Technical Guidelines to Extract and Analyze VGI from Different Platforms

Juhász, Levente; Rousell, Adam; Arsanjani, Jamal Jokar

Published in:
Data

DOI (link to publication from Publisher):
10.3390/data1030015

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

data

MDPI

Data Descriptor

Technical Guidelines to Extract and Analyze VGI from Different Platforms

Levente Juhász [1,*], Adam Rousell [2] and Jamal Jokar Arsanjani [3]

[1] Geomatics Program, Fort Lauderdale Research and Education Center (FLREC), University of Florida, 3205 College Avenue, Fort Lauderdale, FL 33314, USA

[2] GIScience Research Group, Heidelberg University, Im Neuenheimer Feld 348, Heidelberg 69120, Germany; adam.rousell@uni-heidelberg.de

[3] Geoinformatics Research Group, Department of Planning and Development, Aalborg University Copenhagen, A.C. Meyers Vænge 15, Copenhagen DK-2450, Denmark; jja@plan.aau.dk

* Correspondence: levente.juhasz@ufl.edu; Tel.: +1-954-577-6851

Academic Editor: Eric Vaz

Received: 5 July 2016; Accepted: 15 September 2016; Published: 24 September 2016

Abstract: An increasing number of Volunteered Geographic Information (VGI) and social media platforms have been continuously growing in size, which have provided massive georeferenced data in many forms including textual information, photographs, and geoinformation. These georeferenced data have either been actively contributed (e.g., adding data to OpenStreetMap (OSM) or Mapillary) or collected in a more passive fashion by enabling geolocation whilst using an online platform (e.g., Twitter, Instagram, or Flickr). The benefit of scraping and streaming these data in stand-alone applications is evident, however, it is difficult for many users to script and scrape the diverse types of these data. On 14 June 2016, a pre-conference workshop at the AGILE 2016 conference in Helsinki, Finland was held. The workshop was called "LINK-VGI: LINKing and analyzing VGI across different platforms". The workshop provided an opportunity for interested researchers to share ideas and findings on cross-platform data contributions. One portion of the workshop was dedicated to a hands-on session. In this session, the basics of spatial data access through selected Application Programming Interfaces (APIs) and the extraction of summary statistics of the results were illustrated. This paper presents the content of the hands-on session including the scripts and guidelines for extracting VGI data. Researchers, planners, and interested end-users can benefit from this paper for developing their own application for any region of the world.

Data Set: Available as supplementary files at https://github.com/jlevente/link-vgi/tree/master/sample_datasets

Data Set License: Readers should onsult the documentation of each VGI platform for specific license details.

Keywords: Volunteered Geographic Information; OpenStreetMap; Flickr; Instagram; Mapillary

1. Introduction

Since the emergence of Web 2.0 technologies, a massive amount of user-generated content (UGC) have been interactively generated by the public, which has provided us with an alternative source of information [1]. Even though the portion of explicitly geotagged UGC (where geographical coordinates are attached to each piece of data [2]) is not seemingly high (ranging from 2% in Twitter to 25% in Instagram [3,4]), this trend arguably generates a significant mass of data. This is highly important and useful for geography and the GIScience domain, as every citizen can be counted as a volunteer mapper or a "sensor". These volunteers can collect, edit, and share geographical information of their

surroundings [5]. Thanks to the wide availability of GPS-enabled devices (e.g., smart phones) and the location-based services installed on them, as well as the professional infrastructure behind VGI projects, the quality of the collected geoinformation and, therefore, the usefulness of the VGI could potentially be comparable to, or even better than authoritative datasets. This was shown, for example, in the case of OSM compared to map data from national agencies [6,7] and Mapillary images vs. Google StreetView [8]. This is because they can be more detailed in terms of attributes, whilst potentially being collected at a finer geographical scale and being more up-to-date [2,8,9]. Therefore, a new era in gathering geospatial information has begun in GIScience in the form of VGI [10]. The VGI topic has attracted a huge amount of attention in research across various disciplines ranging from environmental management to social sciences and has, in general, the potential to revolutionize science [11]. Recent trends also indicate that creators and users of VGI have started to utilize some user-generated content to improve the quality or enrich another VGI source [12], which is a new research area to explore.

Due to the high importance of access to these data for end-users, it is evident that there is a need for sharing mechanisms and tools for collecting them with a minimal amount of effort invested by users whilst maintaining higher certainty about data. Thus, this paper aims to openly present the guidelines, tools, and scripts for collecting VGI across multiple platforms which were presented and tested at the LINK-VGI workshop. Readers can use these tools and scripts for collecting VGI from various platforms.

This paper addresses methods for accessing explicitly geotagged UGC. However, as mentioned above, this is only a small fraction of user generated content. It is important to note that there are other techniques that can be used to spatially locate UGC with spatially implicit information (where geography is expressed as place names or toponyms) [13]. Studies show that analyzing the textual information embedded in UGC can lead towards the extraction of more geographic data. For example, a prototype system solely relying on the textual information of Tweets and Flickr photos was able to geolocate social media contributions and then report forest fire locations without using any explicit geographic information [14,15]. Recent big data research turned to natural language processing that can potentially be even more useful for enriching location information. The use of more sophisticated natural language processing methods is common in big data research. Applying natural language processing tools to "geoparse" UGC is also an ongoing research direction and is mainly applied to Twitter [16–18]. A recent study also predicts location and estimates errors of Flickr and Twitter data [19].

The remainder of the paper is structured as follows: Section 2 presents a brief description of the sample data provided at the workshop while Section 3 along with a series of Appendices provide working examples of VGI extraction from various sources. Section 4 applies these examples on two case studies with potential applications. All materials (including sample datasets and code examples) along with working case studies can be found in a GitHub repository via the link provided in the Supplementary Materials section.

2. Data Description

Sample datasets (extracted from Twitter, Instagram, Flickr, Foursquare, Wheelmap, and Mapillary) are also provided along with this paper in a GitHub folder (https://github.com/jlevente/link-vgi/tree/master/sample_datasets [20]). The geographic extent of these sample datasets cover Helsinki in Finland and Heidelberg in Germany (Figure 1). Table 1 describes these datasets in greater detail. These datasets serve as examples of VGI data that can be extracted from public APIs (Application Programming Interfaces). In some cases, additional attributes could have been extracted from these platforms according to each API documentation, though, for the case of the exercises in the workshop, these additional attributes were not required. It is also important to note that these services often change over time in terms of data access. Practically, it means that methods to scrape VGI data and the data that can be extracted need to be revised from time to time. Potential applications based on

these datasets would be accessibility analysis, tourism analysis, city planning, and land use/cover monitoring, to name a few [21,22].

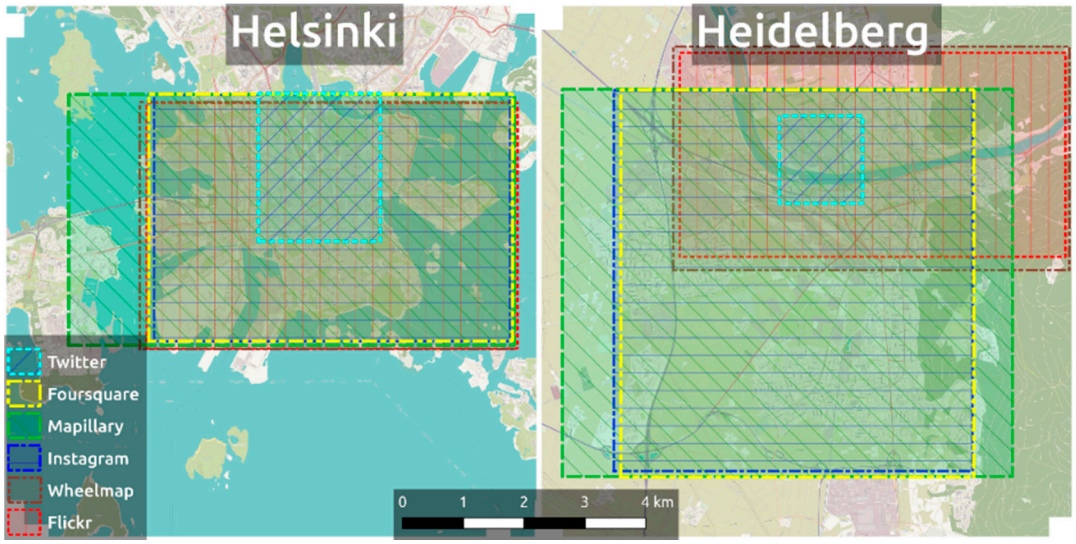

Figure 1. Geographic extent of sample datasets.

Table 1. Some metadata of sample datasets.

Data Source—API Method	Attributes	Data Volume *	File Names
Flickr *photos.search* (https://www.flickr.com/services/api/flickr.photos.search.html)	*"photo_id", "url", "lat", "lon"*	1850 + 625	flickr_*[city]*.csv
Foursquare **—*/venues/search* (https://developer.foursquare.com/docs/venues/search)*/venues* (https://developer.foursquare.com/docs/venues/venues)	Venues: *"id", "name", "tipcount", "checkincount", "usercount", "url", "photo_count", "tags", "lat", "lon"* Photos: *"photo_id", "source", "created_at", "photo_url", "user_id", "user_name", "venue_id"*	Venues: 5141 + 1239; Photos: 12783 + 2233	foursquare_venue_*[city]*.csv foursquare_venue_photos_*[city]*.csv
Instagram**—*/locations/search locations/media/recent* (https://www.instagram.com/developer/endpoints/locations/)	Locations: *"id", "name", "lat", "lng"* Photos: *"id", "username", "user_id", "likes", "tags", "comment", "text", "users_in_photo", "filter", "url", "lat", "lon", "photo_url", "location_id", "created_at"*	Locations: 819 + 286; Photos: 38427 + 7577	instagram_locations_*[city]*.csv instagram_photos_*[city]*.csv
Mapillary—*/search/im* (https://a.mapillary.com/#get-searchim)	*"user", "key", "lon", "lat", "url", "captured_at", "ca" (camera angle)*	21078 + 14052	mapillary_photos_*[city]*.csv
Twitter—*/search/tweets* (https://dev.twitter.com/rest/reference/get/search/tweets)	*"username", "tweet_id", "text", "created_at", "lat", "lon"*	129 + 41	tweets_*[city]*.csv
Wheelmap—*/nodes* (http://wheelmap.org/en/api/docs/resources/nodes)	*"osm_id", "name", "lat", "lon", "category", "type", "accessible"*	2180 + 2436	wheelmap_*[city]*.csv

* Helsinki + Heidelberg; ** Table joins can be defined on the *"id"—"venue_id"* (Foursquare) and *"id"—"location_id"* (Instagram) properties.

3. Material and Methods

3.1. Software Requirements

Programmatically interacting with VGI data from public APIs requires a set of software components to be set up. During the LINK-VGI workshop, we illustrated the process with Python 2.7+ and a set of additional packages (*tweepy, pyshp, psycopg2, python-flickrapi, python-instagram*), along with R statistics and a few of its external libraries (*ggplot2, ggmap, plyr, wordcloud*). As for traditional GIS environments, QGIS and PostgreSQL (+PostGIS) were used. Detailed installation instructions can be found on the public GitHub repository of the workshop (https://github.com/jlevente/link-vgi/blob/master/requirements.md). In the remainder of this section, we illustrate each step with a series of

Appendices. Appendices are code snippets written in Python (unless otherwise noted) with which one can reproduce results and explore the potential of VGI data extraction.

3.2. Interacting with APIs

An API standardizes the ways of interaction between software components. In web environments, it is a well-defined request-response system where servers respond to client application requests. A common practice is having different endpoints for different sets of functionalities. For example, http://api.twitter.com/1.1/users is responsible for operations related to users (e.g., recommending friends to follow), whereas http://api.twitter.com/1.1/geo has methods related to the geospatial domain (e.g., searching for Tweets in a given radius). APIs usually have different, well documented methods (functions) implemented for different functionalities (e.g., querying data, inserting new data). These documentations define accepted parameters for each method along with the expected output (response). A response is usually a JSON or XML document that can be further processed. In addition, many APIs make use of the type of HTTP request (POST, PUSH, GET, etc.) which can be set by the requesting agent. This type is then used by the responding server to identify the type of action being requested (i.e., using GET for requesting data, or PUSH to update information in the server's data structure).

A number of platforms require an API key (i.e., registering for the service) to be provided from the developer's side. This is to protect user's privacy, monitor usage intensity, and, in general, to govern the different levels of data access. A general guideline is that an application should only be able to execute operations for which it is authorized. An example could be OpenStreetMap's JOSM editor, where users can access data (e.g., download via API) but are only able to upload changes once logged in with their credentials (i.e., acquired permissions to perform uploads). Different APIs implemented different authentication systems, some being completely open and public (Overpass API (http://wiki.openstreetmap.org/wiki/Overpass_API)) and some requiring a registered application before any interaction (Instagram API). Table 2 lists the process and parameters needed from each platform.

Table 2. Information on the Volunteered Geographic Information (VGI) services and the corresponding Application Programming Interfaces (APIs).

Service	Registration Address	Instructions	Parameters Needed
Wheelmap.org	http://wheelmap.org/users/sign_in (with an OSM profile)	Once logged in, navigate to your *"Edit profile page"*	authentication_token
Flickr	Yahoo (register for Yahoo or use an existing Yahoo account)	Create a new application (https://www.flickr.com/services/apps/create/) and check *"Sharing & Extending"/Your API keys"*	api_key, api_secret
Mapillary	http://www.mapillary.com/map/signup Create a Mapillary profile	Once logged in, navigate to https://www.mapillary.com/app/settings/developers and hit Register an application.	client_id
Twitter	Sign up for Twitter	Go to Twitter Apps (https://apps.twitter.com/app/new) and hit Create New App. Check your credentials under *"Keys and Access Tokens"*	api_key, api_secret

As mentioned above, different levels of access require different levels of authentication. For example, a registered basic Twitter application is not able to use the */geo* endpoints, nor to pull data from a private Twitter account. However, if a user explicitly authorizes this application to make requests in his name (and, therefore, to reach these endpoints), data can be acquired. Different platforms implement different authentication methods to govern data usage. When developing an application for data mining, one should always consult the API documentation and the Terms of Service (or Terms of Usage). An example of setting up credentials for authenticated requests with Twitter is provided in Appendix A. The example allows the developer to acquire two additional parameters ("*access_token*" and "*token_secret*") in addition to the parameters associated with the application with which authenticated requests can be made.

3.2.1. Making Requests in a Python Environment

Although APIs can be used from any environment that can handle HTTP requests, it is beneficial to make use of existing API wrappers. These wrappers are usually developed based on the API documentation (often by third-parties) for the purpose of tackling some technical challenges for developers and making interactions easier (i.e., handling authentication process, implementing object classes, etc.). Using API wrappers in Python is an easy way to start. Some examples are *tweepy* (http://www.tweepy.org/) for Twitter and *python-instagram* (https://github.com/facebookarchive/python-instagram).

In some cases, however, such as with Wheelmap, easy to use API wrappers are not readily available. With some basic knowledge of HTTP communication, data can still be obtained relatively easily. The bulk of the HTTP communication can be done through built-in packages (*urllib2* in Python, for example) specifically aimed at facilitating communication of data over HTTP connections in a straightforward manner.

3.2.2. API Methods

API methods are the most important elements of data interaction. There are different functionalities defined for every API, together with the list of properties we can use and the expected output of the methods. When working with a platform, the API documentation is the starting point where one can figure out what options are available and what methods suit best for the data collection. The documentation for different platforms can be accessed on their website. Some services have more thorough documentation than others. Appendix B illustrates the process of searching for Tweets within an area using *search/tweets* method from Twitter's REST API. The request contains a *geocode* parameter that consists of a pair of latitude and longitude values and a radius value. This is also an example of using an existing API wrapper from Section 3.2.1.

As mentioned earlier, not all APIs have a wrapper available to make communication easy. One example of these is Wheelmap. In the case that a wrapper class is not available, data can normally be obtained through connection with the API through direct HTTP requests. Appendix C showcases the interactions without an API wrapper. Within Python, there is the *urllib2* built-in package that can perform such tasks. Using the *urllib2* package, calls are made up of two components: a request and a response. The request is sent by the client (the Python script) to the API and basically asks for some specific action to be performed along with additional information such as search parameters, authentication tokens, and requested response data type. This request can be an action such as getting a list of features, creating a new feature, or any other process of the service that the API exposes. In this case, the simplest approach of asking for some data is used. The response component is what the API sends back to the client and contains the data obtained and/or response codes indicating whether the operation was successful. Compared to the previous examples, one can note that there are additional steps performed in this solution. Without available API wrappers, the programmer is responsible for building direct URLs and handling pagination for the response dataset, amongst other methods that may need to be implemented. Authentication for Wheelmap is done by providing the *"api_key"* parameter in the requests.

3.3. Exporting Data from APIs

Since different APIs use different data structures that are often not relational and are not in a tabular format, the next step in VGI data collection after acquiring the data from a service is usually exporting the result set into a widely used data format. The general idea is to reshape the data structure so it can be integrated into other systems. The process involves looping through result sets and adding data to a result set of standard format. Outputs can be anything that can be created in programming environments. If data falls in the geospatial domain, geospatial data formats can be used as well.

3.3.1. Plain CSV

Plain CSV files are often used as an exchange format as they are easy to handle. A Python code snippet to get tweets within a radius of a given point and write results to a CSV file is given in Appendix D. In this example the Python core package *csv* is used to create a CSV file of geocoded Tweets. Properties to export can be decided either when examining the result set or can also be determined from the Result section of Twitter's API documentation (https://dev.twitter.com/rest/reference/get/search/tweets). For simplicity, this code snippet exports some basic information such as username, tweet id, message of the post, timestamp of post, and a latitude-longitude coordinate pair corresponding to the location. As fields can be missing in the response set, it is often useful to implement some error handling solutions in the code (e.g., try—except statements in Python).

3.3.2. GeoJSON

GeoJSON is a common interchange format gaining popularity over the recent years. It is commonly used in web environments to transport geospatial data but most Desktop GIS software can read and write the format as well. Some API methods return GeoJSON directly meaning that there is no need for conversion. However, often GeoJSON documents have to be built manually. The process of building a GeoJSON file is illustrated in Appendix E. This Python function can be used to extend Appendix C with the export functionality so that all Wheelmap nodes could be further processed in a desktop GIS software.

3.3.3. Shapefile

Shapefiles are well known to all GIS professionals, therefore, it is useful to learn how to create them in Python. Appendix F queries Mapillary photos in a given area and exports them as a shapefile. The example uses the *search/im* (https://a.mapillary.com/#get-searchim) method from the Mapillary API to access images nearby a specified location expressed by latitude and longitude coordinates. Similar to the *urrlib2* package seen in the Wheelmap example, another package called *requests* can be used to handle HTTP requests in the absence of an available API wrapper. The process starts with defining the request parameters and manually building the URL for the GET requests. The code snippet could be easily extended to handle multiple result pages as well. This example also uses the *pyshp* package for writing a shapefile.

3.3.4. PostGIS

PostgreSQL is the leading open source Relational Database Management System and can be considered as a geographic data store with its PostGIS extension. In addition, it allows the execution of many geospatial operations in a highly customizable manner. The database can be integrated in the data collection process and then it can be used not just as a data store but as a powerful processing framework or even for collaborative work. Appendixs G and H show the process of exporting locations from OSM's OverpassAPI identifying where drinking water can be obtained. The first step is creating a database and then setting it up for the data. PostGIS is an extension that needs to be enabled for the database. Similarly, *hstore* can be used to store key-value pairs. Since PostgreSQL uses a database schema to describe data, the table structure needs to be defined first (Appendix G). Connecting to the database and populating it with data then can be done within a Python environment (Appendix H).

3.4. Extracting Summary Statistics in an R Environment

In addition to accessing VGI platforms and standardizing formats, exploratory analysis provides better understanding of the nature of these data. R statistics is a widely used statistical framework to analyze VGI sources. In general, however, standardized outputs scraped from APIs can be processed in any statistical software. The purpose of this step is to explore data, create charts, and apply statistical tests.

The following example uses data from the Instagram API. The harvested dataset contains information of photos posted to Instagram since 1 January 2015 in downtown Helsinki. A sample dataset of Instagram locations and Instagram photo metadata are provided in a GitHub folder (https://github.com/jlevente/link-vgi/tree/master/sample_datasets). Please note that Instagram API has changed its policy as of 1 June 2016 (http://developers.instagram.com/post/133424514006/instagram-platform-update). All registered applications start with limited access to data and thus the method presented above does not work with real data. However, API methods have not been changed and in theory it is possible to obtain a higher level of access for Instagram.

This data allows us to extract insights about popular places in Helsinki by quantifying data upload intensity and extracting basic measures. This ultimately leads towards an understanding of how Instagram users post photos.

3.4.1. Data Access

The importance of this first step is to actually load the data into R. It can be done by connecting R to PostgreSQL with the *RPostgreSQL* package. Importing many other formats, such as shapefiles or even JSON documents, is also possible. This example uses the quickest way to get started, which is reading CSV files (Appendix I).

The import of data results in three data frames for different datasets. The *head()* and *summary()* functions can be called to examine if the data is correctly loaded. At this point, all the powerful functionalities of R can be used, such as *nrow(locations)*, which yields that the data frame contains 819 locations. To visualize the spatial distribution, a map can be drawn as an R plot with the *ggmap* package that extends the functionality of *ggplot2* with handy tools to manage spatial data, such as loading background tiles [23]. Once all necessary packages are loaded, Appendix J can be used to generate a map of locations (Figure 2).

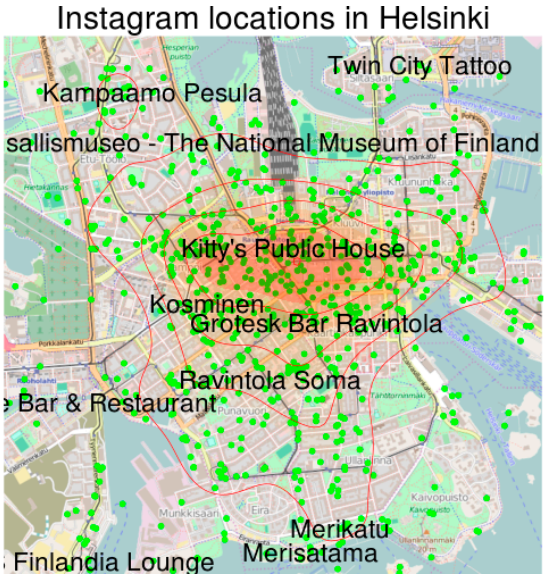

Figure 2. Illustration of Instagram locations along with their names.

3.4.2. Data Exploration and Analysis

Since the data is already imported in an R workspace, further explorations are relatively easy. For example, the total number of photos and summary measures by locations can be extracted with simple commands. Distributions can also be visualized as simple histograms. Appendix K shows a few examples of data explorations along with the extraction of 20 the most popular places in terms of unique users (Figure 3).

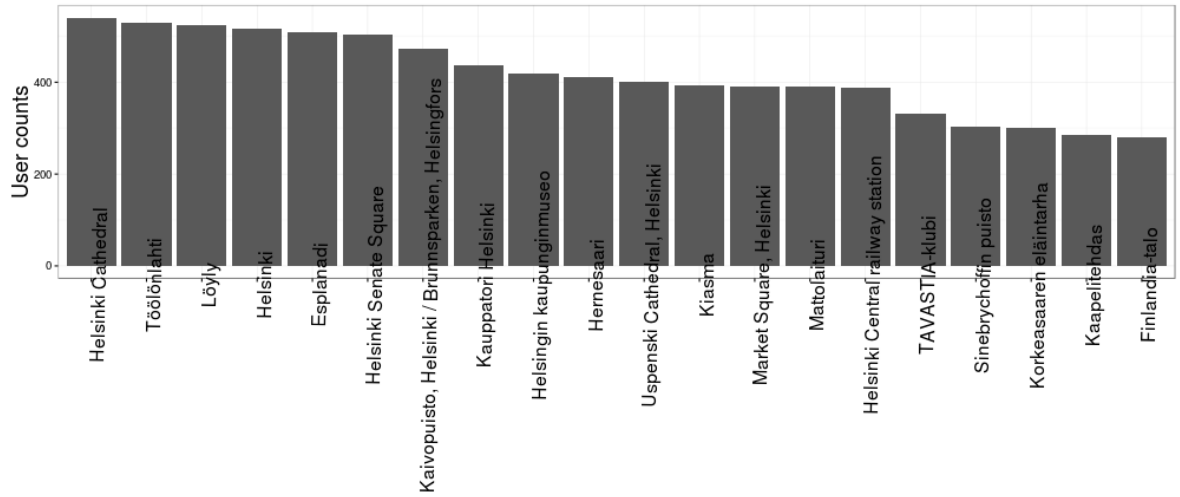

Figure 3. A bar plot showing user counts for the 20 most visited Instagram locations.

Similarly, upload intensity (number of photos per weekday) can be extracted as shown in Appendix L, or even wordclouds can be generated from the hashtags attached to each photo, revealing insights of the conversation from the data collection period (Figure 4, Appendix M).

Figure 4. The word cloud of tags in Helsinki.

As R is a powerful statistical framework, we can easily test hypotheses that we generate. For example, we can fit a simple linear regression between "likes" and "users tagged" in each photo if we suspect any relationship between them (Figure 5, Appendix N).

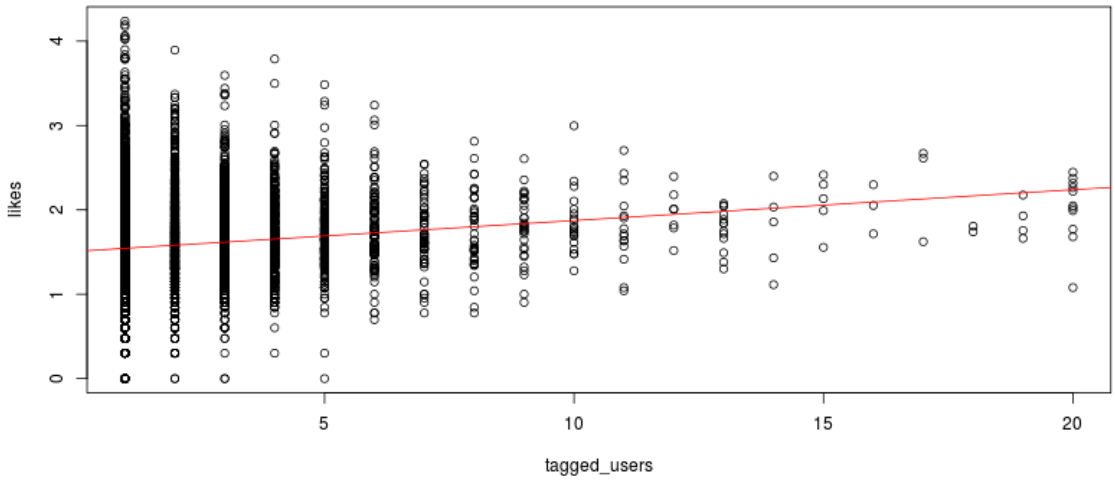

Figure 5. A regression line between the "likes" and "tagged_users".

4. Case Studies

During the workshop, some basic case studies were also provided around extracted VGI data. Step by step tutorials are accessible on the workshop's GitHub repository (https://github.com/jlevente/link-vgi/blob/master/workshop/tutorial.md) in the Case studies section. Below are some short interpretations of case studies whereby data from VGI sources can be linked, visualized, and have secondary datasets created.

In some instances, data scraped from APIs do not contain references to particular real world places but, instead, indicate the locations of particular events, such as the where photographs were taken. Such a dataset is Flickr. In that case, it is possible for a photograph to have data associated with it indicating where the image was captured. When looking at this data overlaid on a map, it is clear that a number of images are clustered in particular locations which could be used as an indicator for points of interest (POIs) within the environment. Using clustering methods (see [24]), the Flickr data can be processed to provide such POIs. It should be noted that if only the location of the photograph is taken into account, and not the direction and focal distance, then the point extracted from the clustering methods will in fact only indicate where photos are taken and will likely not be at the location of the feature which forms the focus of the image. Figure 6 shows some potential POIs extracted from the Flickr dataset for the central Helsinki area.

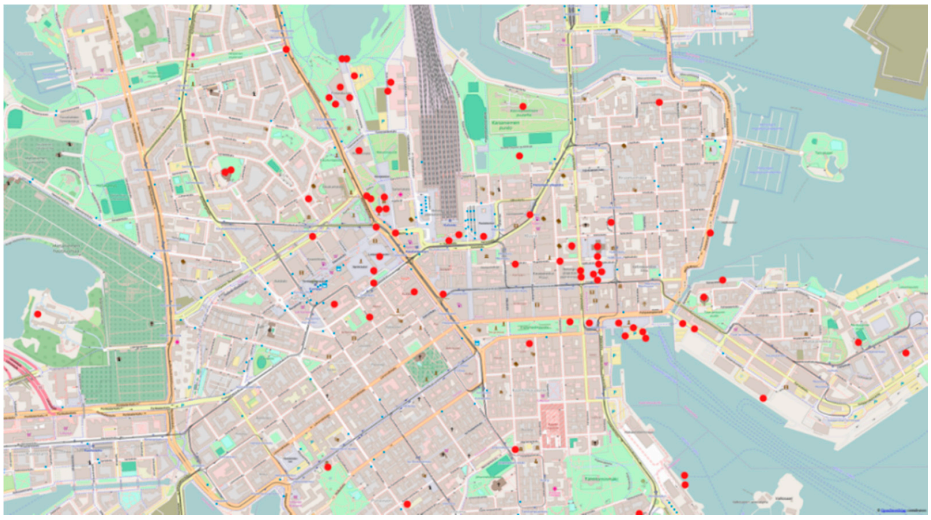

Figure 6. Potential points of interest (POIs) within Helsinki based on Flickr data.

When looking at popular places, for wheelchair users it is important to know if the place is accessible to them or not. For example, it is not particularly useful to plan a trip to several cafes during a vacation in a city if the person doing the trip cannot use the cafe itself.

Wheelmap is a service developed by Sozialhelden in Berlin that aims at allowing users to view and edit whether a place is accessible to wheelchair users. The base data used are POI features from OSM, with the accessibility information being stored against the POI in the OSM dataset. The possible values for this accessibility are "yes", "limited", "no", and "unknown". They provide an API that can be used to harvest the information contributed to the Wheelmap service.

It is possible to create a basic link between the Wheelmap and Foursquare datasets based on the name given to the POI. Though this is by no means perfect as multiple places can have the same name or the same place could have a different spelling between datasets. As a simple exercise for understanding accessibility of popular places it can generate a powerful representation. Using the datasets and QGIS, Figure 7 was produced for the central Helsinki area. In that figure, the size of the circle represents popularity based on Foursquare check-ins (larger = more check-ins), and the color represents the accessibility (red = not accessible, yellow = limited accessibility, green = accessible, and grey = unknown).

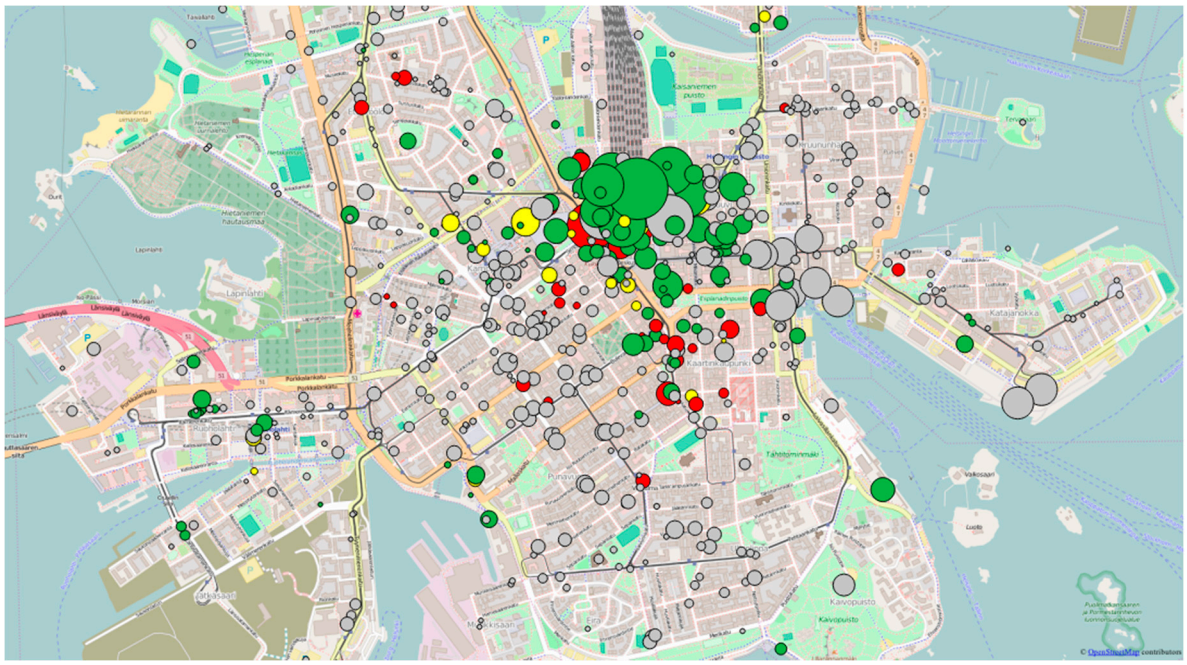

Figure 7. Accessibility of popular places in central Helsinki.

In addition to being able to portray the accessibility of places, the two datasets can be compared to determine the amount of spatial difference between the same POIs in each dataset. As the same POI (say a particular cafe) is recorded in both datasets, but via different methods, there is often a discrepancy which makes linking by location difficult. By measuring the distance between two POIs with the same name, it is possible to get a limited understanding of how much difference is present. Figure 8 shows lines linking features with the same name. All lines greater than 200 m in length were removed from the data as it is likely that these mostly represent differences between places that have the same name multiple times (i.e., McDonald's fast food restaurants). From performing basic statistics on these line features (not longer than 200 m) within QGIS, an average displacement between the POIs in the two datasets is found to be 39 m.

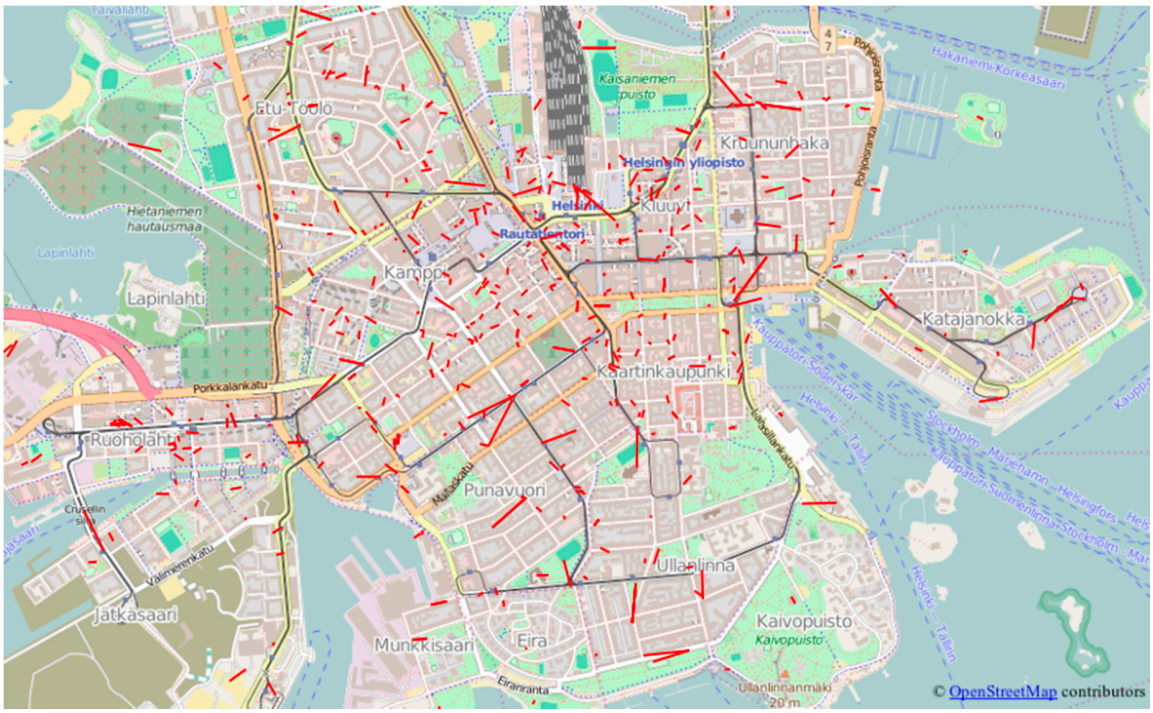

Figure 8. Spatial displacement between Foursquare and Wheelmap POIs in Helsinki.

5. Discussion

With the evolution of user-generated content on the Internet, researchers often face problems with data collection. The collection process contains unique solutions depending on the data source and type of data, and there is no standard way to extract data for research purposes. This can be confusing, especially without a background in programming. However, most VGI platforms provide public APIs, which make it possible to extract potentially useful data for answering many research questions in a relatively straightforward manner. In this paper, we provided an overview of methods used in API interactions when working with VGI data from multiple platforms. Starting from the description of APIs through standardizing output formats, the paper presents some easy and generic ways to extract meaningful information from the dataset in a statistical environment. Our examples provide Twitter, Instagram, Mapillary, Wheelmap, Flickr, and OSM data and aimed to encourage researchers to integrate these solutions in their research methodology. We also provide an open repository consisting of code and step-by-step case studies available on a GitHub page that can be further used for discussion and collaboration. The materials are based on the LINK-VGI pre-conference workshop held in Helsinki, Finland on June 14 during the AGILE conference. This paper serves as a starting point for researchers exploring public APIs of user-generated content providers, allowing them to start their own data collection campaigns. We also provided some sample datasets illustrating some possible outputs.

Although this paper focuses on the extraction and usage of VGI data, it does not address other common concerns about VGI, such as data quality and credibility [25,26] as well as legal and liability issues [27]. By nature, VGI datasets are not homogeneous and they are not representative of the whole population. For these reasons, we strongly advise against using data acquired from VGI platforms without further investigation and quality checks in research studies. Failure to address these concerns can lead to different and unreliable results when performing data analysis.

Supplementary Materials: It is available online https://github.com/jlevente/link-vgi.

Author Contributions: Levente Juhász, Adam Rousell, and Jamal Jokar Arsanjani contributed to organizing the LINK-VGI hands-on session as well as the writing of the paper. The materials for the paper were prepared by Levente Juhász and Adam Rousell. All authors have read and approved the final manuscript.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Acquiring a User Access_Token and Token_Secret for Authenticated Twitter Requests

```
import tweepy

consumer_token = "Your apps token"
consumer_secret = "Your apps secret"

def get_user_tokens(consumer_token, consumer_secret):
    auth = tweepy.OAuthHandler(consumer_token, consumer_secret)
    print "Navigate to the following web page and authorize your application"
    print(auth.get_authorization_url())
    pin = raw_input("Enter the PIN acquired on Twitter website").strip()
    token = auth.get_access_token(verifier=pin)
    access_token = token[0]
    token_secret = token[1]
    print "With the following tokens, your application should be able to make
    requests on behalf of the specific user"
    print "Access token: %s" % access_token
    print "Token secret: %s" % token_secret
    return access_token, token_secret

access_token, token_secret = get_user_tokens(consumer_token, consumer_secret)
```

Appendix B. Searching for Tweets within a Given Radius around a Center Point Using an Existing API Wrapper

```
import tweepy

# Set your credentials as explained in the Authentication section
consumer_key = "Your consumer key"
consumer_secret = "Your consumer secret"
# Access tokens are needed only for operations that require authenticated requests
access_key = "Authorized access token acquired from a user"
access_secret = "Authorized access token secret"

# Set up your client
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_key, access_secret)
api = tweepy.API(auth)

# Get some tweets around a center point (see http://docs.tweepy.org/en/v3.5.0/api.
 html#API.search )
tweets = api.search(geocode='37.781157,-122.398720,1km')

# Print all tweets
for tweet in tweets:
    print "%s said: %s at %s. Location: %s" % (tweet.user.screen_name,
```

```
tweet.text, tweet.created_at, tweet.geo['coordinates'])
    print "---"
```

Appendix C. Accessing Wheelmap API through Direct HTTP Requests in Python

```
import json
import urllib2

api_key = 'xxx'

class WheelmapItem:
    def __init__(self, name, osm_id, lat, lon, category, node_type, accessible):
        self.name = name
        self.osm_id = osm_id
        self.lat = lat
        self.lon = lon
        self.category = category
        self.node_type = node_type
        self.accessible = accessible

    def getName(self):
        if not self.name:
            return "
        else:
            return self.name.encode('utf-8')

def getWheelmapNodes(ll_lat, ll_lon, ur_lat, ur_lon, page, accessible):
    bbox = str(ll_lat) + ',' + str(ll_lon) + ',' + str(ur_lat) + ',' + str(ur_lon)
    url = 'http://wheelmap.org/api/nodes?api_key=' + api_key + '&bbox=' + bbox +
    '&page=' + str(page)
    if accessible != None:
        url = url + '&wheelchair=' + accessible
    headers = {'User-Agent':'Python'}

    req = urllib2.Request(url, None, headers)

    print (url)
    resp = urllib2.urlopen(req).read().decode('utf-8')
    return json.loads(resp)

# When we get the first load of data we can read the meta info to see how many pages
there are in total

firstPage = getWheelmapNodes(8.638939,49.397075,8.727843,49.429415,1,None)
numPages = firstPage['meta']['num_pages']

# so now we need to loop through each page and store the info
pagedData = []
pagedData.append(firstPage)
```

```
for i in range (2,numPages+1):
    pagedData.append(getWheelmapNodes(8.638939,49.397075,8.727843,49.429415,i,None))

# now that we have the data we should go through and create a list of items
# for now we will store the name, location, category, node type, accessibility
and osm id
items = []
for i in range (0,len(pagedData)):
    page = pagedData[i]
    # go through each item
    nodes = page[‘nodes’]
    for node in nodes:
        item = WheelmapItem(node[‘name’], node[‘id’], node[‘lat’], node[‘lon’],
node[‘category’][‘identifier’], node[‘node_type’][‘identifier’], node[‘wheelchair’])
        items.append(item)

print(‘Total items read: ‘ + str(len(items)))
```

Appendix D. Writing Result Tweets as a CSV File

```
import tweepy
import csv

# Set your credentials as explained in the Authentication section
consumer_key = “Your consumer key”
consumer_secret = “Your consumer secret”
# Access tokens are needed only for operations that require authenticated requests
access_key = “Authorized access token acquired from a user”
access_secret = “Authorized access token secret”

# Set up your client
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_key, access_secret)
api = tweepy.API(auth)

# Get some tweets around a center point (see   http://docs.tweepy.org/en/v3.5.0/api.
html#API.search )
tweets = api.search(geocode=‘60.1694461,24.9527073,1km’)

data = {
    “username”: “”,
    “tweet_id”: “”,
    “text”: “”,
    “created_at”: “”,
    “lat”: “”,
    “lon”: “”
}
# Init a CSV writer
```

```
with open(“tweet_export.csv”, “wb”) as csvfile:
    fieldnames = [“username”, “tweet_id”, “text”, “created_at”, “lat”, “lon”]
    writer = csv.DictWriter(csvfile, delimiter=‘,’, quotechar=‘“‘,
    quoting = csv.QUOTE_MINIMAL, fieldnames=fieldnames)
    writer.writeheader()
# Loop through all tweets in the result set and write specific fields in the CSV
    for tweet in tweets:
        try:
            data[“username”] = tweet.user.screen_name
            data[“tweet_id”] = tweet.id
            data[“text”] = tweet.text.encode(‘utf-8’)
            data[“created_at”] = str(tweet.created_at)
            data[“lon”] = tweet.geo[‘coordinates’][0]
            data[“lat”] = tweet.geo[‘coordinates’][1]
            writer.writerow(data)
    except:
            print ‘Error occured. Field does not exist.’
```

Appendix E. Function to Extend Appendix C with GeoJSON Export

```
# This method takes a list of WhelmapItems and writes them to a GeoJSON file

def exportWheelmapNodes(node_list, file_name):
# Init geojson object
    geojson = {
        “type”: “FeatureCollection”,
        “features”: []
    }

    # Loop through the list and append each feature to the GeoJSON
    for node in node_list:
        # Populate feature skeleton
        feature = {
            “type”: “Feature”,
            “geometry”: {
                “type”: “Point”,
                “coordinates”: [node.lon, node.lat]
            },
            “properties”: {
                “name”: node.getName(),
                “osm_id”: node.osm_id,
                “category”: node.category,
                “node_type”: node.node_type,
                “accessible”: node.accessible
            }
        }
        # Append WheelnodeItem to GeoJSON features list
        geojson[“features”].append(feature)
```

```
    # Finally write the output geojson file
    f = open(file_name, "w")
    f.write(json.dumps(geojson))
    f.close()

# Use it as:
exportWheelmapNodes(items, "wheelmap_nodes.geojson")
```

Appendix F. Writing Mapillary Image Dataset to a Shapefile

```
import requests
import json
import shapefile
from datetime import datetime

mapillary_api_url = "https://a.mapillary.com/v2/"
api_endpoint = "search/im"
client_id = "Your Mapillary Client ID"

request_params = {
    "client_id": client_id,
    "min_lat": 60.1693326154,
    "max_lat": 60.17107241,
    "min_lon": 24.9497365952,
    "max_lon": 24.9553370476,
    "limit": 100
}

# Make a GET requests
photos = requests.get(mapillary_api_url + api_endpoint + '?client_id=' + client_id +
'&min_lat=60.1693326154&max_lat=60.17107241&min_lon=24.9497365952&max_lon=24.9553370476')
photos = json.loads(photos.text)

# Init shapefile
writer = shapefile.Writer(shapefile.POINT)
writer.autoBalance = 1
writer.field('image_url', 'C')
writer.field('captured_at', 'C')
writer.field('username', 'C')
writer.field('camera_angle', 'C')

# Add each photo to shapefile
for photo in photos:
    writer.point(photo['lon'], photo['lat'])
    writer.record(image_url='http://mapillary.com/map/im/' + photo['key'],
username=photo['user'], camera_angle=str(photo['ca']), captured_at=str(datetime.
fromtimestamp(photo['captured_at']/1000)))
writer.save('my_mapillary_photos.shp')
file = open(filename + '.prj', 'w')
```

```
# Manually add projection file
file.write('GEOGCS["GCS_WGS_1984",DATUM["D_WGS_1984",SPHEROID["WGS_1984",6378137,298.
257223563]],PRIMEM["Greenwich",0],UNIT["Degree",0.0174532925199433295]]')
file.close()
```

Appendix G. Setting up a PostgreSQL Database before Inserting OSM Data from OverpassAPI (SQL Statements)

```
-- Add PostGIS extension
CREATE EXTENSION postgis;
-- Add hstore extension to store tags as key-value pairs
CREATE EXTENSION hstore;
-- Init table
CREATE TABLE drinking_water (
     id bigint,
     user varchar,
     user_id int,
     created_at timestamp,
     version int,
     changeset int,
     tags hstore
);
-- Add a POINT geometry column to the table
SELECT AddGeometryColumn('drinking_water', 'geom', 4326, 'POINT', 2);
```

Appendix H. Populating a PostgreSQL Table with OSM Data from OverpassAPI

```
import psycopg2
import json
import requests

# This function calls OverpassAPI and asks for all nodes that contain
"amenity"->"drinking_water" tag
def query_nodes(bbox):
    # OverpassAPI url
    overpassAPI = 'http://overpass-api.de/api/interpreter'

    postdata = "'
        [out:json][bbox:%s][timeout:120];
        (
        node["amenity" = "drinking_water"]
        );
        out geom;
        out meta;
        >;
        "'
    # Sending HTTP request toward OverpassAPI
    data = requests.post(overpassAPI, postdata % (bbox))
    # Parse response
```

```
    data = json.loads(data.text)
    return data

# This function uploads the data to the PostgreSQL server
def upload_data(data):
    with_no_geom = 0
    sql = 'INSERT INTO drinking_water (id, user, user_id, created_at, version,
    changeset, tags, geom) VALUES (%s, %s, %s, %s, %s, %s,
    ST_SetSRID(ST_MakePoint(%s, %s), 4326));'

    # Define a connection
    conn = psycopg2.connect(host = 'localhost', user = 'postgres', password='postgres',
    dbname = 'osm_data')
    psycopg2.extras.register_hstore(conn)
    # Initialize a cursor
    cursor = conn.cursor()
    # Loop through all OSM nodes
    for node in data['elements']:
        # Call the INSERT INTO sql statement with data from the current node
        cursor.execute(sql,(node['id'], node['user'], node['uid'], node['timestamp'],
node['version'], node['changeset'], node['lon'], node['lat']))
    # Finally commit all changes in the database
    conn.commit()

# Lago Maggiore
bbox = '45.698507, 8.44299,46.198844,8.915405'

# Start with putting all nodes in the variable called drinking_water
drinking_water = query_nodes(bbox)
# Pass this variable containing all OSM nodes to the upload script that will insert
it to the PostgreSQL database
upload_data(drinking_water)


# Alternatively you can visualize your data in QGIS, or you can simply make a query toward the
# database to see if it worked

# -- Select 10 nodes
# SELECT id, user, created_at, tags from drinking_water limit 10
```

Appendix I. Reading Csv Files in a Statistical Environment (Rscript)

```
locations <-read.csv('locations.csv', as.is=T, quote="")
photos <-read.csv('photos.csv', as.is=T, quote="")
# No quoting in this file. Always check source first!
hashtags <-read.csv('hashtags.csv', as.is=T, header=F)
```

Appendix J. Visualizing Instagram Locations (Rscript)

```
library('ggplot2')
library('plyr')
library('wordcloud')
library('ggmap')
# Once the packages are loaded, continue with the creation of a map of locations.

# Set map center to the arithmetic mean of lat-long coordinates
center <- c(lon = mean(locations$lon), lat = mean(locations$lat))
# Init map background to OpenStreetMap tiles at zoom level 15
background <- get_map(location = center, source = 'osm', zoom = 15)

# Create a ggmap object
map <- ggmap(background, extent = 'device')

# Hint: typing map to the console (calling your object) will print out the
background map

# Populate map plot with points using the geom_point() function
map <- map + geom_point(data = locations, aes(x = lon, y = lat), color = 'green',
size = 2)

# Heatmap style visualization of density with countour lines
map <- map + stat_density2d(data=locations, aes(x = lon,y = lat, fill = ..level..,
alpha = ..level..), geom='polygon', size = .3) +
scale_fill_gradient(low = 'yellow', high = 'red', guide = F) +
stat_density2d(data = locations, aes(x = lon, y = lat), bins = 5, color = 'red',
size = .3) +
scale_alpha(range = c(0, .2), guide = F)

# Finally, we can annotate our plot

map <- map + ggtitle('Instagram locations in Helsinki') + geom_text(data =
locations[sample(1:nrow(locations),20),], aes(x = lon, y = lat, label = name),
size = 5, check_overlap=T)

# Type "map" to the console again to see your final map.
```

Appendix K. Simple Data Exploration (Rscript)

```
# Total number of photos
nrow(photos)

# Total number of unique users posting photos in these locations
length(unique(photos$username))

# Summarizing data by location
# Check out also count(), join() and merge() functions!
```

```
locations[‘user_count’] <- NA
locations[‘photo_count’] <- NA
# sapply() summarizes users by applying the length() function for all location_ids
(i.e., what is the
# length of the list of users for a location?)
locations[‘user_count’] <- sapply(locations$id, function(x)
length(unique(photos[photos$location_id==x,]$username)))
# Again, we answer to the question “How many rows do we have after truncating our
photos data
# frame to the specific location?” with sapply()
locations[‘photo_count’] <- sapply(locations$id, function(x)
nrow(photos[photos$location_id==x,]))

# Draw histograms to see how popularity of places is distributed

# Histogram of user counts by location
hist(locations$user_count)
# Histogram of uploaded photos for each location
hist(locations$photo_count)

# Extract top 20 places with most users
plot_data <- locations[order(-locations$user_count),][1:20,]
plot_data <- transform(plot_data[,c(‘name’,’user_count’)],
name = reorder(name, order(user_count, decreasing=T)))

# Create a bar plot of user counts for the 20 most visited locations
ggplot(plot_data, aes(x = name, y = user_count)) + geom_bar(stat = ‘identity’) +
theme_bw() + theme(axis.text.x = element_text(angle = 90),
axis.title.x = element_blank()) + ylab(‘User counts’)
```

Appendix L. Exploring Instagram Photo Upload Intensity over the Days of Week (Rscript)

```
# Extract the Day of Week from the timestamp
days <- format(as.Date(photos$created_at), format = ‘%A’)

# Calculate frequencies using the count() function from the plyr package
freq_table <- count(days)

ggplot(freq_table, aes(x = x, y = freq)) + geom_bar(stat = ‘identity’) +
theme_bw() + xlab(‘Day of week’) + ylab(‘Photos uploaded’)
```

Appendix M. Generating a Wordcloud of Instagram Hashtags (Rscript)

```
# Let’s count the occurances of each hashtag with the count() function
hashtag_freq <- count(hashtags)

# We can also rename the columns
names(hashtag_freq) <- c(“hashtag”,”freq”)
```

```
# Finally, let's generate a wordcloud.
wordcloud(words = hashtag_freq$hashtag, freq = sqrt(hashtag_freq$freq),
min.freq = 1, max.words=200, rot.per = 0.25,colors = brewer.pal(8,"Dark2"))

# Hint: type ?wordcloud in the console if you're using RStudio to see what additional
 parameters you can use to control the appearance of your wordcloud.
```

Appendix N. Fitting a Linear Regression on Instagram "likes" and Number of Tagged Users (Rscript)

```
# Let's drop those photos with 0 tagged users and 0 likes
subset <- photos[photos$tagged_users > 0 & photos$likes > 0,]

# Since we're about to build a linear regression model, let's do a dummy
normality check
hist(subset$likes)

# Log transform like counts since they're not normally distributed
likes <- log10(subset$likes)
tagged_users <- subset$tagged_users

# Build simple linear regression
reg <- lm(likes ~  tagged_users)
plot(tagged_users, likes)
abline(reg, col = 'red')
# Check the summary to see if the relationship is statistically significant
summary(reg)
```

References

1. Krumm, J.; Davies, N.; Narayanaswami, C. User-generated content. *IEEE Pervasive Comput.* **2008**, *4*, 10–11. [CrossRef]
2. Sester, M.; Arsanjani, J.J.; Klammer, R.; Burghardt, D.; Haunert, J.-H. Integrating and Generalising Volunteered Geographic Information. In *Abstracting Geographic Information in a Data Rich World*; Springer: Cham, Switzerland, 2014; pp. 119–155.
3. Hecht, B.; Hong, L.; Suh, B.; Chi, E.H. Tweets from justin bieber's heart: The dynamics of the location field in user profiles. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Vancouver, BC, Canada, 7–12 May 2011; pp. 237–246.
4. Flatow, D.; Naaman, M.; Xie, K.E.; Volkovich, Y.; Kanza, Y. On the accuracy of hyper-local geotagging of social media content. In Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, Shanghai, China, 2–6 February 2015; pp. 127–136.
5. Haklay, M. Citizen Science and Volunteered Geographic Information: Overview and Typology of Participation. In *Crowdsourcing Geographic Knowledge*; Sui, D., Elwood, S., Goodchild, M., Eds.; Springer: Berlin, Germany, 2013; pp. 105–122.
6. Girres, J.F.; Touya, G. Quality assessment of the french openstreetmap dataset. *Trans. GIS* **2010**, *14*, 435–459. [CrossRef]
7. Haklay, M. How good is volunteered geographical information? A comparative study of openstreetmap and ordnance survey datasets. *Environ. Plan. B Plan. Des.* **2010**, *37*, 682–703. [CrossRef]
8. Juhász, L.; Hochmair, H.H. User contribution patterns and completeness evaluation of Mapillary, a crowdsourced street level photo service. *Trans. GIS* **2016**. [CrossRef]

9. Mooney, P.; Corcoran, P.; Winstanley, A. A study of data representation of natural features in openstreetmap. In Proceedings of the 6th GIScience 2010 International Conference, Zurich, Switzerland, 14–17 September 2010; pp. 150–156.
10. Goodchild, M.F. Citizens as voluntary sensors: Spatial data infrastructure in the world of web 2.0 (editorial). *Int. J. Spat. Data Infrastruct. Res.* **2007**, *2*, 24–32.
11. Arsanjani, J.J.; Zipf, A.; Mooney, P.; Helbich, M. An Introduction to Openstreetmap in Geographic Information Science: Experiences, Research, and Applications. In *Openstreetmap in Giscience*; Springer: Cham, Switzerland, 2015; pp. 1–15.
12. Juhász, L.; Hochmair, H.H. Cross-Linkage between Mapillary Street Level Photos and OSM edits. In *Geospatial Data in a Changing World*; Springer: Cham, Switzerland, 2016; pp. 141–156.
13. Antoniou, V.; Morley, J.; Haklay, M. Web 2.0 geotagged photos: Assessing the spatial dimension of the phenomenon. *Geomatica* **2010**, *64*, 99–110.
14. Spinsanti, L.; Ostermann, F. Automated geographic context analysis for volunteered information. *Appl. Geogr.* **2013**, *43*, 36–44. [CrossRef]
15. Schade, S.; Díaz, L.; Ostermann, F.; Spinsanti, L.; Luraschi, G.; Cox, S.; Nuñez, M.; Longueville, B.D. Citizen-based sensing of crisis events: Sensor web enablement for volunteered geographic information. *Appl. Geomat.* **2013**, *5*, 3–18. [CrossRef]
16. Lingad, J.; Karimi, S.; Yin, J. Location extraction from disaster-related microblogs. In Proceedings of the 22nd International Conference on World Wide Web, Rio de Janeiro, Brazil, 13–17 May 2013; pp. 1017–1020.
17. Kinsella, S.; Murdock, V.; O'Hare, N. I'm eating a sandwich in glasgow: Modeling locations with tweets. In Proceedings of the 3rd International Workshop on Search And Mining User-Generated Contents, Glasgow, UK, 24–28 October 2011; pp. 61–68.
18. Gelernter, J.; Mushegian, N. Geo-parsing messages from microtext. *Trans. GIS* **2011**, *15*, 753–773. [CrossRef]
19. Intagorn, S.; Lerman, K. Placing user-generated content on the map with confidence. In Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas, TX, USA, 4–7 November 2014; pp. 413–416.
20. Sample Datasets. Available online: https://github.com/jlevente/link-vgi/tree/master/sample_datasets (accessed on 20 September 2016).
21. Laakso, M.; Sarjakoski, T.; Sarjakoski, L.T. Improving accessibility information in pedestrian maps and databases. *Cartographica* **2011**, *46*, 101–108. [CrossRef]
22. Antoniou, V.; Fonte, C.C.; See, L.; Estima, J.; Arsanjani, J.J.; Lupia, F.; Minghini, M.; Foody, G.; Fritz, S. Investigating the feasibility of geo-tagged photographs as sources of land cover input data. *ISPRS Int. J. Geo-Inf.* **2016**, *5*, 64. [CrossRef]
23. Kahle, D.; Wickham, H. Ggmap: Spatial visualization with ggplot2. *R J.* **2013**, *5*, 144–161.
24. Ayala, G.; Epifanio, I.; Simó, A.; Zapater, V. Clustering of spatial point patterns. *Comput. Stat. Data Anal.* **2006**, *50*, 1016–1032. [CrossRef]
25. Goodchild, M.F.; Glennon, J.A. Crowdsourcing geographic information for disaster response: A research frontier. *Int. J. Digit. Earth* **2010**, *3*, 231–241. [CrossRef]
26. Flanagin, A.J.; Metzger, M.J. The credibility of volunteered geographic information. *GeoJournal* **2008**, *72*, 137–148. [CrossRef]
27. Scassa, T. Legal issues with volunteered geographic information. *Can. Geogr.* **2013**, *57*, 1–10. [CrossRef]