SoftwareX 15 (2021) 100777

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

INACITY - INvestigate and Analyze a CITY

Artur André Almeida de Macedo Oliveira *, Roberto Hirata Jr.

Computer Science Department, Instituto de Matemática e Estatística - Universidade de São Paulo, Rua do Matão 1010, São Paulo, SP 05508-090, Brazil

ARTICLE INFO

Article history:
Received 18 April 2021
Received in revised form 19 July 2021
Accepted 20 July 2021

MSC:
68-04
68U01
68U105
68U3
97R50

Keywords:
Geographical information system
Geoportal
Computer vision

ABSTRACT

INACITY is a platform that integrates Geo-located Imagery Databases (GIDs), Geographical Information Systems (GIS), digital maps, and Computer Vision (CV) to collect and analyze urban street-level images. The platform's software architecture is a client–server model, where the client-side is a simple Web page that allows the user to select regions of a map and select filters to analyze and visualize urban features. The server side is a Django-powered Web service with PostgreSQL and Neo4j databases. Users can select a region of a map, an image filter, and geographical features to analyze relevant urban characteristics as trees, for instance, using the platform. An open-source implementation of the platform is available. The architecture is extensible, and it is easy to add new modules or replace the existing ones with new digital maps, GIS databases, other CV filters, or other GIDs.

Code metadata

Current code version	v1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00075
Code Ocean compute capsule	N/A
Legal Code License	Mozilla Public License 2.0
Code versioning system used	git
Software code languages, tools, and services used	Python, Django, Javascript, jquery, openlayers, bootstrap, Docker, PostgreSQL, neo4j
Compilation requirements, operating environments & dependencies	Docker, docker-compose
If available Link to developer documentation/manual	http://inacity.org/docs
Support email for questions	arturao@ime.usp.br

Software metadata

Current software version	v1.0
Permanent link to executables of this version	https://github.com/arturandre/inacity
Legal Software License	Mozilla Public License 2.0
Computing platforms/Operating Systems	Linux, OS X, Microsoft Windows, Docker
Installation requirements & dependencies	Docker, docker-compose
If available, link to user manual – if formally published include a reference to the publication in the reference list	http://inacity.org/tutorial
Support email for questions	arturao@ime.usp.br

1. Introduction

Open-Government data, or crowd-sourced data, or even enterprise companies data, is usually available on the Internet, and the correct and proper use of it may be significant for both governments and citizens [1]. Several social investigations and

* Corresponding author.
E-mail address: arturao@ime.usp.br (Artur André Almeida de Macedo Oliveira).

https://doi.org/10.1016/j.softx.2021.100777

projects are done by inspecting places in person and collecting data through surveys or photographs. Some of these investigations had been conducted in order to establish a relationship between neighborhood audition data and health related subjects like adolescents mental health [2], overall and lower-extremity functional loss [3,4], poverty and mortality [5], low birth weight, asthma and respiratory diseases [6,7], physical activity [8], self-rated health [9], depression and stress [10,11]. Some studies targeted more specifically the relationship between neighborhood physical characteristics (e.g. greenery, graffiti, visual urban decay), psychosocial characteristics and obesity [12–16]. In some studies the causes and consequences of physical and social disorder at the neighborhood level are analyzed [17–19].

Collecting data can be an expensive and time-demanding process [20]. To the best of our knowledge, there is no open source platform to gather, combine, analyze, and visualize multimodal data. When image data at the street level is needed, Geo-located Imagery Databases (GIDs), that is, image retrieval systems, can be used to collect the urban images [20,21]. Looking for some feature in thousands of images can be unfeasible, and Computer Vision (CV) systems can be employed to mitigate this problem.

INACITY is an open-source platform that integrates GIDs, Geographical Information Systems (GIS) databases, digital maps, and CV techniques to collect and analyze urban street-level images. The software architecture of the platform is a client–server model, where the client-side is a simple Web page that allows the user to select regions of a map and also select filters to analyze and visualize urban features. The server side is a Django powered Web service with PostgreSQL and Neo4j databases. The platform is not a replacement for a Geographical Information System, such as QGIS [22] for example. It can be complementary to a GIS to define geographical locations of interest. The workflow would start using an Imagery database system to collect images from defined locations, extracted features from those images, and stored the data back into a GIS, as will be described later when we explain how we use the store geographical data into the graph-oriented Database Management System Neo4j [23].

From the perspective of an end-user, the platform allows him/her to select a region of interest in a map, select a feature (e.g. greenery, bus stops, traffic signs, etc.) and then analyze the presence, or even the distribution of that feature based on images collected, inside the selected region, on GIDs. Besides, the end-user can also use INACITY's front-end to visualize geographical entities from a GIS such as bus stops, schools, overpasses, and others, over the same region of interest.

From a developer's perspective, INACITY offers three kinds of extensions: new GIS, GIDs, and CV techniques. The architecture is extensible, and it is easy to add new modules or replace the existing ones with new digital maps, GIS databases, CV filters, or GIDs.

2. Methodology and software description

This section presents the software architecture, components, classes, current functionalities, and the way to extend it to add new features. The components responsible for collecting data in INACITY's back-end keep a common interface, so that integrating a new data collecting component is just a matter of implementing that interface. Besides, custom objects allow heterogeneous data to be combined and displayed at INACITY's front-end.

2.1. Software architecture

The INACITY's client–server model [24] is twofold: a *back-end* (data access layer) and a *front-end* (presentation layer) component. This model was chosen in order to maximize the accessibility of the platform, that is, instead of having a client application for each major OS (e.g., Linux, macOS, and Windows) by making the platform available as a web platform, any http-based client front-end can consume it. To make the platform even more accessible, the main deploy tool used is Docker system [25].

The programming language adopted to develop the back-end was Python 3. The reason to choose Python rather than a more business-driven language like C# or even Java is that those languages usually come coupled with an environment of their own, like the Java Virtual Machine or proprietary libraries from Microsoft. Python tends to be more friendly for newcomers and has a rich set of libraries and packages publicly available. Besides, there is a considerably large body of scientific work produced with Python (e.g., the Scikit libraries family [26]) and web-development frameworks (e.g., Django and Flask), which in turn are tuned to deal easily with database modeling even with geographical data.

The choice for the Django framework [27] as the back-end core was because Django provides all the machinery for handling web-based requests (i.e. HTTP or, more specifically REST based requests), database access and modeling, user authentication and authorization, real-time communication (with the extension Django-channels) and a stable and large community. Having all of these pieces managed and put together by the Django framework leaves only the main concepts (data integration) to be dealt with in the INACITY platform. In the back-end, the main body of work, besides setting up the Django machinery, is modeling the classes responsible for collecting and integrating data.

The front-end development also followed the Django guidelines. Using Django templates, we developed a coupled front-end provided by the same back-end rather than having a framework to deal with the server stuff (e.g., processing and database access) and front-end serving (e.g., NodeJS). The front-end provided by the INACITY platform is a visualization tool that consumes data from the back-end, also allowing user account creation, login, and the management of users' work sessions. The front-end was developed using Django template language (based on HTML) and Javascript.

The back-end holds a manager system[1] responsible for keeping track of classes that collect data from GIS, GIDs, and classes that implement CV techniques for extracting and processing data from images. The flow of a request to the back-end is as following:

1. A request is made to some end-point in the back-end.
2. Django infrastructure delegates this request to the Manager Component (MC).
3. The MC delegates it to the class responsible for collecting/generate data for the request.
4. The data collecting/generating class returns data to the MC.
5. The MC formats the data received (if needed) and returns it to the Django infrastructure that, in turn, returns it to the caller that originated the request.

It is worth noting that the communication with the back-end is performed through a REST API. Any client application can request to the back-end, not only the front-end developed in the INACITY platform. A diagram describing the back-end components along with some comments about their functions is available at the project's Github entry [30]. The diagram also illustrates the relationship between managers, abstract classes and

[1] A class that follows the design pattern known as Strategy [28], that is, when a class is a manager (usually called Manager Components) it delegates a request to an associated class and the response depends on the associated class. These components have nothing to do with Django's Manager class, which is an interface to allow Django models to query a database [29].

derived counterparts. Each manager is responsible for delegating requests from some front-end client to the components responsible for collecting urban imagery, GIS data, and data extracted by some Image Filter. The `MapMinerManager`, `ImageProviderManager`, and `ImageFilterManager` are abstract classes that define a common interface to enable the manager classes to delegate requests in the same fashion to different external systems. The **URLs** component define end-points that external clients can call; those end-points define the REST API functions of the back-end.

A class diagram describing the Manager classes, arranged according to the Strategy design pattern, is available at the project's Github entry [31]. The diagram shows the abstract base classes allowing the extension of the INACITY data sources and processors. For example, the class `OSMMiner` is a subclass derived from `MapMinerManager`, and it provides a unified way to collect data from the `OpenStreetMap` GIS [32]. The `OSMMiner` class implements functions whose signatures are defined in its base class. By calling those functions, the `MapMinerManager` class can collect data from the `OpenStreetMap` GIS seamlessly, without the need to be adjusted to the data model or even the connection details from the `OpenStreetMap` GIS. This design is possible because the `OSMMiner` class translates requests from the `MapMinerManager` to queries for the `OpenStreetMap`, and also translates the response from the latter to a common format (i.e., GeoJSON [33]) that can be transmitted back to the front-end client.

The front-end part comprises a website for the end-user. A diagram describing the front-end diagram is available at the project's Github entry [34]. The main components are its pages and communication classes. The pages essentially enable the end-user to interact with the website and render updated information as the user makes requests. The communication components are responsible for encapsulating requests for the back-end and, at the current version, for Google Street View (GSV) servers. It is noteworthy that the GSVService component, responsible for communication with GSV servers, is implemented at the front-end due to restrictions on GSV. However, the signing key and the GSV request formulation algorithms are implemented at the back-end. The Google Street View [35] was chosen as the standard GID due to its worldwide coverage and because it is a platform that is commonly used in scientific papers targeted at analyzing the urban environment through street-level urban imagery.

2.2. Software functionalities

This section presents the components of the system that allow one to add functionalities to the system. By combining geo-located features from GIS and geo-located images from an image provider, one can enrich the system's functionalities. One example of such functionality consists of observing specific kinds of trees cataloged in a city greenery database like the Pasadena Urban Trees dataset [36], or the road afforestation ("Arborização viária") layer from the GeoSampa dataset [37] of the city of São Paulo, Brazil.

The INACITY platform concerns integrating imagery data and information extracted from it with a GIS. Such integration allows a comprehensive range of applications, the most direct ones assessing the quality or presence of urban features. This integration allows one to assess visually and automatically the quality of a segment of road, spotting precisely cracks, potholes, paint damage, and other signs of degradation. Concerning the detection of urban features, one possibility would be to implement deep learning neural networks to detect traffic signs [38] in images collected from a crowdsourced imagery platform such as KartaView (previously known as OpenStreetCam) [39]. By creating subclasses, from the abstract base classes `ImageProvider` and `ImageFilter`, one can integrate KartaView (to collect the images) to INACITY to process the collected images and detect the traffic signs or other urban elements. From a developer's perspective, INACITY offers three kinds of extensions: new GIS, GIDs, and CV techniques.

2.3. Extending the platform

The platform provides the means for a user to use his/her dataset or new image filter algorithms, so new components can be easily integrated with previously implemented components thanks to its design. Integrating a new component to extend the platform requires the user to implement the new component(s) directly in Python's source code. There are three main ways to extend the platform, new geographical databases, new imagery platforms, and new Computer Vision algorithms.

Each possible extension is made by implementing a subclass of a specific base class, as detailed in the following sections, that defines an interface for a corresponding Manager Component.

2.3.1. GeoImage

To facilitate the integration between imagery data and geographical data, one can use the GeoImage object. A class diagram describing the GeoImage component based on GeoJSON is available at the project's Github entry [40].

The implementation of this object is similar to the GeoJSON object to achieve better interoperability. As specified by RFC 7946 [33], the GeoJSON object has its fields well defined with a proper semantic, except for the `properties` field of the `Feature` object. This field can contain any JSON (JavaScript Object Notation) object. Therefore, we keep the imagery data related to the coordinates of a `Feature` object inside the `properties` field under the key **geoimages**. Every GeoJSON object is either a `FeatureCollection`, a `Feature` or one of seven kinds of geometries [33]. We consider each geographical entity as a `FeatureCollection`, usually containing only a single `Feature`.

Every geographical entity is treated as a `FeatureCollection`, possibly containing just a single `Feature`. Each request to `ImageProvider` will contain a `FeatureCollection` with an array of Features. The coordinates of the latter will define the coordinates of the images to be retrieved by the `ImageProvider`. Fig. 1 shows a diagram of the `GeoJSON` as an abstract class with nine possible subclasses (considering that the `Geometry` class could be one of six distinct types).

The GeoImage object keeps metadata about some image collected from an image provider and data extracted from that particular image using some Computer Vision algorithm. The extracted data will be kept in a separate object called `ProcessedImageData`. Notice that the same GeoImage can hold a reference to multiple `ProcessedImageData`, because each image can be processed by different Computer Vision algorithms, yielding multiple distinct extracted data.

We add a new entry with the key `geoimages` into the JSON field `properties` of the `Feature` object to keep the same indexes between the coordinates of the geometry property. That is an easy way to access the `GeoImage` related to a particular coordinate. The `geoimages` entry may have the same structure (i.e., nesting indexes) of the coordinates in the geometry property of the `Feature`. When an image is not available for a particular coordinate, an error string will fulfill that particular index position in the `geoimages` entry.

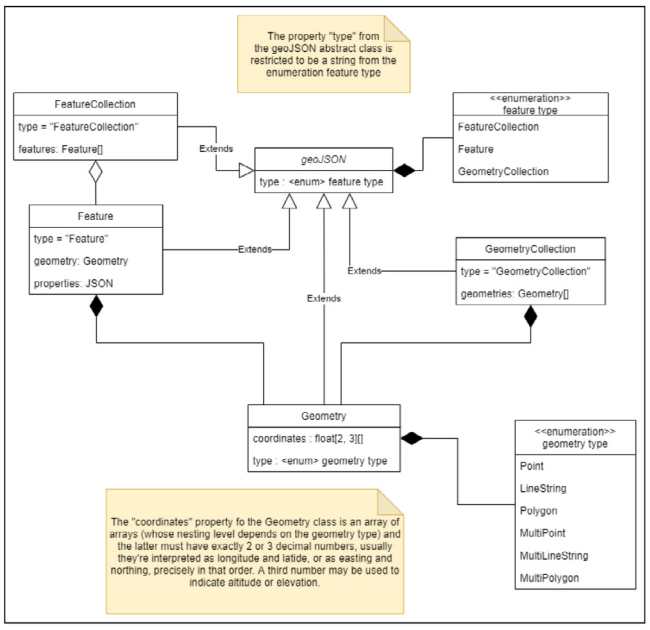

Fig. 1. GeoJSON class diagram.

2.3.2. Image filter module

Fig. 2 illustrates how the platform can be extended. The subclasses from the `Image Filter` components are meant to process images. The components without any highlight are already implemented while the ones highlighted in yellow represent new components in development, and the `Trees & Powerlines` is under integration in the platform. It represents a detector of overhead power-lines that intersect trees' canopies and branches.

The `Scene Captioning` class could, for example, be such that given an image, it provides a textual description of the scene. A geo-located image's description can be stored in the geographical feature related to the image or even in a completely decoupled GIS.

The `Pavement quality` class is another example of an object that could provide a degree to indicate to what extent the pavement of a road is damaged. If the input image's point of view is such that the camera points downwards, the main object of interest will be the pavement and its eventual defects (e.g., cracks and potholes). The component could provide a mask image obtained by segmenting the defects that would highlight the pavement's most relevant damages.

The interface design and the guidelines provided by the abstract class `ImageFilter` can be used to add a new component that derives from the `ImageFilter`. The newly implemented component can then be readily used by the `ImageFilterManager` and integrated into the INACITY platform.

2.3.3. Image Provider module

Similarly, the same extensibility and modularity concepts can be extended to the `Image Provider` and the `Map Miner` classes.

Fig. 3 presents the same concept as presented for image filters but applied to image providers. The main difference is that instead of processing an input image, the component is responsible for collecting an image associated with a given coordinate. For example, the GSV API allows one to query for the closest panorama with relation to some given coordinate. It also allows selecting the vertical and horizontal angles, the field of view, image resolution, and other parameters regarding an image from some specific panorama.

The component `GoogleStreetView` in Fig. 3 receives as input a set of coordinates and converts them into appropriate requests for the GSV platform, treat possible exceptions and finally return a formatted response (typically a `GeoImage` instance) back to the `ImageProviderManager` class.

To extend the *Image Provide* module a subclass derived from the `Image Provider` abstract class must be created, as exemplified in Fig. 3 by the highlighted (yellow boxes) subclasses `Mapilary`[41], `Baidu Total View`[42] and `Crowdsource`. Such subclasses encapsulate all the code responsible for communicating with the target Image Provider system. Given a set of coordinates, the subclasses must formulate a query to the target system, and then merge the response from that system with the input coordinates, composing a `GeoImage` response that is returned to the `ImageProviderManager` class. In the case of the `Crowdsource` subclass, its target system is the INACITY platform back-end itself, and this component provides the means for the users to provide urban images themselves, which in turn can be retrieved later by the `Image Provider Manager` component under another user request.

2.3.4. Map Miner module

The *Map Miner* module is responsible for integrating GIS databases to INACITY. It has some particularities that are worth mentioning. Fig. 4 presents the *Map Miner* module and its connections with some GIS databases. As in the other module figures, the blue components correspond to those implemented, and the yellow ones are example components to be implemented in future versions of the INACITY platform. The GeoSampa component defines the means to collect data, from the internal PostgreSQL database, regarding bus stops from the city of São Paulo. An extract from the GeoSampa [37] database was introduced directly into the INACITY database to mitigate the number of external queries. The database also keeps users' access and working session data.

The `OSMMiner` class implements the means to collecting streets' information. The streets are represented as a collection of interconnected `LineString` objects (according to GeoJSON specification [33]), from the OpenStreetMap [32] platform.

The sequence of collected `LineString` objects (each holding its geographical coordinates) can be the input to get geolocalized images from some Image Provider system. Besides that, the relationships (e.g. direction) between each pair of objects can be used to determine camera angles between two adjacent *panoramas*.

The `PanoramaMiner` class performs queries over a Neo4j database instance, a graph-oriented Database Management System (DBMS) [23]. The main advantage of using a graph-oriented database is to model the streets, the regions, and other geographical objects. The Neo4j instance is hosted together with the Django server into the same Docker container in the current version. This database is responsible for relating data from physical entities (e.g., objects from some GIS), their images (sampled from some Image Provider system), and even data extracted from those images (using some Image Filter component).

Fig. 5 shows an example of some GSV's panoramas, images metadata from each panorama, and data extracted from the images, as seen by Neo4j Browser User Interface. An orange circle represents each panorama. It holds information like address, pitch, heading of the camera, and the shot's time. Each panorama may span different images, each with a pitch and heading. Therefore, each panorama may be related to multiple images, so the metadata regarding these images are stored in the blue circles called **Views**. The data resulting from processing the images is stored in vertices called **FilterResult** represented by the gray circles in Fig. 5 (related to the **View** that correspond to the processed image). This data representation in a graph-oriented DBMS allows faster retrieval of results already collected and processed, reducing the time a user needs to wait for his/her request to be completed, keeping the system flexible and extensible.

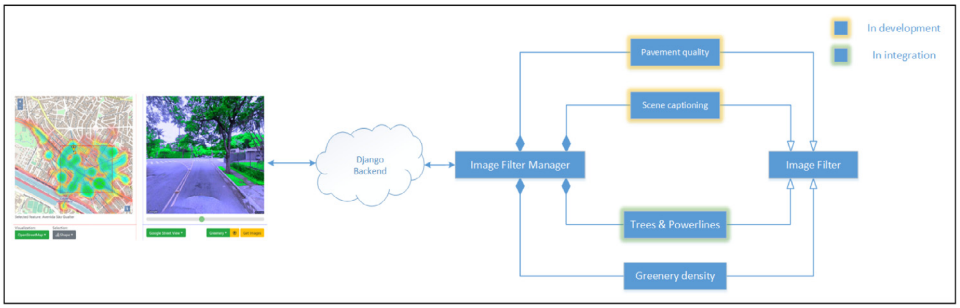

Fig. 2. *Image Filtering* module. All the subclasses of the *Image Filter* class have a common interface that is used by the *ImageFilterManager* class during a request. The yellow shadowed subclasses are being developed in the current version of the INACITY platform, the green shadowed one is being integrated in the platform and the others are already available. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

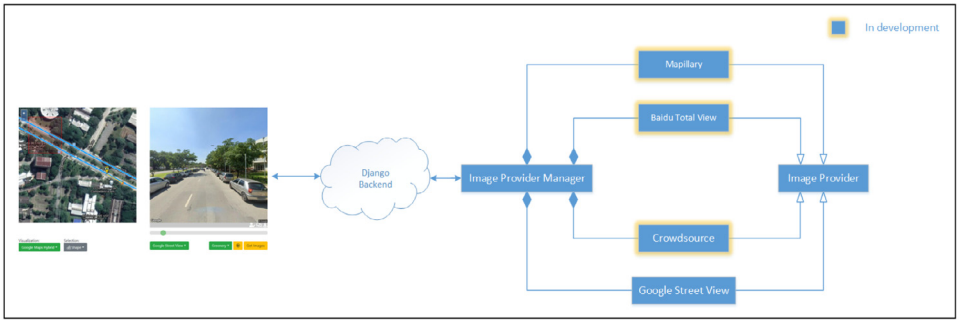

Fig. 3. *Image Provider* module. Each user request is received by the Manager component (e.g. *ImageProviderManager*) and routed to the *ImageProvider* subclass specified in the request. Each of these subclasses provide to the Manager component a common interface to collect images from a given external image provider (e.g. GSV), thus abstracting the particularities of the API and rules (e.g. minimum time between requests) of each external image provider.

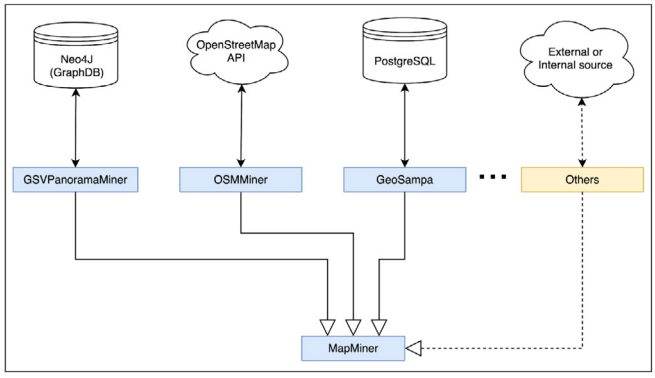

Fig. 4. *Map Miner* module. The Neo4j and the PostgreSQL databases (cylinder icons) are locally deployed, while the components depicted with a cloud icon denote external systems accessed through the subclasses of the *Map Miner* class. As with the other modules, user requests are routed by a Manager component (omitted in this figure; see Figs. 2 and 3 for examples) to the *Map Miner* subclass specified in the request. This subclass must translate the user request into a request to the target GIS (e.g., OpenStreetMap) or local database (e.g., Neo4j), which contains geographical data (e.g., the location of GSV panoramas or even bus stops imported from GeoSampa into the local PostgresSQL database). Notice that the Django ORM is used to interact with the PostgresSQL database, while the interaction with the Neo4j is done through the packages for python provided by the Neo4j, Inc.

3. Results and illustrative examples

The platform's current implementation includes the OpenStreetMap GIS as a source of coordinates to be sampled from the GSV Image Provider. This combination allows multiple uses of the platform in a practical way. In this section, we present two use case examples of the platform.

3.1. Neighborhood visual inspection

The platform's most simple use case involves selecting a region of interest and fetching images from that region. In the current implementation, the OpenStreetMap and the GSV platforms are the coordinates' source and the Image Provider. Once collected, the streets define a route for the user to view pictures from streets in the selected region as if he/she was there. This use case is handy for auditing neighborhoods [21]. A short video showing the use case accompanies this article [43].

The pipeline begins when a person using INACITY's front-end selects a region and presses the "Get Images" button. We assume that the user keeps the default options of GIS (**MapMiner**). This action triggers a request from the front-end class `UIModel` to INACITY's back-end. The request consists of the selected region's geographical entities collected and sent to the Image Provider. A diagram of the process that follows the `UIModel` request is available at the project's Github entry [44]. Since this is a request for a geographical entity, the request is received by the `MapMinerManager` component, which in turn delegates it to the appropriate `MapMiner` subclass. In this example, the `OSMMiner` is the referred subclass since it is responsible for treating requests to the OpenStreetMap platform. The `OSMMiner` subclass will formulate a query written using the Overpass Query Language [45] and will send it to the OpenStreetMap platform.

After the OpenStreetMap returns the query results, the `OSMMiner` subclass will format the response using the GeoJSON [33] specification and return it to the MapMinerManager, which in turn will return it to the front-end. INACITY's front-end will then display the returned geographical entities, streets in this case. To better visualize the results, the system represents streets as blue lines in the digital map.

Following this step, a second request is issued by the `UIModel` class. This request consists of the geographical entities collected in the first step and an Image Provider.

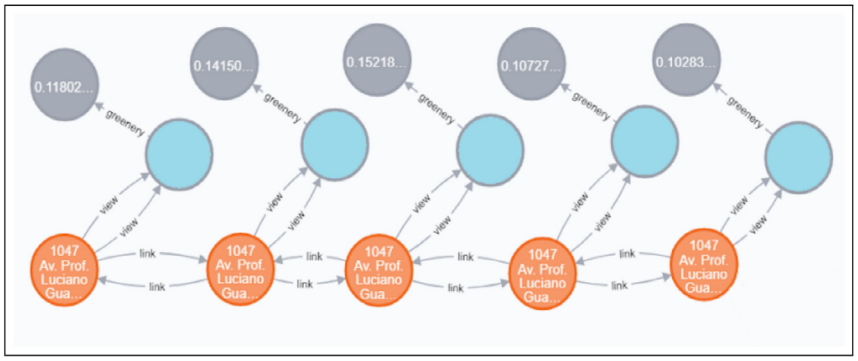

Fig. 5. Visualization of some nodes stored in the local Neo4j instance. Panorama nodes in orange represent locations (specified in latitude and longitude coordinates) where images were taken. Views (in blue) represents an image and encapsulates the horizontal (heading) and vertical (pitch) angles of the camera (with relation to the true North and a flat ground, respectively) at the moment of the shot. Filter results (nodes in gray) maintain data extracted from a view by some *ImageFilter* subclass (Fig. 2).

Following this step, the `UIModel` class issues a second request consisting of the geographical entities collected in the first step and an Image Provider. This request follows a similar flow as the one used to collect street geographical data, except that this time the `ImageProviderManager` receives the request. Assuming that the default option for the Image Provider is the GSV, the `ImageProviderManager` will delegate the request to the `GoogleStreetViewImageProvider` subclass of the `ImageProvider` component.

3.2. Urban feature visualization (greenery)

Visualizing the distribution of urban features, like greenery, is another use case of the INACITY platform in a given geographical region.

The pipeline starts as before (a user selecting a region of interest) and then triggers a request for processing the images collected during a neighborhood audition. In the video that accompanies this paper, the user applies a filter called `Greenery filter` to estimate the `Green View Index` by segmenting which parts of the image correspond to green vegetation. The proportion of the image (in relation to the image size) regarded as green vegetation will be stored in the **density** property of a `ProcessedImageData` object associated with the `GeoImage` of the processed image. When the `FeatureCollection` is returned back to the front-end each of its features and corresponding `GeoImages` (if available) will have an associated `ProcessedImageData` which in turn may have extracted data as the density of Green View Index which will be displayed as Fig. 6, for example.

Besides the heat-map visualization, INACITY can present some urban features overlayed to the original images. Fig. 7(b) presents an example of an image in which the greenery parts (as classified by the back-end) are highlighted in green and the non-greenery ones in blue overlayed to the image. This kind of visualization allows a fine-grained inspection over each image rather than over the analyzed region. The greenery image filter module uses the Python packages numpy [46] and scikit-image [47].

4. Demo site, impact and limitations

INACITY is available at http://inacity.org to anyone who wishes to try the implemented analysis without deploying it. A user-level quota system is necessary to allow more users to try the platform through the demo site (due to GSV costs). Despite that, users can supply the platform with their own GSV credentials (i.e., there is no general quota use).

A non-specialist user can use the public instance to select a region, query images from it, and extract features from those

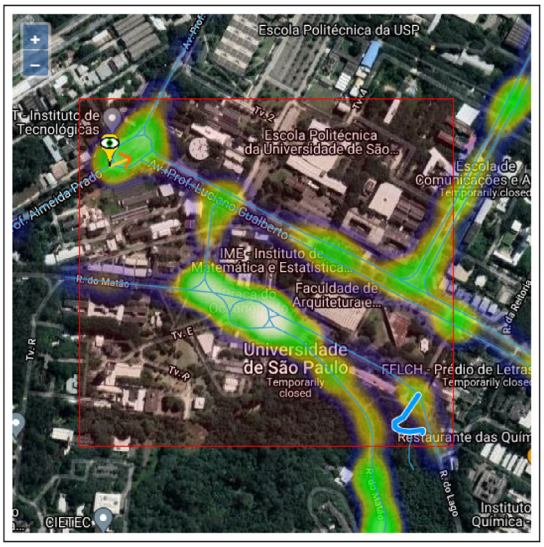

Fig. 6. Greenery heat-map (lighter colors for higher values) for a region selected (i.e. the red square). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

images. Nevertheless, this approach limits the user to the already available capabilities implemented in the public instance, those are, street network locations from OpenStreetMap [32], Bus Stops from GeoSampa [37], images from Google Street View [35] and currently the only image processing algorithm implemented in INACITY's public instance is the greenery one. Additional capabilities can be implemented by developers and researchers in the future, possibly in locally deployed instances of the INACITY platform.

INACITY can be part of a more extensive pipeline of research. For instance, in [48], we collect images from some locations in the cities of Porto Alegre (BR) and São Paulo to build a machine learning model to detect entanglements between electric wires and tree branches. If the model is successful, it can be coupled into INACITY by subclassing the ImageFilter class, thus enabling the platform to help city managers to detect tree and wire entanglements and prevent accidents.

4.1. Quota module

The quota module keeps track of how many calls are performed by a registered user. An anonymous user, identified by a Django session-id, can also use the system.

(a) Unfiltered image (b) Filtered (greenery) image

Fig. 7. (a) An example image as provided by Google Street View. (b) Image shown in (a) after being filtered by the "Greenery density" subclass. In the filtered image the detected greenery regions are highlighted in green while the rest of the image is blue tinted. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The main components of the quota subsystem is shown at the project's Github entry [49]. The class `QuotaManager` is responsible for registering a new entry in the database and keeping track of each registered user's available quota. The decorator factory **quota_request_decorator_factory** is used as a decorator over the functions whose usage is tracked. The parameters of the decorator factory are **default_user_quota**, **default_anonymous_quota** and **skip_condition**. These parameters specify how many calls will be available for a registered, and an anonymous user, respectively. The last parameter is a Boolean function called to test if the quota manager should be used (e.g., the user is using his/her own GSV credentials).

4.2. Performance tests

We created a benchmark test to assess the effect of multiple simultaneous requests to the back-end. The tests consist of collecting streets and images for two disjoint urban regions. Each region has a different size and consequently a different number of streets and images. Table 1 presents the results and information details of both areas used in the benchmark. The features are the size (in squared meters) of each region, the number of streets, the number of images collected. The minimum, maximum, average, and standard deviation times are split between collecting streets in each region and collecting the images for the corresponding streets collected. Finally, the time spent processing all the collected images using the currently implemented `Greenery` image filter. To collect the response times, 100 requests were performed, 50 for **area 1** and 50 for **area 2**. At any given time, ten simultaneous requests have been made. The requests for **area 1** were intertwined with requests for **area 2**, that is, the first request performed was related to **area 1**, the second to **area 2**, the third to **area 1** again, and so on. The server hosting the platform during the benchmark was an Intel Xeon E5420 2.5 GHz with eight cores.

Table 1
Multiple statistics taken upon the execution times and requests sizes (in terms of streets and images collected) for two distinct regions.

	Both areas	Area 1	Area 2
Area	348906 m^2	20931 m^2	327975 m^2
Num. Streets	27	2	25
Num. Images	429	71	358
Min Time (Streets)	7.78 s	7.78 s	7.89 s
Avg. Time (Streets)	9.14 s	9.36 s	8.91 s
Std. Time (Streets)	2.2 s	2.34 s	2.05 s
Max Time (Streets)	16.76 s	16.68 s	16.76 s
Min Time (Images)	8.57 s	8.57 s	108.36 s
Avg. Time (Images)	84.55 s	18.79 s	152.04 s
Std. Time (Images)	68.79 s	9.52 s	19.81 s
Max Time (Images)	194.92 s	50.13 s	194.92 s
Min Time (Greenery filter)	153.70 s	153.70 s	812.96 s
Avg. Time (Greenery filter)	494.61 s	170.53 s	843.63 s
Std. Time (Greenery filter)	342.97 s	10.97 s	15.32 s
Max Time (Greenery filter)	867.48 s	187.24 s	867.47 s

5. Discussion and conclusions

In the context of smart cities and the Internet of the future, using government public data or even private data available on the Internet is essential to assess features in a city. We created the INACITY platform with three concepts in mind: geolocated images, geolocated data, and algorithms to extract information from images. These concepts directed the platform modules' design and architecture such that implementing a new image provider, GIS, or a new CV/image processing algorithm can be done without impacting any of the other modules. In other words, by following each module base class's specifications, new components can be seamlessly integrated.

The front-end is simple and allows end-users (e.g., citizens, developers, researchers, or government administration agents) to use the platform to gather data for future use and, because it is open-source, further improve the platform by modifying it to their needs.

The use cases presented are useful but straightforward, and we plan to extend them to other city problems as studying the problem of intersection of power lines with trees in the future.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq 465446/2014-0, FAPESP 14/50937-1 and 15/24485-9. The authors acknowledge São Paulo Research Foundation (FAPESP) 18/10767-0 and 15/22308-2. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

[1] OpenStreetMap contributors. Providing your data to openstreetmap. 2021, https://blog.openstreetmap.org/wp-content/uploads/2020/07/Providing-data-to-OpenStreetMap.pdf. Retrieved: 18/06/2021.
[2] Aneshensel CS, Sucoff CA. The neighborhood context of adolescent mental health. J Health Soc Behav 1996;293–310.
[3] Balfour JL, Kaplan GA. Neighborhood environment and loss of physical function in older adults: evidence from the Alameda County Study. Am J Epidemiol 2002;155(6):507–15.
[4] Schootman M, Andresen EM, Wolinsky FD, Malmstrom TK, Miller JP, Miller DK. Neighborhood conditions and risk of incident lower-body functional limitations among middle-aged African Americans. Am J Epidemiol 2006;163(5):450–8.
[5] Cohen DA, Farley TA, Mason K. Why is poverty unhealthy? Social and physical mediators. Soc Sci Med 2003;57(9):1631–41.
[6] Cagney KA, Browning CR. Exploring neighborhood-level variation in asthma and other respiratory diseases. J Gen Intern Med 2004;19(3):229–36.
[7] Nepomnyaschy L, Reichman NE. Low birthweight and asthma among young urban children. Am J Public Health 2006;96(9):1604–10.
[8] Hoehner CM, Ramirez LKB, Elliott MB, Handy SL, Brownson RC. Perceived and objective environmental measures and physical activity among urban adults. Am J Prev Med 2005;28(2):105–16.
[9] Agyemang C, van Hooijdonk C, Wendel-Vos W, Lindeman E, Stronks K, Droomers M. The association of neighbourhood psychosocial stressors and self-rated health in Amsterdam, The Netherlands. J Epidemiol Commun Health 2007;61(12):1042–9.
[10] Latkin CA, Curry AD. Stressful neighborhoods and depression: a prospective study of the impact of neighborhood disorder. J Health Soc Behav 2003;34–44.
[11] Kim D. Blues from the neighborhood? Neighborhood characteristics and depression. Epidemiol Rev 2008;30(1):101–17.
[12] Ellaway A, Macintyre S, Bonnefoy X. Graffiti, greenery, and obesity in adults: secondary analysis of European cross sectional survey. BMJ 2005;331(7517):611–2.
[13] Glass TA, Rasmussen MD, Schwartz BS. Neighborhoods and obesity in older adults: the Baltimore Memory Study. Am J Prev Med 2006;31(6):455–63.
[14] Boehmer T, Hoehner C, Deshpande A, Ramirez LB, Brownson RC. Perceived and observed neighborhood indicators of obesity among urban adults. Int J Obes 2007;31(6):968.
[15] Stafford M, Cummins S, Ellaway A, Sacker A, Wiggins RD, Macintyre S. Pathways to obesity: identifying local, modifiable determinants of physical activity and diet. Soc Sci Med 2007;65(9):1882–97.
[16] Grafova IB. Overweight children: assessing the contribution of the built environment. Prevent Med 2008;47(3):304–8.
[17] Skogan WG. Disorder and decline: Crime and the spiral of decay in American neighborhoods. Univ of California Press; 1992.
[18] Sampson RJ, Raudenbush SW. Systematic social observation of public spaces: A new look at disorder in urban neighborhoods. Am J Sociol 1999;105(3):603–51.
[19] Skogan W. Disorder and decline: The state of research. J Res Crime Delinquency 2015;52(4):464–85.
[20] Wilson JS, Kelly CM, Schootman M, Baker EA, Banerjee A, Clennin M, et al. Assessing the built environment using omnidirectional imagery. Am J Prev Med 2012;42(2):193–9.
[21] Rundle AG, Bader MD, Richards CA, Neckerman KM, Teitler JO. Using Google Street View to audit neighborhood environments. Am J Prev Med 2011;40(1):94–100.
[22] QGIS Development Team. QGIS geographic information system. Open Source Geospatial Foundation; 2009, http://qgis.osgeo.org.
[23] ©2019 Neo4j, Inc.. Neo4j graph platform. 2019, https://neo4j.com/. Retrieved: 2019-12-04.
[24] Reese G. Database programming with JDBC and JAVA. " O'Reilly Media, Inc."; 2000.
[25] ©2019 Docker Inc.. Docker overview. 2019, https://docs.docker.com/engine/docker-overview/. Retrieved: 2020-02-28.
[26] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine learning in Python. J Mach Learn Res 2011;12:2825–30.
[27] Django Software Foundation. About the django software foundation. 2021, https://www.djangoproject.com/foundation/. Retrieved: 16/06/2021.
[28] Vlissides J, Helm R, Johnson R, Gamma E. Design patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley 1995;49(120):11.
[29] Django Software Foundation. Managers - django documentation - django. 2021, https://docs.djangoproject.com/en/2.2/topics/db/managers/. Retrieved: 16/06/2021.
[30] Oliveira AAAM. INACITY's Back-end diagram. 2021, https://github.com/arturandre/INACITY/raw/master/django_website/docs/diagrams/backendDiagram.png. Last access 28/06/2021.
[31] Oliveira AAAM. INACITY's Back-end class diagram. 2021, https://github.com/arturandre/INACITY/raw/master/django_website/docs/diagrams/inacity_components.png. Last access 28/06/2021.
[32] OpenStreetMap contributors. 2017, Planet dump retrieved from https://planet.osm.org, https://www.openstreetmap.org.
[33] Butler H, Daly M, Doyle A, Gillies S, Schaub T, Schaub T. The GeoJSON format. RFC 7946, RFC Editor; 2016, http://dx.doi.org/10.17487/RFC7946, https://rfc-editor.org/rfc/rfc7946.txt.
[34] Oliveira AAAM. INACITY's Front-end diagram. 2021, https://github.com/arturandre/INACITY/raw/master/django_website/docs/diagrams/frontendDiagram.png. Last access 28/06/2021.
[35] Google Inc. Google street view. 2017, https://www.google.com/maps/streetview/. Last access 05/11/2017.
[36] ©RegisTree 2016. Pasadena urban trees. 2016, http://www.vision.caltech.edu/registree/publications-and-dataset.html. Retrieved: 2019-12-04.
[37] Prefeitura de São Paulo. Mapa digital da cidade de São Paulo. 2020, http://geosampa.prefeitura.sp.gov.br/PaginasPublicas/_SBC.aspx. Last access 01/03/2020.
[38] Arcos-Garcia A, Alvarez-Garcia JA, Soria-Morillo LM. Evaluation of deep neural networks for traffic sign detection systems. Neurocomputing 2018;316:332–44.
[39] OpenStreetMap Wiki. KartaView — OpenStreetMap Wiki. 2021, https://wiki.openstreetmap.org/w/index.php?title=KartaView&oldid=2163193. Retrieved: 18/06/2021.
[40] Oliveira AAAM. INACITY's GeoImage class diagram. 2021, https://github.com/arturandre/INACITY/blob/master/django_website/docs/diagrams/GeoImage_example.png. Last access 28/06/2021.
[41] Mappilary. About - Mappilary. 2021, https://www.mapillary.com/about. Last access 10/06/2021.
[42] ©2019 Baidu. Baidu map (Translated from chinese). 2019, https://map.baidu.com/. Retrieved: 2019-12-04.
[43] Oliveira AAAM. INACITY use cases. 2021, https://youtu.be/K525hS7SsAg. Last access 10/03/2021.
[44] Oliveira AAAM. INACITY's Street sampling diagram. 2021, https://github.com/arturandre/INACITY/blob/master/django_website/docs/diagrams/diagram_street_sampling.png. Last access 28/06/2021.
[45] OpenStreetMap contributors. Overpass API/Overpass QL - openstreetmap wiki. 2019, https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL.
[46] Oliphant T. NumPy: A guide to NumPy. USA: Trelgol Publishing; 2006, http://www.numpy.org/. Retrieved: 2020-02-28.
[47] van der Walt S, Schönberger JL, Nunez-Iglesias J, Boulogne Fc, Warner JD, Yager N, et al. Scikit-image: image processing in Python. PeerJ 2014;2:e453. http://dx.doi.org/10.7717/peerj.453, https://doi.org/10.7717/peerj.453.
[48] Oliveira AAAM, Hirata Jr R, Buckeridge MS. Detecting trees near electric wires with deep learning. 2021, [submitted for publication].
[49] Oliveira AAAM. Quota subsystem diagram. 2021, https://github.com/arturandre/INACITY/blob/master/django_website/docs/diagrams/quota_system.png. Last access 28/06/2021.