Trajectory analysis and identification of significant points

Alessia Caputo - Naomi Demolli
alessia.caputo@studenti.unimi.it - naomi.demolli@studenti.unimi.it

Introduction

In the last decades we have witnessed the stunning spread of smartphones equipped with more and more precise and accurate sensors. The huge availability of this geographical information brings us challenges as well as opportunities to discover valuable knowledge from raw data.

The aim of this project is to develop a tool which allows us, at first, to manage raw trajectories, collected from our smartphones and, later, to extrapolate additional information from them.

In order to do this, we use Pyhton as programming language and Jupyter Notebook as environment. We use PostgreSQL - a free and open-source relational database management system - in order to memorize the trajectories and PostGIS to add support for geographic objects to the database. In order to visualize data and the results of our analyses, we use QGIS, a free and open-source cross-platform application that supports viewing, editing and analysis of geospatial data.

Our work is structured as follows:

- First of all, we explain how we collected the trajectories and we describe the characteristics of the dataset used in this project.
- In the next section, we show the pre-processing made on raw data. In particular the map-matching made on the trajectories in order to map points with low precision coordinates onto real roads.
- Then we implement the stay point detection algorithm, this allows us to detect the geographic regions in which the user stays for a while.
- After we implement and running on trajectory's points a density-based algorithm, DBSCAN. Later we add a temporal feature to DBSCAN algorithm. This allows us to detect more *reliable* clusters, indeed cluster expansion also considers interval time between points.
- We show metrics used to evaluate the effectiveness of the algorithms and the chosen thresholds.
- Finally, we show the results of the analysis made on all the trajectories of the dataset.

Data

We recorded 4 trajectories with the Android application "GeoTracker", in the period from December 2020 to January 2021. Most were recorded in central/southern Milan, one trajectory was recorded in Bollate, a city near by Milan. Each trajectory was exported in format GPX. We read GPX files using the library gpxpy. We imported into the database the points of each trajectory.
The tables have five attributes: id, latitude, longitude, geom, timestamp - [figure 1]

	id integer	lat numeric	lon numeric	geom geometry	time timestamp without time zone
1	0	45.470822	9.199488	0101000020E6100...	2020-12-27 13:33:30
2	1	45.470888	9.199592	0101000020E6100...	2020-12-27 13:33:41
3	2	45.470949	9.199713	0101000020E6100...	2020-12-27 13:33:54
4	3	45.471041	9.199732	0101000020E6100...	2020-12-27 13:34:04

Figure 1: example of table in database

Map-matching

The problem of GPS being non ideally accurate is not new, this led us to use an algorithm that matches raw GPS trajectories onto real roads networks. The most common approach is to consider each recorded point and relate it to an edge in an existing street graph.

First of all, we need the streets' map of the area we are interested in: we need one that contains the streets of Milan and another one containing the streets of Bollate. We got this data from OpenStreetMap (OSM), a collaborative project to create a free map of the world, using a QGIS plugin, QuickOSM. Once we have this information, we create a table which contains, for each point in the trajectory, the distance to the closest point of the closest road, as shown in figure 2.

```
create table {0}_pt_roads as
select distinct on (pid)pid, Mi_id, p_geom, Mi_geom, distance
from
    (select p.id as pid, p.geom as p_geom, Mi.id as Mi_id,
     Mi.geom as Mi_geom,
     ST_distance(
         ST_transform(p.geom, 3857),
         ST_transform(ST_LineInterpolatePoint(Mi.geom, ST_LineLocatePoint(Mi.geom,p.geom)),3857)) as distance
     from {0} as p, stradevicine as Mi) as point_road_couple
order by point_road_couple.pid,
    st_distance(point_road_couple.p_geom, point_road_couple.Mi_geom);
```

Figure 2: ptroads contains the closest road to a point p and the distance road-point

After that, we create a table that contains the interpolation of each point on the nearest road, provided that the distance between point and road is less than a certain amount of meters, figure 3. This last condition is necessary due to the fact that OSM map does not contain all of the streets: points in a trajectory close to an unregistered roads could be mapped to distant and non significant roads.

```
create table {0}_mapmatching as
    (select pid, ST_LineInterpolatePoint(Mi_geom, ST_LineLocatePoint(Mi_geom,p_geom)), distance
    from {0}_pt_roads
    where distance < 2 )
```

Figure 3: Table mapmatching contains only interpolated points

Since we need all the points, we join the table containing the interpolated points with the original table in which all the points are contained. We keep the original geometry if the point is too far from a road, otherwise we use the interpolated geometry, figure 4.

```
# Selezione punti proiettati - con geom modificata espressa da st_lineinterpolarepoint
query = "select st_lineinterpolatepoint, id from {0}_final_points where id = pid ".format(table)
cursor.execute(query)
risultati1 = cursor.fetchall()

# Selezione punti rimasti uguali - hanno pid nulla dal leftjoin
query = "select geom, id from {0}_final_points where pid is null".format(table)
cursor.execute(query)
risultati2 = cursor.fetchall()

risultati = risultati1 + risultati2

for element in risultati:
    query = "UPDATE {0}_final_points set final_geom = '{1}' where id = {2}".format(table,str(element[0]), str(element[-1]))
    cursor.execute(query)
    connection.commit()
```

Figure 4:

We can see a graphic example in figure 5: interpolated geometry is taken only when there's a road close to the original geometry

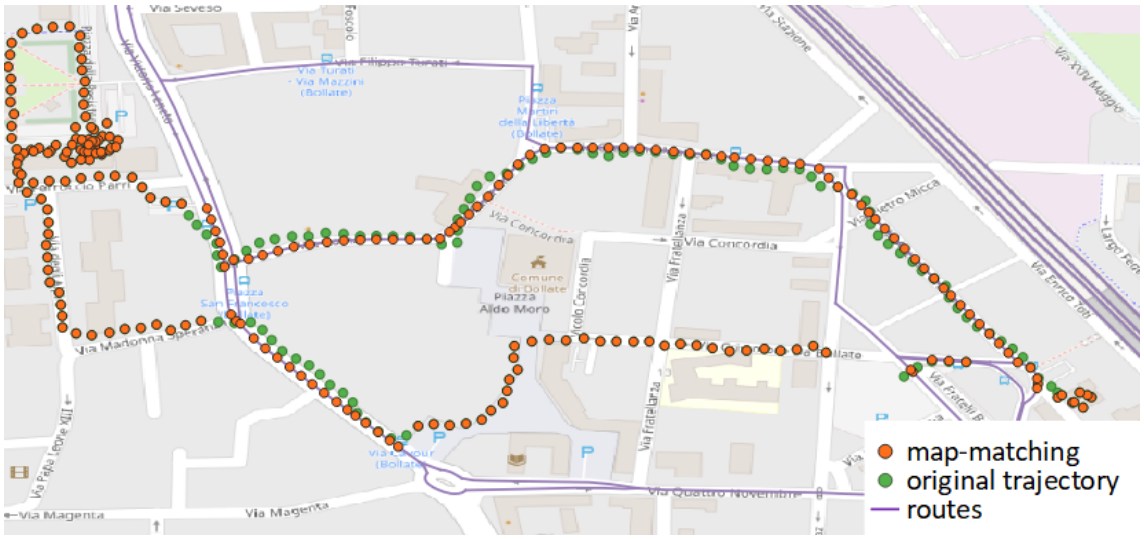

Figure 5: Example of map matching

There is only one threshold to set: the distance between point and road beyond which we keep the original geometry of the point, rather than the interpolated one.

Threshold	Description	Value
Map matching	Distance between point and road	< 10m

Stay point detection

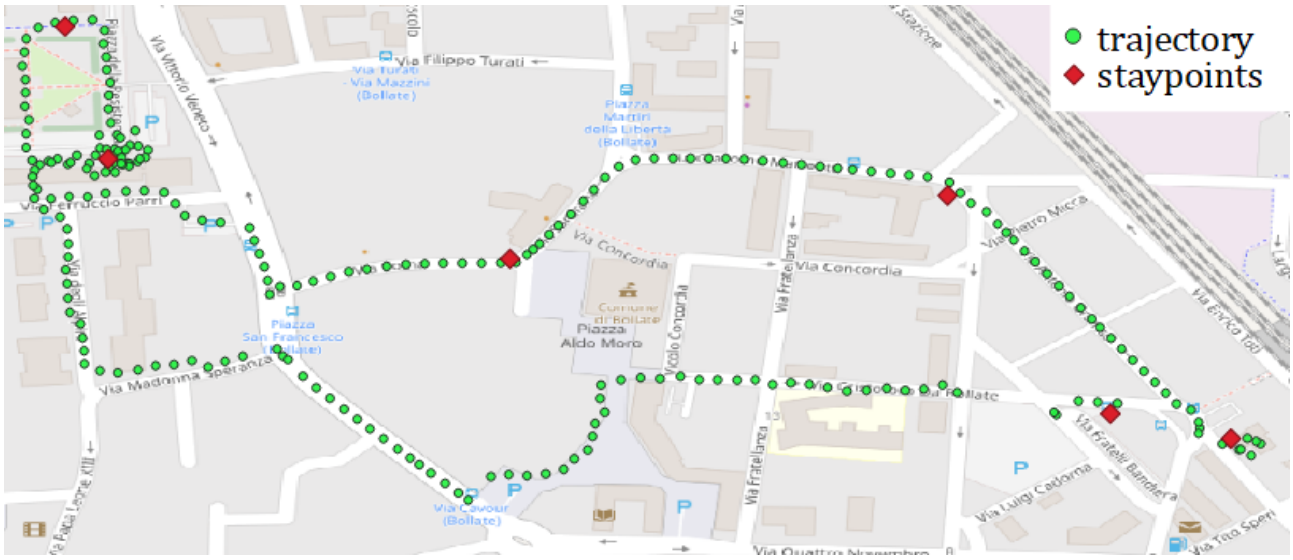

Figure 6: results of running staypoint algo on a trajectory

We are interested in stops made by the user during his/her trajectory. We define a stay point as a geographic area in which the user stays for a while, this probably indicates that it is a point of interest for the user. [1]

First of all we fix two thresholds: $distThre$ for distance and $timeThre$ for the time. Then we consider pairs of points in our trajectories, we compute the distance Δ_d and the interval Δ_t between them, with method delta-dist and delta-time, in figure 7.

```
def delta_dist(x, y): # X,Y geom
    cur = connection.cursor()
    cur.execute("select st_distance(ST_TRANSFORM('{}', 3857),ST_TRANSFORM('{}', 3857));".format(str(x), str(y)))
    dd = cur.fetchall()
    return dd[0][0]

def delta_time(x, y): # X,Y timestamp
    cur = connection.cursor()
    dt = y-x
    return dt.total_seconds()
```

Figure 7: methods to compute distance and time between two points

If points p_i, p_j have $\Delta_d(i,j) < distThre$ we continue to increment p_j until we find a couple p_i, p_j having $\Delta_d(i,j) > distThre$. When we detect it, we compute the time interval between the two points with method delta-time, in figure 7.

If $\Delta_t(i,j) > timeThre$, it means the user covered the distance between two points in a large amount of time, so he probably stopped. We calculate the centroid of points between $p_i, p_{i+1}, ..., p_{j-1}$ using method calc-centroid, in figure 8. We can define a centroid of a plane figure as the arithmetic mean position of all points in the figure, that centroid represents a stay point. At the end we saves all detected stay points in the database.

```python
def calc_centroide(listapunti, table = TABLE): # X,Y id dei punti
    cur = connection.cursor()
    listapunti = tuple(listapunti)
    query = """ SELECT ST_CENTROID(st_union(t.geom)) as cent_point
                FROM {} as t
                WHERE t.id in {} """.format(TABLE, listapunti)
    cur.execute(query)
    centroide = cur.fetchall()
    geom_cent = centroide[0][0]
    return geom_cent
```

Figure 8: method to calculate the centroid

```python
def staypoint(distThre, timeThre):
    cur = connection.cursor()
    cur.execute("select distinct * from {} order by id".format(TABLE))
    data = cur.fetchall()

    query = """ ALTER TABLE {0} DROP COLUMN IF EXISTS id_centroide;
                ALTER TABLE {0} ADD COLUMN id_centroide INTEGER """.format(TABLE)
    cursor.execute(query)
    connection.commit()

    i = 0
    num_points = len(data)
    id_centroide = 0
    stay_points = []

    while i < len(data):
        j = i+1

        while j < num_points:

            dist = delta_dist(data[i][3], data[j][3])

            if dist > distThre:
                temp = delta_time(data[i][4], data[j][4])

                if temp > timeThre:
                    lista_id = [i for i in range(i,j+1)]
                    query = """ UPDATE {0} SET id_centroide = {1}
                                WHERE id in {2}""".format(TABLE, id_centroide, tuple(lista_id))
                    cursor.execute(query)
                    connection.commit()
                    geom_cent = calc_centroide(lista_id)
                    time_inizio = data[i][4]
                    time_fine = data[j][4]
                    point = [id_centroide, geom_cent, temp, time_inizio, time_fine]
                    stay_points.append(point)
                    id_centroide += 1

                break
            j += 1
        i = j
```

Figure 9: stay point algorithm

We also set an attribute *centroid id* on the trajectory points used to calculate the stay point. It's helpful to visualize those points and have a graphic view of which points contribute to determine each centroid, we can see an example in figure 10.

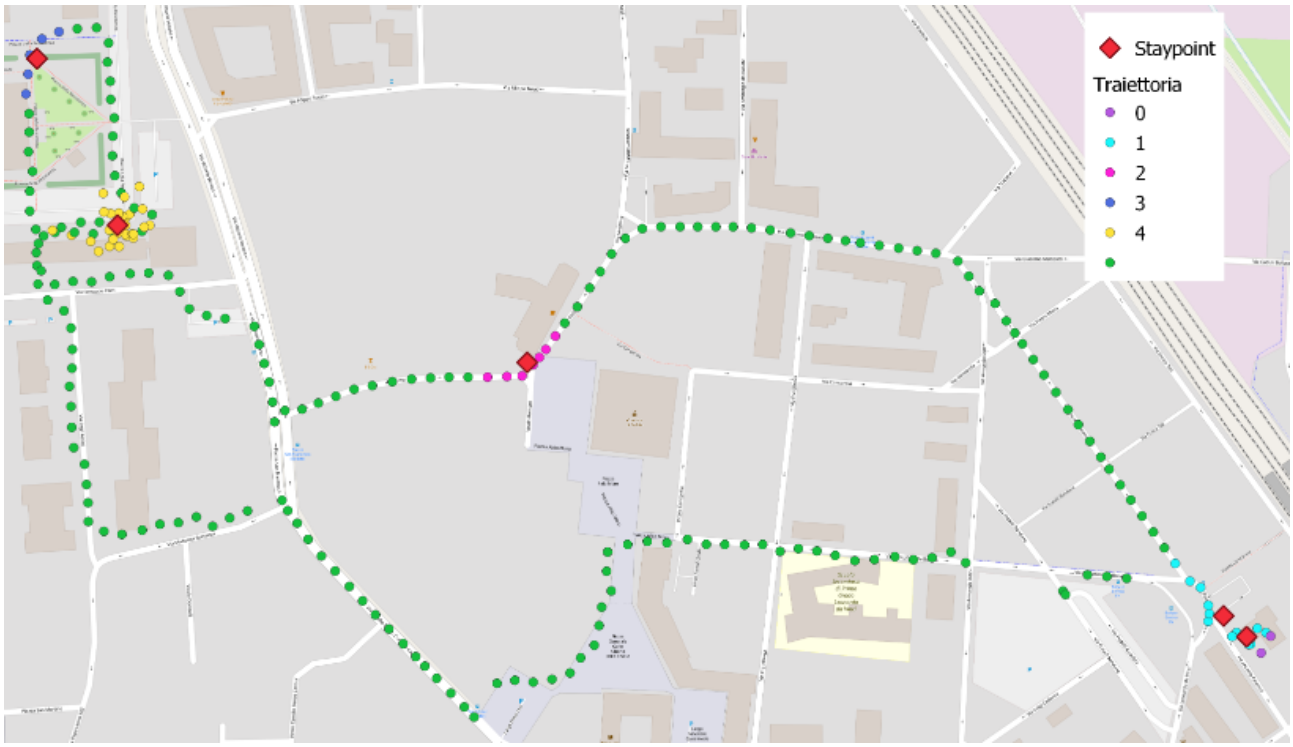

Figure 10: Points used to determine centroids

DBSCAN algorithm

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a data clustering and density based algorithm: we can define clusters as areas of higher density than the remainder of the data set. The key idea is that for each point of a cluster the neighborhood of a given radius has to contain at least a minimum number of points [2].

DBSCAN algorithm requires two parameters:

1. Eps: two points are considered to be neighbors if the distance between them are less than or equale to Eps.

   We can define the Eps neighborhood of a point p as

   $$N_{eps}(p) = \{q \in D | distance(p, q) \leq Eps\} \tag{1}$$

2. MinPts: number of neighbors, within the Eps distance, needed to create a cluster.

We have to consider two types of point within a cluster: points inside of the cluster (core points) and point on the border of the cluster (border points). DBSCAN forms a cluster starting from a core point, then expands the cluster by incorporating its neighborhood.

To address this, we introduce the notion of **density-reachability** [2]: a point p is directly density-reachable from a point q wrt Eps, MinPts if $p \in N_{Eps}(q)$, namely, if the two points are close, and $|N_{Eps}(q)| \geq MinPts$, q is a core point.

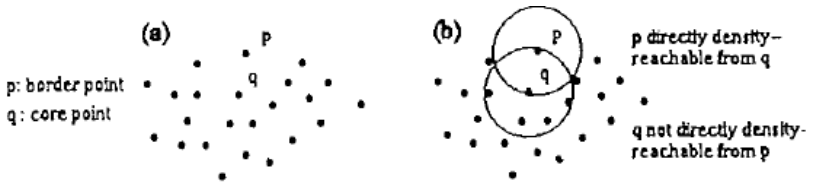

Figure 11: density reachability.

We need another definition [2], we state that a point p is **density-reachable** from a point q wrt Eps, MinPts if there is chain of points $p_1, ..., p_n$ with $p_1 = q, p_n = p$ such that p_{i+1} is directly density-reachable from p_i. Two border points of the same cluster C are possibly not density reachable from each other, as we can see in figure 12(a), therefore we introduce the notion of **density-connectivity**: a point p is density connected to a point q wrt. Eps and MinPts if there is a point o such that both, p and q, are density-reachable from o.

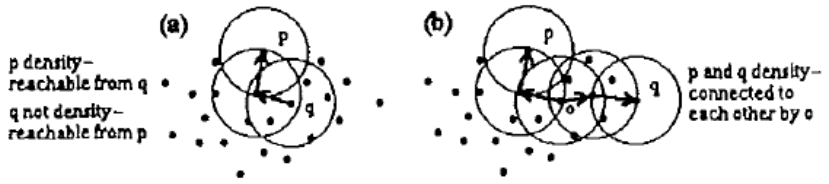

Figure 12: density connectivity

DBSCAN algorithm forms a cluster starting from a core point, the cluster is then extended to include all points density-reachable from the core point. In this way we can detect density-connected cluster.

DBSCAN algorithm works as follows: a starting point p is selected at random and it's neighbourhood area, $Neigh(p)$, is determined using radius Eps. If $|Neigh(p)| \geq MinPts$, p is marked as a core point (changing the value of its label to clusterindex) and a cluster formation starts (method espansione-cluster). Otherwise, the point p is classified as potential noise (changing the value of its label to 0), we don't know yet if it's noise or a border point.

```
def DBSCAN(eps, minpts, tempo = False, deltat = 0):

    cursor.execute("select count(*) from {}_final_points".format(TABLE))
    dataset_size = cursor.fetchone()[0]

    labels = [-1 for i in range(dataset_size+1)] # -1 è nodo non visitato
    cluster_index = 1

    for i, lbl in enumerate(labels):
        if lbl == -1:
            vicinato = get_vicini(i, eps, tempo, deltat)
            if len(vicinato) >= minpts:
                labels[i] = cluster_index
                espansione_cluster(i, vicinato, cluster_index, eps, minpts, labels, tempo, deltat)
                cluster_index += 1
            else:
                labels[i] = 0 # 0 nodo rumore

    new_column = 'clus_index_tempo' if tempo else 'clus_index'
    cursor.execute(""" ALTER TABLE {0}_final_points DROP COLUMN IF EXISTS {1};
                    ALTER TABLE {0}_final_points ADD COLUMN {1} INTEGER """.format(TABLE, new_column))
    connection.commit()

    for i in range(len(labels)):
        cursor.execute(""" UPDATE {0}_final_points SET {1} = {2}
                        WHERE {0}_final_points.id = '{3}'""".format(TABLE, new_column, labels[i], i))
        connection.commit()

    print('Numero cluster: ' + str(cluster_index-1))
```

Figure 13: DBSCAN algorithm's implementation in Python

The expansion of a cluster A begins by considering $Neigh(p)$ of the initial point p. All the points $q_1, ..., q_n \in Neigh(p)$, if they do not belong to any cluster, become a part of cluster A. Then we compute $Neigh(q_i)$ for each of these nodes, if $|Neigh(q_i)| \geq MinPts$, q_i is also a core node. Therefore we add $Neigh(q_i)$ to the set *seed*, which contains the points to be checked and density-reachable from p. Otherwise, if $|Neigh(q_i)| \leq MinPts$, q_i is a border point belonging to the cluster: its neighbors aren't added to the set *seed*. The algorithm proceeds - in the same way - until it runs out of points in the *seed*.

```
def espansione_cluster(p, vicinato, cluster_index, eps, minpts, labels, tempo, deltat):
    seed = vicinato
    for nodo in seed:
        if labels[nodo] == -1 or labels[nodo] == 0: # non visitato o noise
            labels[nodo] = cluster_index
            vicini_nodo = get_vicini(nodo, eps, tempo, deltat)
            if len(vicini_nodo) >= minpts:
                seed.extend(vicini_nodo)
```

Figure 14: Cluster expansion in Python

DBSCAN algorithm groups neighboring points if they are spatially close: clusters are areas where the density of points is higher. We are managing trajectories, lists of points with an essential

temporal feature. If we don't consider this last fact in cluster's creation, we could obtain clusters of points that are close in space but non necessary close in time. An example: a user passes the same point a few times but in different moments. If we use "traditional" DBSCAN we probably obtain a single cluster incorporating all of the points, without considering the time dimension.

In order to fix this, we introduced a time condition in DBSCAN algorithm, as we can see in figure 15. Considering a node, its neighbors have to be close both in space and in time.

```
def get_vicini(id_p, eps, tempo, deltat):

    if tempo:
        cursor.execute("""  SELECT t2.id FROM {0}_final_points as t1, {0}_final_points as t2
                        WHERE st_distance(st_transform(t1.geom, 3857), st_transform(t2.geom, 3857)) <= {1} and t1.id = '{2}'
                        AND abs(extract(epoch from t2.time::timestamp - t1.time::timestamp )) <{3}
                        """.format(TABLE, eps, str(id_p), deltat))
    else:
        cursor.execute(""" SELECT t2.id FROM {0}_final_points as t1, {0}_final_points as t2
                        WHERE st_distance(st_transform(t1.geom, 3857), st_transform(t2.geom, 3857)) <= {1}
                        AND t1.id = '{2}' """.format(TABLE, eps, id_p))

    data = cursor.fetchall()
    vicini = []
    for i in range(len(data)):
        vicini.append(int(data[i][0]))

    return vicini
```

Figure 15: add a time component to the algorithm

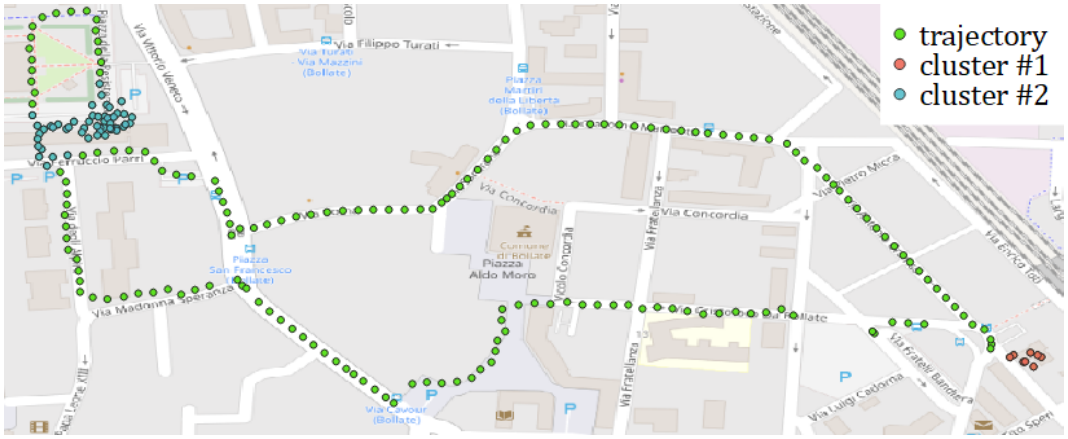

Figure 16: results of DBSCAN algo on the second trajectory

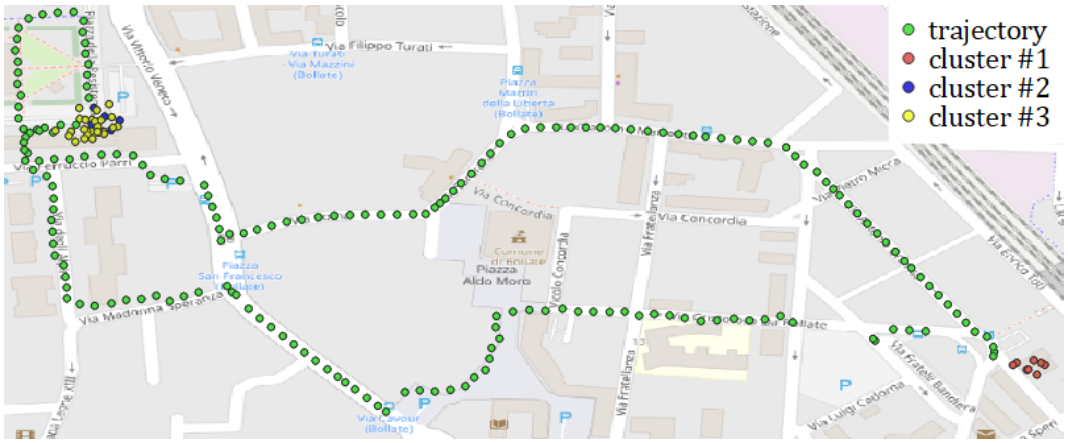

Figure 17: results of DBSCAN algorithm on the second trajectory with time condition

In figure 16, after the execution of the DBSCAN algorithm we get only two clusters, instead - as we can see in figure 17 - we obtain three clusters after running temporal DBSCAN algorithm. The second cluster of the first image is divided in two smaller clusters, this probably indicates that the user has passed there twice in two different moments. In order to be sure about this fact, we plot the trajectory in three dimensions, figure 18 and figure 19.

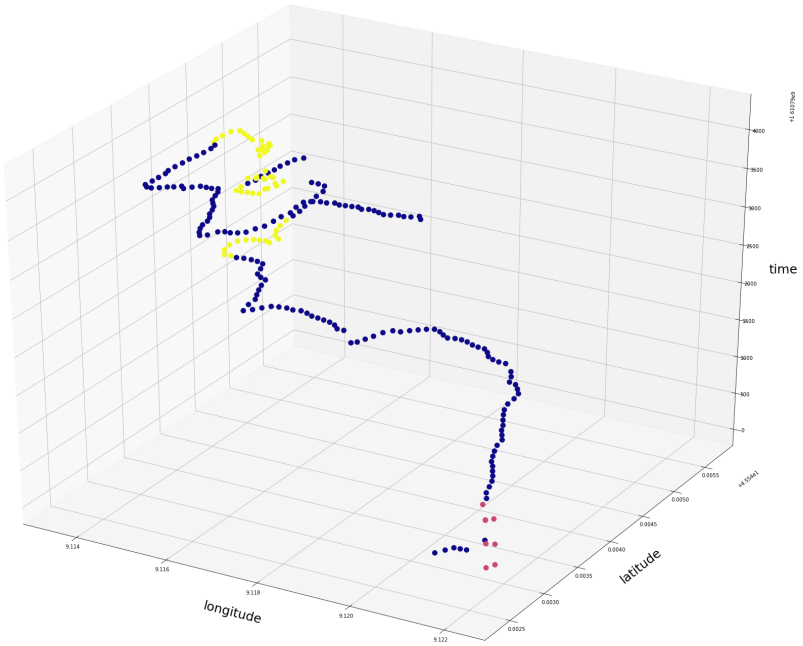

Figure 18: results of DBSCAN algorithm on the second trajectory

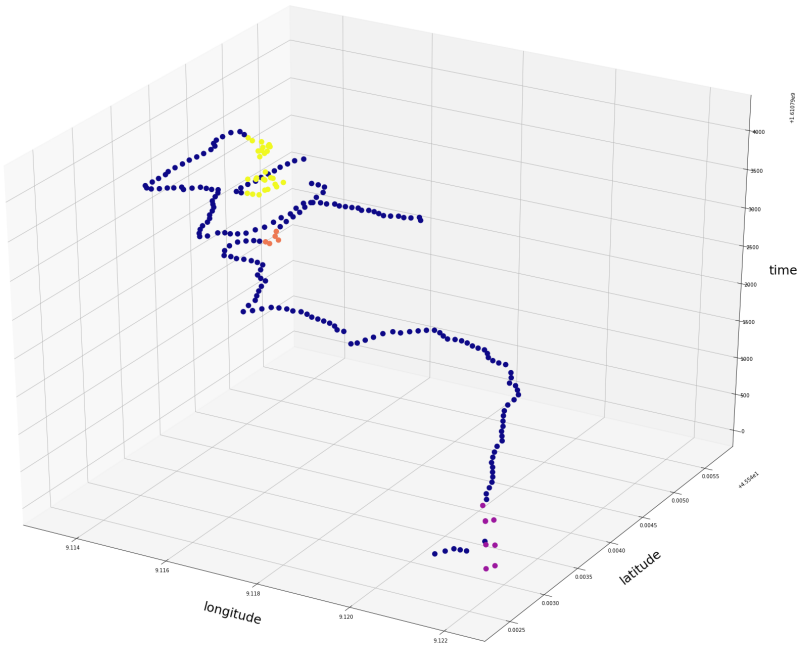

Figure 19: results of DBSCAN algorithm on the second trajectory with time condition

Figure 19 shows that user has passes in the same area in two different moments. Figure 18 confirms that using the "traditional" DBSCAN algorithm this fact does not emerge, so we can assume that the DBSCAN with time condition works better in our project.

Once we find clusters, we calculate the centroid of each of them, in order to do that we use the function already presented in figure 8. These points represent the points where the user probably stopped in the trajectory. We can compare the results obtained with stay point algorithm and with "temporal" DBSCAN. For example, in figure 20 we can see the stop points detected by DBSCAN. If we compare figure 20 and figure 10 we can see that some centroids and staypoints are in the same area.

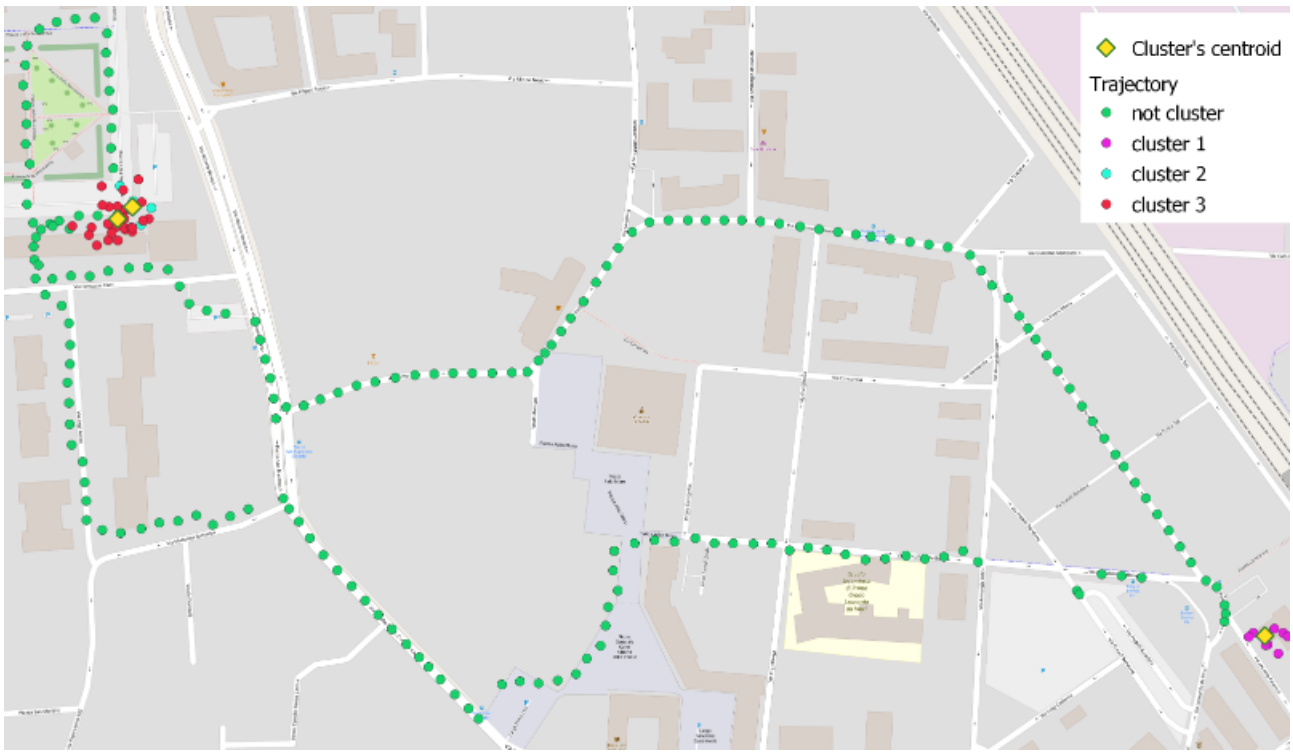

Figure 20: Temporal dbscan: centroids clusters

Analysis metrics

Once the staypoint algorithm and temporal dbscan have been implemented, we need a way to evaluate their performances. In particular, we'd like to know how effective the two algorithms are in finding points of interest for the user. To achieve this, we introduce the notion of waypoints.
A **waypoint** is a point added to the trajectory by the user. We can see it as a veracity label indicating the area around the point as a real and important stop.

At first, it's interesting to calculate the percentage of waypoints close to a centroid over the total number of waypoints, figure 21. This is done separately for the centroids obtained by stay point detection algorithm and for those calculated from clusters. This tells us how good our two algorithms are at detecting user's points of interest.

$$\text{Total waypoints detected} = \frac{WP\ near\ centroid}{tot\ WP}$$

Figure 21: total waypoints detected

Ideally, we would like to identify only the stops that are relevant to the user, so we define noises all the centroids far from any waypoints. We calculate the percentage of noises returned by an algorithm as total number of centroids far from waypoints over the total number of centroids.

$$\text{Total noises detected} = \frac{centroids\ far\ WP}{tot\ centroids}$$

Figure 22: total noises detected

Assuming that the user manually added all his points of interest as waypoints, we have perfect algorithms if they identify all the waypoints without any mistake. So, the ideal percentage of detected waypoints is 100%, the ideal percentage of noise is 0%.

Results

We use the metrics described in the precedent paragraph to evaluate the best thresholds for each algorithm. We want to maximize the percentage of detected waypoints and minimize the percentage of noises. We obtained the best staypoint detection results with the thresholds listed in the following table:

Threshold	Description	Value
Stay point: Δ_d	Distance covered by the user	>50m
Stay point: Δ_t	Amount of time to cover a distance	$>$ 180s

Instead as regards the DBSCAN algorithm, with respect to the metrics above, we get these thresholds:

Threshold	Description	Value
DBSCAN eps:	neighborhood distance	$<$ 20m
DBSCAN MinPts:	min number of neighbors within eps	$=$ 5
DBSCAN deltat:	neighborhood interval time	$=$ 240s

Finally, we present the results of the two algorithms applied to the 4 trajectories. The staypoint detection algorithm detected 84% of the waypoints with a noise percentage of 25%. As regards the temporal DBSCAN algorithm, we can obtained a 46% of detected waypoints and 45% of the centroids as noise.

In conclusion, we can say that the stay point algorithm seems to behave better than the temporal dbscan with the indicated thresholds. The stay point algorithm gives good results, the percentage of detected waypoints is very high. The percentage of noise that we obtained it's low but not close to the ideal value: the algorithm found some points where the user stopped for few minutes without adding a waypoint on the trajectory.

Graphic overview of the trajectories

Finally we offer a 2D and a 3D graphic representation of our 4 trajectories: in 3D the points of the trajectory are blue, we've indicated the waypoints with a black X and the staypoint detected by the algorithm with red X. We can also observe clusters of points represented with different colors.

Figure 23: 2D overview of t1

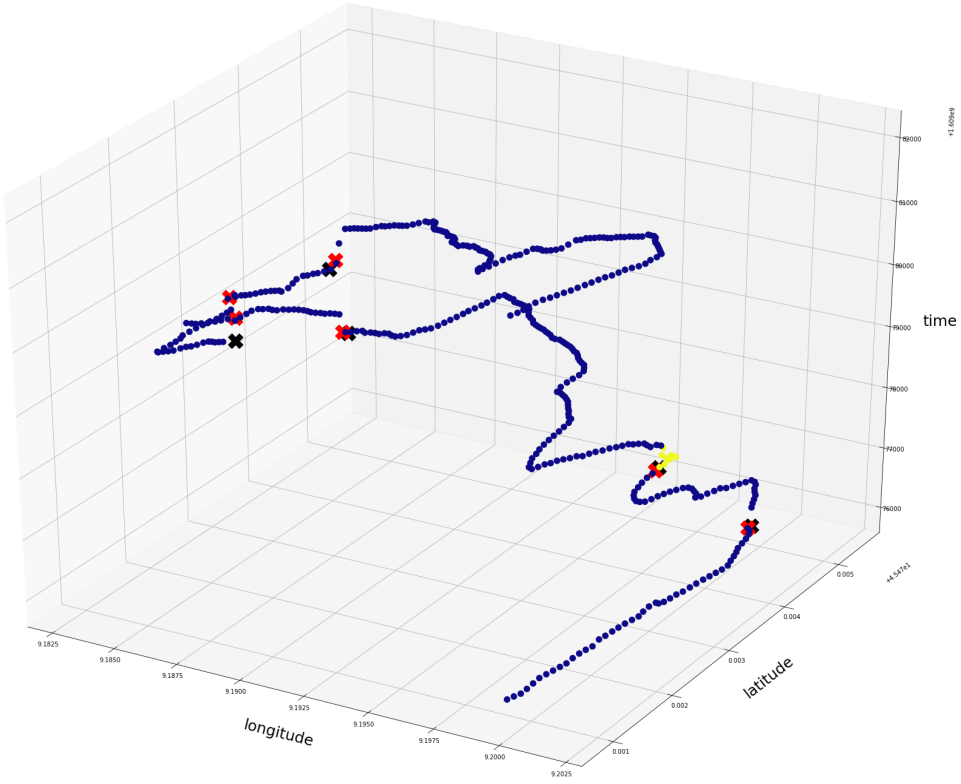

Figure 24: 3D overview of t1

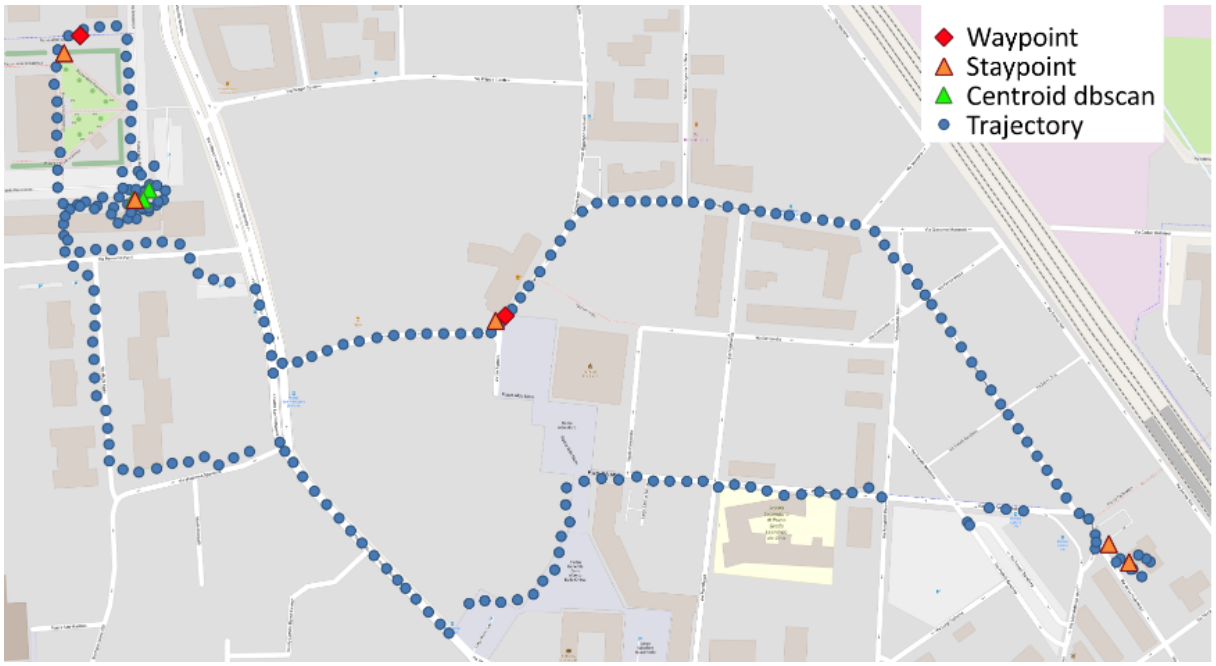

Figure 25: 2D overview of t2

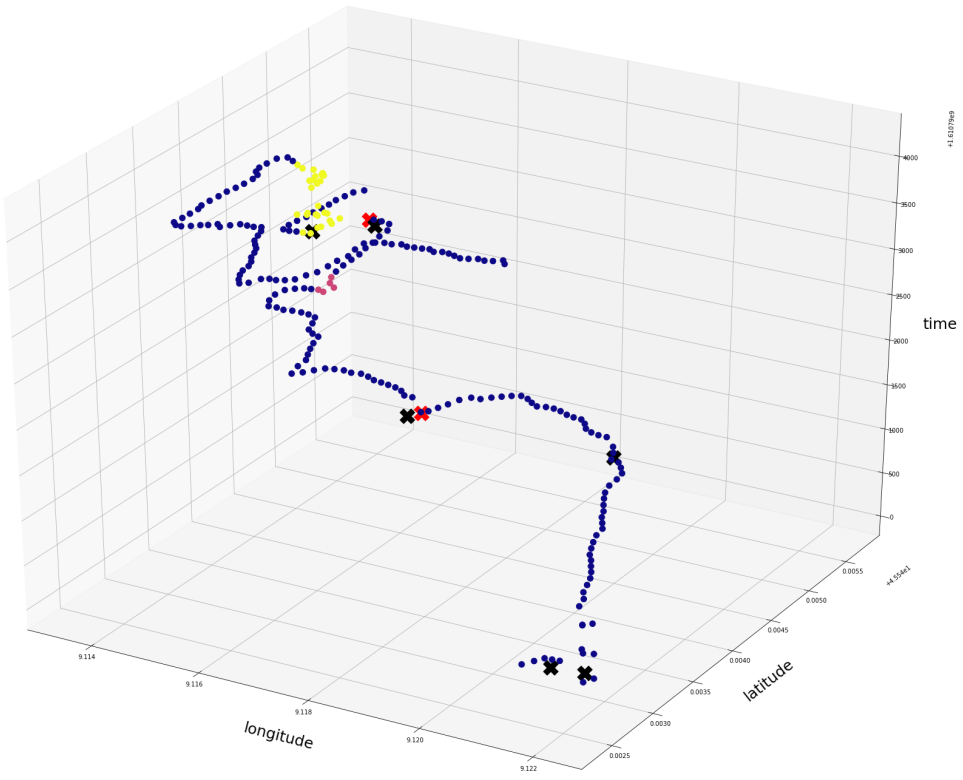

Figure 26: overview of t2

Figure 27: 2D overview of t3

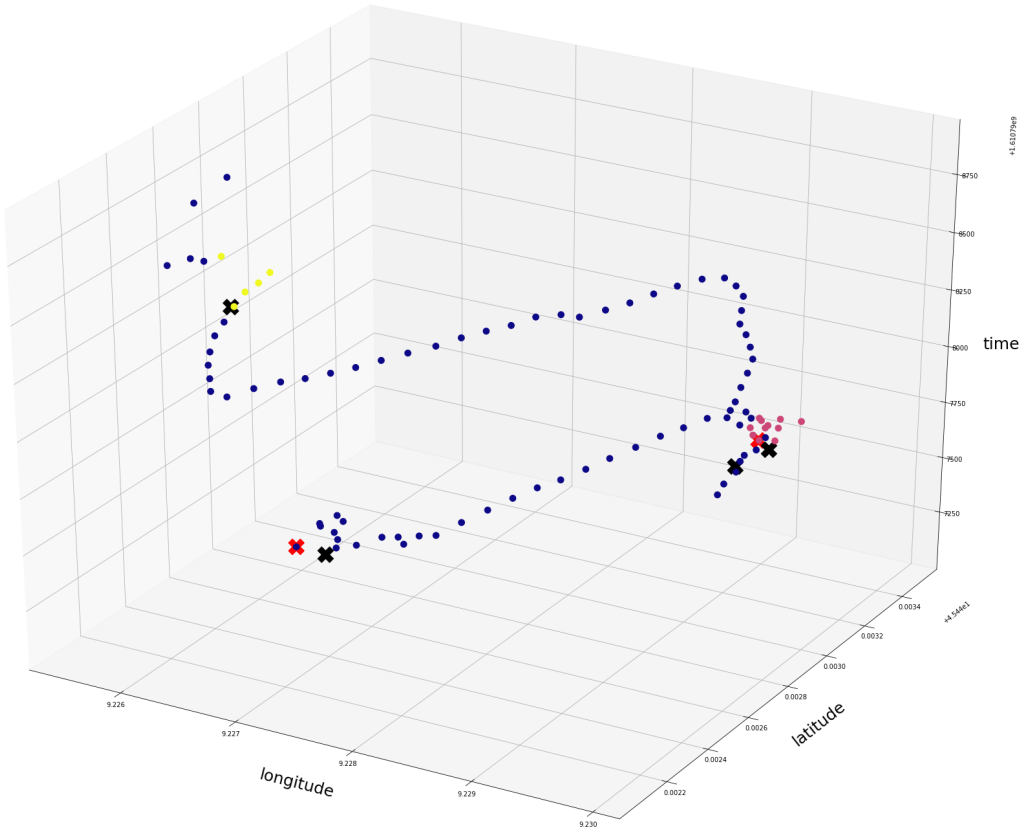

Figure 28: overview of t3

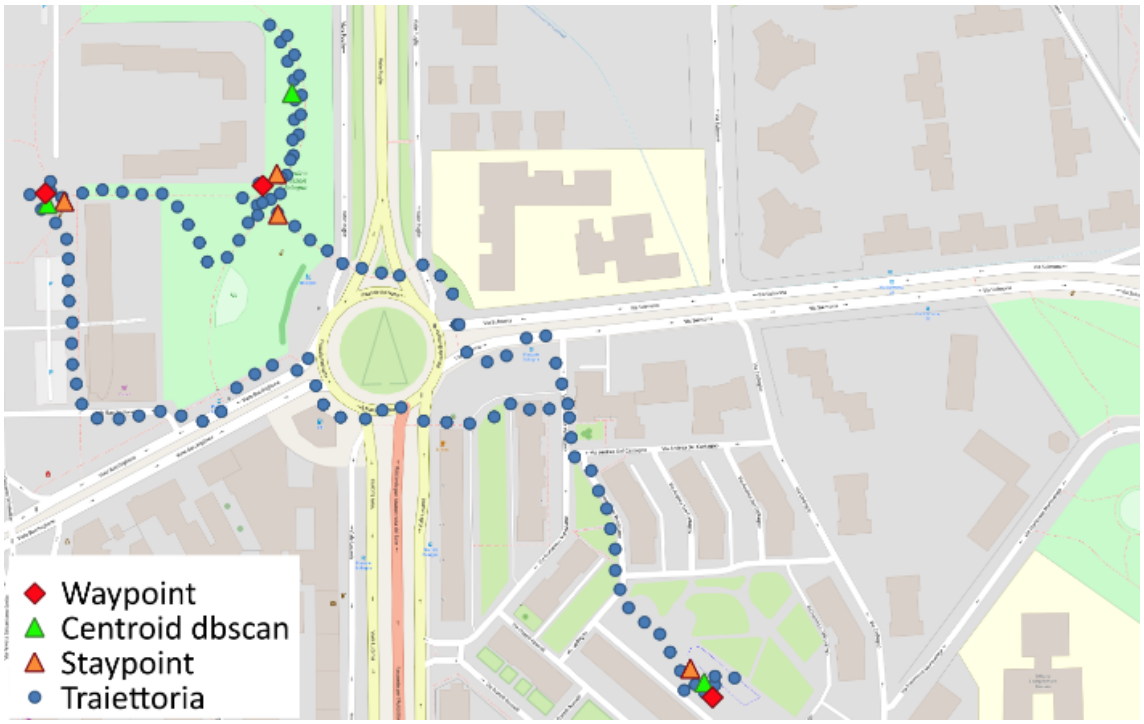

Figure 29: 2D overview of t4

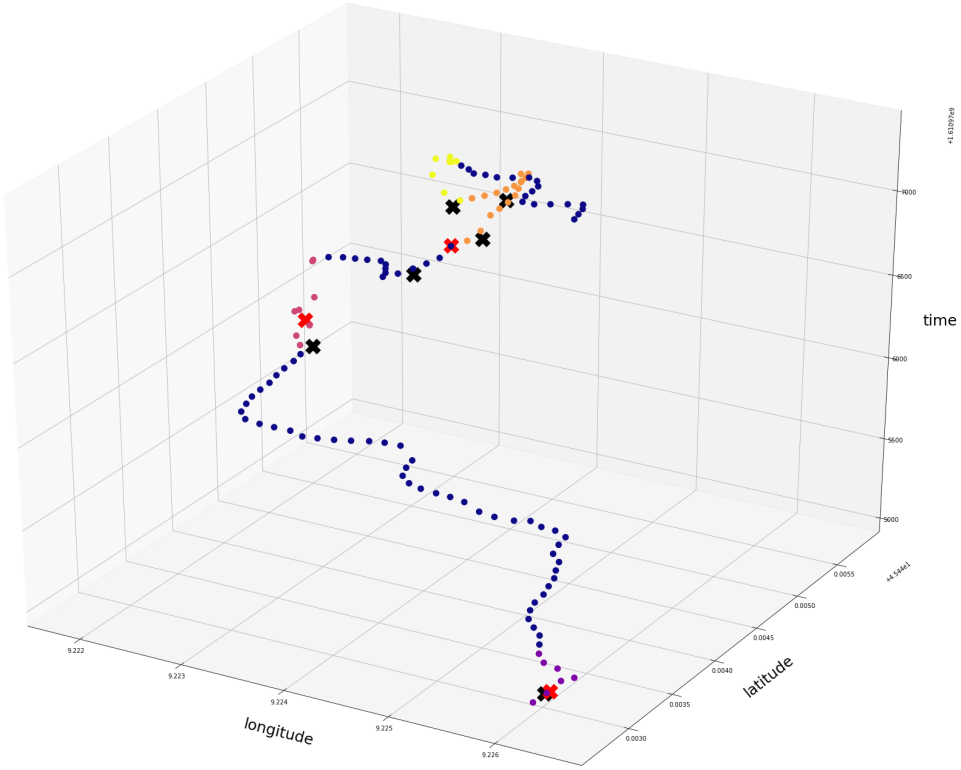

Figure 30: overview of t4

References

[1] Yang Ye, Yu Zheng, Yukun Chen, Jianhua Feng, Xing Xie *Mining individual life pattern based on location history*. 2009.

[2] Martin Ester, Hans-Peter Kriegel, Jorg Sander, Xiaowei Xu *A Density-Based Algorithm for Discovering Clusters*. 1996.

We declare that this material, which we now submit for assessment, is entirely our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should we engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.