UNIVERSITY OF ZAGREB

FACULTY OF GEODESY

Filip Todić

QUANTIFICATION AND QUALIFICATION OF CHANGES IN THE OPENSTREETMAP PROJECT

Master’s thesis

Zagreb, 2015

I. Author

Name and surname: Filip Todić

II. Master's thesis

Subject: Spatial data analysis

Title: Quantification and qualification of changes in the OpenStreetMap project

Mentor: Distinguished professor Damir Medak, PhD

Advisor: Mario Miler, PhD

III. Grade and defense

Assignment issue date: January 15th, 2015

Defense date: July 3rd, 2015

Master's thesis defense committee:

Distinguished professor Damir Medak, PhD

Docent Robert Župan, PhD

Mario Miler, PhD

Acknowledgements

People often choose celebrities or other famous people as their role models, the most important criteria often being professional achievements or financial success. Although there is nothing wrong in admiring wealthy and successful people, I have always found it much better to admire people who have made a substantial impact on my life.

My parents are one of them. Someone told me that being a parent is the most difficult job there is. Thinking about the sacrifices my parents have made for me, I have to agree. There are no words that can adequately express my gratitude. They are, and always will be, my true role models.

The definition of mentor is "*a wise and trusted counselor or teacher*". These words perfectly describe professor Medak and his two assistants, Mario Miler and Dražen Odobašić. I have collaborated with them on several projects and they never cease to amaze me. Their dedication to their work and science in general is nothing short of amazing, but that is topped by their treatment of students and other colleagues who have come to them for help. With their help, I have learned that it is more important to share knowledge as opposed to possessing it.

Finally, I would like to thank my colleague, Nikolina Vidonis, for her support and understanding.

Abstract

The OpenStreetMap project has developed into one of the largest VGI datasets today. Over the years, its quality was analyzed and compared to commercial or authoritative datasets by various researchers and in different scenarios. Time and again, it proved to be a reliable alternative to commercial and authoritative datasets, despite being maintained mostly by amateurs and enthusiasts with little or no background in cartography or spatial sciences. After a decade of admirable results, a keen interest has arisen in the evolution of the project, most notably in the activities of its members and the changes they leave behind them. The aim of this research is to develop a generic model that would aid researchers and the OSM community in determining the nature of changes present in OSM objects, as well as their currency and credibility. The model described in this paper is able to analyze all OSM objects, regardless of their geometry type, location and semantic attributes.

Key words: OpenStreetMap, quantification of changes, qualification of changes, spatial data currency

Sažetak

OpenStreetMap projekt razvio se u jedan od najvećih skupova VGI podataka današnjice. Tijekom proteklih godina analizirana je njegova kvaliteta i brojni istraživači proveli su usporedbe s komerijalnim i službenim skupovima podataka u raznim slučajevima. Projekt se često pokazao kao pouzdana alternativa komercijalnim i službenim skupovima podataka, usprkos činjenici da ga održava zajednica amatera i entuzijasta koji posjeduju malo ili nimalo znanja o kartografiji i prostornim znanostima. Nakon desetljeća sjajnih rezultata, pojavio se interes za tijek razvoja samog projekta, posebno u vezi aktivnosti njegovih članova i promjena koje uzrokuju. Cilj ovog istaživanja je razvoj generičkog modela koji bi pomogao istraživačima i OSM zajednici u određivanju prirode promjena prisutnih u OSM objektima te njihovoj ažurnosti i kredibilitetu. Model koji je objašnjen u ovom radu u mogućnosti je analizirati OSM objekte, neovisno o tipu geometrije, lokaciji i semantičkim atributima.

Ključne riječi: OpenStreetMap, kvantifikacija promjena, kvalifikacija promjena, ažurnost prostornih podataka

Table of Contents

1. Introduction

The term *Web 2.0* was first introduced at the beginning of the 21st century. Unlike its predecessor *Web 1.0*, which was primarily characterized by the consumption of predefined content, the term *Web 2.0* relates to a platform where users can customize their own applications on the World Wide Web (WWW) and more importantly, create their own data and edit existing data. The online encyclopedia *Wikipedia*, where volunteers share their knowledge on various topics, is based on this phenomenon. This approach is referred to as *user-generated content* (UGC) and is present on other websites such as *YouTube* and *Flickr* (Neis & Zipf, 2012).

One of the most interesting aspects of *Web 2.0* was the emergence of crowdsourced information, which also represents one of the most significant and potentially controversial developments in *Web 2.0*. Crowdsourcing refers to large groups of users performing functions that are either difficult to automate or expensive to implement. It can be used in large-scale community activities focused on the development of software or on the collection and sharing of information. These activities are carried out by large groups of volunteers who work independently and without much coordination (Haklay, 2010).

Similar efforts are the foundation of the OpenStreetMap (OSM) project. Unlike other platforms that rely on user contributions in form of collected information about a particular subject, the OSM project contains more specific details about spatial elements (e.g. streets, buildings, etc.) which always include a geographic reference. This type of data is often described as *Volunteered Geographic Information* (VGI), while the whole process is described as *crowdsourcing geospatial data* (Neis & Zipf, 2012).

The OSM project has developed into one of the largest sources of VGI in recent years, and with the change of the licensing model by Google Maps in early 2012, more and more businesses are moving toward the free option offered by the OSM project. The location-based social network *FourSquare* and the *Nestoria Property Search* are the two major examples. Furthermore, professional spatial data providers and companies have created their own platforms which allow users to edit their own data on the provided maps (e.g. Google Map Maker, TomTom Map Share). These developments show that the success of the VGI approach to data collaboration and sharing is undeniable (Neis & Zipf, 2012). Recently, OSM

has been the focus of many new developments such as routing applications, 3D city models and *Location-Based Services* (LBS) (Neis, et al., 2011).

On the other hand, most of the VGI projects (including OSM) rely on volunteers that do not necessarily have professional qualifications and background in geodata collection or surveying. Contribution to the project largely depends on the technical aspects (e.g. PC, Internet connection, GPS receiver, Smartphone, etc.) as well as the population density of specific areas. However, the local knowledge of most participants should make them local experts (Neis, et al., 2011). In light of the data collection by amateurs, the distributed nature of data collection and the loose coordination in terms of standards, one of the significant questions about VGI is the quality of information collected through such activities (Haklay, 2010).

Recently, the quality of the OSM project has been analyzed in several studies by comparing the OSM dataset to authoritative or commercial datasets. Haklay (2010) analyzed the quality of the OSM street and road network by comparing it with the Ordnance Survey (OS) dataset. The results of this research indicate that OSM information can be fairly accurate: on average within 6 m of the position recorded by the Ordnance Survey and with approximately 80% of overlap of motorway objects.

Neis et al. (2011) compared the German OSM street network with a proprietary dataset provided by TomTom. At the time of publication, the difference between the OSM street network for car navigation in Germany and the TomTom dataset was only 9%. Furthermore, the OSM dataset exceeds the information provided by the proprietary dataset in some areas by 27%. An analysis regarding topological errors and completeness of street name information showed that the OSM dataset is not flawless, but the trend shows that the relative and absolute number of errors is decreasing.

Despite the good results and evident improvements, a few concerns still remain. Haklay (2010) noticed the lack of coverage in rural and poorer areas. Furthermore, the temporal issue is of special interest to VGI. Due to the leisure-activity aspect of the involvement in such projects, the longevity of engagement can be an issue, depending on the enthusiasm of its participants. OSM is still going through a period of rapid growth and it is important to note that many other commons-based peer-production projects are able to engage participants over longer periods of time, as shown by the Apache Web server and Wikipedia.

Neis and Zipf (2012) analyzed the contributor activity in the OSM project and came to the conclusion that only 38% of the registered users carried out at least one edit and that only 5% of all members actively contributed to the project in a more productive way.

As to the temporal changes, Mooney and Corcoran (2012) analyzed the characteristics of heavily edited objects in the OSM project in hopes of informing potential consumers of OSM data that the data itself is changing over time. The heavily edited objects in question refer to OSM objects which have been edited 15 or more times. Their results indicate that there is no strong relationship between the increasing number of contributors to a given object and the number of tags assigned to it.

In this research, temporal, contextual and geometric changes were evaluated. The quantification of changes refers to the act of counting or measuring the magnitude of certain types of changes. Since the magnitude of certain changes is not enough and can sometimes be misleading, the qualification of changes is important for determining the level of change present in certain objects. For instance, frequently visited places can experience high amounts of changes that mostly consist of correcting certain typographical errors, while places visited less frequently can experience lower amounts of significant changes, such as changes in names or operators.

Considering the fact that the OSM project has been active for more than a decade and its undeniable increase in quality was confirmed by the aforementioned researches, the purpose of this research is to determine the quality of spatial data by analyzing the changes in the OSM project. For this reason, a generic model was developed and tested on several types of OSM objects. The model in question was designed to detect and analyze changes in OSM tags as well as the geometries of OSM objects and implement present measures of spatial data quality.

2. Materials

In order to develop a generic model capable of analyzing all OSM objects, several open source technologies were used. First of all, the study area in this research is the Republic of Croatia. Several OSM PBF (*Protocolbuffer Binary Format*) files pertaining to the study area were downloaded from the Planet OSM website (URL1). Given the magnitude of these datasets, only several types of OSM objects were extracted from the downloaded PBF files and imported into a PostgreSQL/PostGIS database. The model was implemented in the Python programming language.

2.1. OpenStreetMap

The OSM project was founded in 2004 at the University College London. Its objective is to create a free database with geographic information of the entire world. For this purpose, a set of detailed instructions was published on the OSM Wiki pages. A wide range of spatial data such as roads, buildings, land use areas or points of interest (POI) are entered into the project's database on a daily basis. There are several ways of contributing new data to the project. The most common approach is to record data using a GPS receiver and to edit the collected data using one of the various freely available editors. The second approach is the digitization of streets from satellite images provided by companies such as Yahoo or Microsoft Bing. The third approach is the import of other freely available data, such as the TIGER dataset of the United States. The fourth and final approach refers to the local knowledge of the contributor (Neis & Zipf, 2012).

As to retrieving data from the OSM project, there are three different methods. The first method refers to the OSM *dump files* which include the latest versions of the objects stored in the OSM database. These files are updated on a weekly basis. The second method refers to a complete database dump file with all available versions of the objects. This file is released once every quarter. The third method provides *diff files* that contain the latest changes to the database (Neis & Zipf, 2012).

The geographic information in the OSM database is stored by using three object types: *Nodes*, *Ways* and *Relations*. A *Node* contains the location information of a point in the form of latitude and longitude. Lines are stored as *Ways* and *Relations* which define logical or geographic relationships between the objects. The term *Ways* includes both polylines and polygons. Furthermore, each OSM object contains additional information such as version

number, ID, creation or modification date, the name of the editor and further attributes stored in *Tag/Value* pairs (Neis & Zipf, 2012).

An OSM object can have any number of tags. The OSM wiki contains a page that describes the most popular features and tags in detail (URL2). Contributors are free to add their own arbitrary tags if necessary. However, only the tags listed on the *Map Features* list are the ones usually supported by GIS software capable of consuming OSM data and cartographic software for rendering OSM data as map image tiles (Mooney & Corcoran, 2012).

The PBF format in which the OSM data were downloaded is primarily intended as an alternative to the XML (*Extensible Markup Language*) format. The format was designed to support future extensibility and flexibility and is superior to GZIP and BZIP file formats in terms of compression size while at the same time being 5x faster to write 6x faster to read than a gzipped planet (URL3).

2.2. PostgreSQL/PostGIS

In this research, the PostgreSQL database was used for OSM data storage. In order to manipulate spatial data, PostGIS was used.

2.2.1. PostgreSQL

PostgreSQL is the world's most advanced open source object-relational database system. It has more than 15 years of active development and a strong reputation for reliability, data integrity and correctness. It runs on all major operating systems, including Linux, UNIX and Windows. It is fully ACID (*Atomicity, Consistency, Isolation, Durability*) compliant, includes most SQL:2008 data types, has full support for foreign keys, joins, views, etc. It has an exceptional documentation and native programming interfaces for several programming languages, including C++, Java and Python (URL4).

Furthermore, PostgreSQL supports compound, unique, partial, and functional indexes which can use any of its B-tree, R-tree or GiST storage methods. GiST (Generalized Search Tree) indexing is an advanced system which brings together a wide array of different sorting and searching algorithms including B-tree, B+-tree, R-tree, partial sum trees and many others. It also serves as a foundation for many public projects that use PostgreSQL such as PostGIS (URL4).

2.2.2. PostGIS

PostGIS is a spatial database extension for the PostgreSQL database. It adds support for geographic objects allowing location queries to be run in SQL. It offers many features rarely found in other competing spatial databases such as Oracle Locator/Spatial and SQL Server. PostGIS adds extra types to the PostgreSQL database (e.g. geometry, geography, raster) as well as functions, operators and index enhancements that apply to these spatial types. These additions augment the power of the core PostgreSQL DBMS, making it a fast and robust spatial database management system (URL5).

2.2.3. Hstore

One of the key data types used in this research is the PostgreSQL's *hstore* data type. The hstore module implements the hstore data type for storing sets of key/value pairs within a single PostgreSQL value. It is useful in case of rows with many attributes that are rarely examined or semi-structured data (URL6).

Keys and values are text strings. The text representation of an hstore includes zero or more *key=>value* pairs separated by commas, while the order of the pairs is insignificant. Each key in an hstore is unique. If an hstore with duplicate keys is declared, only one will be store in the hstore without guarantee as to which will be kept (URL6).

2.3. Python

The developed model was implemented in the Python programming language. In this process several Python modules were used. First of all, *GDAL/OGR* API was used to read the input data and to write the output data in GeoJSON format. The *SQLAlchemy* module was used for interactions with the database. In order to manipulate spatial data, *GeoAlchemy2* module was used as well. The *Argparse* module was used to develop a command-line interface for the input of arguments. This section presents a brief overview of the Python programming language and the aforementioned modules.

Python is an interpreted, interactive and object-oriented programming language that incorporates modules, exceptions, dynamic typing, very high level dynamic data types and classes. It combines remarkable power with very clear syntax. It is also portable: it runs on many Unix variants, Mac and PCs (URL7).

Furthermore, Python is a high-level general-purpose programming language that can be applied to many different classes of problems. It comes with a large standard library that covers areas such as string processing (e.g. regular expressions, Unicode), Internet protocols (e.g. HTTP, FTP), software engineering (e.g. unit testing, logging, etc.), and operating system interfaces (e.g. filesystems, TCP/IP sockets). Several Linux distributions (e.g. Red Hat) have written part or all of their installer and system administration software in Python. Companies that use Python include Google, Yahoo and Lucasfilm Ltd (URL7).

2.3.1. SQLAlchemy

SQLAlchemy is the Python SQL toolkit and Object Relational Mapper (ORM) that gives application developers the full power and flexibility of SQL. It provides a full suite of well-known enterprise-level persistence patterns designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language (URL8).

The reasons behind the development of SQLAlchemy are that SQL databases behave less like object collections the more size and performance starts to matter and object collections behave less like tables and rows the more abstraction starts to matter. SQLAlchemy considers the database to be a relational algebra engine. Rows can be selected from tables, joins and other statements. It is most famous for its ORM, an optional component that provides the data mapper pattern where classes can be mapped to the database in multiple ways, thus allowing the object model and database schema to develop in a cleanly decoupled way from the beginning (URL8).

2.3.2. GeoAlchemy2

GeoAlchemy 2 provides extensions to SQLAlchemy for working with spatial databases and is primarily focused on PostGIS. It supports PostGIS' *geometry*, *geography* and *raster* types and adds *to_shape* and *from_shape* functions for a better integration with Shapely (URL9).

2.3.3. GDAL/OGR

Geospatial Data Abstraction Library (GDAL/OGR) is a cross platform C++ translator library for raster and vector geospatial data formats. It is released under an X/MIT style Open Source license by the Open Source Geospatial Foundation. GDAL supports over 50 raster formats and OGR over 20 vector formats. It provides the primary data access engine for many applications (e.g. QGIS, GRASS) and is the most widely used geospatial data access library.

Some of its features include library access from Python, Java, Ruby; a vector data model closely aligned with OGC Simple Features; and a coordinate system engine built on PROJ.4 and OGC Well Known Text coordinate system descriptions. It supports a number of vector formats, including ESRI Shapefile, PostGIS and GML (URL10).

2.3.4. Shapely

Shapely is a Python package for set-theoretical analysis and manipulation of planar features using functions from the well-known and widely deployed GEOS (*Geometry Engine Open Source*) library. GEOS is a C++ port of the Java Topology Suite (JTS) and is the geometry engine of the PostGIS spatial extension for the PostgreSQL RDBMS (*Relational Database Management System*). The designs of JTS and GEOS are largely guided by the Open Geospatial Consortium's Simple Features Access Specification and Shapely remains devoted to the same set of standard classes and operations. Shapely's first premise is that Python programmers should be able to perform PostGIS type geometry operations outside of an RDBMS (URL11).

2.3.5. Argparse

The Argparse module is a parser for command line options, arguments and sub-commands. This Python module makes it easy to write user-friendly command-line interfaces. It also automatically generates help and usage messages and issue errors when users give the program invalid arguments (URL12).

2.4. String matching techniques

The changes in certain OSM tags were evaluated using two well-known string matching techniques, the Jaro-Winkler and the Levenshtein distance. These two algorithms are normally used for the identification of data records that refer to equivalent entities but come from heterogeneous information sources. Furthermore, records that describe the same object might differ syntactically. Variations in representation across information sources can arise from differences in formats that store data, typographical errors, and abbreviations. Considering the fact that individual records are often stored as strings, functions that accurately measure the similarity between two strings are important in duplicate identification (Bilenko, et al., 2003).

2.4.1. Levenshtein distance

An important class of string matching metrics are *edit distances*, in this case the Levenshtein distance. The distance between two strings *s* and *t* is the cost of the best sequence of *edit operations* that converts *s* to *t*. The strings are mapped using these edit operations (Bilenko, et al., 2003):

- *Copy* the next letter in *s* to the next position in *t*;
- *Insert* the new letter in *t* that does not appear in *s*;
- *Substitute* a different letter in *t* for the next letter in *s*;
- *Delete* the next letter in *s* (i.e. do not copy it to *t*).

Levenshtein distance is defined as the minimal number of characters that have to be replaced, inserted or deleted in order to transform one string to another (Mooney & Corcoran, 2012). Edit distance metrics are widely used in text processing as well as biological sequence alignment and many variations are possible (Bilenko, et al., 2003).

2.4.2. Jaro-Winkler metrics

Another effective similarity metric is the *Jaro metric.* It is based on the number and order of common characters between two strings *s* and *t*. William Winkler proposed a variant of the Jaro metric that also uses the length of the longest common prefix of *s* and *t* which emphasizes matches in the first few characters. The Jaro and Jaro-Winkler metrics are intended primarily for short strings, such as personal names (Bilenko, et al., 2003). The Jaro-Winkler metric is normalized in such a way that 0 equates to no similarity and 1 is an exact match between two strings (Mooney & Corcoran, 2012).

2.5. Spatial data currency

Currency, or *data validity*, is a parameter of how up-to-date the data are. It can be viewed as a form of semantic accuracy which is a parameter of how well a real-world object is actually described in the data. Other forms of semantic accuracy include attribute accuracy, completeness and consistency (Devillers & Jeansoulin, 2010).

From a data producer's perspective, currency is a parameter of data capture and update policy. For instance, Ordnance Survey's update policy for large scale vector data distinguishes

between two categories of change. The first category concerns features of significant business and national interest, such as housing areas, commercial and industrial developments and communication networks. These features are captured within six months of construction under a *continuous revision* policy. The second category concerns features of generally lower benefit for customers, such as vegetation and small building-extensions. These features are captured within a cyclic revision process every five years, or ten years in mountain and moorland areas (Devillers & Jeansoulin, 2010).

Achievement of these currency targets is only as good as the organization's knowledge of real world change. Measurement of conformance depends upon quantifying and comparing known real-world change and with captured change in the data. Detection and quantification of real-world changes relies on a combination of local knowledge, third-party information sources and the potential automatic change detection from the available imagery. Although it is presently difficult for a national mapping organization to achieve the ultimate goal of 100% currency, levels of conformity aim to meet or exceed *acceptable quality levels* (Devillers & Jeansoulin, 2010).

Within the captured vector dataset itself, each feature is attributed with the date it was last updated. However, this does not necessarily inform the user about currency. For instance, a building feature last update in 1950 may still be up to date with respect to the real world and current update policy. From a user's perspective, it is important to note that the vector feature's date of update primarily relates to the suppliers capture policy, and not with the date at which the change actually occurred in the real-world. This difference could present a significant factor in the assessment of data quality (Devillers & Jeansoulin, 2010).

2.6. Spatial data processing

In order for this model to be completely generic, various characteristics of spatial objects need to be considered. The OSM project (along with Google Maps, Bing maps, etc.) uses the Web Mercator projection, which is also known as Google Web Mercator, Spherical Mercator or WGS84/Pseudo-Mercator. Web Mercator is a special case of Mercator on a sphere and projected from latitude and longitude coordinates from the World Geodetic System 1984 (WGS84) ellipsoid. Since it uses a spherical Earth, the difference between these two projections manifests itself as a function of latitude (Battersby, et al., 2014).

Web Mercator is a good choice for online mapping, particularly at a global scale, because it simplifies the standard Mercator projection by mapping the Earth onto a sphere which allows simpler and faster calculations. It also readily supports Web map service requirements for indexing the world map and allowing continuous panning and zooming to any area, location and scale. Although it is not technically a conformal projection, the visual difference between Web Mercator and Mercator is non-existent, and for most general purpose mapping the distortions to local angles are minimal (Battersby, et al., 2014).

However, the technical issues with Web Mercator can be seen in the *km/deg* measurements and area measurements. The difference between a map's *km/deg* calculation and the real-Earth values varies considerably as a direct result of the projection used. This difference can be small in some cases (e.g. less than 1 *km/deg* for Sinusoidal Equal Area), but the values for Web Mercator can reach nearly 100 *km/deg* of difference in high altitudes. These issues increase in severity as one moves closer to the poles (Battersby, et al., 2014).

Considering these issues, the Web Mercator projection is unfit for spatial calculations in this research. Since the study area is the Republic of Croatia, another possible solution is to transform the downloaded OSM data from WGS84 to the local Mercator projection, i.e. the official Croatian projection, HTRS96/Croatia TM. Unlike the WGS84 which uses degrees (latitude and longitude) as a unit, the Croatia TM uses meters (Easting and Northing) as a unit. This makes the Croatia TM optimal for metric calculations used for calculating areas and distances.

However, in order for the model to be applicable worldwide, without using transformations between projections, another approach using PostGIS was taken. Unlike coordinates in Mercator, UTM or Stateplane, geographic coordinates (the coordinate format of the downloaded OSM data) are not Cartesian coordinates. Geographic (spherical) coordinates do not represent a linear distance from an origin as plotted on a plane, but angular coordinates on a globe. In spherical coordinates a point is specified by the angle of rotation from a reference meridian and the angle from the equator, i.e. latitude and longitude (Figure 1) (URL13).

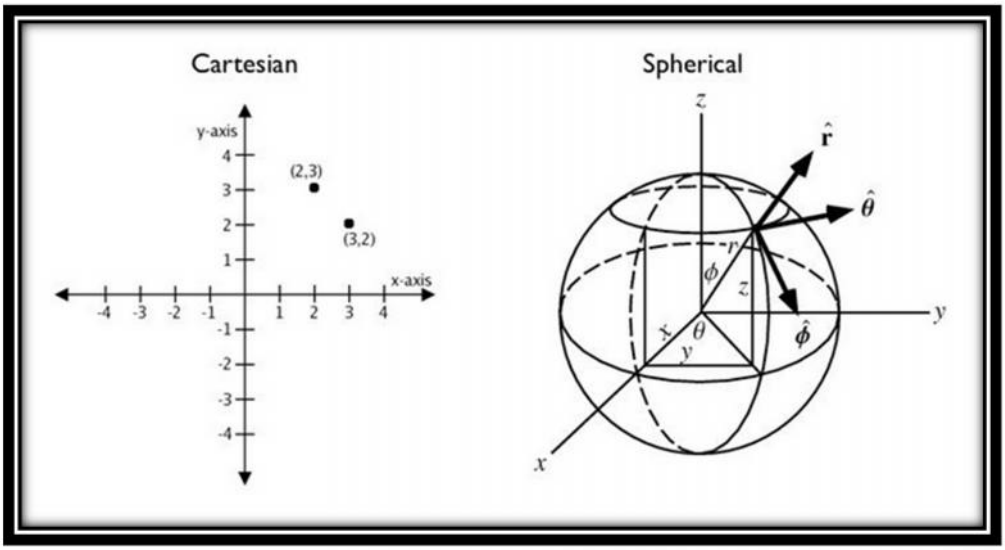

Figure 1. *A comparison of the Cartesian and the Spherical coordinates* (URL13)

Geographic coordinates can be treated as approximate Cartesian coordinates and spatial calculations can be made. However, measurements of distance, length and area are nonsensical. Considering that spherical coordinates measure angular distance, the units are in degrees. Furthermore, the approximate results from indexes and True/False tests, such as intersections, can become terribly wrong. The distances between points increase as problematic areas like the poles or the international dateline are approached (URL13).

In order to calculate a meaningful distance, geographic coordinates must not be treated as approximate Cartesian coordinates but rather as true spherical coordinates. The distances between points must be measured as true paths over a sphere, i.e. a portion of a great circle (Figure 2). As of version 1.5, PostGIS provides this functionality through the *geography* type (URL13).

Figure 2. *Comparison of distances between Los Angeles and Paris calculated using PostGIS (purple – geographic coordinates on a Cartesian plane; red – great circle route)* (URL13)

Furthermore, the Cartesian approach to handling geographic coordinates breaks down entirely for features that cross the international dateline (Figure 3). The shortest great circle route from Los Angeles to Tokyo crosses the Pacific Ocean. The shortest Cartesian route crosses the Atlantic and Indian Oceans (URL13).

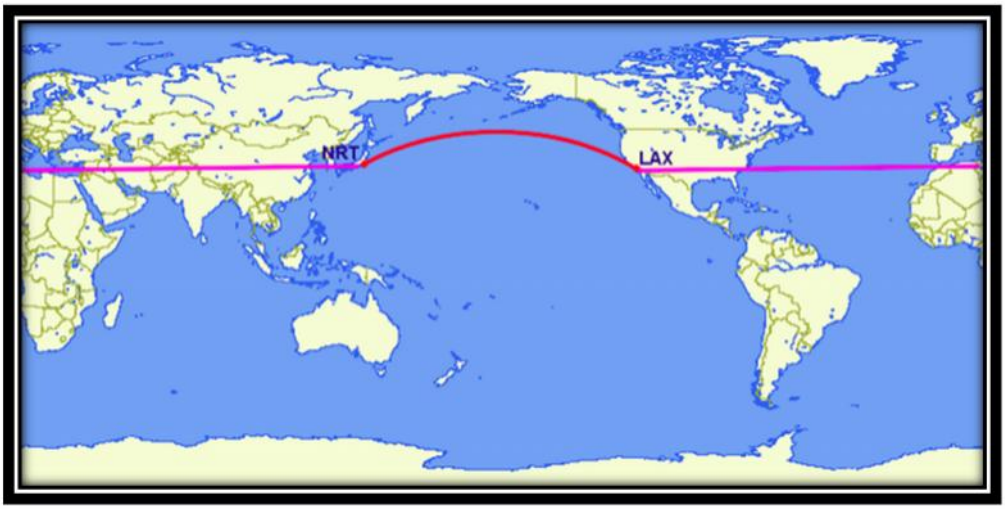

Figure 3. *Comparison of distances between Los Angeles and Tokyo using PostGIS (purple – geographic coordinates on a Cartesian plane; red – great circle route)* (URL13)

PostGIS uses two spatial types, *geometry* and *geography*. In order to load geometry data into a geography table, the geometry first needs to be projected into EPSG:4326 (longitude/latitude) using the *ST_Transform(geometry, srid)*. Then it needs to be changed into geography using the *Geography(geometry)* function which casts them from geometry to geography. Building a spatial index for geography is exactly the same as for geometry. The

difference between these two types is under the covers: the geography index will correctly handle queries that cover the poles or the international dateline, while the geometry index will not (URL13).

Despite the universal acceptability and familiarity with geographic coordinates, the geography type has two disadvantages. Firstly, a small number of functions is currently available that directly supports the geography type. Secondly, the calculations on a sphere are computationally far more expensive than Cartesian calculations. For instance, the Cartesian formula for distance (Pythagoras) involves one call to sqrt(). On the other hand, the spherical formula for distance (i.e. Haversine distance) involves two sqrt() calls, an arctan() call, four sin() calls and two cos() calls. Spherical calculations involve many trigonometric functions which are very costly (URL13).

3. Methodology

Given the size, heterogeneity and complexity of the OSM dataset, only several types of OSM features were analyzed. The OSM PBF files downloaded from the Planet OSM website are daily dump files that represent the Croatian OSM dataset at the time of their creation. The analyzed time frame spans from January 1st, 2013 to March 1st, 2015, with three month time intervals. In order to extract data from the previously downloaded PBF files, two open source programs were used: *osmconvert* and *osmfilter*.

3.1. OSMconvert

The *osmconvert* program can be used to convert and process OSM files. Unlike the commonly used Osmosis, it offers no way of accessing the database. However, it is faster and offers several special functions. The program was designed to run from the command line. It can be downloaded and built by running the command shown in Figure 4 (URL14).

```
wget -O - http://m.m.i24.cc/osmconvert.c | cc -x c - -lz -O3 -o osmconvert
```

Figure 4. *osmconvert download and build process* (URL14)

In this research, the *osmconvert* program was used to convert the OSM PBF format into a format more suitable for the extraction process, the *O5M* format. An example of the conversion process is displayed in Figure 5.

```
~/osmconvert 20130301-croatia.osm.pbf -o=20130301-croatia.o5m
```

Figure 5. An example of the conversion process

3.2. OSMfilter

The *osmfilter* program is used to filter OSM data for specific tags. The O5M is one of the supported input and output formats. In order to allow fast data processing, it is recommended to use O5M format at least for input (URL15). The program was designed to be run from the command line and it can be downloaded and built by running the command as shown in Figure 6.

```
wget -O - http://m.m.i24.cc/osmfilter.c |cc -x c - -O3 -o osmfilter
```

Figure 6. *osmfilter download and build process* (URL15)

In this research, the *osmfilter* program was used to filter OSM data based on specific tags. The input data were the O5M files generated in the previous step. The filtering process was focused on OSM feature values (e.g. amenity=restaurant), and the output was once again the O5M file (Figure 7).

```
~/osmfilter 20130301-croatia.o5m --keep="amenity=school" -o=20130301-school.o5m
```

Figure 7. *An example of the filtering process*

The output O5M file containing only the features of interest was converted to PBF format.

3.3. Database schema design

In this research a specific schema was designed to handle different versions of OSM features as well as different geometries simultaneously. The schema was divided into two sets of tables. Figure 8 displays the first set which is focused on storing all versions of OSM features into one table and connecting them to their source file and geometric data through a series of foreign keys. Each table has an auto incrementing primary key *id*.

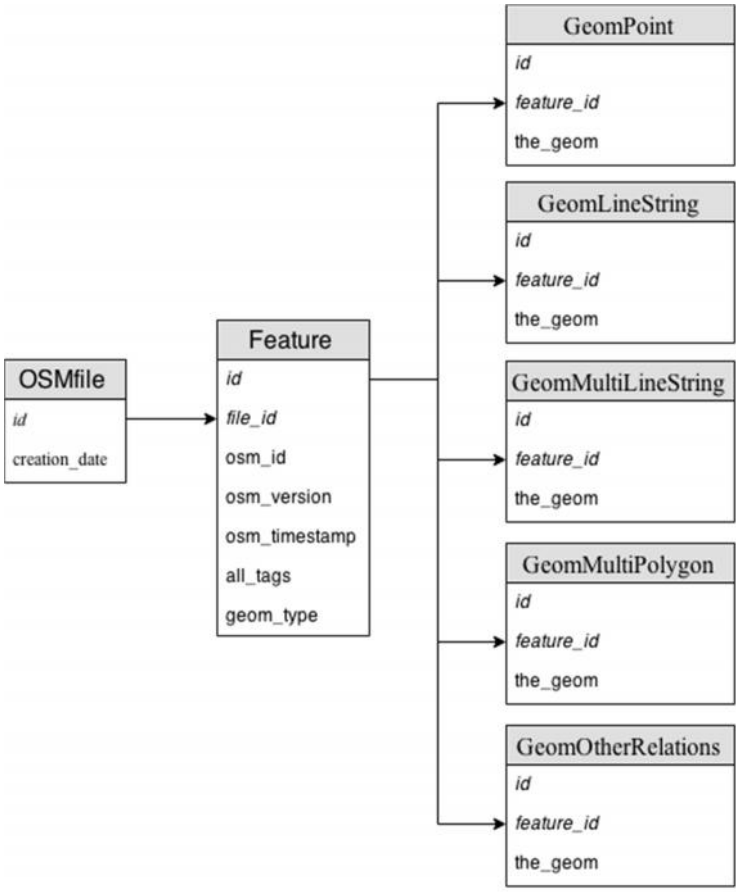

Figure 8. *The first part of the database schema regarding OSM features*

The table *OSMfile* has the *creation_date* field which represents the creation date of the downloaded OSM PBF file (type *date*). The data in this table assists in keeping track of current OSM features and their versions. For instance, if a feature remained present but unchanged throughout the analyzed time period, only the feature's *file_id* was updated. Removed features are detected using the file_id because their file_id does not correspond to the ID of the latest OSM file.

The OSM data extracted from the downloaded PBF files are stored in the *Feature* table. Apart from its primary key, it has a foreign key *file_id* connecting it to the aforementioned *OSMfile* table. The fields *osm_id, osm_version* and *osm_timestamp* contain data that uniquely define each OSM feature. These data are generated automatically by the OSM. Although *osm*_id is the unique identifier of a feature in the OSM project, in this case it is not unique because several different versions of an single feature are stored in the same table. Considering the fact there are three types of OSM features (Node, Way and Relation), each feature's *osm_id* is modified by adding a prefix to the value stored under *osm_id* depending on the type of the feature (e.g. a node's *osm_id* turns from "123456" to "N123456"). The *osm_version* pertains

to the version of an OSM feature, and the *osm_timestamp* pertains to the date and time of the feature's creation (in which case the *osm_version* is 1) or its latest update (the *osm_version* is larger than 1).

On the other hand, the *all_tags* field contains all of the tags defined by the users as *key=>value* pairs. Although there are tools that successfully extract specific tags from the PBF file (e.g. *ogr2ogr*), the *Feature* table is designed to store all OSM features regardless of their type and data.

The final field in the *Feature* table is the *geom_type*. It holds data regarding the geometry type of the OSM feature and this value determines into which table the feature's geometry will be stored.

The final five tables represent the tables that store the geometric data of OSM features. In this case, there are five types of geometries: *Point, LineString, MultiLineString, MultiPolygon* and *OtherRelations*. These tables also contain foreign keys connecting them to the *Feature* table.

Figure 9 displays the second set of tables which is focused on user inputs. The user specifies the name of the new *Job*, parameters that define which type of OSM features are to be processed and several *Functions* that need to be executed in order to get the desired *Results*.

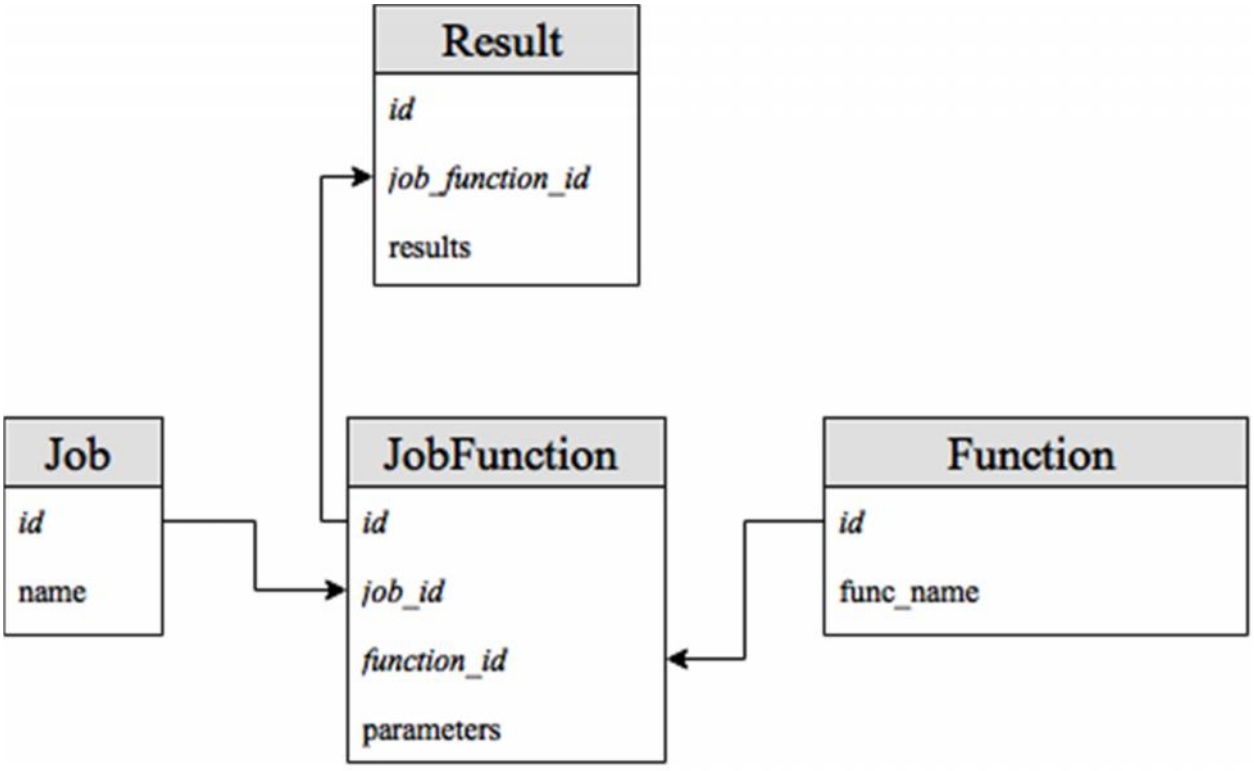

Figure 9. *The second part of the database schema regarding user inputs*

Similar to the aforementioned set of tables, each table in this set contains an auto incrementing primary key *id*. The *Job* table contains the *name* field which contains the string specified by the user describing the job at hand. The *Function* table contains the *func_name* field which stores the names of the functions available to the user. These two tables are connected through a *many-to-many* relationship because each job can call each function. This *many-to-many* relationship is implemented by the *JobFunction* table which has a composite primary key (*id, job_id, function_id*). Furthermore, the *JobFunction* table contains a set of parameters defined by the user for a particular *job=>function* combination. The *Result* table stores the results of a particular *job=>function* combination in form of *key=>value* pairs where the key is the osm_id and the value is the result returned by the called function. This table also has a foreign key connecting the results to each *job=>function* combination (*job_function_id*).

3.4. OSM data extraction

The first part of the developed model refers to OSM data input. The user defines which data are going to be inputted into the database and runs a Python script that uses the GDAL/OGR API to read the desired OSM PBF files and SQLAlchemy/GeoAlchemy2 to send the processed data to the database.

Prior to the extraction process, the OSM configuration file *osmconf.ini* was downloaded from the GDAL website regarding the driver for reading OSM data (URL16). The configuration file declares several tags as dedicated fields, depending on the geometry type. It was subsequently modified to produce the exact same fields for each geometry type. The fields of interest in this research were *osm_id*, *osm_version*, *osm_timestamp* and *all_tags*. The *all_tags* field contains all of the tags (and their values) stored within a feature in a *key=>value* pair whose syntax is compatible with the PostgreSQL *HSTORE* type. The customization is essential for determining which OSM tags should be translated into OGR layer fields. This file is one of the necessary inputs since it determines the way how the PBF file is going to be read.

The downloaded PBF files were not used as input files. Instead, modified files containing specific types of OSM features were generated using the *osmconvert* and *osmfilter* programs and used as input files. This step is not necessary if a user wants to evaluate the entire OSM dataset and all of its features. However, given the complexity of the OSM dataset, only several types of OSM features were analyzed in this research.

Upon reading the PBF file, the first step is date extraction from the filename. This date is cross referenced with the existing entries in the *OSMfile* table and if no match is found, the date is saved into the table. This date is later used for referencing purposes, i.e. all OSM features extracted from this file will have a foreign key linking them to this date.

After reading the PBF file, the extraction process begins with reading the layers and features stored within the file. According to the GDAL documentation (URL17), the driver for reading OSM data will categorize features into five layers:

- *Points*: *Node* features with significant tags attached;
- *Lines*: *Way* features that are recognized as non-area;
- *MultiLineStrings*: *Relation* features that form a MultiLineString;
- *MultiPolygons*: *Relation* features that form a MultiPolygon and *Way* features that are recognized as area;
- *Other_relations*: *Relation* features that do not belong to the above 2 layers.

In this process each feature's *osm*_id is modified based on its geometry type. For instance, the *osm*_id of a *Way* feature turns from '123456' to 'W123456'. This modification allows the storage of features with different geometry types, but same *osm_id*s. In order to avoid duplicates, if the *Feature* table already contains an entry with the same (modified) *osm_id* and *osm_version* as does the feature that is currently being processed, the processed feature is discarded and the existing entry's *file_id* and *osm_timestamp* fields are updated. This step is necessary because several versions of the same feature are stored in the same table thus making the *osm_id* field unsuitable to be the unique identifier (i.e. primary key).

Furthermore, the feature's geometry is evaluated. The goal is to discard features with corrupt geometric data, such as (URL11):

- *Self-intersections* (in LinearRings);
- *Overlapping interior and exterior rings* (in Polygons);
- *Overlapping polygons* (in MultiPolygons).

If a feature passes all of the aforementioned tests, its attribute data is sent to the *Feature* table, while its geometric data is sent to the geometric table that stores its geometric type.

3.5. Detection of changes in the OSM dataset

The second part of the developed model evaluates changes in the OSM dataset. Based on the user's input data, the OSM data is processed by running the Python script designed for that purpose and specifying several positional and optional arguments. The positional arguments specify the OSM features that are going to be processed while the optional arguments determine which functions are going to do the processing.

3.5.1. Positional and optional arguments

The use of positional and optional arguments is achieved by using the *Argparse* Python library. The positional arguments specify the name of the job at hand and the OSM features that need to be processed. The use of positional arguments is obligatory. The OSM features are specified by defining several arguments that describe them, such as the key, value and version of an OSM feature (Table 1).

Table 1. *An overview of the available positional arguments*

Positional argument	**Description**	**Example**
new_job	The name of the job	'changes'
fkey	OSM feature's key	'highway'
fvalue	OSM feature's value	'secondary'
osm_version	Minimal version of an OSM feature	'6'

Table 1 displays an overview of the currently available positional arguments, their short descriptions and an example for each argument. The *fkey* and *fvalue* arguments represent the *key=>value* pair that defines the OSM feature. The *osm_version* argument is used for establishing the lower boundary of frequently edited OSM features, depending on what the user considers a frequently edited feature.

Apart from the positional arguments, a user can define several optional arguments. Most optional arguments are related to functions used for data processing and are not obligatory. However, there are a few exceptions. The only obligatory positional argument is the one that defines the output format. The only output format available at the moment is the GeoJSON, but other output formats such as CSV (*Comma Separated Value*), ESRI Shapefile and GML (*Geography Markup Language*) could also be produced by increasing the functionality of the

developed model. Furthermore, the only optional argument not related to a function is the one that defines the *tag of interest* (TOI). If a user wants to analyze certain tags (e.g. name, operator, etc.), the TOI argument needs to be defined along with the OSM tag.

The remaining optional arguments are related to functions used for data processing. These functions are divided into four groups displayed in Table 2. Each group of functions will be described in the following subsections.

Table 2. *An overview of the available groups of functions*

Group of functions	**Description**	**Input data**
Count functions	Counts the number changes between different versions of the same OSM feature	Tag/Geometry
Change functions	Evaluates the level of change by calculating the Jaro-Winkler or the Levenshtein Distance	Tag
Currency functions	Estimates the level of data currency	Timestamp
Geometry functions	Evaluates the level of change in certain types of geometries	Geometry

3.5.2. Count functions

This group contains four functions that count the number of changes in OSM features. Three functions analyze contextual data (i.e. tags) while the last function analyzes geometric data (Table 3).

Table 3. *An overview of count functions*

Function name	Optional argument (short)	Description
count_inserts	-oci	Counts the number of new tags added to the feature
count_updates	-ocu	Counts the number of modified tags
count_deletions	-ocd	Counts the number of removed tags
count_geom_changes	-ocg	Counts the number of changes in the feature's geometry

Each function analyzes OSM features by comparing all available versions of the same feature, and each version of a feature is compared to its successor in an ascending order. The *count_inserts* and *count_deletions* functions only search for the presence or absence of certain tags, while the *count_updates* function uses exact string matching for the evaluation of *key=>value* pairs and can also be used when analyzing a TOI.

The *count_geom_changes* function is the only function in this group that evaluates geometric changes. This function is specific because *Floating Point Arithmetic* needs to be considered. Floating-point numbers are represented in computer hardware as binary fractions. Unfortunately, most decimal fractions cannot be represented exactly as binary fractions which results with the approximation of the entered decimal floating points by binary floating points actually stored in the machine. This problem can be easily explained in base 10. The fraction 1/3 can be approximated as a base 10 fraction as 0.3, 0.33, 0.333, etc. regardless of the number of digits, the result will never be exactly 1/3, but it will be an increasingly better approximation. The same thing occurs in base 2 used by the machine (URL18).

This problem is also present in storing and manipulating geometric data (i.e. coordinates). To eliminate this effect, Shapely's *almost_equals* method was used. This method compares two geometries and returns *True* if they are approximately equal at all points to specified decimal place precision (URL11).

3.5.3. Change functions

In this research, the aforementioned string matching techniques were used for calculating the average change rate of specified *key=>value* pairs between different versions of OSM

features. The average change rate is used for determining whether a feature has experienced drastic or minor changes in the past. Minor changes indicate an increase in quality by correcting typographical errors while drastic changes could indicate changes in ownership. These functions can only be called if a TOI (i.e. key) was specified. If that condition is satisfied, only features with a specified tag present in two or more versions of the feature are evaluated. The functions and their optional arguments are displayed in Table 4.

Table 4. *An overview of change functions*

Function name	**Optional argument (short)**	**Description**
change_jaro_winkler	-ocjw	Calculates the average Jaro-Winkler percentage of similarity for the TOI
change_levenshtein	-ocl	Calculates the average Levenshtein distance for the TOI

3.5.4. Currency functions

Spatial data currency was implemented in this research with the purpose of determining the currency of OSM features. A feature's currency was determined based on the feature's timestamp which gets updated as soon as new changes are committed to the OSM project by the user. The currency coefficient is calculated for each feature as the difference in days between the feature's timestamp and the median timestamp of the entire dataset associated with the feature's type. All features with timestamps located between the newest and the median timestamp are assigned a negative currency coefficient and are considered up-to-date, whereas the features with timestamps located between the median and the oldest timestamp are assigned a positive currency coefficient which indicates they require revision in the near future.

The objective of this test is to establish a time period after which a feature type becomes outdated (e.g. cafes need to be reviewed every two years) and to avoid the presence of features that were created 5+ years ago and never reviewed again. Table 5 shows an overview of the currency functions currently implemented.

Table 5. *An overview of currency functions*

Function name	Optional argument (short)	Description
currency_coefficient	-occ	Assigns a currency coefficient to the object. The greater the value, the greater the need for revision
currency_revision_needed	-ocrn	A quick test that shows which features are in need of revision. 1 => needs revision 0 => no revision needed

3.5.5. Geometric functions

The last group of functions analyzes geometric changes in features. This group incorporates four functions, two of which process Points, one for linear objects (i.e. LinesStrings, MultiLineStrings) and one for MultiPolygons (Table 6).

Table 6. *An overview of geometric functions*

Function name	Optional argument (short)	Description
geom_from_original	-ogfo	Calculates the distance from the first registered point. Requires at least two points.
geom_sum_dist	-ogsd	Calculates the cumulative distance accross points. Function requires at least three points.
geom_area_diff	-ogad	Calculates the difference in areas between the first and the current version of a feature. Function requires at least two MultiPolygons.
geom_length_diff	-ogld	Calculates the difference in legths between the first and the current version of a feature. Function Requires at least two linear objects.

Unlike the previous groups of functions that analyze a set of tags which is in most cases predefined by the OSM community, this group of functions analyzes spatial data which depends on several factors, such as spatial reference systems, coordinate formats, etc.

The geography type makes the model developed in this research applicable worldwide, regardless of reference systems, projections, etc. All of the spatial calculations were handled by PostGIS due to its spatial indexes and functions that support geography type. The coordinates of OSM features come in longitude/latitude format so the only thing that needs to be done is the conversion to geography type. Only three functions were used in this research:

- ST_Distance(geography, geography) [m],
- ST_Length(geography, geography) [m],
- ST_Area(geography, geography) [m^2].

3.6. Output and visualization

The only output format currently implemented in this model is the GeoJSON format generated using the OGR library. GeoJSON is a format for encoding geographic data structures. A GeoJSON object can represent geometry, a feature, or a collection of features. GeoJSON supports the following geometry types (URL19):

- Point;
- LineString;
- Polygon;
- MultiPoint;
- MultiLineString;
- MultiPolygon;
- GeometryCollection.

Features in GeoJSON contain a geometry object and additional properties, while a feature collection represents a list of features. The entire GeoJSON data structure is always an object. In GeoJSON, an object consists of a collection of *name=>value* pairs, i.e. members. For each member, the name is always a string, while the values are can be a string, number, object, array, or a literal (true, false, null). The GeoJSON object must have a member with the name *type* that defines the object's type (e.g. Point, LineString, Feature, etc.), but it can also have an optional *crs* member whose value must be a coordinate reference system object (URL19).

In this research, the type of the GeoJSON object is set to *FeatureCollection* so it can accept all geometry types, and the *crs* value is set to WGS84. GeoJSON data can be used with a number of different tools, such as QGIS, an open source GIS application, or the *geojson.io* tool for creating, changing and publishing maps (URL20).

The processed data were visualized and analyzed using QGIS. QGIS, previously known as Quantum GIS, is a user friendly open source geographic information system (GIS) published under the GNU General Public License. It is an official project of the Open Source Geospatial Foundation (OSGeo). It runs on Linux, UNIX, Mac OSX, Windows and Android operating systems and supports numerous vector, raster and database formats and functionalities (URL21).

With QGIS, point data were analyzed using heat maps, a method that shows the geographic clustering of a phenomenon. Heat maps show locations of higher densities of geographic entities so that patterns of higher than average occurrence can emerge (e.g. crime activity, traffic accidents, store locations, etc.). They are created by interpolating discrete points and creating a continuous surface known as a density surface. Three parameters have to be determined when calculating a density surface: cell size, bandwidth or search radius, and type of interpolation. The cell size determines the level of detail in terms of coarseness of the resulting density surface (e.g. a smaller cell size results in a smoother surface but increases the processing time and the size of the output raster file). The bandwidth, or the search radius, is the area around each cell the GIS software will factor into the density calculation. A too small search radius restricts the density patterns to the immediate area of the point features, while the density patterns become too generalized with a too large search radius. The final parameter is the type of calculation used in interpolating the density surface. The simplest calculation is a straightforward count of features within the search radius, although weighted calculations (e.g. Inverse Distance Weighting) are more common (URL22).

4. Results

The results of this research were divided into four categories in accordance with the groups of functions that produced them. The following OSM features were analyzed (Table 7):

Table 7. *Analyzed OSM features*

Feature key	Feature value	Geometry type
Amenity	Café	Point
Amenity	Restaurant	Point
Building	School	MultiPolygon
Highway	Secondary	LineString
Highway	Tertiary	LineString

Cafés and restaurants represent frequently visited objects which have a higher probability of being reviewed by a member of the OSM community. A preliminary inspection of the dataset indicated various geometric changes in MultiPolygons regarding their level of detail. At first, schools and their respective grounds were mapped together, whereas later they were separated and various other features (e.g. trees) were added to these areas. Schools and highways represent long standing objects that experience semantic changes less frequently. However, given their relative positions and relationships to surrounding spatial objects, they are bound to experience frequent indirect changes.

4.1. Quantification of changes

The following results represent the quantification of changes in the OSM dataset. These results primarily show the activities of certain local branches of the OSM community. The quantification is split into four categories, inserts, updates, deletions and geometric changes. Although these results indicate the locations of major amounts of changes, the exact levels of changes remains unknown and needs to be properly estimated, i.e. qualified.

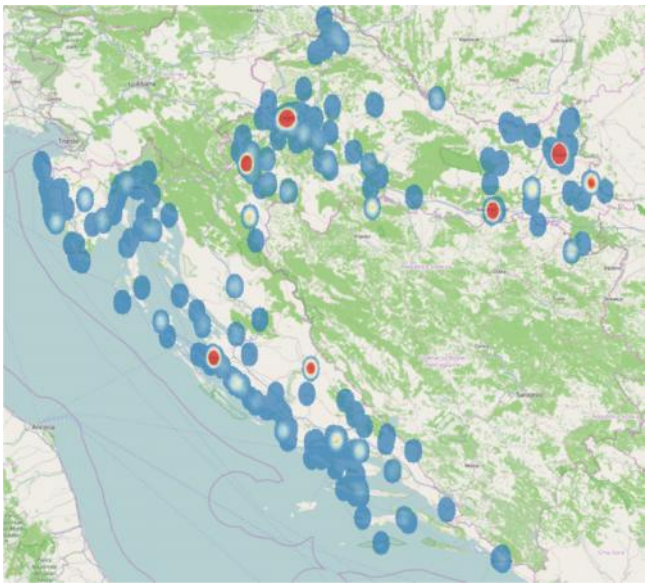

Figure 10. *Areas with frequent additions of new tags to cafés*
(blue - less frequent, red- more frequent)

Figure 10 shows areas with frequent additions of new tags to cafés. This dataset only includes cafés with new tags added after their initial creation. The results indicate that the majority of newly added tags are located in urban areas (e.g. Zagreb, Osijek, Karlovac, etc.), whereas the rural areas experience far less additions.

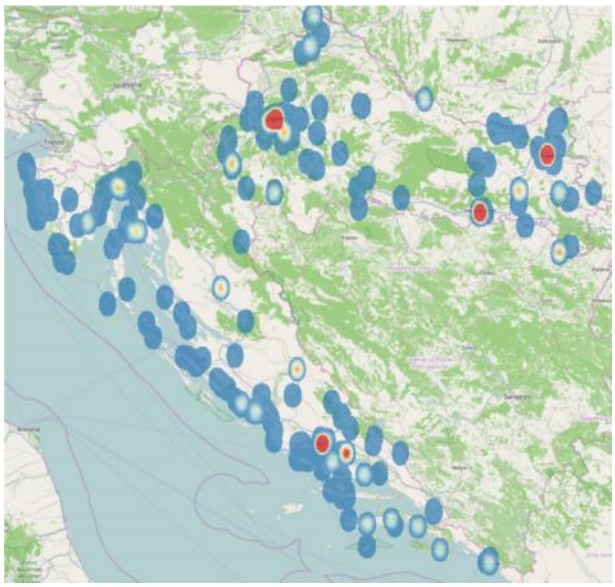

Figure 11. *Areas with frequent changed tags in cafés*
(blue - less frequent, red- more frequent)

Figure 11 displays areas with frequent changes to tags stored in café features. Similar to previous results, frequent changes often occur in urban areas (e.g. Zagreb, Split, Osijek, etc.)

and several suburban areas. The changes in name tags were further analyzed using the Jaro-Winkler and Levenshtein distance algorithm.

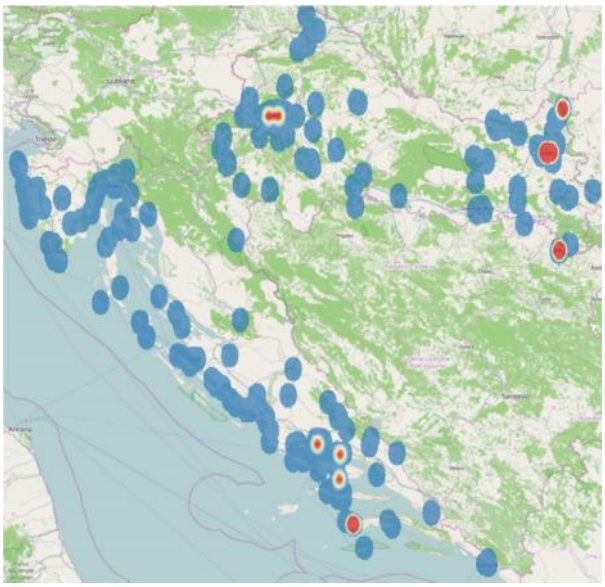

Figure 12. *Areas with frequent deletions of tags in cafés*
(blue - less frequent, red- more frequent)

Figure 12 represents areas with frequent deletions of tags in cafés. Unlike previous cases, most deletions occur in rural and suburban areas. As to urban areas, a large part of deletions occurred in Osijek, while Zagreb and Split experienced a far lower rate of deletions.

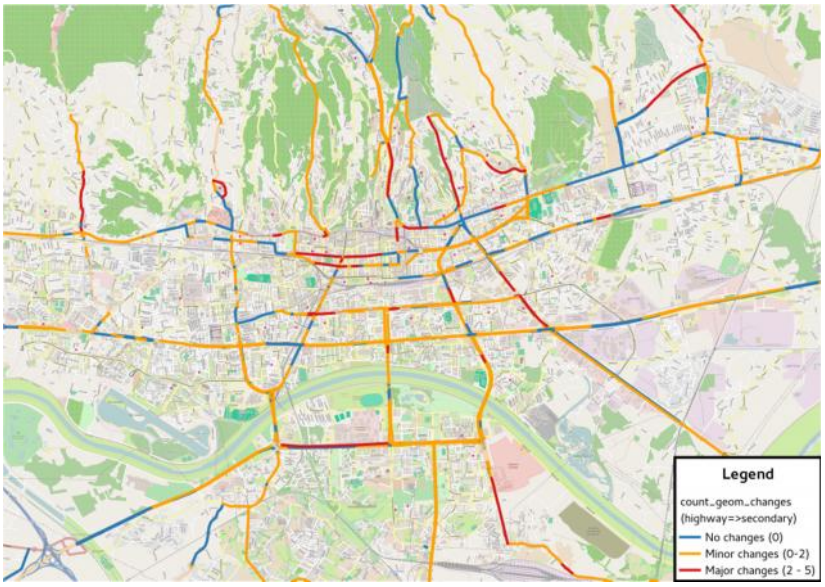

Figure 13. *The rate of changes in linear geometries*
(highway=>secondary, Zagreb)

Figure 13 shows the change rate in geometric data of secondary highways. Most roads experienced minor changes which are often the result of generating new connections between roads, extensions and reductions.

4.2. Qualification of changes using string matching techniques

In order to adequately estimate the level of change in certain features and their tags, string matching techniques were implemented in the developed model. This research was primarily focused on the changes in name tags of cafés and highways. High similarity percentages produced using the Jaro-Winkler algorithm or small Levenshtein distances indicate minor changes in name tags. This is attributed to corrections of certain typographical errors or similar variations and is an indicator of the rising quality of the OSM dataset.

On the other hand, a low Jaro-Winkler similarity percentage or a high Levenshtein distance indicates drastic changes in name tags, which could be indicative of changes in ownership. The change history of these features should be inspected along with the real-world object.

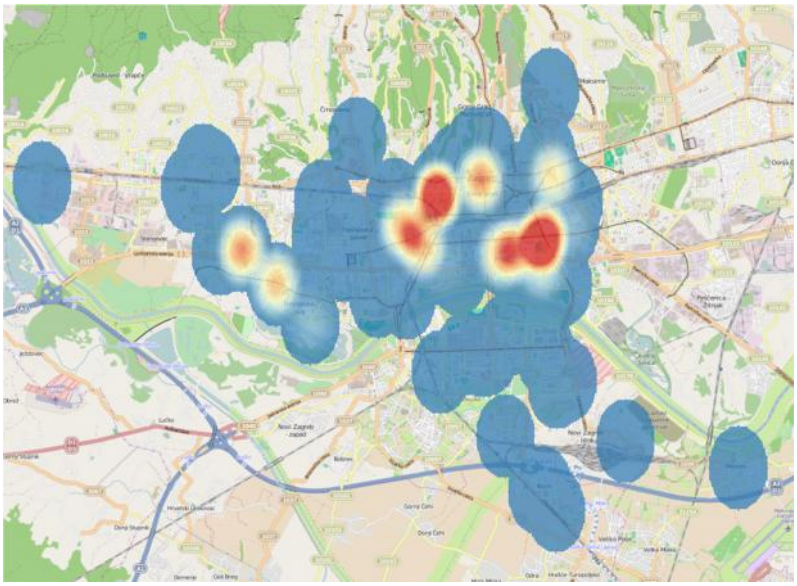

Figure 14. *Estimating the similarity between name tags in cafés using the Jaro-Winkler technique*
(blue – more similar, red – less similar)

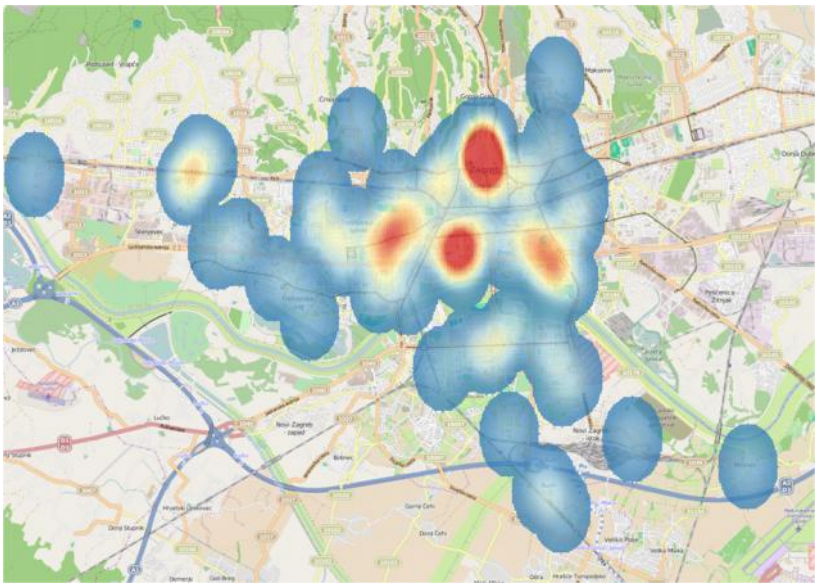

Figure 15. *Estimating the similarity between name tags in cafés using the Levenshtein distance technique*
(blue – more similar, red – less similar)

Figures 14 and 15 outline the differences between the Jaro-Winkler and the Levenshtein distance algorithm. Specifically, the number of different characters between two names does not necessarily indicate a significant difference between them. The results achieved using the Levenshtein distance algorithm show two significant hot spots and two minor ones, while at the same time only one of the significant hot spots is registered by the Jaro-Winkler algorithm. The same analogy can applied to figures 16 and 17.

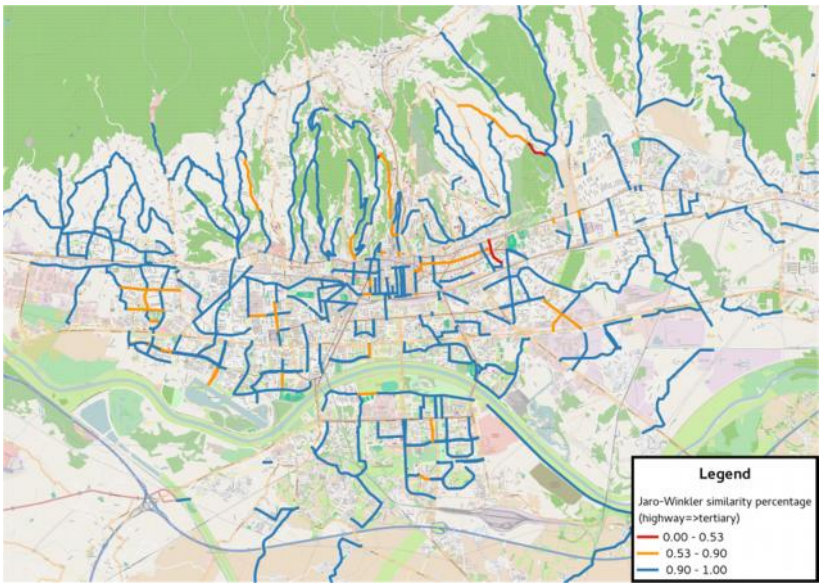

Figure 16. *Estimating the similarity between name tags in tertiary highways using the Jaro-Winkler technique*

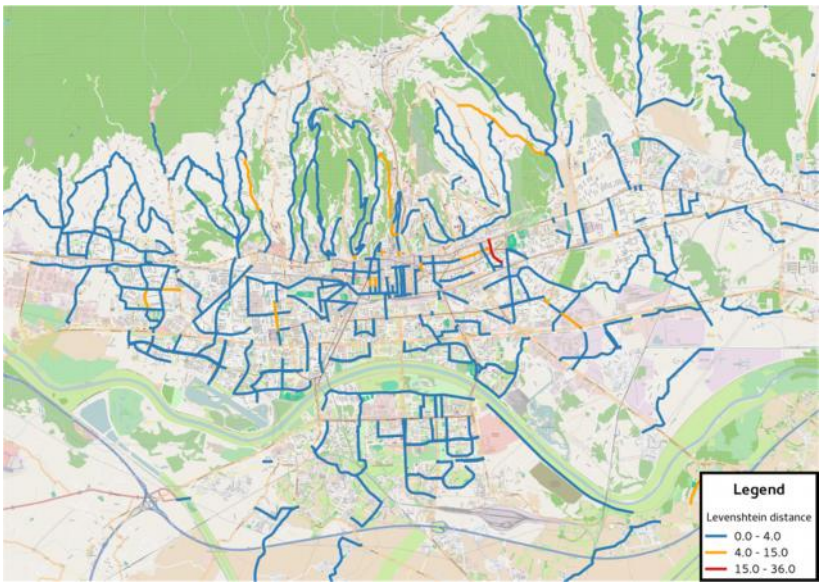

Figure 17. *Estimating the similarity between name tags in tertiary highways using the Levenshtein distance technique*

4.3. OSM data currency

There is a great necessity for estimating the currency of spatial object, especially in open source projects such as the OSM. In this research, restaurants were evaluated on a temporal basis and divided into two groups. The median timestamp in this subset is estimated at May

2nd, 2013, which renders restaurants that were evaluated more than two years ago obsolete and in need of revision.

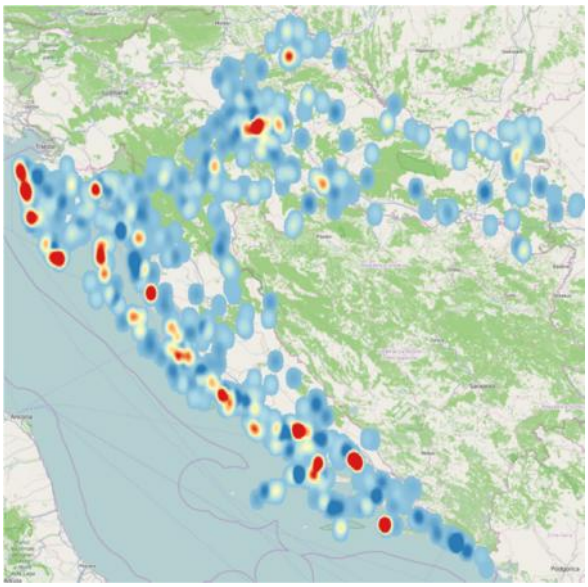

Figure 18. *Currency of restaurants in Croatia*
(blue – up-to-date, red – out-of-date)

Figure 18 represents the currency of restaurants in Croatia. The majority of out-of-date restaurants are located in coastal areas with several exceptions located in the continental part, such as Zagreb.

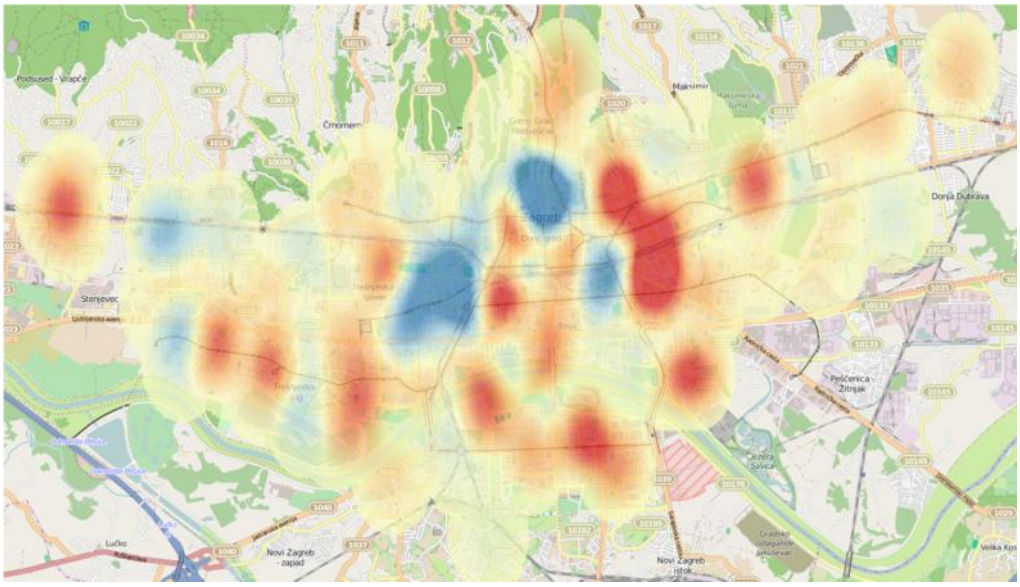

Figure 19. *Currency of restaurants in Zagreb*
(blue – up-to-date, red – out-of-date)

Figure 19 displays the currency of restaurants located in Zagreb. The majority of restaurants is out-of-date, but the restaurants located near the town center, near the main bus station and the northern part of the Trešnjevka district are up-to-date and reviewed regularly.

4.4. Geometric changes

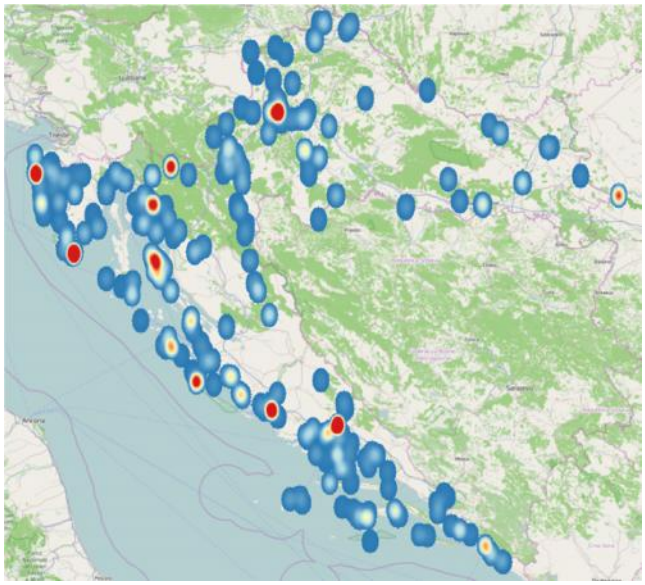

Figure 20. *Areas with high levels of changes in geometric data of restaurants (blue – small changes, red – significant changes)*

Figure 20 shows areas with significant movements of restaurants. In this case, the distance between the current and the original position of a restaurant is analyzed. The hot spots indicate areas of high levels of movement.

Figure 21. *Geometric changes of a MultiPolygon (school)*

Figure 21 displays changes in a particular MultiPolygon feature, in this instance, a school. This figure also shows that the geometry of spatial objects changes over time which can lead to higher levels of detail and accuracy.

5. Discussion

The results of this research demonstrate the effectiveness of the developed model regarding the detection and evaluation of changes. For this to be achieved, various methods were implemented, ranging from simple counters to string matching techniques and spatial calculations on a sphere. Although the model was tested on just a handful of OSM features, each feature is a representative of a certain geometry type. In this section, each aspect of the model will be thoroughly analyzed and potential improvements will be suggested.

First of all, in order to create a solid foundation for the model in terms of tables and connections in the database schema, several advanced features were implemented, such as the *hstore* type. OSM objects, depending on their importance and real-world features, can have a wide range of tags describing them to the very last detail. Furthermore, given the development rate of the OSM project and the user's freedom to introduce new or custom tags to a particular object, the construction of a database schema that would take all these possibilities into consideration is challenging. That is why the *hstore* data type is used. Apart from its ability to store all tags in one field, it can also be indexed or used as a dictionary in Python. This greatly reduces the complexity of the database schema design by saving most tags and information about features into one single table. Each feature is subsequently connected to its geometry and the file of origin.

This method of data storage has its merits, but what about the results? How are they connected to their respective features? Considering the adopted design, it was impractical to store each result into a feature's *hstore* field. Therefore, the second part of the schema pertaining to the user's inputs was developed. The main disadvantage of this methodology is that the process of rendering results is slower than it would be if the results were stored next to the features. However, the advantages of this approach are the increased flexibility of the output, reusability of results and the separation of results from the original data (i.e. the original data remains intact when used by several different users simultaneously).

One of the drawbacks of this model is the fact that the maximum number of changes that the algorithm can register is equal to the number of input files, in this case OSM dump files. This means that unless the update interval of OSM features corresponds to the time interval of the input files (e.g. a feature is updated regularly every three months), the changes detected by the algorithm are not going to be a real indicator of changes experienced by the object. In this

research, several *highway* objects experienced over 50 changes since their creation which qualifies them as frequently updated objects. The algorithm can only detect changes that occurred in the analyzed time period (in this case from January 1st, 2013 to March 1st, 2015), which disregards the changes that happened before, regardless of their amount or importance. Furthermore, it is possible that a certain object can experience frequent changes in the first couple of days since its creation. The registration of these changes depends on the time interval between the input files. If the time interval of these updates is shorter than the time interval of the input files, these changes are going to be disregarded.

There are two possible solutions to this issue. The first solution is to minimize the time interval between the input files by downloading and inserting new data to the database on a daily basis. The second solution is to use the *planet.osm/full* file made available by OSM every few months. This file includes almost all OSM data ever collected and would certainly solve the issues described earlier. However, the size of this file is extensive and the developed algorithm should be reconfigured to handle large amounts of data, select data using bounding boxes, modify geometry handling, etc. For instance, Mooney and Corcoran (2012) developed two Python scripts that process the OSM history file for the UK and Ireland first by inserting all nodes and the inserting all ways. The duration time of this process is 305 h. Given these issues, this approach was deemed unfit for this research.

String matching techniques are greatly influenced by these issues. In this model the string matching techniques are used for determining the level of changes in certain features and their tags. These changes can indicate whether the changes are negligible (e.g. typos) or are they considerable. If the changes are considerable (e.g. changes in name or operator tags), this can be an indication that the ownership of the object has changed. However, the results of automatic qualification methods highly depend on the tags of interest as well as the used functions, which in turn depends on the user. In this research, only string matching techniques were used to qualify changes in OSM data. This set of functions can be further expanded with functions that analyze timestamps, the use of unstandardized tags, etc.

The only functions impervious to the aforementioned issues are the currency functions. These functions rely solely on the current data, i.e. the data from the last input file. The calculation is fairly simple; features are divided into two groups (up-to-date and out-of-date) defending on the difference between their timestamps and the median timestamp. This approach produces excellent results when applied to service objects that change frequently, such as restaurants,

cafés, and bakeries. However, it is not applicable to long-standing, fixed objects of social value, such as historical monuments, castles, governmental buildings, and motorways. These objects are expected to endure and even though they may be frequently updated, the probability they would suddenly change drastically is negligible. Therefore, the currency coefficient of such object should be determined by the community after reaching a consensus. The currency coefficient should prevent the existence of obsolete data by indicating which objects need to be revisited, and if necessary, updated. In the event that there are no more tags that could be added to an object, the currency coefficient can be used as a reminder of when the object needs to be revisited and confirmed that it has not changed.

The geometric functions are modified to handle the *geography* type for two reasons. Firstly, the user does not need to worry about spatial reference systems and transformation of coordinates. All geometric data is available and stored in the database in WGS84. Secondly, most countries use the metric system. Therefore, the end result is in meters or square meters, depending on the type of geometry.

In this research, the end results are visualized in QGIS and analyzed using heat maps and data classification. Heat maps can only be generated when using points as input data. They provide a solid basis for detecting and analyzing clusters, and the results of particular functions called by the user can be used as a weighting parameter. On the other hand, LineStrings and MultiPolygons can only be analyzed using classifications. This approach is practical for large scale maps, but impractical for small scale maps since they are either barely visible, or some form of generalization is needed to reduce the amount of details. This can be improved by extracting characteristic points from LineStrings or centroids from MultiPolygons and creating a heat map from the available point data.

6. Conclusion

The model developed in this research is able to analyze all types of OSM features regardless of their geometry type. String matching techniques are used to evaluate the level of change in certain tags, and the currency of each feature can be determined. The spatial calculations are conducted on a sphere which slightly increases the computation time, but allows for the model to be applicable worldwide, regardless of spatial coordinate systems. The model is currently implemented as a command line application that allows the user to define which features need to be processed and what functions should be used to get the desired results. It also allows the user to define tags of interest which should be processed separately. The output is a single GeoJSON file that stores all processed features, their data and the results of called functions. The entire model is available at https://github.com/fitodic/osm-changes under the GNU General Public Licence version 2.

Literature

Battersby, S. E., Finn, M. P., Usery, E. L. & Yamamoto, K. H., 2014. Implications of Web Mercator and Its Use in Online Mapping. *Cartographica: The International Journal for Geographic Information and Geovisualization,* 49(2), pp. 85-101.

Bilenko, M. i dr., 2003. Adaptive name matching in information integration. *IEEE Intelligent Systems,* 18(5), pp. 16-23.

Devillers, R. & Jeansoulin, R., 2010. *Fundamentals of Spatial Data Quality.* London: Wiley Online Library.

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,* Issue 37, pp. 682-703.

Mooney, P. & Corcoran, P., 2012. Characteristics of heavily edited objects in OpenStreetMap. *Future Internet,* 4(1), pp. 285-305.

Neis, P., Zielstra, D. & Zipf, A., 2011. The street network evolution of crowdsourced maps: OpenStreetMap in Germany 2007-2011. *Future Internet,* 4(1), pp. 1-21.

Neis, P. & Zipf, A., 2012. Analyzing the contributor activity of a volunteered geographic information project—The case of OpenStreetMap. *ISPRS International Journal of Geo-Information,* 1(2), pp. 146-165.

URL1: Index of /croatia/archive, http://data.osm-hr.org/croatia/archive/ (Accessed June 16th, 2015)

URL2: OpenStreetMap Map Features, http://wiki.openstreetmap.org/wiki/Map_Features (Accessed June 17th, 2015)

URL3: PBF Format, http://wiki.openstreetmap.org/wiki/PBF_Format (Accessed April 23rd, 2015)

URL4: About PostgreSQL, http://www.postgresql.org/about/ (Accessed April 25th, 2015)

URL5: About PostGIS, http://postgis.net/ (Accessed April 25th, 2015)

URL6: hstore, http://www.postgresql.org/docs/current/static/hstore.html (Accessed April 25th, 2015)

URL7: General Python FAQ, https://docs.python.org/2/faq/general.html#what-is-python (Accessed April 27th, 2015)

URL8: SQLAlchemy Home, http://www.sqlalchemy.org/ (Accessed April 28th, 2015)

URL9: About GeoAlchemy2, https://geoalchemy-2.readthedocs.org/en/0.2.4/ (Accessed April 27th, 2015)

URL10: GDAL/OGR Info Sheet, http://www.osgeo.org/gdal_ogr (Accessed April 27th, 2015)

URL11: The Shapely User Manual, http://toblerity.org/shapely/manual.html (Accessed May 6th, 2015)

URL12: argparse, https://docs.python.org/2/library/argparse.html#module-argparse (Accessed April 28th, 2015)

URL13: PostGIS Geography Type, http://workshops.boundlessgeo.com/postgis-intro/geography.html#casting-to-geometry (Accessed May 8th, 2015)

URL14: Osmconvert, http://wiki.openstreetmap.org/wiki/Osmconvert (Accessed April 23rd, 2015)

URL15: Osmfilter, http://wiki.openstreetmap.org/wiki/Osmfilter (Accessed April 25th, 2015)

URL16: Configuration file for OSM import, http://svn.osgeo.org/gdal/trunk/gdal/data/osmconf.ini (Accessed June 17th, 2015)

URL17: OSM - OpenStreetMap XML and PBF, http://www.gdal.org/drv_osm.html (Accessed April 28th, 2015)

URL18: Floating Point Arithmetic: Issues and Limitations, https://docs.python.org/2/tutorial/floatingpoint.html (Accessed May 6th, 2015)

URL19: The GeoJSON Format Specification, http://geojson.org/geojson-spec.html (Accessed May 9th, 2015)

URL20: geojson.io, https://github.com/mapbox/geojson.io (Accessed June 17th, 2015)

URL21: Discover QGIS, http://qgis.org/en/site/about/index.html (Accessed May 9th, 2015)

URL22: Heat Maps in GIS, http://www.gislounge.com/heat-maps-in-gis/ (Accessed May 9th, 2015)

List of figures

List of tables

Resume

PERSONAL INFORMATION

Name	Filip Todić
Email	todic.filip@gmail.com
LinkedIn	hr.linkedin.com/in/filiptodic
GitHub	github.com/fitodic

WORK EXPERIENCE

May, 2015 – Present	Student helper, Ericsson Nikola Tesla d.d., Zagreb
October, 2014 – May, 2015	Teaching fellow, Chair of Geoinformatics, Faculty of Geodesy, Zagreb
October, 2014 – March, 2015	Computer programmer, Faculty of Transport and Traffic Sciences, Zagreb
May, 2014 – October, 2015	Cartographer, Karlovac County Fire Department, Karlovac
August, 2014 – September, 2014	Surveyor, Geovizija d.o.o., Osijek

EDUCATION

September, 2013 – July, 2015	Geoinformatics Master Study programme, Faculty of Geodesy, University of Zagreb
July, 2014	GIS Summer School, Faculty of Geodesy, University of Zagreb
July, 2010 – June, 2013	Geodesy and Geoinformatics Bachelor Study programme, Faculty of Geodesy, University of Zagreb
September, 2006 – May, 2010	III. gymnasium, Osijek

PERSONAL SKILLS

Mother tongue: Croatian

Other languages	**UNDERSTANDING**		**SPEAKING**		**WRITING**
	Listening	**Reading**	**Interaction**	**Production**	
English	C1	C1	C1	C1	C1
	Certificate in Advanced English (CAE), University of Cambridge				
German	A1	A1	A1	A1	A1
	Foreign language school Prospero, Zagreb				

COMPUTER SKILLS

Programming languages	Python, JavaScript, Java
Operating system skills	Linux, Windows
Databases	PostgreSQL, PostGIS, SQLite
GIS software	QGIS, GRASS GIS, SAGA GIS
CAD software	Autodesk Map 3D
Other skills	HTML, CSS, Latex, GIT, OpenLayers, GeoServer

ADDITIONAL INFORMATION

Publications

- Miler, M., Todić, F., Ševrović, M. (2015), Validating traffic accident locations using OpenStreetMap and the Jaro-Winkler string matching technique, Transportation Research Part C: Emerging Technologies, Elsevier (under review)
- Antolović, J., Giljanović, M., Jurić, V., Kozić, R., Todić, F., Vidonis, N. (2014), Construction of a tourist Web map of Duga Resa with GIS Cloud technology, Ekscentar No. 17, pg. 45-49
- Todić, F., Jurinović, A., Mustač, A. (2013), Registry of geographic names, Ekscentar No. 16, pg. 50-53

Projects

- *Web GIS for the visualization of traffic accident frequencies*. Programmer in charge of developing the algorithm for the evaluation of traffic accident locations (October 1st, 2014 – March 31st, 2015)

Seminars

- Todić, F., Šimunović, T., Tomac, G., Everywhere navigation (2015) (presented at the 8th Colloquium of the Chair for Satellite Geodesy (January 19th, 2015))

Honours and rewards

- Rector's award for the best student paper: Šimunović, T., Todić, F., (2014) Geostatistical analysis of traffic accident spatial distribution in the City of Zagreb from 2010 to 2013
- Among the top 10% of the most successful students (Bachelor study programme, Faculty of Geodesy, generation 2010/11)
- Certificate in Advanced English (CAE), Cambridge English Language Assessment, University of Cambridge