Analyzing Transportation Modes

Geospatial Data Management (2024)

Prof. Maria Luisa Damiani

Behnam Najafloo

Master student in Computer Science

University of Milan

Abstract:

This project focuses on analyzing transportation modes of Bergamo city (such as walking and riding a bus) using location tracking data collected via a GPX file. The goal is to derive transportation modes based on the variation of speed throughout a trajectory, using QGIS for data processing and Python for calculating speed and determining modes of transport. The project demonstrates how to process real-world movement data, analyze the results, and compare them with a manually logged ground truth of transportation times.

Introduction:

In today's world, location-based services play a crucial role in understanding human mobility. Whether for research, urban planning, or personal fitness tracking, trajectory data reveals significant insights. The purpose of this project is to analyze movement patterns recorded in GPX format, calculate speed, and classify transport modes (such as walking and bus travel) in QGIS.

The workflow involves collecting trajectory data, importing it into QGIS, preprocessing the data, calculating speed, and determining the transport mode based on predefined speed thresholds. Finally, the results are compared against manually logged ground truth data for verification.

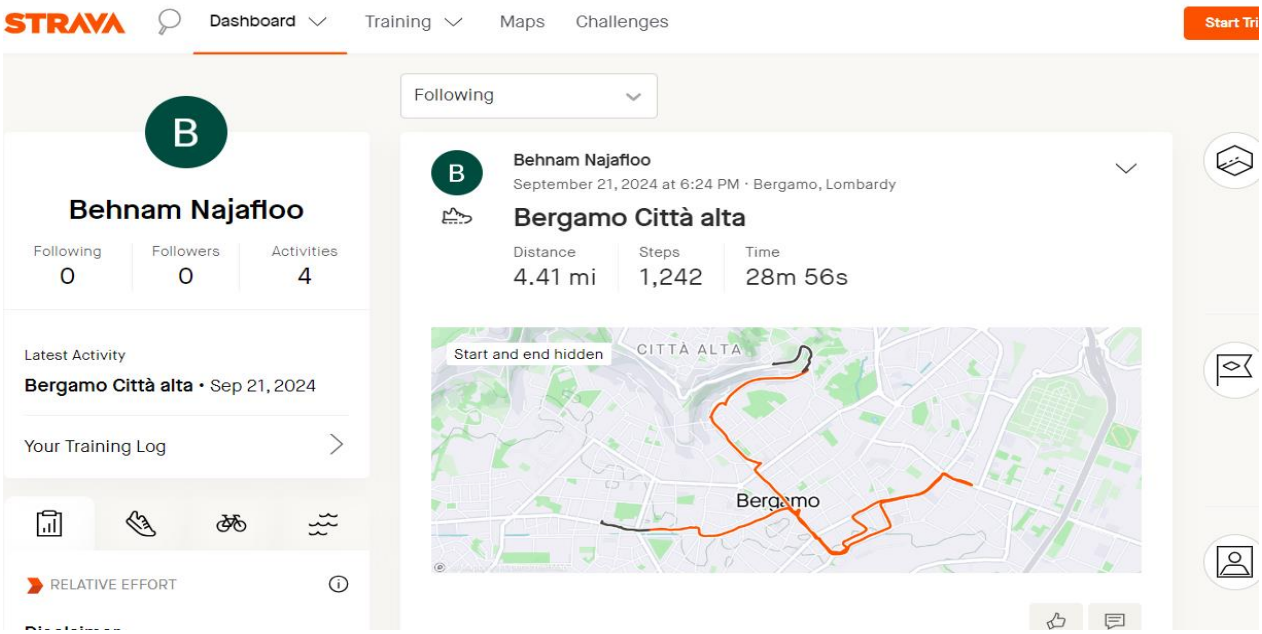
Methodology:

- Data Collection
- Import GPX File into QGIS
- Preprocessing the Data
- Calculating Speed
- Assigning Transportation Modes
- Comparison with Ground Truth

Step 1: Data Collection

The first step was collecting location-based data using a tracking app that exports in GPX format. The GPX file was generated by recording a trajectory while switching between walking and bus travel. The following are the key steps of this phase:

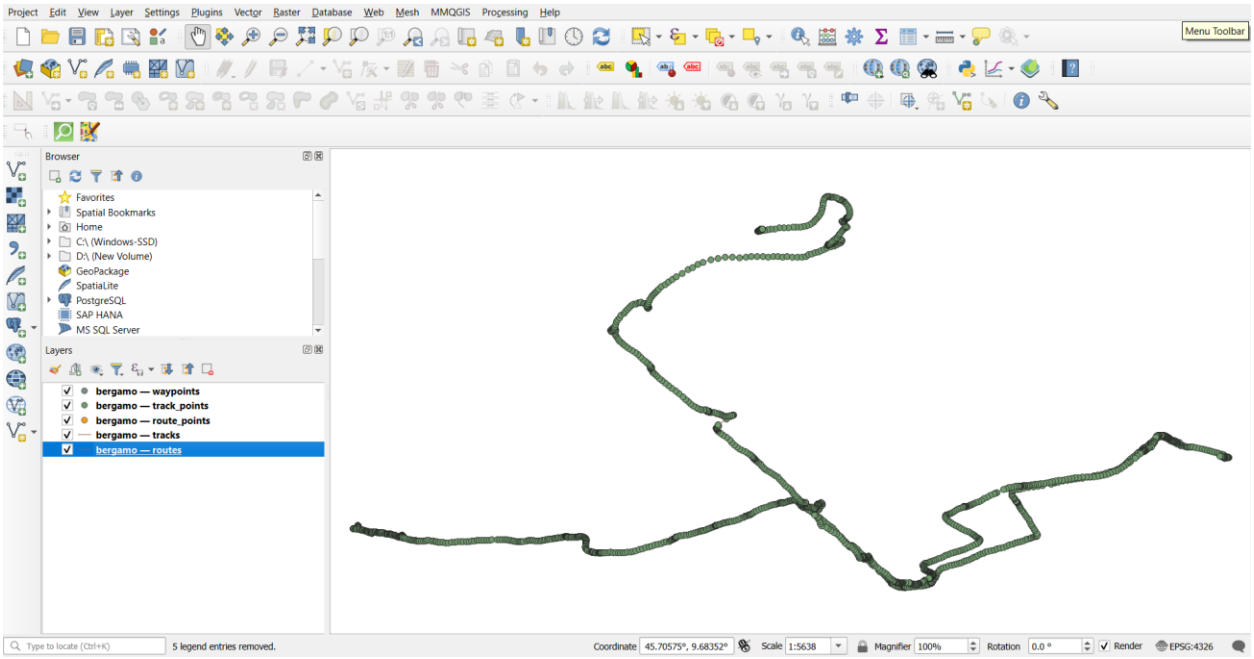
- **Tools Used**: A GPS tracking app (such as Geo Tracker or Strava) to collect the data.
- **Data Collected**: Time-stamped geographic coordinates along the movement route.



Step 2: Import GPX File into QGIS

The GPX file was imported into QGIS, a powerful open-source GIS tool, for analysis. QGIS enabled us to visualize the trajectory, view attributes (such as time, latitude, and longitude), and perform geospatial operations.

- **Layer Setup**: The trajectory data was transformed into a format suitable for further analysis, where each data point was associated with a timestamp and geographic coordinates.



Before step 3, since the data is likely in latitude and longitude (WGS84, EPSG:4326), we need to transform it to a metric projection (EPSG:3857) for distance calculations in meters. Here's how to do that:

1. Right-click on the `bergamo — track_points` layer.
2. Choose **Export** > **Save Features As...**.
3. In the **CRS** section, choose **EPSG:3857**.
4. Save the new layer as `track_points_transformed`.

Step 3: Preprocessing the Data

Once the data is in metric units, we can calculate the time differences between consecutive track points.

- Transforming the 'time' field
- Calculating the time difference between each point and the previous one.
- Calculating distances between points

Calculating Time Difference: A time difference between consecutive points was also computed using the `timestamp` field. This difference is crucial for determining the speed.

- Adding a new field called `timestamp` (Field type: `Date` or `DateTime`).
- Using the following expression in the **Field Calculator** to convert the `time` field to a standard timestamp format:

```
to_datetime("time")
```

Calculating Distance: Using QGIS's spatial functions, the distance between consecutive points was calculated.

- In the attribute table of `track_points_transformed`, adding a new field called `time_diff` (in seconds).
- We can use the `LAG()` function in SQL to calculate the time difference between each point and the previous point.

Here's the SQL query for time difference calculation:

```
WITH time_diffs AS (
  SELECT
    fid,
    EXTRACT(EPOCH FROM (timestamp - LAG(timestamp) OVER (ORDER BY timestamp))) AS
time_diff
  FROM track_points_transformed
)
UPDATE track_points_transformed
SET time_diff = time_diffs.time_diff
FROM time_diffs
WHERE track_points_transformed.fid = time_diffs.fid;
```

Calculating Distances Between Points: Now that we have time differences, we can calculate the distances between consecutive points in meters.

To add the `distance_meters` field, we will recalculate the distance between consecutive points.

Here's SQL query:

```
ALTER TABLE track_points_transformed ADD COLUMN distance_meters DOUBLE PRECISION;
```

Calculate distance in meters using the `ST_Transform` and `ST_Distance` functions:

We will calculate the distance between points and store them in the `distance_meters` field using Python in the QGIS console.

```
# Get the layer
layer_name = 'track_points_transformed'
layers = QgsProject.instance().mapLayersByName(layer_name)

if not layers:
    print(f"Layer '{layer_name}' not found.")
else:
    layer = layers[0]

    # Start editing the layer
    layer.startEditing()

    # Create a new field for speed if it doesn't exist
    if layer.fields().indexFromName('speed_kmh') == -1:
        layer.dataProvider().addAttributes([QgsField('speed_kmh',
QVariant.Double)])
        layer.updateFields()

    # Iterate over the features
    for feature in layer.getFeatures():
        # Get the time difference and distance in meters
        time_diff = feature['time_diff']  # Assuming time_diff is in seconds
        distance_meters = feature['distance_meters']  # Distance in meters

        # Check if time_diff and distance_meters are not None
```

```
        if time_diff is not None and distance_meters is not None and time_diff >
0:
            # Calculate speed in km/h (distance in meters / time in seconds *
3.6)
            speed = (distance_meters / time_diff) * 3.6
            # Update the feature with the calculated speed
            layer.changeAttributeValue(feature.id(),
layer.fields().indexFromName('speed_kmh'), speed)
        else:
            # Optionally log or handle cases where values are None
            print(f"Skipping feature {feature.id()} due to None values
(time_diff: {time_diff}, distance_meters: {distance_meters})")

    # Commit the changes to the layer
    layer.commitChanges()
```

Step 4: Calculating Speed

Using the time difference and distance fields, speed (in km/h) could be calculated for each trajectory point. The formula for speed is:

$$\text{Speed (km/h)} = \frac{\text{Distance (meters)}}{\text{Time (seconds)}} \times 3.6$$

Add a Speed Field:

- First, adding a `speed_kmh` field in the table.
  SQL command:

```sql
ALTER TABLE track_points_transformed ADD COLUMN speed_kmh DOUBLE PRECISION;
```

Calculate Speed:

- We can then write an SQL query to calculate the speed based on the distance and the time difference (which we should also have in our table, likely in seconds).
  Here's a sample query:

```sql
UPDATE track_points_transformed
SET speed_kmh =
    CASE
        WHEN time_diff IS NOT NULL AND time_diff > 0
        THEN (distance_meters / time_diff) * 3.6
        ELSE NULL
    END;
```

The formula `(distance_meters / time_diff) * 3.6` converts the speed from meters per second to kilometers per hour. The factor 3.6 comes from converting seconds to hours and meters to kilometers (since 1 hour = 3600 seconds and 1 kilometer = 1000 meters).

Verify and Analyze Speed Values

- After executing the query, we can run a simple SELECT query to verify that the speeds are calculated correctly:

```sql
SELECT id, distance_meters, time_diff, speed_kmh
FROM track_points_transformed
WHERE speed_kmh IS NOT NULL
ORDER BY id;
```

Step 5: Assigning Transportation Modes

To assign values to the `transport_mode` field (like "walking" and "on the bus"), we can use conditional logic based on the calculated speed or other criteria that we have established in our data analysis. Below is a Python code snippet that demonstrates how to assign values to the `transport_mode` field based on the calculated speed.

```
from datetime import datetime
from qgis.core import QgsProject, QgsField
from PyQt5.QtCore import QVariant

# Get the layer
layer_name = 'track_points_transformed'
layers = QgsProject.instance().mapLayersByName(layer_name)

if not layers:
    print(f"Layer '{layer_name}' not found.")
else:
    layer = layers[0]

    # Start editing the layer
    layer.startEditing()

    previous_time = None
    for feature in layer.getFeatures():
        # Get the current timestamp
        current_time = feature['timestamp']

        # Convert QDateTime to Python datetime
        if current_time.isValid():
            current_time = current_time.toPyDateTime()

            if previous_time is not None:
                time_diff = (current_time - previous_time).total_seconds()
                distance_meters = feature['distance_meters']

                # Logging for better visibility
                print(f"Feature ID: {feature.id()}, Previous Time:
{previous_time}, Current Time: {current_time}, Time Diff: {time_diff}, Distance:
{distance_meters}")
```

```
                # Check if time_diff and distance_meters are valid
                if time_diff > 0 and distance_meters is not None:
                    speed = (distance_meters / time_diff) * 3.6  # km/h
                    layer.changeAttributeValue(feature.id(),
layer.fields().indexFromName('speed_kmh'), speed)

                    # Assign transport mode based on speed
                    if speed == 0:
                        transport_mode = 'stop'
                    elif 0 <= speed < 5:
                        transport_mode = 'walking'
                    else:
                        transport_mode = 'on the bus'

                    layer.changeAttributeValue(feature.id(),
layer.fields().indexFromName('transport_mode'), transport_mode)
                else:
                    print(f"Skipping feature {feature.id()} due to invalid
time_diff: {time_diff} or distance_meters: {distance_meters}")

            previous_time = current_time
        else:
            print(f"Skipping feature {feature.id()} due to invalid timestamp.")

    # Commit the changes to the layer
    layer.commitChanges()
```

Define Speed Thresholds: Establish thresholds for different modes of transportation:

- **Stop**: Speed = 0
- **Walking**: Speed between 0 and 5 km/h
- **On the Bus**: Speed greater than 5 km/h

This logic was implemented in Python, and the transport mode for each feature was derived based on the computed speed.

```
if speed == 0:
    transport_mode = 'stop'
elif 0 < speed < 5:
```

```
    transport_mode = 'walking'
else:
    transport_mode = 'on the bus'
```

Attribute table respectively for the situation of ‘on the bus’ and ‘walking’ with field of respectively ‘timestamp’, ‘time_diff’, ‘speed_kmh’, ‘distance_meters’, and ‘transport_mode’:

	9/21/2024 17:30:43 (W....	0.01666666666...	11.7373375289...	3.26037153581...	on the bus
	9/21/2024 17:31:00 (W....	0.01666666666...	17.0850356748...	4.74584324301...	on the bus
	9/21/2024 17:31:01 (W....	0.01666666666...	6.29924891585...	1.74979136551...	on the bus
	9/21/2024 17:29:54 (W....	0.01666666666...	26.5683972318...	7.38011034217...	on the bus
	9/21/2024 17:29:55 (W....	0.01666666666...	16.8324148676...	4.67567079657...	on the bus
	9/21/2024 17:29:56 (W....	0.01666666666...	11.1714969749...	3.10319360415...	on the bus
	9/21/2024 16:25:32 (W....	0.01666666666...	0	0	stop
	9/21/2024 16:26:04 (W....	0.01666666666...	0	0	stop
	9/21/2024 16:26:08 (W....	0.01666666666...	0	0	stop
	9/21/2024 16:26:12 (W....	0.01666666666...	0	0	stop

NULL	9/21/2024 17:29:11 (W....	0.01666666666...	0	0	stop
NULL	9/21/2024 17:29:16 (W....	0.01666666666...	0	0	stop
NULL	9/21/2024 17:29:22 (W....	0.01666666666...	0	0	stop
NULL	9/21/2024 17:29:34 (W....	0.01666666666...	0	0	stop
NULL	9/21/2024 17:31:04 (W....	0.01666666666...	0	0	stop
NULL	9/21/2024 17:29:05 (W....	0.01666666666...	0	0	stop
NULL	9/21/2024 17:31:19 (W....	0.01666666666...	0	0	stop
NULL	9/21/2024 16:25:03 (W....	0.01666666666...	11.9144662057...	3.30957394604...	unknown
NULL	9/21/2024 16:24:23 (W....	0.01666666666...	10.2668120329...	2.85189223136...	unknown
NULL	9/21/2024 16:24:42 (W....	0.01666666666...	0	0	unknown

Step 6: Comparison with Ground Truth

Ground truth is used for **validation**—comparing the predicted results (in this case, our transportation mode classification) with what actually happened. This helps us evaluate how accurate our model or approach is.

How to Collect Ground Truth in Your Case:

External information:

- You could also rely on external data, like a **schedule for the bus route** or any **timing information** for specific segments.

Manually record activities:

- While collecting the GPX data, you can **keep a log** or **note down** what you were doing at specific times (e.g., noting when you were walking, on the bus, or stopped).

My Ground Truth Notes on 21/09/2024 start from 18:25 in Bergamo:

- Get on bus at 18:25
- Get off bus at 18:40
- paused the application
- Get-on bus number 7 at 19:06
- Get-off bus number 8 at 19:11
- Walking at 17:11 until 19:14
- Get-on bus number 1 at 19:14
- Get-off bus number 8 at 19:29

However, there was an issue, I noted the time that I was walking or getting on the bus, but the time that I saw in the trajectory was different. For example, I started at 18:25 but I discovered it in the table time started from 16:25.

This problem is likely due to a discrepancy between **time zones**. The times recorded in my GPX file or trajectory data are likely using **UTC** (Coordinated Universal Time) rather than my local time zone.

Solution: Adjust Time Zone in the Data

To correct this, we can adjust the timestamps in our table by converting them to our local time zone. This can be done by adding the appropriate time offset.

Here's how you can fix this:

Determine the Time Difference:

- Identify the time zone difference between **UTC** and your **local time**. If you're in a UTC+2 zone, you'll need to add 2 hours to the recorded timestamps.

Python Code to Adjust Time: We can also modify our timestamps by adding the offset:

```
from datetime import datetime, timedelta

layer_name = 'track_points_transformed'  # Your layer name
layers = QgsProject.instance().mapLayersByName(layer_name)

if not layers:
    print(f"Layer '{layer_name}' not found.")
else:
    layer = layers[0]

    # Start editing the layer
    layer.startEditing()

    # Iterate over the features
    for feature in layer.getFeatures():
        timestamp = feature['timestamp']  # Assuming 'timestamp' is the field
with the original time
        if timestamp is not None:
            # Convert QDateTime to Python datetime
            timestamp_py = timestamp.toPyDateTime()

            # Adjust for time zone (e.g., UTC+2)
            adjusted_time = timestamp_py + timedelta(hours=2)

            # Update the feature with the corrected time
            layer.changeAttributeValue(feature.id(),
layer.fields().indexFromName('timestamp'), adjusted_time)

    # Commit the changes to the layer
    layer.commitChanges()
```

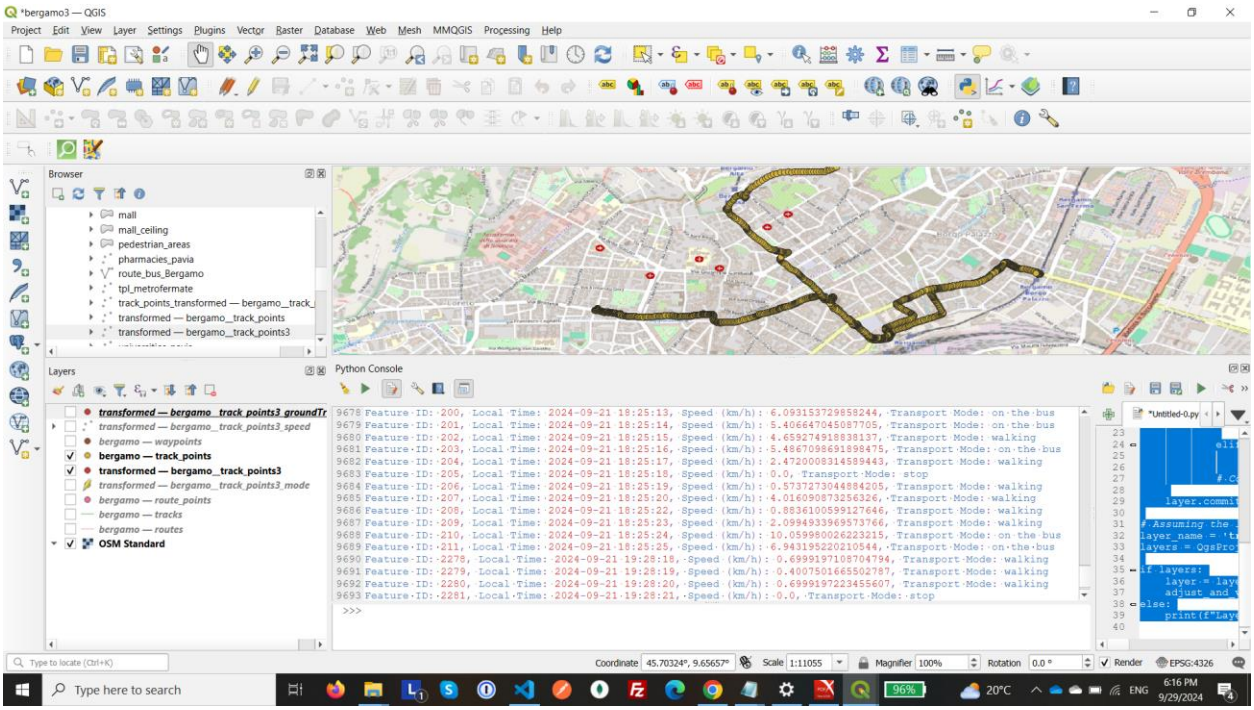
Python Code to Analyze and Log Transport Mode:

You can now verify if your Python code accurately assigns transport modes during these specific time periods. Here's an example of how you can print the data around your manually recorded times for verification.

```
from datetime import timedelta

# Adjust timestamps in your GPX file
def adjust_and_verify_transport_modes(layer, offset_hours=2):
    layer.startEditing()

    for feature in layer.getFeatures():
        timestamp = feature['timestamp']

        if timestamp is not None:
            # Adjust to local time zone
            local_time = timestamp.toPyDateTime() + timedelta(hours=offset_hours)

            # Log key transport moments based on your ground truth
            print(f"Feature ID: {feature.id()}, Local Time: {local_time}, Speed
(km/h): {feature['speed_kmh']}, Transport Mode: {feature['transport_mode']}")

            # Adjusting time range for comparison
            if local_time >= datetime(2024, 9, 21, 16, 25) and local_time <=
datetime(2024, 9, 21, 16, 40):
                print(f"Check if 'bus number 8' transport mode is correct.")

            elif local_time >= datetime(2024, 9, 21, 17, 6) and local_time <=
datetime(2024, 9, 21, 17, 11):
                print(f"Check if 'bus number 7' transport mode is correct.")

            elif local_time >= datetime(2024, 9, 21, 17, 11) and local_time <=
datetime(2024, 9, 21, 17, 14):
                print(f"Check if 'walking' mode is correct.")

            # Continue for other time ranges...

    layer.commitChanges()

# Assuming the layer name is 'track_points_transformed'
```

```
layer_name = 'track_points_transformed'
layers = QgsProject.instance().mapLayersByName(layer_name)

if layers:
    layer = layers[0]
    adjust_and_verify_transport_modes(layer)
else:
    print(f"Layer '{layer_name}' not found.")
```

With running this Python code, in the console we can see the logs as the result:



How This Helps:

- **Speed/Mode Validation:** This script logs the features' timestamps, speeds, and transport modes, and highlights critical time ranges based on your ground truth log.
- **Manual Comparison:** We can manually check if the **speeds and transport modes** (e.g., walking, on the bus, stop) in the data align with the actual activities at those times.

Expected Outcome:

- **At 16:25-16:40:** I should see that my **transport_mode** is set to "on the bus" and speeds are relatively high.
- **At 17:06-17:11:** The same applies for bus number 7.
- **At 17:11-17:14:** I should observe **walking** mode with a speed of approximately 3-5 km/h.
- **At 17:14-17:29:** Again, expect a return to "on the bus" with a higher speed.

Results:

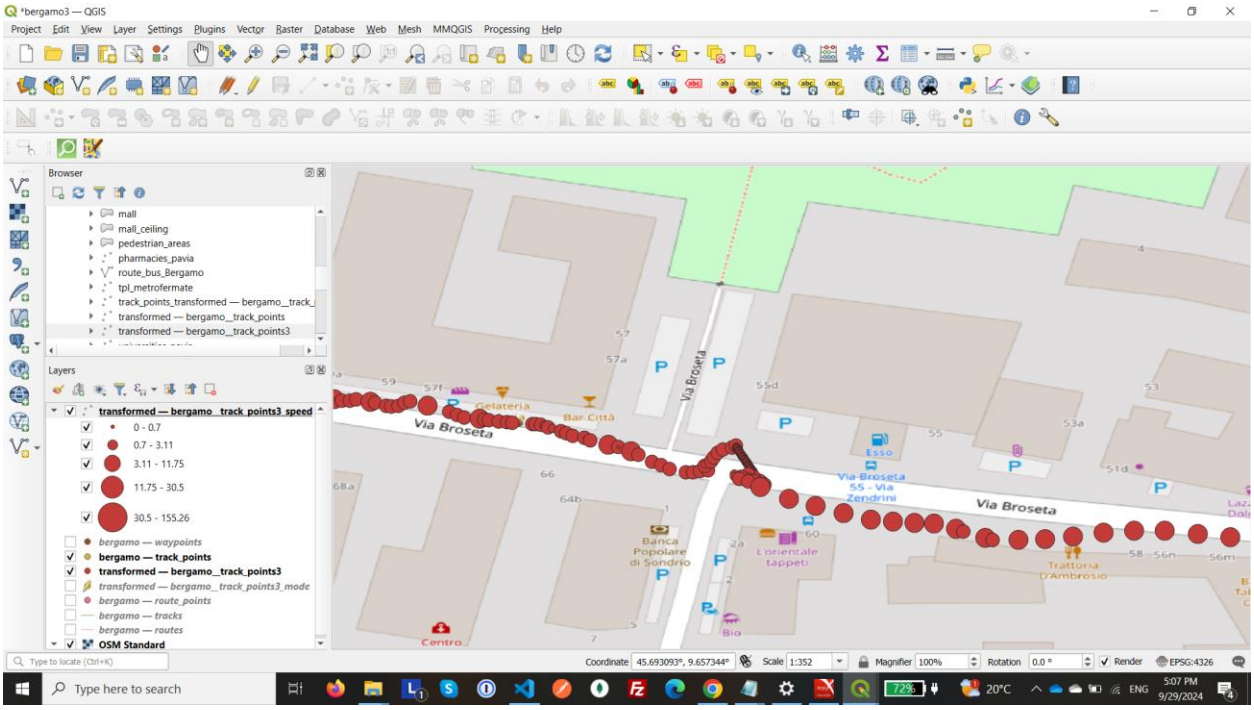
The processed data yielded reasonable results, with the transport modes being correctly categorized in most instances based on speed thresholds. Here are some of the key findings:

- The **speed** ranged from around 3 km/h (walking) to over 10 km/h (bus rides), aligning with expected values.
- The **transport modes** accurately reflected the transitions between walking and riding a bus when compared to the ground truth, although there was a time zone discrepancy that had to be accounted for.

Visualization and Analysis:

Add Symbology:

- We can symbolize the points based on speed or distance to visualize variations in our data.
- Right-click the layer, choose **Properties**, then go to the **Symbology** tab.
- For example, use a **Graduated** style based on the speed field to see how speed varies across different points.

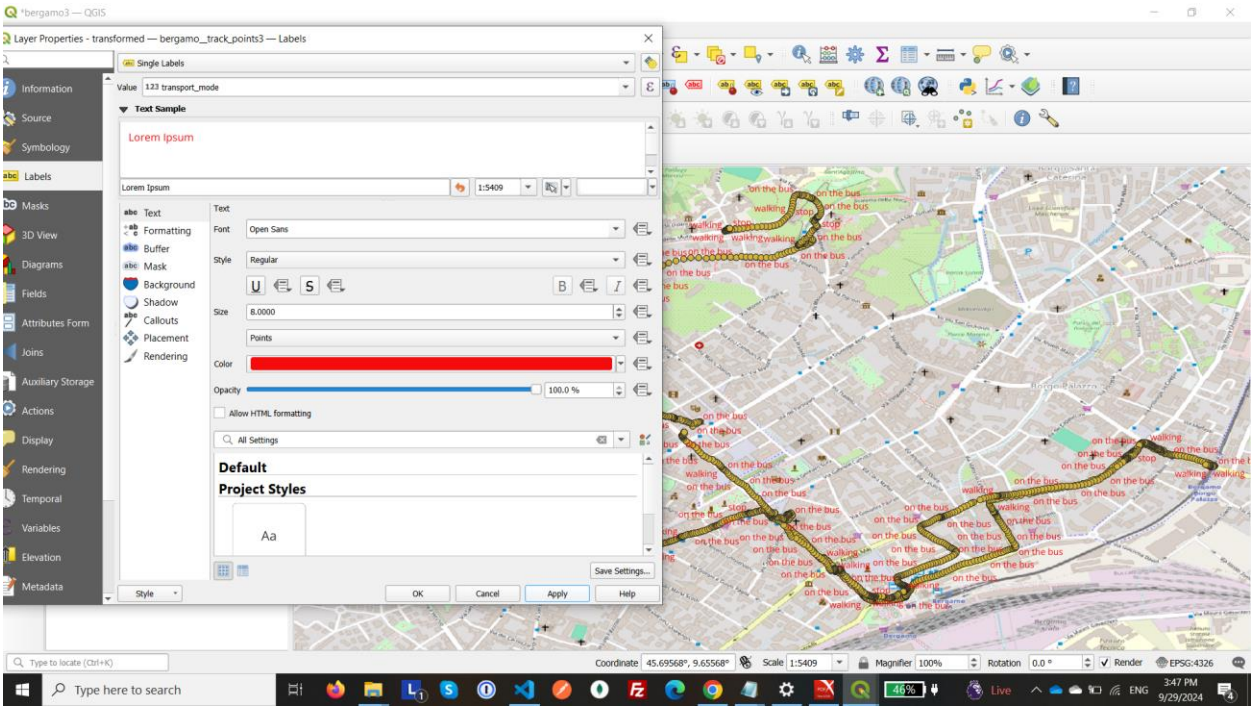


Style by Labels:

To make it more visual on the map:

- **Right-click your layer** → **Properties** → **Labels**.
- Choose **Single Labels** and select the field you created (`transport_mode`) to display the transportation mode on your map.

This will label each point in our trajectory with the mode of transportation, so we can visually see where we were walking, on the bus, or any unknown segments.



Challenges and Solutions:

- **Time Zone Mismatch**: A key issue that arose was the time discrepancy between the trajectory data and the ground truth logs. This was corrected by adjusting the time zone to ensure proper comparison.
- **Noise in Data**: Occasionally, the GPS data included points with errors (such as distances of 0 meters). These data points were filtered out to improve accuracy.
- **Zero or Negative Time Differences**: In certain instances, erroneous or null values were present in the time differences between consecutive points, which were addressed by skipping over these features.

Conclusion:

This project successfully demonstrated the process of collecting, analyzing, and classifying transportation modes using trajectory data. By leveraging QGIS and Python, I calculated speed variations in the movement data and classified transport modes with a good level of accuracy.

The comparison with manually recorded ground truth logs further validated the accuracy of the analysis. Future work could focus on refining the speed thresholds for transport modes or expanding the analysis to include additional modes such as cycling or driving.

This project showcases the value of spatial data analysis and its practical applications in understanding human mobility patterns.

References:

- QGIS Documentation
- Python API for QGIS
- GPX File Data Sources (Geo Tracker, Strava)