure 24.

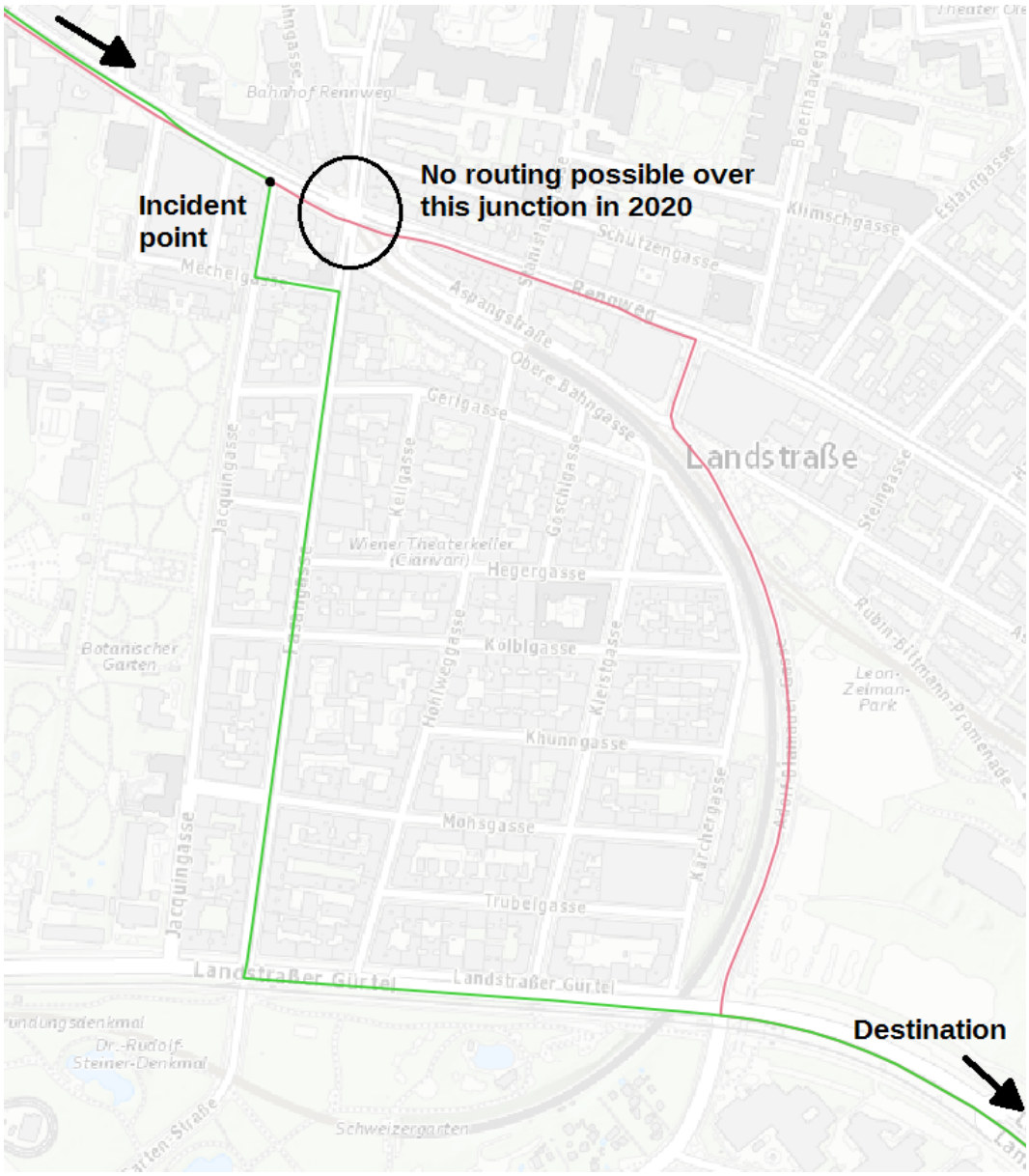


Figure 24: Example around Rennweg (Route ID 387)

This makes the routes from 2014 to 2019 (pictured in green) longer by just some seconds than the route from 2020 (not pictured). The usual fastest route from 2014 to 2019 (red) would be faster than the one in 2020, but is not possible and therefore another good example where an out-dated dataset leads to problems.

Apart from the two major special cases, 23 other minor changes in fastest routes happened in this analysis. These can have multiple factors, which can lead to changes in both the travel time and distance in very different degrees:

- U-turns are prohibited in some datasets but allowed in others, which is sometimes reflecting the real world, but could also just be missing from the dataset, affecting the completeness of the dataset.

- Various turn restrictions are present in some datasets but not in others, which is sometimes reflecting the real world, but could also just be missing from the dataset, affecting the completeness of the dataset.

- Roads are (temporarily or permanently) closed, whether reflecting the real world or just use attributes, that make it impossible to route over.

- The road layout changed significantly (see “St.Marx” example described above).

- Added or changed one-way restrictions (mostly in smaller, less-used streets).

- Changed speed restrictions, whether reflecting the limits in the real world or just changed in the attributes of the road.

Especially for the changing speed restrictions it can happen that a fastest route has a longer driving distance but a shorter travel time than a fastest route from another year. This is only logical, as a route might allow a higher maximum speed limit (e.g., motorways) and therefore allows to traverse a certain part of the route faster.

6 Conclusion

By computing enough random routes across the city of Vienna, this analysis could show that the currentness of a dataset indeed results in extended travel times, which increase depending on how out-dated the dataset really is. Following lessons were learned in the conduction of this experiment:

- The age of a spatial dataset has a demonstrable effect of the quality of the routing decision, which noticeably increases over time.
- The effect of the usage of older datasets on aspects of the solution other than the optimised one (e.g., most economical route in terms of fuel consumption) is unpredictable.
- The quality control was done completely by hand and therefore very time consuming. Parts of this analysis could be automatised with an reasonable amount of time and work.

The drawback of this analysis is the manual processing pipeline. The automation of the whole process with an implementation of all the special cases would have gone beyond the scope of this thesis. It could be automated with a reasonable amount of work and time and the usage of the confidence parameter in the matching part of the OSRM routing engine. Also other cities or even agglomerations should be investigated as the different road network layouts around the world might have an effect on the results of this thesis. Another method for assessing the quality, that is worth having a look at, is to count the total segments of a reference route and see how many segments of a computed route match those of the reference route.

It might also be interesting to investigate the effect of the path from the outdated dataset (if it is different of course). An agent-based approach could be used to make an agent follow the proposed route. If the agent is having a problem, he relies on using a specific strategy to continue a trip in a certain way, e.g., move straight ahead for a given amount of time until the route from the current position to the original destination is computed successfully. In this, different strategies might play an important role to finally provide a better assessment according to what a real user would do and therefore minimise the effect of outdated datasets on the travel time.

Finally, this approach could be used with any application that tries to answer spatial decisions based on data from the OpenStreetMap project not relying on the navigation aspect. This might, for example, include queries like searching for the best-suited hospital to the users current location and needs or finding a hiking route that matches certain user-defined criteria the most.

7 References

Boin, A., & Hunter, G. (2007). What Communicates Quality to the Spatial Data Consumer? In: A. Stein, W. Shi, & W. Bijker (Eds.), Quality Aspects in Spatial Data Mining. 285–296. https://doi.org/10.1201/9781420069273

Butler, H., Daly, M., Doyle, A., Gillies, S., Hagen, S., Schaub, T., et al. (2016). The GeoJSON Format. Internet Engineering Task Force (IETF). https://www.hjp.at/(de)/doc/rfc/rfc7946.html

Chrisman, N. R. (1984). Part 2: Issues and Problems Relating to Cartographic Data Use, Exchange and Transfer: The Role Of Quality Information In The Long-Term Functioning Of A Geographic Information System. Cartographica, 21(2–3), 79–88. https://doi.org/10.3138/7146-4332-6J78-0671

Fogliaroni, P., D'Antonio, F., & Clementini, E. (2018). Data trustworthiness and user reputation as indicators of VGI quality. Geo-Spatial Information Science, 21(3), 213–233. https://doi.org/10.1080/10095020.2018.1496556

Fonte, C. C., Antoniou, V., Bastin, L., Estima, J., Arsanjani, J. J., Bayas, J-C. L., See, L. and Vatseva, R. (2017): Assessing VGI Data Quality. In: Foody, G., See, L., Fritz, S., Mooney, P., Olteanu-Raimond, A-M., Fonte, C. C. and Antoniou, V. (Eds.) Mapping and the Citizen Sensor. 137–163. https://doi.org/10.5334/bbf.g.

Girres, J.-F., & Touya, G. (2010). Quality Assessment of the French OpenStreetMap Dataset: Quality Assessment of the French OpenStreetMap Dataset. Transactions in GIS, 14(4), 435–459. https://doi.org/10.1111/j.1467-9671.2010.01203.x

Goodchild, M. F. (2007). Citizens as sensors: The world of volunteered geography. GeoJournal, 69(4), 211–221. https://doi.org/10.1007/s10708-007-9111-y

Goodchild, M. F., & Li, L. (2012). Assuring the quality of volunteered geographic information. Spatial Statistics, 1, 110–120. https://doi.org/10.1016/j.spasta.2012.03.002

Guptill, S. C., Morrison, J. L. (1995). Elements of Spatial Data Quality. https://doi.org/10.1016/C2009-0-14900-0

Haklay, M. (2010). How Good is Volunteered Geographical Information? A Comparative Study of OpenStreetMap and Ordnance Survey Datasets. Environment and Planning B: Planning and Design, 37(4), 682–703. https://doi.org/10.1068/b35097

Heuvelink, G. B. M. (2006). Analysing Uncertainty Propagation in GIS: Why is it not that Simple? In G. M. Foody & P. M. Atkinson (Eds.), Uncertainty in Remote Sensing and GIS. 155–165. https://doi.org/10.1002/0470035269.ch10

ISO (2013). ISO 19157:2013 Geographic information - Data quality

ISO (2014). ISO 19115-1:2014 Geographic information - Metadata - Part 1: Fundamentals

Karssenberg, D., & De Jong, K. (2005). Dynamic environmental modelling in GIS: 2. Modelling error propagation. International Journal of Geographical Information Science, 19(6), 623–637. https://doi.org/10.1080/13658810500104799

Krek, A. (2002). An Agent-based Model for Quantifying the Economic Value of Geographic Information, Ph.D. thesis, Institute for Geoinformation, TU Wien

Meek, S., Jackson, M. J., & Leibovici, D. G. (2014). A flexible framework for assessing the quality of crowdsourced data. 17th AGILE Conference on Geographic Information Science, 3-6 June 2014, 3–6. http://hdl.handle.net/10234/98927

Schmidl, M., Navratil, G., Giannopoulos, I. (2021). An Approach to Assess the Effect of Currentness of Spatial Data on Routing Quality. 24th AGILE conference on Geographic Information Science (forthcoming)

Senaratne, H., Mobasheri, A., Ali, A. L., Capineri, C., & Haklay, M. (Muki). (2017). A review of volunteered geographic information quality assessment methods. International Journal of Geographical Information Science, 31(1), 139–167. https://doi.org/10.1080/13658816.2016.1189556

Tsutsumida, N., & Comber, A. J. (2015). Measures of spatio-temporal accuracy for time series land cover data. International Journal of Applied Earth Observation and Geoinformation, 41, 46–55. https://doi.org/10.1016/j.jag.2015.04.018

Will, J. (2014). Development of an automated matching algorithm to assess the quality of the OpenStreetMap road network : a case study in Göteborg, Sweden, Student thesis series INES, http://lup.lub.lu.se/student-papers/record/4464336

Yu, R., Liu, R., Wang, X., & Cao, J. (2014). Improving Data Quality with an Accumulated Reputation Model in Participatory Sensing Systems. Sensors, 14(3), 5573–5594. https://doi.org/10.3390/s140305573

Zielstra, D. and Zipf, A. (2010): A comparative study of proprietary geodata and volunteered geographic information for Germany, in: 13th AGILE International Conference on Geographic Information Science, http://agile2010.dsi.uminho.pt/pen/shortpapers_pdf/142_doc.pdf

Appendix – Source codes

- **Script 1 – Computing a set of fastest routes (*01_get_routes.py*)**

```
import numpy as np
import requests from geojson
import Point, LineString, Feature, FeatureCollection, dump

def nearest_road_point(lon, lat, year):
	loc = "{},{}".format(lon, lat)
	y = year + 3000
	url = "http://127.0.0.1:" + str(y) + "/nearest/v1/driving/"
	r = requests.get(url + loc)
	if r.status_code!= 200:
		return {}
	res = r.json()
	nearest_point = res['waypoints'][0]['location']
	return nearest_point

def get_route(start_lon, start_lat, end_lon, end_lat, year):
	loc = "{},{};{},{}".format(start_lon, start_lat, end_lon, end_lat)
	y = year + 3000
	url = "http://127.0.0.1:" + str(y) + "/route/v1/driving/"
	url2 = "?overview=full&geometries=geojson"
	r = requests.get(url + loc + url2)
	if r.status_code!= 200:
		return {}
	res = r.json()
	geometry = res['routes'][0]['geometry']
	coords = res['routes'][0]['geometry']['coordinates']
	distance = res['routes'][0]['distance']
	duration = res['routes'][0]['duration']
	start_point = [res['waypoints'][0]['location'][1],
			res['waypoints'][0]['location'][0]]
	end_point = [res['waypoints'][1]['location'][1],
			res['waypoints'][1]['location'][0]]
	out = { 'geometry':geometry, 'coords':coords, 'distance':distance,
		'duration':duration, 'start_point':start_point, 'end_point':end_point }
	return out
```

```
# random coords
lons = np.random.uniform(16.22,16.53,2).round(6)
lats = np.random.uniform(48.14,48.28,2).round(6)
start_point = nearest_road_point(lons[0], lats[0], 2020)
end_point = nearest_road_point(lons[1], lats[1], 2020)
features2 = []
for year in range(2014,2021):
	req = get_route(start_point[0], start_point[1], end_point[0], end_point[1], year)
	route_json = req['geometry']
	route_time = req['duration']
	route_dist = req['distance']
	features2.append(Feature(geometry=route_json, properties={"year": year,
		"duration": route_time, "distance": route_dist}))

feature_collection2 = FeatureCollection(features2)
with open('id_fastest.geojson', 'w') as file:
	dump(feature_collection2, file)
```

- **Script 2 – Matching the fastest routes (*02_match_routes.py*)**

```
import requests
from geojson import Point, LineString, Feature, FeatureCollection, dump, load

def route_list_to_string(ref_route_list):
	ref_route_string = ""
	for c in range(0,len(ref_route_list[:-1])):
		this_c = ref_route_list[c]
		next_c = ref_route_list[c+1]
		loc = "{},{};{},{}".format(this_c[0], this_c[1], next_c[0], next_c[1])
		ref_route_string = ref_route_string + loc + ";"
	return ref_route_string[:-1]

def match_route(coord_string, year):
	y = year + 3000
	url = "http://127.0.0.1:" + str(y) + "/match/v1/driving/" + coord_string + "?
		overview=full&geometries=geojson"
	r = requests.get(url)
	if r.status_code!= 200:
		return {}
	res = r.json()
	geometry = res['matchings'][0]['geometry']
	distance = res['matchings'][0]['distance']
	duration = res['matchings'][0]['duration']
	confidence = res['matchings'][0]['confidence']
	try:
		start_point = [res['tracepoints'][0]['location'][1],
			res['tracepoints'][0]['location'][0]]
	except:
		start_point = [0,0]
	try:
		end_point = [res['tracepoints'][-1]['location'][1],
			res['tracepoints'][-1]['location'][0]]
	except:
		end_point = [0,0]
	out = { 'geometry':geometry, 'distance':distance, 'duration':duration,
		'confidence':confidence,'start_point':start_point,'end_point':end_point }
	return out
```

```
number_array = [1001]
for number in number_array:
	number_str = str(number)
	file_string1 = number_str[1:] + '_fastest.geojson'
	features = []
	with open(file_string1) as read_file:
		gj1 = load(read_file)
	fastest_list = []
	for i in range(0,7):
		fastest_geometry = gj1['features'][i]['geometry']
		fastest_route = route_list_to_string(gj1['features'][i]['geometry']
			['coordinates'])
		fastest_time = gj1['features'][i]['properties']['duration']
		fastest_distance = gj1['features'][i]['properties']['distance']
		fastest_list.append(fastest_route)
		features.append(Feature(geometry=fastest_geometry,
			properties={"year": i+2014, "duration": fastest_time,
				"distance": fastest_distance}))
	year = 2014.2020
	for fastest in fastest_list:
		if year == 2020.2020:
			break
		req_year = match_route(fastest, 2020)
		year_route_json = req_year['geometry']
		year_route_time = req_year['duration']
		year_route_dist = req_year['distance']
	features.append(Feature(geometry=year_route_json, properties={"year": str(year)
		+"0", "duration": year_route_time, "distance": year_route_dist}))
		year = year + 1

	feature_collection = FeatureCollection(features)
	file_string2 = number_str[1:] + '_routes.geojson'
	with open(file_string2, 'w') as write_file:
		dump(feature_collection, write_file)
	print("File " + number_str[1:] + ' done!')
```

- **Script 3 – Matching the fastest routes (*02_match_routes.py*)**

```
import geojson
import csv
with open('route_data.csv', mode='w', newline='') as write_file:
	csv_writer = csv.writer(write_file, delimiter=',')
	csv_writer.writerow(['route_id', 'time20', 'time14', 'time15', 'time16',
		'time17', 'time18', 'time19', 'dist20', 'dist14', 'dist15', 'dist16',
		'dist17', 'dist18', 'dist19'])
	for number in range(1001,1501):
		number_str = str(number)
		file_string1 = './routes/' + number_str[1:] + '_routes.geojson'
		with open(file_string1) as read_file:
			gj1 = geojson.load(read_file)
	time20 = gj1['features'][6]['properties']['duration']
	time14 = gj1['features'][7]['properties']['duration']
	time15 = gj1['features'][8]['properties']['duration']
	time16 = gj1['features'][9]['properties']['duration']
	time17 = gj1['features'][10]['properties']['duration']
	time18 = gj1['features'][11]['properties']['duration']
	time19 = gj1['features'][12]['properties']['duration']
	dist20 = gj1['features'][6]['properties']['distance']
	dist14 = gj1['features'][7]['properties']['distance']
	dist15 = gj1['features'][8]['properties']['distance']
	dist16 = gj1['features'][9]['properties']['distance']
	dist17 = gj1['features'][10]['properties']['distance']
	dist18 = gj1['features'][11]['properties']['distance']
	dist19 = gj1['features'][12]['properties']['distance']
	print(number_str[1:] + " done")
	csv_writer.writerow([number_str[1:], time20, time14, time15, time16, time17,
		time18, time19, dist20, dist14, dist15, dist16, dist17, dist18, dist19])
```