OpenStreetMap Data Wrangling with Python and MongoDB

Ivailo Kassamakov

November 2015

Abstract

The present project is an effort towards fulfilling the requirements of the Data Analyst Nanodegree at Udacity. The goal is to perform data cleaning and exploration on an extract of the OpenStreetMap GIS database.

All data extraction and exploration tools (an indispensable part of this project) have been developed in Python, using the `xml.etree` package for XML processing and `pymongo` for interfacing with a locally installed MongoDB server.

Some useful tools used during the work on this project were: www.sharelatex.com, www.tablesgenerator.com, and overpass-turbo.eu.

Contents

List of Figures

List of Tables

1 Introduction

The OpenStreetMap (OSM) GIS database is a community-driven project collecting freely accessible world-wide map data. Cartographic data can be exported from the database via the public Overpass API, using the OSM XML or PBF (Protocolbuffer Binary Format) file formats. The PBF format is binary in nature, therefore it results in much smaller files and is much quicker to work with. However, for the purpose of this project I will be using data in the OSM XML format.

2 Rights on the OSM data

All of the OSM cartographic information is published under the Open Database License (ODbL) v1.0, and as such it is considered *open* data that can be freely shared, queried and updated.

3 Choosing and obtaining the cartographic data

The chosen area was that of the Zurich city in Switzerland. The cartographic data was obtained using the Overpass API via the `dl_osm_xml_data.py` Python script.

The Overpass QL query sent to the API was:

```
area[name='Zürich'];
(
  rel(pivot)->.a;
  way(r.a)->.b;
  node(area)->.c;
  way(area)->.d;
  rel(area)->.e;
);
out meta;
```

The resulting OSM XML file has a size of 104 MB, which meets the requirements for this project. Figure 1 visualizes the downloaded data. This map was prepared from the downloaded OSM file with the help of the free Maperitive tool.

4 Preliminary dataset analysis

The Python script `get_xml_schema.py` analyzes the XML structure of the downloaded OSM file and counts the number of occurrences of each XML tag. It also lists all attributes found for each of the tags. Table 1 shows this statistics.

The second XML processing Python script `get_xml_values.py` script extracts all attribute values from the input OSM file. It dumps the attribute values of each XML element into a separate file–for example, the values of the 'role' attribute of the 'relation.member' XML element are output to the `attr-relation-member-role.txt` file. The format of these files is a valid JSON and looks like this:

```
    {'.//relation/member[@role]': {'across',
                                   'admin_centre',
                                   'alternate',
                                   'backward',
                                   'backward_stop',
                                   ...}}
```

The *k:v* attributes found in the 'node.tag', 'way.tag' and 'relation.tag' XML elements are extracted in pairs and also saved to files in a valid JSON format. For example, the `kv-node-tag.txt` dump file looks like this:

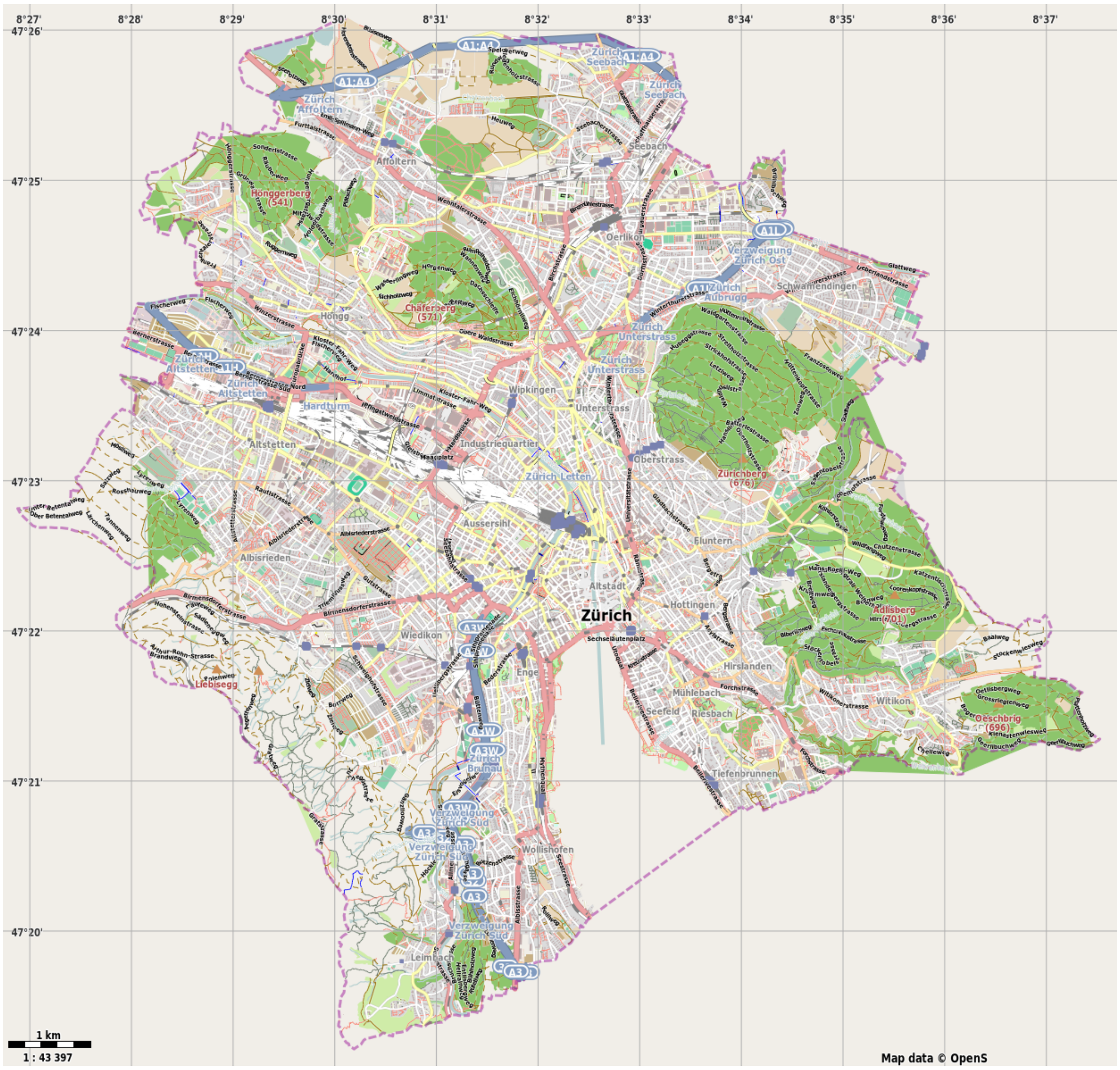
Figure 1: Map of the selected area–Zurich city

```
{'.//node/tag': {'abandoned:railway': {'station'},
                 'access': {'customers',
                            'destination',
                            'no',
                            'permissive',
                            'private',
                            'public',
                            'unknown',
                            'yes'},
                 'addr.source:housenumber': {'survey'},...}}
```

5 Importing the data into MongoDB

The OSM XML map data is transformed to a JSON representation via the `osm_to_xml.py` Python script. This script is heavily based on the code found in Lesson 6 of the Udacity "OpenStreetMap Data Wrangling with MongoDB".

The script transforms every *node* and *way* OSM XML element into a JSON document. For example, the following OSM XML element:

Tag name	Count	Attributes
’osm’	1	’generator’, ’version’
’osm.meta’	1	’areas’, ’osm_base’
’osm.node’	417676	’changeset’, ’id’, ’lat’, ’lon’, ’timestamp’, ’uid’, ’user’, ’version’
’osm.node.tag’	87143	’k’, ’v’
’osm.note’	1	none
’osm.relation’	1503	’changeset’, ’id’, ’timestamp’, ’version’, ’uid’, ’user’
’osm.relation.member’	65783	’ref’, ’role’, ’type’
’osm.relation.tag’	6977	’k’, ’v’
’osm.way’	67451	’changeset’, ’id’, ’timestamp’, ’version’, ’uid’, ’user’
’osm.way.nd’	534218	’ref’
’osm.way.tag’	265276	’k’, ’v’

Table 1: XML statistics of the OSM input file

```
<node id="2406124091" lat="41.9757030" lon="-87.6921867" version="2"
      timestamp="2013-08-03T16:43:42Z" changeset="17206049"
      uid="1219059" user="linuxUser16" visible="true">
    <tag k="addr:housenumber" v="5157"/>
    <tag k="addr:postcode" v="60625"/>
    <tag k="addr:street" v="North Lincoln Ave"/>
    <tag k="amenity" v="restaurant"/>
    <tag k="cuisine" v="mexican"/>
    <tag k="name" v="La Cabana De Don Luis"/>
    <tag k="phone" v="1 (773)-271-5176"/>
</node>
```

is translated into the following JSON document:

```
{
    "id": "2406124091",
    "type: "node",
    "visible":"true",
    "created": {
          "version":"2",
          "changeset":"17206049",
          "timestamp":"2013-08-03T16:43:42Z",
          "user":"linuxUser16",
          "uid":"1219059"
          },
    "pos": [41.9757030, -87.6921867],
    "address": {
          "housenumber": "5157",
          "postcode": "60625",
          "street": "North Lincoln Ave"
          },
    "amenity": "restaurant",
    "cuisine": "mexican",
    "name": "La Cabana De Don Luis",
    "phone": "1 (773)-271-5176"
}
```

The JSON output file is then imported into MongoDB via the following command:

```
    > mongoimport --db osm --collection map < zurich-area.json
```

Some statistics about the imported data is shown below:

```
# Number of distinct users having contributed to the map
    > db.map.distinct('created.user').length
    904
```

```
# Number of documents:
    > db.map.find().count()
    485127   (This matches the sum of 'osm.node' and 'osm.way' occurrences
              counted by the get_xml_schema.py script)

# Number of nodes and ways:
    > db.map.find({'type':'node'}).count()
    417474
    > db.map.find({'type':'way'}).count()
    67425
```

Oops! The number of nodes and ways does not add up to the total number of documents. This means that the JSON document 'type' attribute has been contaminated by unexpected values.

```
# Which are the offending 'type' values?

    > db.map.distinct('type')
    ['way', 'node', 'broad-leaved', 'Laubbaum', 'water', 'destination_sign',
                'esotheric', 'bags', 'floating', 'gas', 'travel', 'glass']

# Which is the most frequent offending value?
    > db.map.aggregate(
        [
            {'$project': {'type':1, '_id':0}},
            {'$match': {'type': {'$nin': ['way', 'node']}}},
            {'$group': {'_id':'$type', 'count':{'$sum':1}}},
            {'$sort': {'count':-1}},
            {'$limit': 1}
        ]
    )
    ...{ "_id" : "gas", "count" : 200 }...
```

This just pointed out a bug in the `osm_to_json.py` script: obviously some *node* and *way* elements in the OSM data are tagged with 'type' keys (i.e. contain `<tag k="type"...>`), which due to the bug will overwrite the document's 'type' JSON attribute defined by the Python script. NOTE: This bug has not been corrected and is still present in the Python script.

6 Data audit

The OSM database is freely modified by a large community of volunteering cartographers. Therefore, it is natural to expect that it will contain a lot of *dirty* data.

6.1 General considerations

Performing an audit of the input data to determine how dirty it is, and the subsequent cleaning is a major part of the ETL (Extract-Transform-Load) process.

Following the ETL data wrangling principles, the input data audit includes checking for the following *data qualities*:

- **Validity**–to determine if the data conforms to a certain schema, and has the necessary data types and ranges
- **Accuracy**–to determine if the data is factually correct
- **Completeness**–to determine if all necessary data items are present
- **Consistency**–to determine if the data in one field matches/correlates with data in other fields
- **Uniformity**–to determine if the data consistently uses the same units or textual representation

6.2 OSM data audit

The following *validity* checks can be performed on the OSM dataset:

- Do the XML tags follow the official OSM XML schema?
- Do the telephone numbers follow the structure, recommended in the key:phone documentation at wiki.osm.org?
- Are email and web addresses correctly constructed?
- Are longitudes and latitudes valid in term of structure and ranges?
- Are the cartographic items described with the correct set of free-form *key:value* pairs? There are a lot of requirements for this on wiki.osm.org.
- Do the *key* attributes have values that are commonly used? This requirement is pretty elastic, since the tagging philosophy of OSM is free-form. However, the cartographers should strive not to invent keys that already exist.
- Do the *value* attributes have the expected/common values for their respective *keys*?
- Are all elevations (the 'ele' key) presented as pure floating point numbers?

Evaluating the *accuracy* of the audited data, on the other hand, is difficult due to the general lack of GIS golden standards. Still, there are some simple checks that can be performed, such as:

- Correct orthography of the attribute values.
- The ranges of the postal code in the node addresses, as well as the range of nodes' longitudes and latitudes can be easily verified.
- At least the 'city' and 'country' part of the nodes' addresses can be verified.
- Do the telephone numbers have the expected area codes?
- Is the country code in 'is_in:country_code' correct in terms of ISO 3166-1?

As for the *completeness* of the dataset, there is not much that can be done. The OSM database is an ongoing project and by definition is very incomplete. Some obvious checks are:

- Do all "buildings" have address tags?
- Do all addresses contain the "implicitly" known tag keys 'addr:city' and 'addr:country', which remain the same for all nodes in the selected area?

The evaluation of the data *consistency* can employ checks such as:

- Do all referenced items (e.g. nodes in a way) really exist?
- Does the relation "relation.member.type" (node, way, relation) correspond to the real type of the referenced item in a relation?

Data *uniformity* on the other hand is much easier to check for. For example:

- Are all elevations in the 'ele' tag expressed in meters?
- Are all way 'width' tag values expressed in meters?
- Are there values that differ only by their case?
- Are the name of the streets spelled in a uniform way, i.e. not having a mixture of "strasse", "str." and "st.".
- Are all phone numbers written in a uniform way?

6.3 Audit results and quality problems found

Initial auditing the of input data was performed by manually browsing the values of the OSM *k:v* pairs, as extracted by the `get_xml_values.py` script. The *k:v* pairs form the information backbone in the OSM database.

For this project only an audit on the *k:v* attributes of the *node.tag* elements was performed.

Some of the identified issues are presented in table 2.

Quality attribute	OSM keys	Issues found	Expected
Validity	'phone', 'fax', 'contact:phone', 'contact:mobile', 'contact:fax'	Telephone numbers written in many different ways, some of them wrong.	Phone numbers must conform to E.164/2002 specification.
Accuracy	'phone', 'fax', 'contact:phone', 'contact:mobile', 'contact:fax'	There are telephone numbers with wrong area code.	Area codes in Zurich are 43 and 44.
Validity	'url', 'website', 'contact:website', 'facebook', 'site'	URLs not using in a uniform way 'http(s)' and '/'.	"http://my.web.ch/page"
Uniformity	'addr:postcode'	Some postal codes are like '8005', others are like 'CH-8005'.	'8005'
Accuracy	'addr:postcode'	There is a wrong postcode 6330.	Postcodes in Zurich are 8xxx.
Completeness	'addr:*'	The tags 'addr:city' 'addr:state' and 'addr:country' do not exist always in all addresses.	These tags shall be present (with the same values) for all addresses in Zurich.
Validity	'addr:city'	Sometimes appears as postal code.	'Zürich'
Uniformity	'addr:city'	Different spellings and capitalization of 'Zürich', e.g. 'zuerich', 'Zurich'	'Zürich'
Validity	'ele'	Some values are float, some are integer	'ele' shall be integer, in meters, in the range 300–900 m.
Accuracy	'cuisine', 'craft', 'defibirllator' [sic]	Spelling mistakes	

Table 2: Audit results

7 Cleaning of phone numbers

After the quality audit, some data cleaning is in order. For this project only a cleaning of the telephone numbers was performed. Auditing phone numbers has the big advantage of allowing checks for validity, accuracy and uniformity at the same time. It also contributes a significant value to the OSM community, since phone numbers are a really critical piece of data.

The audit and cleaning is done programmatically via the `audit_phones.py` Python script. This script performs basically the following tasks:

- Extracts the phone numbers tagged via different OSM *keys* like 'contact:phone', 'fax', 'mobile:phone', etc.
- Checks the numeric validity of each number by analyzing it according to the Swiss E.164/2002 Numbering Plan
- At the same time, the phone number accuracy is partly checked by verifying that the country and area code prefixes have correct values.
- The phone numbers get classified according to the way they have been input, e.g. some numbers start with "+41", others with "0041", others wrongly prefix the area code with a "+", some have "(0)" in front of the area code, others not, etc.
- The numbers that are found to be numerically valid are then brought to a uniform text representation, i.e. they get cleaned. All numbers get represented in one of the following two shapes common in Switzerland: "+41 (0)xx yyy yy yy" for 2-digit area codes, or "+41 (0)xxx yyy yyy" for 3-digit area codes.
- If the script is working on a MongoDB database, the results of the audit for each number are stored under separate keys ('x-audit-op' and 'x-audit-clean:...') in the respective document (via 'update' calls). Invalid numbers are appropriately tagged, so that they can be retrieved later for manual inspection and correction.

The script can take input from one of the following sources:

- A text file containing a dump of k:v OSM values. A suitable input can be the `kv-node-tag.txt` file output by the `get_xml_values.py` script. In this case, the results of the audit and the cleaned phone numbers will be printed to `stdout` .

- A MongoDB database containing map data prepared and imported as described in section 5. As pointed out above, the results of the audit will be written back to the database. The will also be printed to `stdout` .

The following statistics of the audit results was then obtained:

```
# Number of phone numbers present in the database
    > db.map.find({'$or':[{'phone':{'$exists':1}}, {'contact:phone':{'$exists':1}},
                          {'fax':{'$exists':1}}, {'contact:fax':{'$exists':1}},
                          {'contact:mobile':{'$exists':1}}]}).count()
    511

# Number of phone numbers audited:
    > db.map.find({'x-audit-op':{'$exists':1}}).count()
    511

# Number of phone numbers found numerically valid and rewritten (cleaned):
    > db.map.find({'x-audit-op':'fix'}).count()
    498

# Number of phone numbers found numerically valid and already having the right text shape:
    > db.map.find({'x-audit-op':'none'}).count()
    1

# Which one?
    > db.map.find({'x-audit-op':'none'})[0]['phone']
    "143"  (OK, this is a valid short number)

# Number of phone numbers found to contain invalid characters

    > db.map.find({'x-audit-op':'reject'}).count()
    5

# Which ones?
    > db.map.aggregate(
    [
        {'$project':{'x-audit-op':1,
                     'phonenum':{'$ifNull': ['$phone',
                                 {'$ifNull':['$contact:phone',
                                  {'$ifNull':['$contact:mobile', '$contact:fax']}]}]}}},
        {'$match':{'x-audit-op':'reject'}},
        {'$group':{'_id':'$x-audit-op', 'bad_phones':{'$addToSet':'$phonenum'}}}
    ])
    'bad_phones': ['+41 44 635 87 56 oder +41 44 635 81 12;Notfall: +41 635 81 11',
                   '043/2992010',
                   '+41 44 299 44 1?',
                   '+41 44 301 02 01;+41 79 449 63 00',
                   '+41 44 557 4348 \u200e']...

# Number of phone numbers that couldn't be recognized by the script and need
  manual inspection

    > db.map.find({'x-audit-op':'inspect'}).count()
    7

# Which ones?
    > ...code similar to the one above, except that we look for 'x-audit-op'='inspect'
    'strange_phones': ['+41 44 266 25 2', '+41 62 765 8222', '+41 01 311 70 00',
                       '+41 44 268 66 9', '+41 32 5133158', '+413116100',
                       '+41 44 288 388']
```

All phones in the 'bad_phones' and 'strange_phones' contain problems that need manual resolution. I fixed many of these phone numbers directly in the real OSM database (see user ihk123's change history) with the iD in-browser editor. For some of them it was necessary to visit the web page of the respective point-of-interest to find the correct phone.

8 Additional exploration with MongoDB

As an additional exercise, I tried to find the restaurants that are closest to the geographical center of Zurich. This can happen via *geospatial* queries based on the 'pos' coordinates available for each node.

Before performing such queries we need to create a *2d* geospatial index of the data:

```
> db.map.createIndex('pos', '2d')
```

We can find the geographical center of Zurich with the following query:

```
> db.map.find({'place':'city'})[0].pos
[ 47.3685586, 8.5404434 ]
```

The nearest restaurants can be found like this:

```
> db.map.find({"amenity":"restaurant",
              "pos":{'$near':[47.3685586, 8.5404434]}},
             {"name":1, "pos":1, "_id":0})
{ "name" : "Old Fashion Bar", "pos" : [  47.3686486,  8.5406992 ] }
{ "name" : "Milchbar",        "pos" : [  47.3689961,  8.540223 ] }
{ "name" : "Strozzi's",       "pos" : [  47.3693,     8.5405687 ] }
{ "name" : "Münsterhof",      "pos" : [  47.3700185,  8.5403502 ] }
{ "name" : "Orsini",          "pos" : [  47.3701032,  8.5401242 ] }...
```

9 Additional ideas

9.1 More automated cleaning

There are many data fields that are amenable to automated data cleaning. This can be more or less difficult and efficient depending on how free-form or formalized the data is.

One tool that can provide great help in this endeavour is the OSM's Taginfo web service. This is basically a database containing information about all used OSM tags, including their usage statistics, and all the values that have ever been attributed to them. Being a web service with a public API, it can assist us in programmatically verifying from within our audit/clean scripts if a certain tag or its value is well used.

As already said tags and their values differ significantly in terms of effort needed to audit and fix them. We can make an informal "cleaning difficulty" rating of the tags giving at the same time some recipes for auditing them:

- **Easiest** The easiest tags are fields like 'country' which can contain only a limited set of well know values and for which golden standards readily exist. The extreme case are keys like 'automated' that can contain only "yes" or "no" Boolean values. It suffices to check them against the golden standard list.
- Next come tags that contain *structured* data (like URLs, emails, phone numbers, street names, opening hours and many numeric fields like 'ele', 'pos', and 'width'). These can be audited and cleaned relatively easy via regular expressions.
- Next are tags that can have a (potentially large) variety of values, which are more or less specified/standardized by the OSM community. Examples are the 'amenity', 'building', 'crossing' and 'cuisine' keys. The biggest issues with these keys are spelling errors and non-uniform writing of the same value (e.g. different letter case). Sometimes the authors invent new values which can be synonymous to something already existing. The way to audit & fix such fields is 1.) first applying a spell check (e.g. via PyEnchant spell checking module), 2.) then consolidating values having different textual representation, and 3.) finally using the OSM Taginfo database to see if the given value is commonly used or if there are synonyms that are more common.
- More difficult are tags that contain a well-specified information type, but which have simply too many ways for entering valid data values. This means it is not easy to come up with an all-encompassing regexp scheme for them. Example is 'addr:housename'. One possibility for approaching this kind of tags could be employing machine learning algorithms for classifying them. Such an algorithm could be trained against the whole value set stored in Taginfo.
- **Hardest** Other keys like 'description', 'brand', 'name' and 'note' contain completely arbitrary natural language data. Apart from verifying them with some advanced linguistic AI algorithm, the most practical check otherwise

would be a spelling check. It is also possible to try to detect the language (via simple "keywords" search) in which the value is written and to add a language identifier to the key–e.g. for notes written in German the key can be changed from 'note' to 'note:de'. For some keys like 'brand' or 'name' at least some checks can be performed by correlating with other info sources (e.g. phone directories) (see also below).

The audits listed above are concerned primarily with the *validity* of the tags values. Addressing the data *accuracy* is however also of great importance. The challenges with this are due mainly to the absence of comprehensive golden standards (which is an issue with GIS in general). Below I present some ideas for ways of verifying the accuracy of certain tags:

- **URLs** in keys like 'website', 'facebook', and 'wikipedia' can be easily checked by sending an HTTP request to them.
- **Phone numbers** at least in Switzerland, where reverse phone number lookup is legal, can be easily searched for and correlated to address/name on tel.local.ch. The downside of this site is that it does not have a RESTful API, so the information has to be extracted via web scraping.
- **Company names** in the 'name' field can also be checked for some accuracy (in case the node contains also a 'phone' key) using the same reverse phone directory approach.
- **Addresses** in general can be verified by using a *reverse geocoding* service based on the 'pos' coordinates given for each node. An example for such a service is ArcGIS. I only hope the chosen service does not use the same OSM data in the background.

9.2 Ideas for improving the OSM dataset

In my opinion one of the biggest issues with a community-compiled database like OSM is the handling of *stale* data. GIS data can be notorious for quickly becoming outdated. This includes both cartographic features like roads and buildings, and manually entered tag attributes. Devising a clever way to handle this challenge would be of extreme value to the community.

Some proposals to this end:

- Many notes stored under the 'note' key contain time-boxed data. Such values for example can be inspected for dates or date phrases in the past, e.g. a note saying "road closed until 11/25/2014".
- There are also a lot of tags like 'FIXME' left by the contributors to mark records that need verification or cleanup. I can think of an automated system that extracts such records and presents them as *Human Intelligence Tasks* on micro-work platforms like Amazon's Mechanical Turk. (Who will pay for this is another question).
- A possible way to verify if roads and buildings have changed is by employing image processing and recognition algorithms, whereby an aerial photo (e.g. from Bing) is overlayed on a map generated from the OSM data, and both images are compared. This approach can probably be extended to identify different terrestrial objects that are still not mapped in OSM.

Another aspect of the OSM dataset improvement is of course its *expanding* with new cartographic data. There is a myriad of things that due to being inherently geolocated make an excellent *geoinformation* suitable for entry into OSM. These range from demographic statistics, to geodetic data, to ecologic data to economic data. The crucial moment here is to be able to automatically extract such data from their specialized databases, transform and store it into OSM–actually a typical implementation of an ETL process. Some possibilities I find attractive are:

- **Geodetic & Geological information** The official Swiss topographic site SwissTopo for example contains a wealth of geodetic & geological information, such as cadastral points, survey networks and soil/rock types. Probably ETL-ing information from this site alone into OSM is worth a ton of accolades from the OSM community.
- **Aviation information** Enhancing OSM with aeronautical information (airspaces, airways and waypoints) can also prove to be of value, if not to the professional aviation circles (for which data quality and reliability is of utmost importance), at least to the research, construction and hobby communities.
- **Lake and river relief** How about adding relief and depth information to lakes and rivers. The scuba-diving and sailing clubs will be happy. I know such bathymetric data exists for oceans (e.g. the freely available NOAA's ETOPO1 Global Relief Model), maybe someone has also compiled something similar for inland water bodies?

9.3 Some application ideas based on analyzing open-source map data

Having a free GIS database opens unlimited possibilities for all kind of applications. Several examples related to some of my interests:

- Learning geospatial analysis techniques–OSM, being freely accessible forms a perfect basis for training oneself in cartographic and geoanalysis/geocoding techniques (e.g. with the Turf.js library).
- Calculating electromagnetic field intensities generated by high-voltage overhead power lines and pinpointing excessive irradiation of inhabited areas. This is easily achievable, since OSM normally contains information about the power lines and their voltages and carried power.
- Calculating the best locations from where plane-spotters can have optimal view at planes flying on a given route. This is achievable if OSM would contain the aeronautical information mentioned above.

10 Conclusion

The OSM database has a very lightweight schema. As a consequence, the major information payload is carried by a very liberal tagging mechanism based on free-form key:value pairs. This gives enormous expressive power to the cartographers, but at the same time provides ample opportunities for entering ”dirty” data. As a result the database contains significant amount of discrepancies and inaccuracies, which in turn necessitate very complex software for robust processing.

A very promising way to deal with this kind of challenges is by employing automated data analysis and machine learning techniques to perform data quality audit and cleaning.

The possibility to use a public API for querying and updating the OSM database, together with the enormous data analysis and processing capabilities of Python, provide a very powerful toolbox for implementing such an automatic data cleaning.

These capabilities are greatly leveraged by using a NoSQL database like MongoDB, which is excellently suited for importing and querying such free-form data.

I’m currently considering enhancing the phone number auditing script so that it can store the cleaned results directly into the OSM database (after validating this with the Swiss OSM community).