UNIVERSITÀ DI PISA

Master in Big Data Analytics and AI for Society

Andrea Vitali

Toward Data-Driven Mobility:

Harnessing Big Data for a Smarter Transport Network in Tuscany

Supervisors:
Dr. Leonardo Piccini (IRPET)
Prof. Mirco Nanni (UNIPI)

Part 1
Background and Main Discussion

1.1 Introduction

Big Data has been a buzzword since the early '90s, yet it is only in the last two decades that the concept has truly evolved into a major discipline. This surge of academic interest mirrors the **groundbreaking impact** that data is having on our societies and economies. Indeed, by analysing and manipulating Big Data we can unlock capabilities that were simply unimaginable just a decade ago, enabling businesses and public institutions to streamline their processes and operations.

While healthcare, finance, retail, and logistics have led this **Big Data Revolution**, it is increasingly evident that every economic sector stands to gain significant benefits by leveraging Big Data to enhance decision-making and drive innovation. Transportation is no exception. In fact, thanks to the pervasive integration of **location-based technologies**—from **GPS tracking** and **mobile phone records** to **social networks** with geotagged data—we now have a rich, **continuous** stream of information on how people and goods move. Additionally, recent **advances in spatial analytics** have opened entirely new avenues for translating these raw inputs into **actionable insights**, with the potential to transform infrastructure development and traffic management.

As we will see in Section 1.3, this dissertation harnesses a suite of spatial and mobility datasets—along with advanced analytical tools—to support more **evidence-based transport strategies**. This approach is applied to Tuscany as a case study, highlighting both **regional-scale patterns** and specific **local infrastructures**.

1.2 The Importance of Mobility

The importance of mobility planning in the modern world could not be overstated. Studies from around the globe have repeatedly confirmed that **well-connected areas** tend to enjoy **faster economic development and greater social mobility** [1,2,3,4,5]. Conversely, isolated rural areas often face significant barriers to accessing employment, healthcare, and education, leading to a self-reinforcing cycle of depopulation and increasing isolation.

Addressing social disparities should be a top priority for any government committed to a more inclusive and equitable development. Consequently, ensuring the efficient and timely movement of people and goods is more than just a logistical issue; it should instead be seen as a crucial step towards a **fairer society**. By bridging the gap between large urban centres and smaller rural communities, we can help guarantee that every citizen has access to comparable economic and social opportunities.

This relationship between mobility and socio-economic development is also apparent in **Tuscany**, as shown by Figures 1–4 [8], which overlay socio-economic indicators [6] onto the region's key railway stations [7], motorways (thick orange lines), and other important roads (thin orange lines). Municipalities with stronger transport links tend to have higher average incomes, a larger share of innovative companies, and a younger demographic. While correlation does not always prove causation, these observations underscore the **significance of this study**: enhancing Tuscany's transport network could boost the region's **socio-economic vitality** and improve its citizens lives.

Socio-economic indicators vs Transport Infrastructures

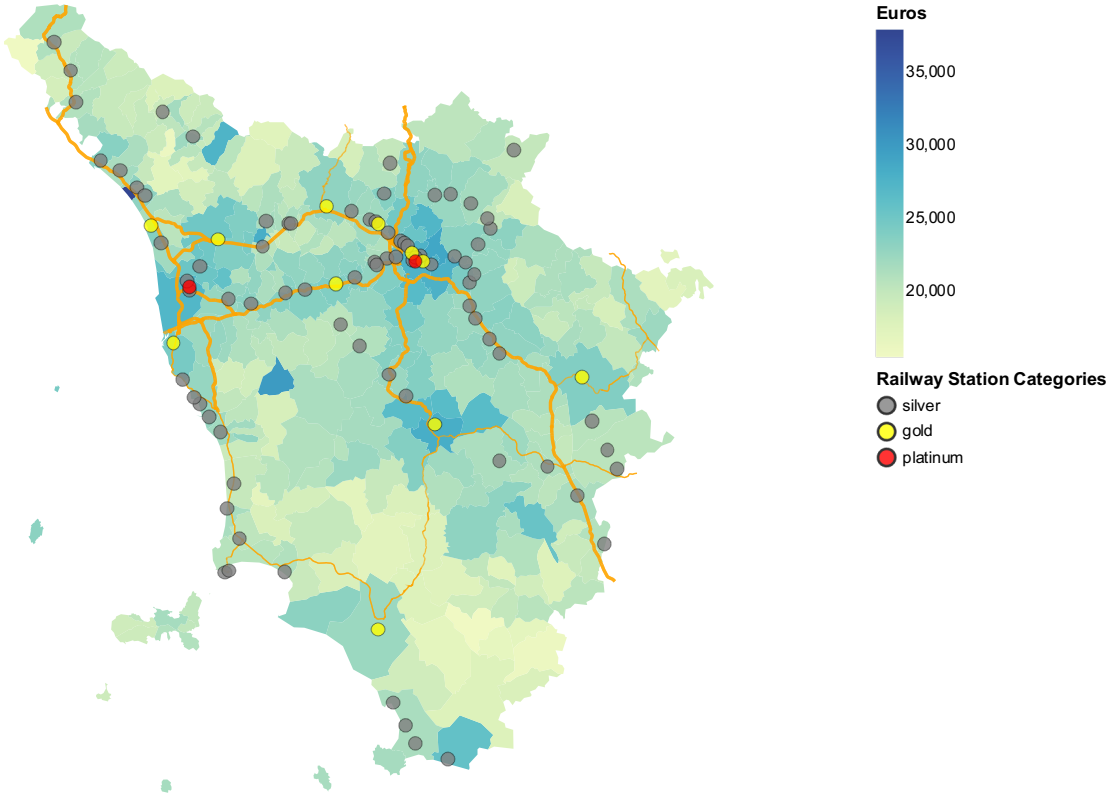


Figure 1: Average Taxable Income in Euros

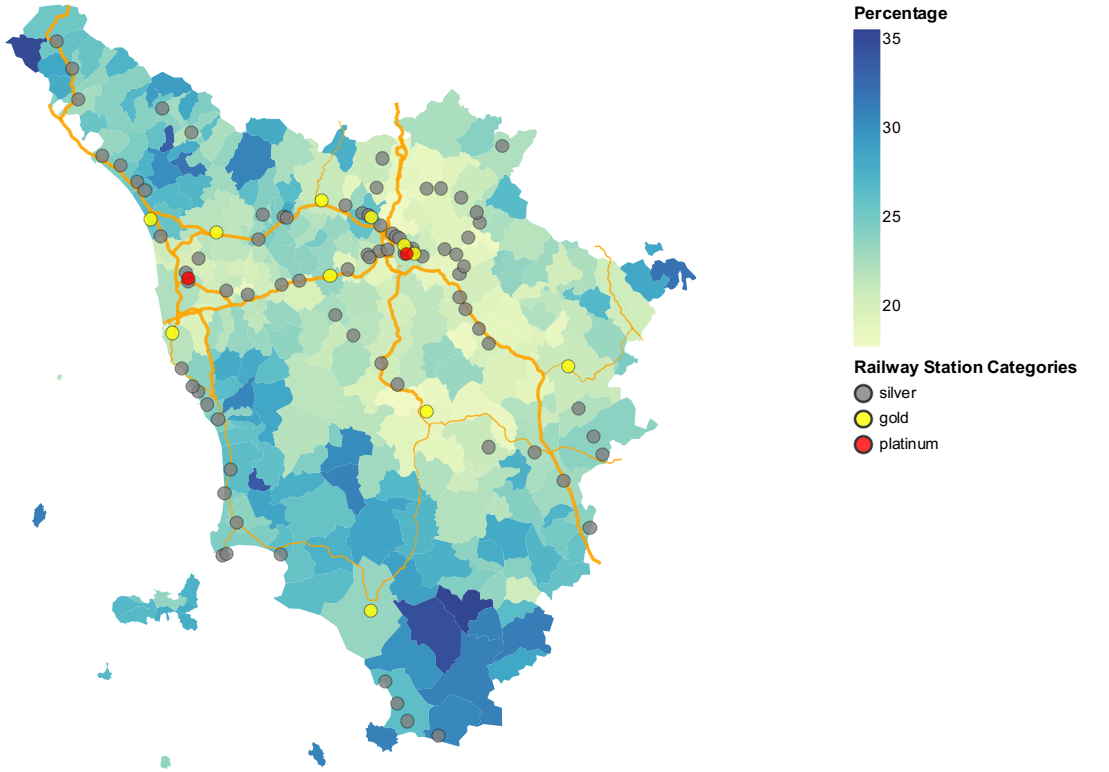


Figure 2: Percentage of Taxpayer with Income below 10000 Euros

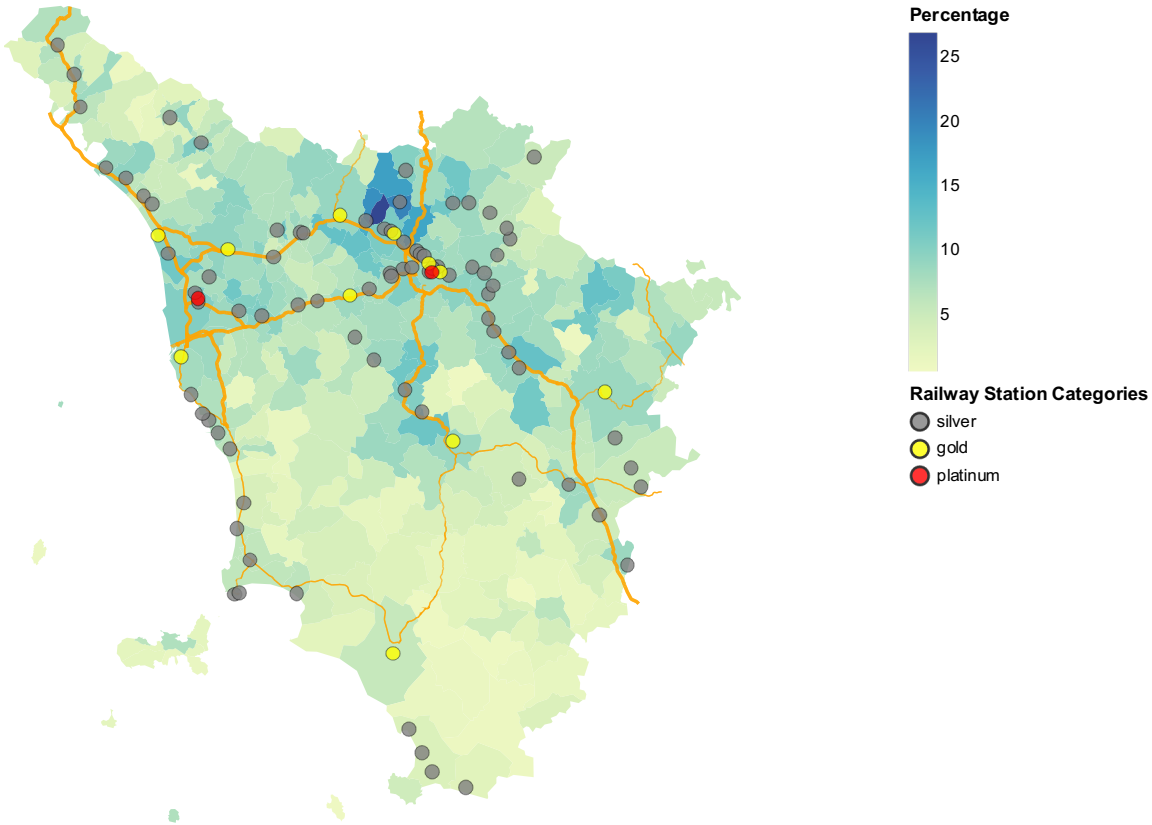


Figure 3: Percentage of Active Companies in Innovative Sectors

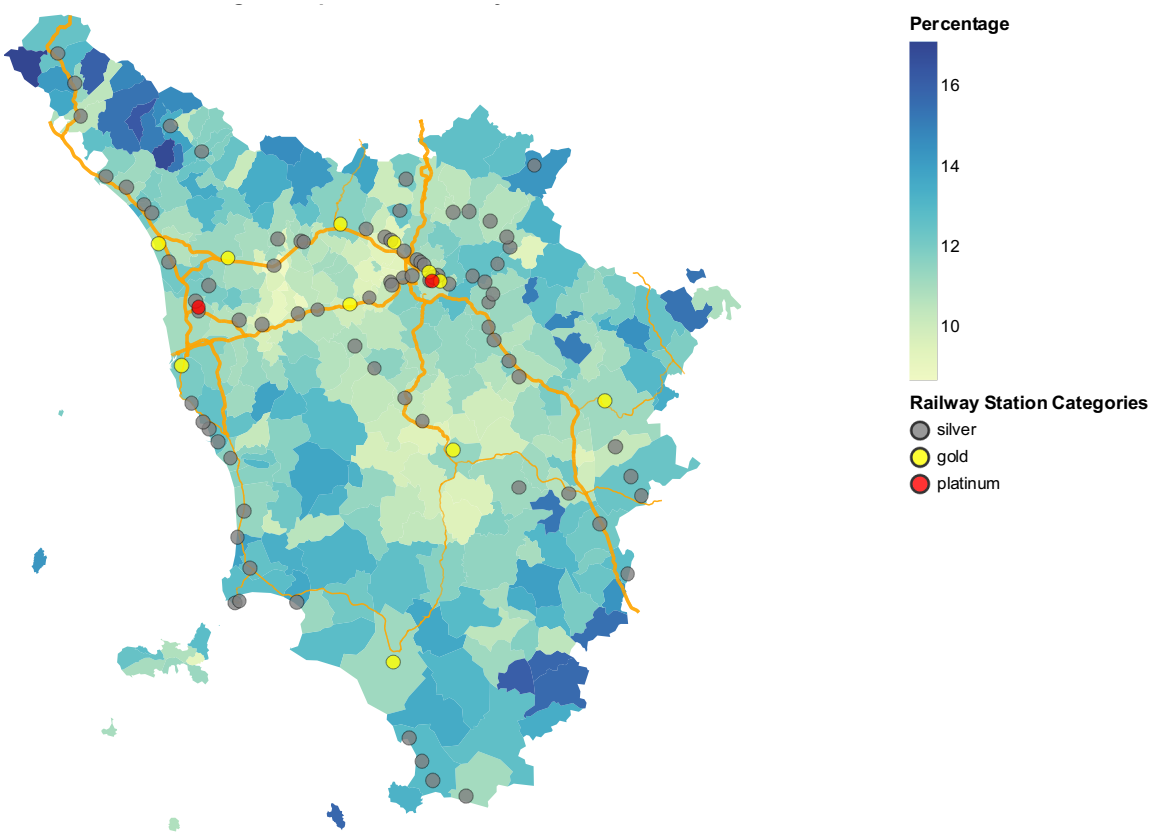


Figure 4: Percentage of Population over 80 Years Old

1.3 The Purpose and Data of the Project

We have seen how transport infrastructures can significantly benefit a region's economic vitality and social well-being. Yet, try saying that to a person stuck in a traffic jam, or a taxpayer whose hard-earned money has been spent on a costly, environmentally impactful, and chronically underutilised new motorway. You would likely be met with a look of disbelief, or perhaps even an angry reaction. After all, the true **value of infrastructures** is not measured by their mere existence, but in how effectively they **serve their users**.

Ill-planned or poorly managed infrastructure can leave communities feeling frustrated and deceived. Often, this mismanagement stems from a lack of powerful, data-driven tools for making informed decisions during both planning and operational phases. Thankfully, the era of blind planning may soon be over, thanks to the **transformative power of Big Data and machine learning**.

Ultimately, our goal is to ensure that infrastructures serve communities effectively and sustainably. Access to **large geodatasets** and other spatial information can help in this task by allowing us to analyse infrastructure performance, identify patterns, and predict future demand with unprecedented accuracy. These **cutting-edge tools and technologies** can provide insights once unimaginable, enabling us to build, and manage, smart transport systems that truly meet the needs of their users and, at the same time, give taxpayers better value for their money.

Recognising this potential, the project outlined in this dissertation aims to showcase a range of data-driven instruments, with **two main objectives**:

1. Illustrate how spatial and mobility data can provide actionable insights for **regional-level transport planning** that policymakers and other stakeholders in Tuscany can employ to create a more efficient transport system.

2. Apply advanced mobility analytics to a critical transport link of the Tuscan network, **the FI-PI-LI motorway**, to identify current challenges and potential improvements.

To support these objectives that capture both macro-scale trends and fine-grained details of mobility across the region, the **primary datasets** used for this project are the following:

- **Mobile Phone Records of Tuscan municipalities.** Mobility patterns derived from anonymised mobile phone records. They provide insights into large-scale population movement and travel behaviours across the region.

- **GPS-Based Vehicle Dataset of Northern Tuscany**. Records collected from GPS signals of “black boxes” installed in vehicles for insurance purposes. They offer high-resolution on how roads are used through the day with vehicle-specific data, including detailed trajectories. They allow an in-depth analysis of road usage and traffic dynamics.

Data Privacy and Ethical Considerations

While Big Data offers unprecedented insights into human mobility, they also raise important ethical questions, particularly around **privacy**. Using mobile phone records or GPS signals for large-scale analysis means collecting **sensitive location details** about real individuals.

The personal data used in this project have been anonymised in compliance with the General Data Protection Regulation (GDPR). By using sequential IDs, we can reduce the likelihood that individuals can be re-identified. Nevertheless, even hashed data can be vulnerable to **re-identification** if combined with external datasets. Furthermore, prolonged GPS data collection can reveal **habitual patterns**, including users’ homes, workplaces, and other frequently visited locations. Techniques like **spatial or k-anonymisation** of trajectories can reduce the risk of disclosing personally identifying information. Future expansions of this project might explore **advanced anonymisation frameworks**, thus balancing robust mobility insights with **stronger safeguards** for citizen privacy.

1.4 A Region-wide Perspective

Transport infrastructures can become inefficient and even counterproductive when planned and managed in isolation, focusing only on local needs without considering the **bigger picture**. This narrow vision could in fact lead to wasted resources on low-priority projects that fail to align with wider mobility needs. Worse, overlooking the flows of people and goods at regional and national levels could exacerbate issues, creating bottlenecks and congestion instead of alleviating them.

A **broader perspective** is therefore essential for tackling these challenges effectively. Thanks to the advancements afforded by the Big Data era, data scientists now have an array of instruments to help policymakers to make smarter, evidence-based decisions. Among these tools, **visualisations** can play a pivotal role in providing a **comprehensive and immediate understanding** of movement dynamics, empowering transport planners to make informed choices.

Mapping Large-Scale Movement

An important initial step is to assess large-scale movement patterns within the region of interest to identify mobility hotspots and areas needing more investments. This is crucial for our **first objective** of providing actionable regional-level insights. The two maps below (Figures 5 and 6) show the total average **daily movements in and out for each municipality in Tuscany** [9,10] and highlight that the macro corridor stretching between Florence, Prato, Lucca, Pisa, and Livorno is a focal point for regional travel.

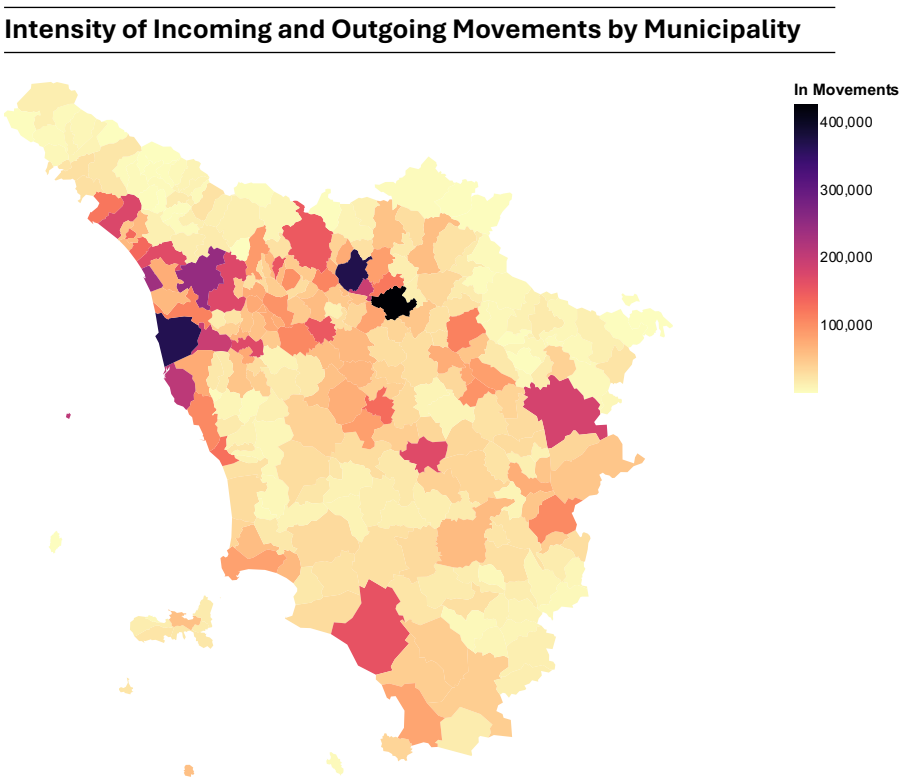


Figure 5: Intensity of Incoming Movements by Municipalities

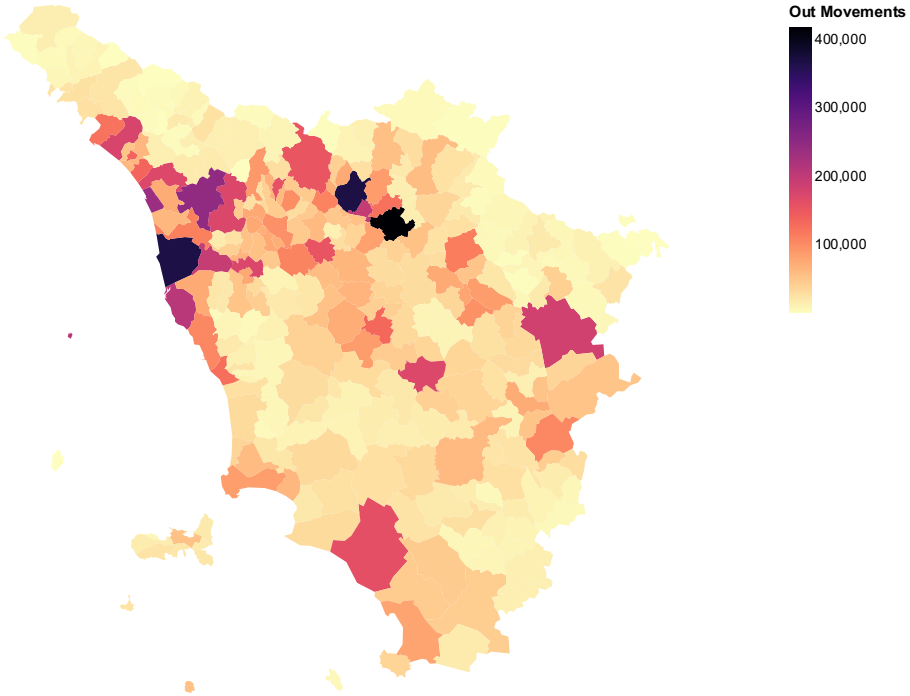


Figure 6: Intensity of Outcoming Movements by Municipality

While visualising in-and-out movements is valuable, the analysis becomes far more revealing when we trace **where these movements originate from** and **where are headed**. Here, **flow maps** become indispensable: they offer a bird's-eye view of **inter-municipal connections**, exposing the complex network of movement dynamics. Figures 7 and 8 —shown below— highlight such flows across Tuscany, filtered to include a minimum daily average of 3000 and 15000 movements respectively [11,12,13]. Thanks to them, it becomes immediately evident where the main regional transport corridors lie, and which municipalities exchange the greatest volumes of traffic.

Average Daily Movement Flows

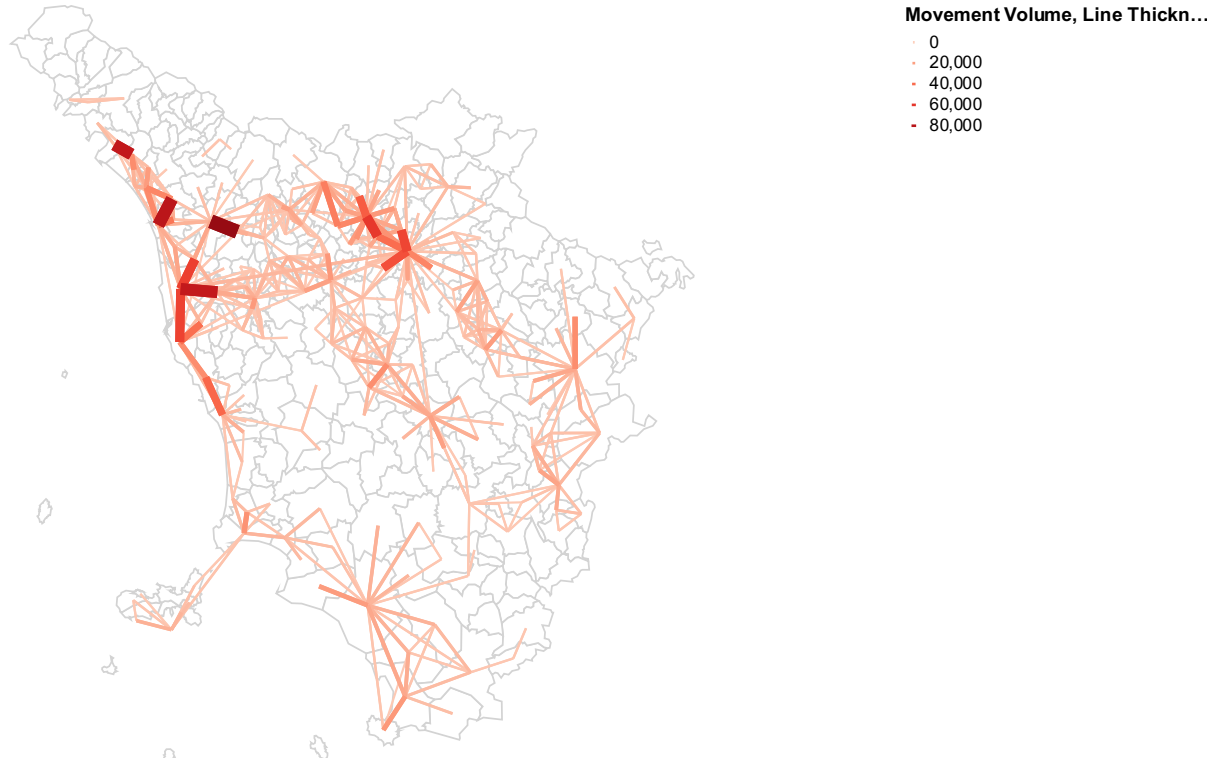


Figure 7: Average Daily Movement Flows between Municipalities (Filtered at 3000)

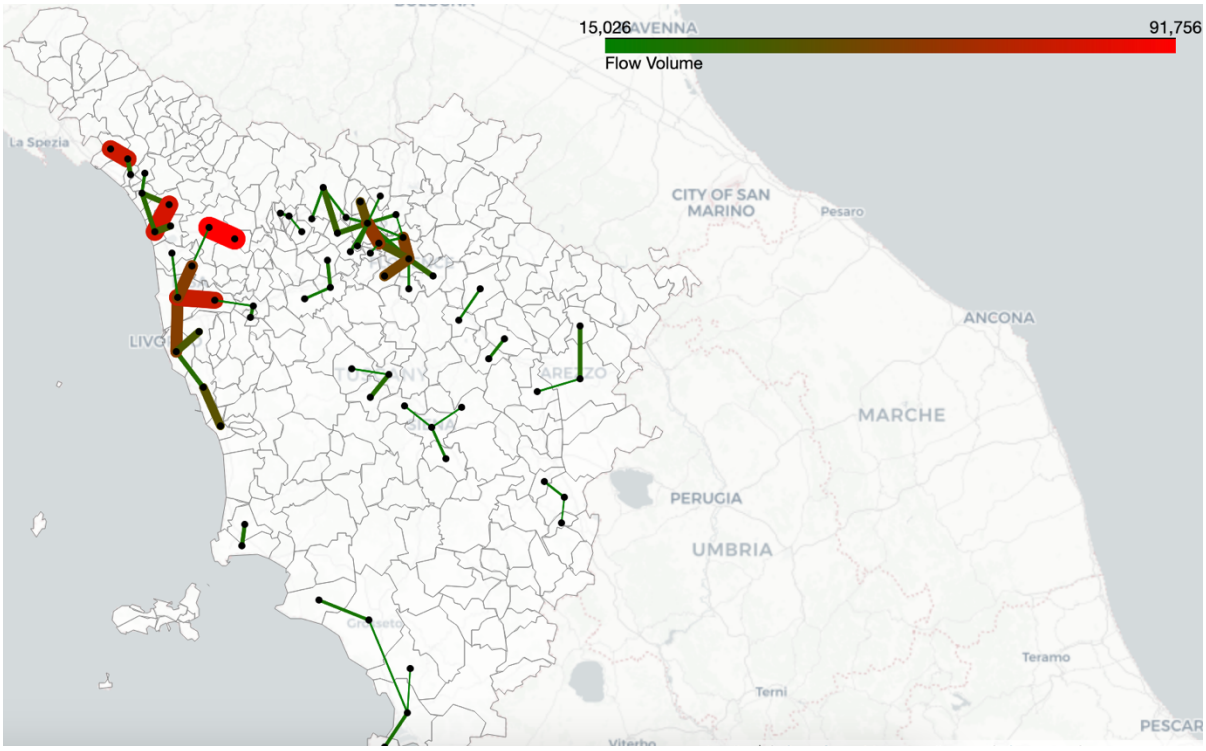


Figure 8: Average Daily Movement Flows between Municipalities (Filtered at 15000)

Policymakers can delve even deeper into transportation dynamics for a specific municipality by employing **interactive maps**, **bar charts**, **Chord diagrams**, and **Sankey diagrams**. These tools go beyond static data representation, enabling planners to analyse evolving patterns much more **quickly than traditional tabular data** ever could.

For example, **interactive maps** integrated dynamically with **bar charts** (Figure 9) [14] can pinpoint areas with high outflows, thereby facilitating the recognition of hotspots and corridors that require attention, all in a straightforward and easily interpretable format. **Sankey diagrams** (Figure10) [15] and **Chord diagrams** (Figure 11) [16] can add another layer of insight by illustrating the relationships and interactions between multiple municipalities. They provide a **clear and immediate visual representation** of the magnitude and direction of movements, highlighting the most significant flows as well as potential bottlenecks.

Volume of Average Outgoing Movements by Destination

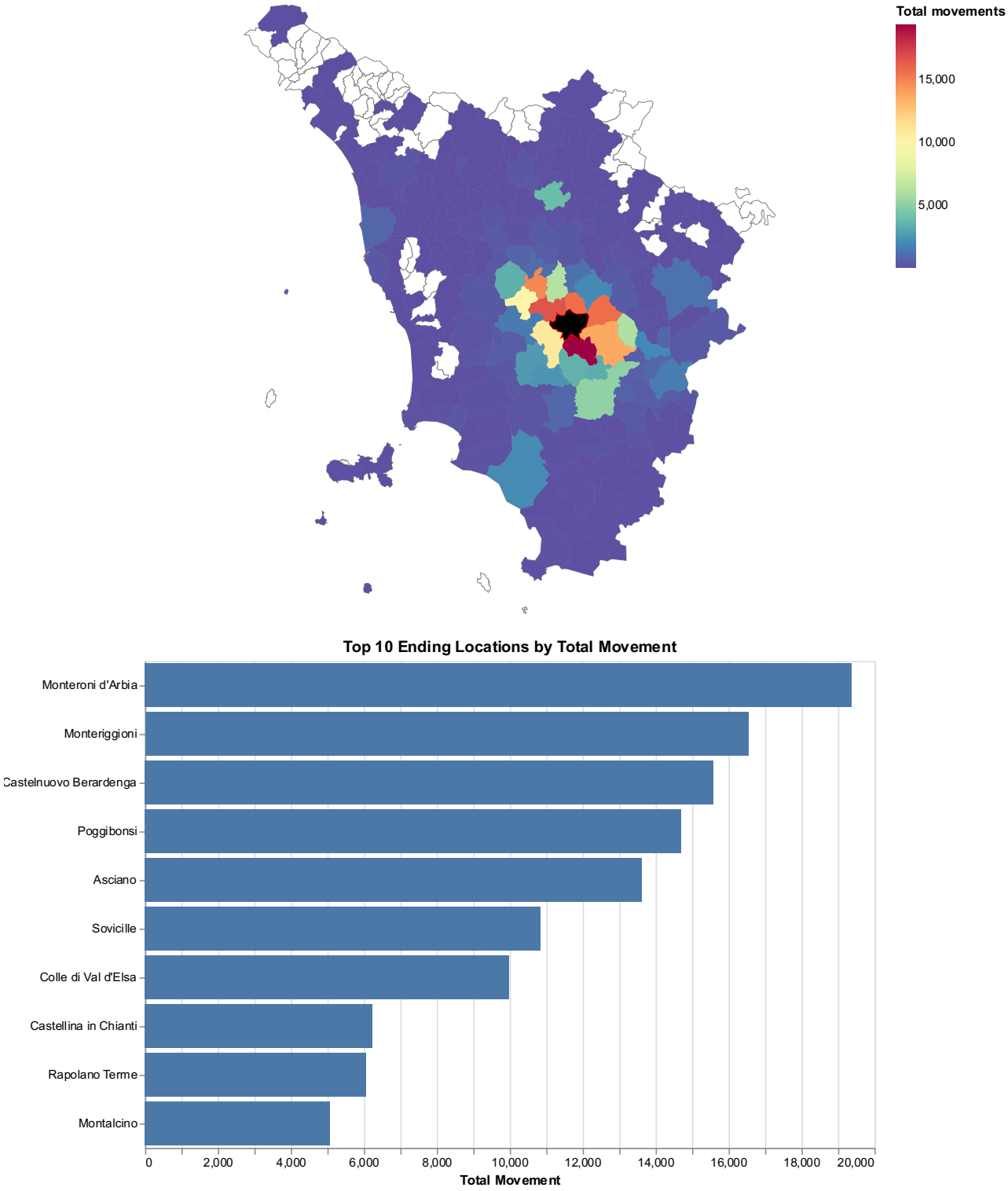


Figure 9: Total Outgoing Municipality Movements by Destination

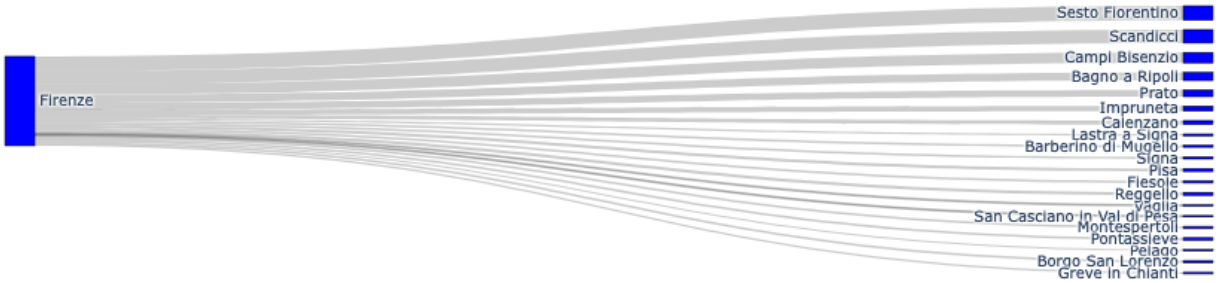


Figure 10: Average Daily Outgoing Movements from Florence (>3000)

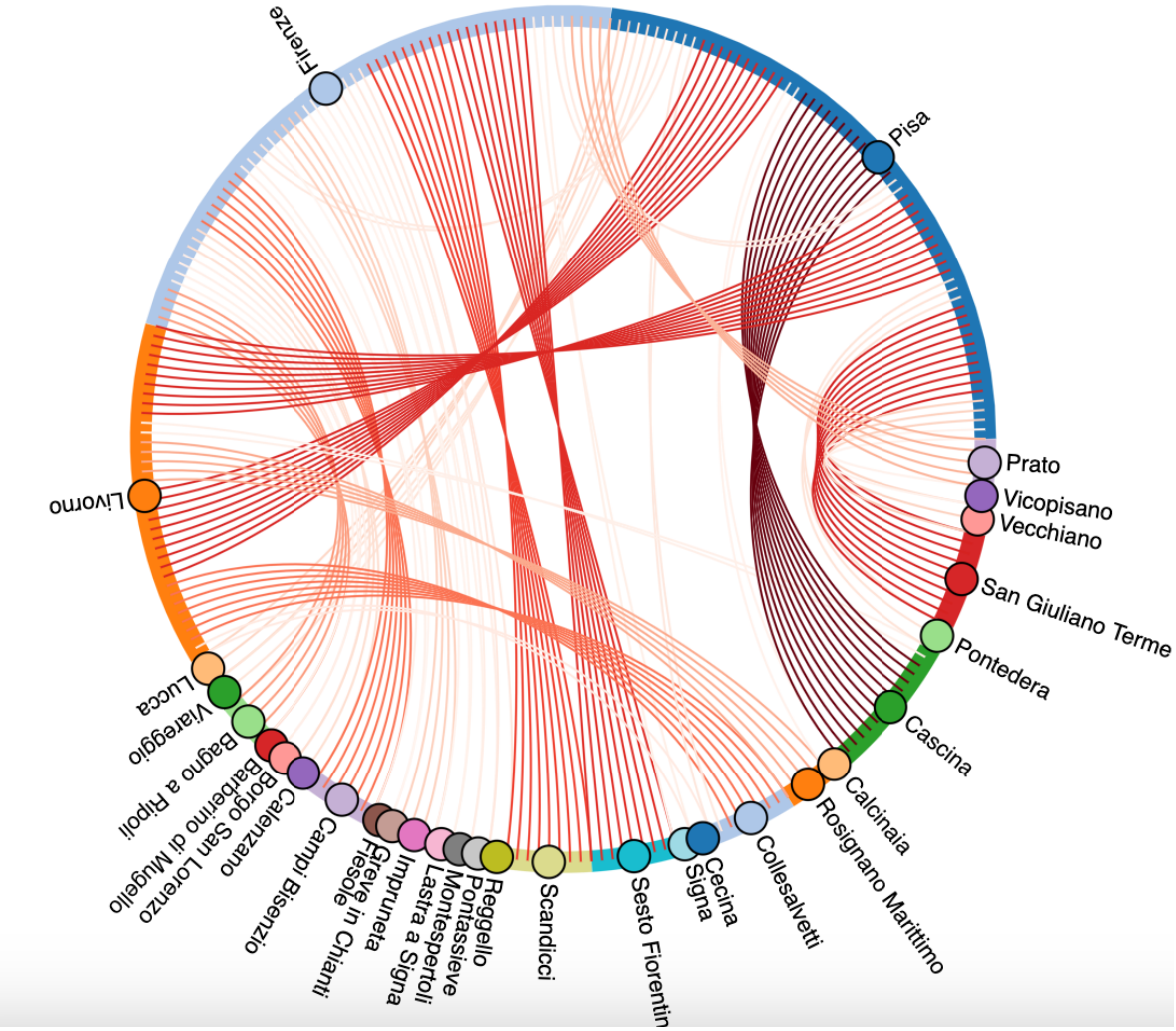


Figure 11: Average Daily Outgoing Movements from Firenze, Pisa, and Livorno (>5000)

Understanding Modes of Transport

Knowing **how people move** is also crucial for shaping transportation policies, optimising infrastructure investments, and promoting sustainable mobility. Analysing modes of transport helps policymakers address capacity issues and identify alternative routes during disruptions caused by accidents or planned closures.

As highlighted earlier, **charts and maps offer a much faster interpretation of movement dynamics than traditional tables**. For example, the bar charts below (Figures 12 and 13) show how people travel at a regional level and in the 10 busiest municipalities [17,18].

Movements by Modes of Transport

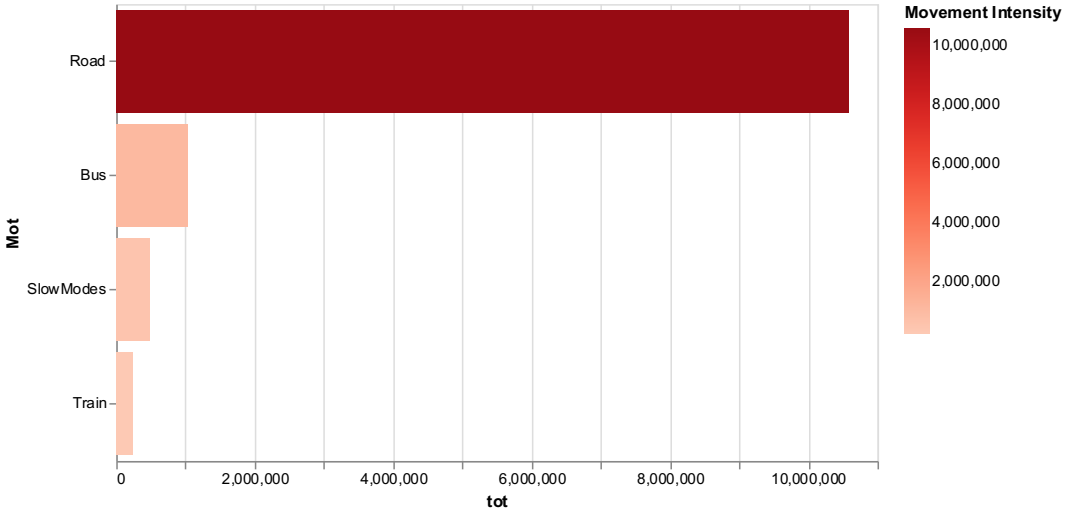


Figure 11: Volumes of Movements by Means of Transport

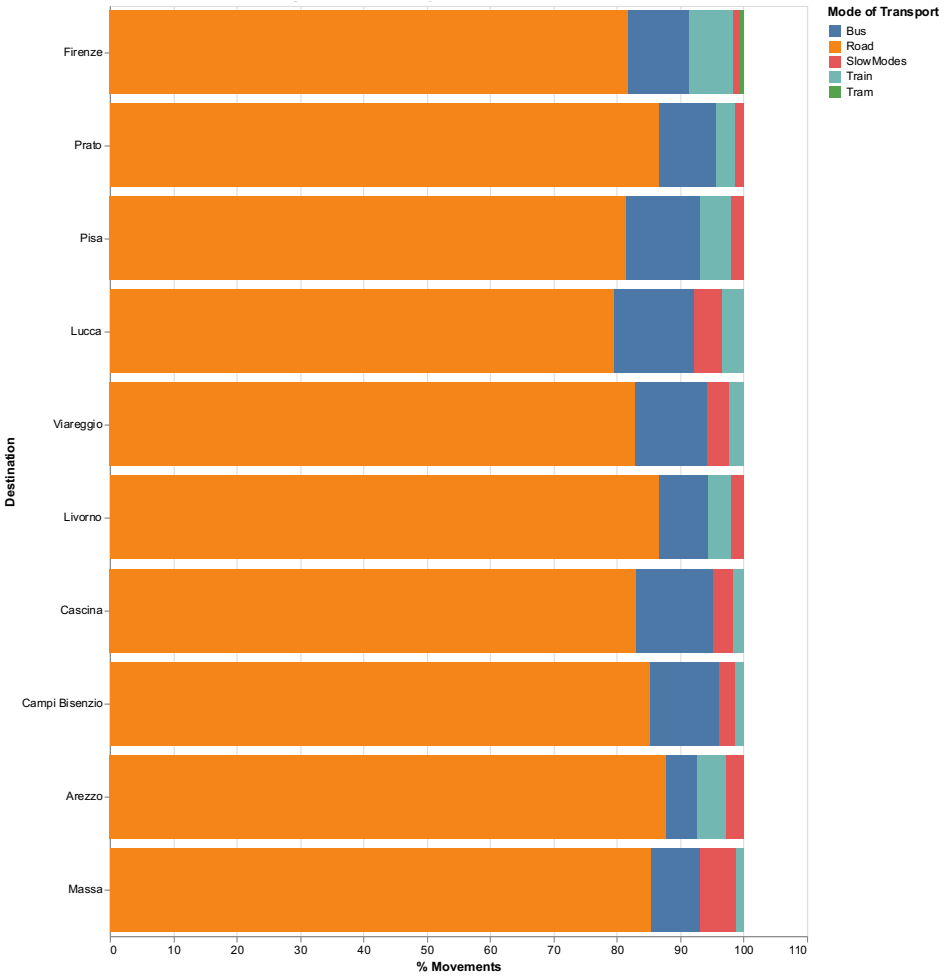
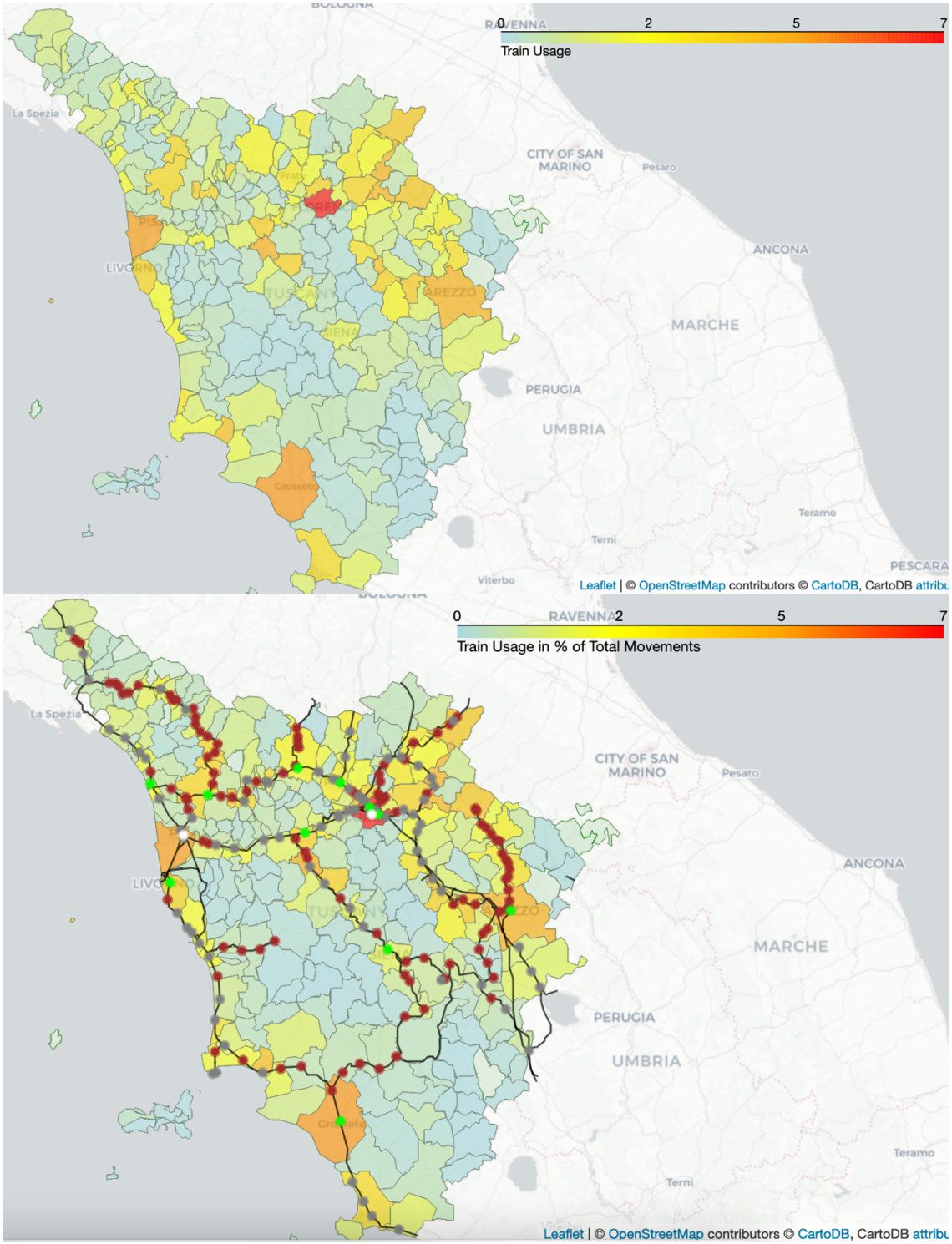


Figure 12: Percentage of Movements by Mode of Transport for the Top 10 Destinations

Monitoring Train Usage

Focusing on a **single transport mode** can reveal underperforming infrastructure and show where policy interventions might be most beneficial. The **choropleth maps** in Figures 14 and 15 depict the **percentage of train usage** across Tuscan municipalities, overlaid with **railway lines and stations** [19,20]. Notably, the **Pisa–La Spezia** line seems to be **underused** given its robust station coverage and frequent service. On the other hand, the **Arezzo–Stia** line sees **high demand** even though its stations are classified as offering minimal services—suggesting a potential **over-usage** scenario. Recognising these patterns can help planners allocate resources more effectively, whether by promoting underused lines through targeted campaigns, improving station multimodal accessibility, or upgrading rail tracks to offer more reliable, frequent service.

Percentage of Train Usage



Figures 13-15: Percentage of Train Usage by Municipality with Railways Tracks and Stations (classified by importance)

A Broader and Holistic View

In conclusion, **combining the visualisation tools** discussed in this section gives administrators **a comprehensive and multifaceted grasp of movement dynamics**. This broader and holistic view opens new opportunities for more effective, data-driven

planning of transportation infrastructure, ensuring that **local needs align with regional-wide objectives**. Additionally, the flexible nature of these tools means that they can be continuously refined with new data once in operation, and swiftly adapted to evolving needs and demands.

1.5 A More Detailed Perspective: The FI-PI-LI Motorway.

While a region-wide outlook is essential to get a general understanding of movement dynamics, zooming in on a **particular infrastructure** or even **specific sections** of it allows for a more **detailed examination** of local issues and opportunities. This granular focus helps to identify usage patterns, inefficiencies, and potential alternatives in case of planned or unplanned disruptions.

Among the infrastructures known for being the most problematic for mobility in Tuscany, the no-toll **Firenze-Pisa-Livorno motorway** (Figure 16), also referred to as FI-PI-LI, is perhaps the most notorious. Built in the 1960s to connect key Tuscan cities, the FI-PI-LI motorway has faced **persistent challenges**: excessive traffic volumes, narrow lanes, and limited safety features, which often result in severe congestion and serious accidents that have ripple effects across the regional traffic network [21-22].

Over the years, numerous proposals for infrastructural improvements and enhanced traffic management have been put forward, but to no avail, and the FI-PI-LI remains one of the most controversial points of discussion in Tuscan mobility policies.

Figure 16: The Firenze-Pisa-Livorno Motorway

The Lastra a Signa–Florence Segment: A Key Bottleneck
Among the various sections of the FIPILI, the stretch between **Lastra a Signa and Florence** (Figure 17) is particularly problematic. Known as one of the **busiest and most accident-prone** parts of the motorway, it serves as a critical route for both commuters and freight traffic—connecting the FI-PI-LI to the A1 MILANO-NAPOLI motorway, a major arterial road in Italy [23-24].

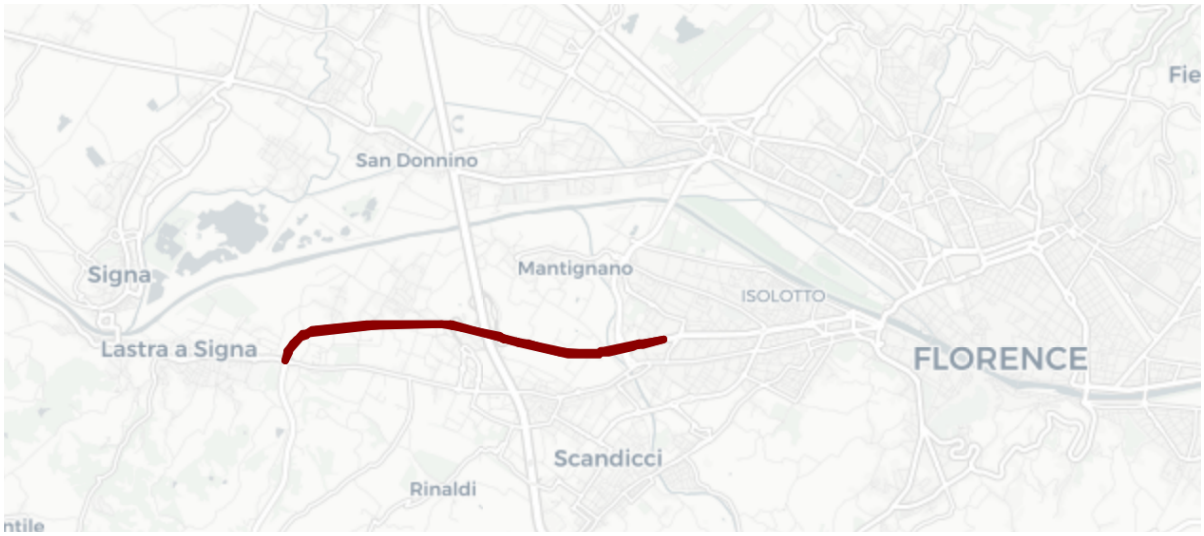


Figure 17: The Firenze-Lastra a Signa Segment

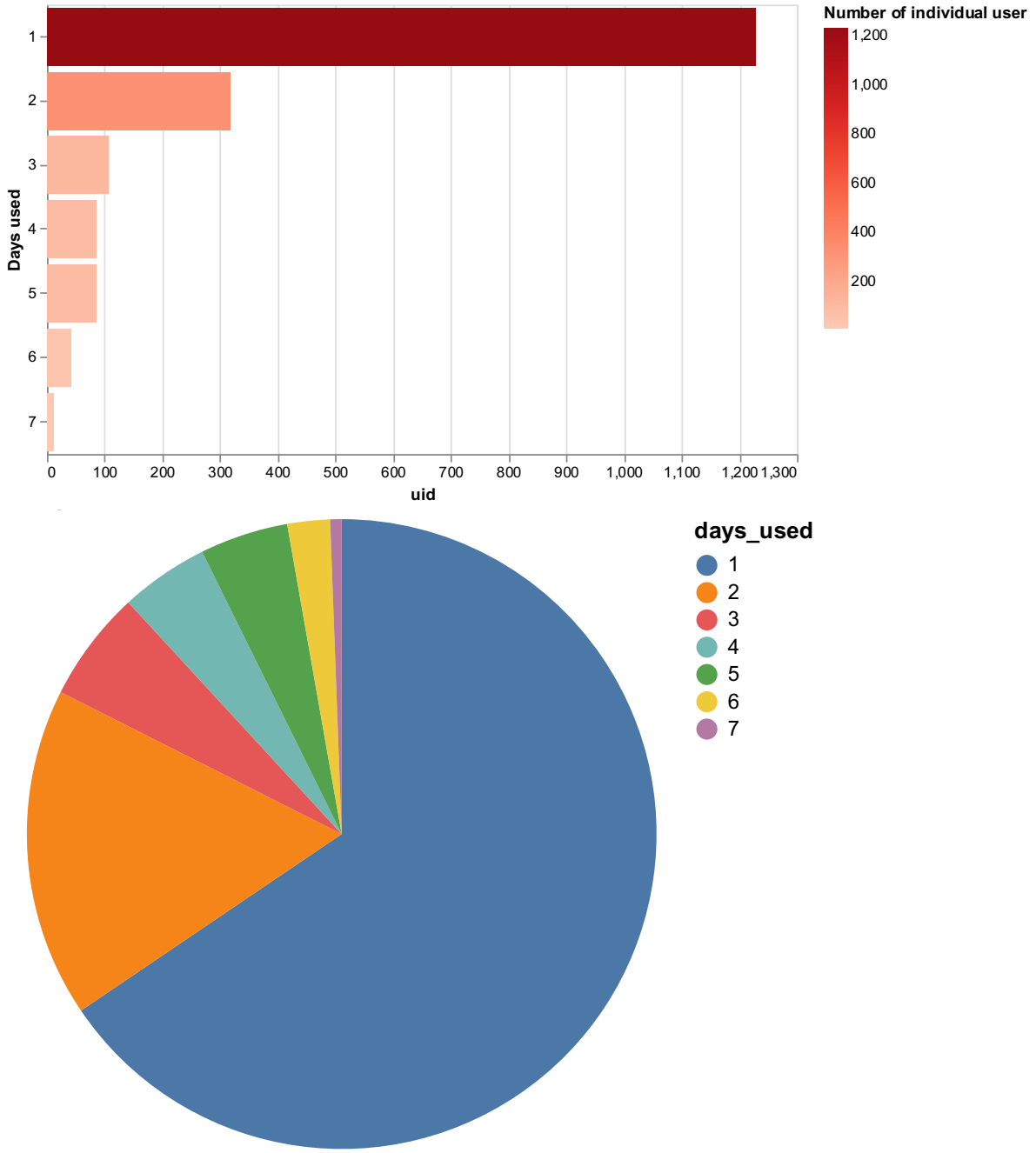
To develop **long-lasting solutions** for this problematic road segment, policymakers and stakeholders should begin by gaining an **in-depth understanding of its users and their behaviours**. By examining commuting patterns, traffic flows, and decision-making factors, they can uncover the root causes of congestion and inefficiencies, thereby paving the way for more targeted and durable improvements.

User Analysis: Who's Driving and When?

To start, it is crucial to assess the **frequency of use** among different types of users over an average week, distinguishing between **occasional ones, regular drivers, and commuters.**

The charts below (Figures 17-20) [25-28] reveal two notable patterns:

- **Most users travel on this segment only once a week**, typically on weekends.
- Those who use the segment **five times a week primarily do so on weekdays**



Figures 17-18: Frequency of use by Individual Drivers of the Section Firenze-Lastra a Signa (2nd Week February 2019)

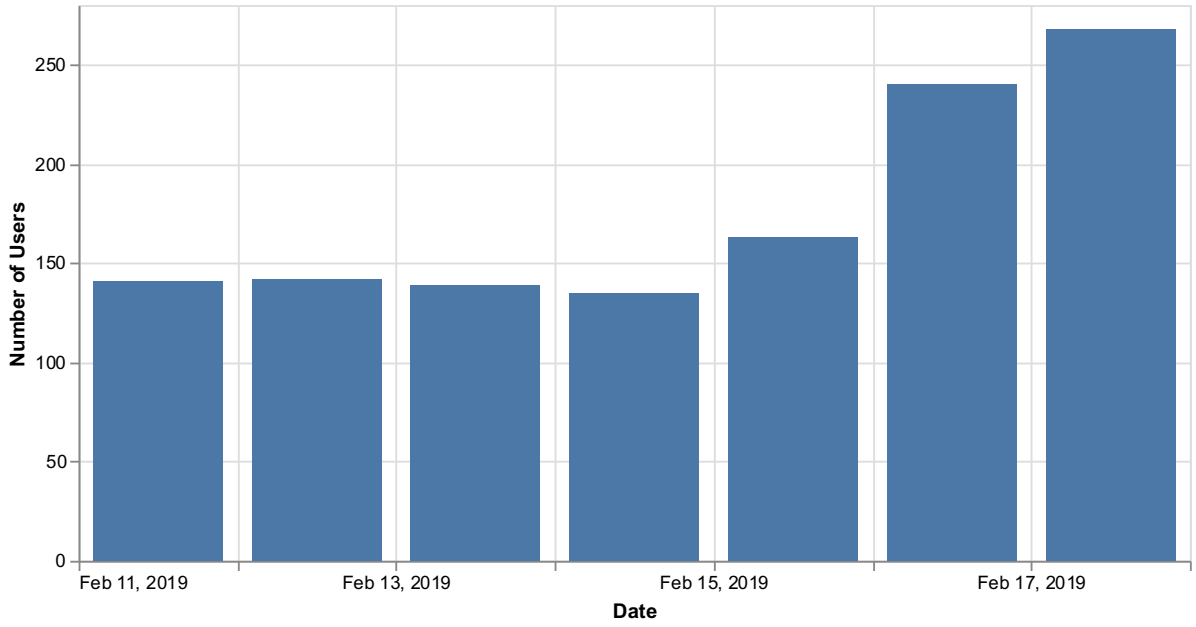


Figure 19: Distribution of Days for Drivers that Use the Firenze-Lastra a Signa Once a Week

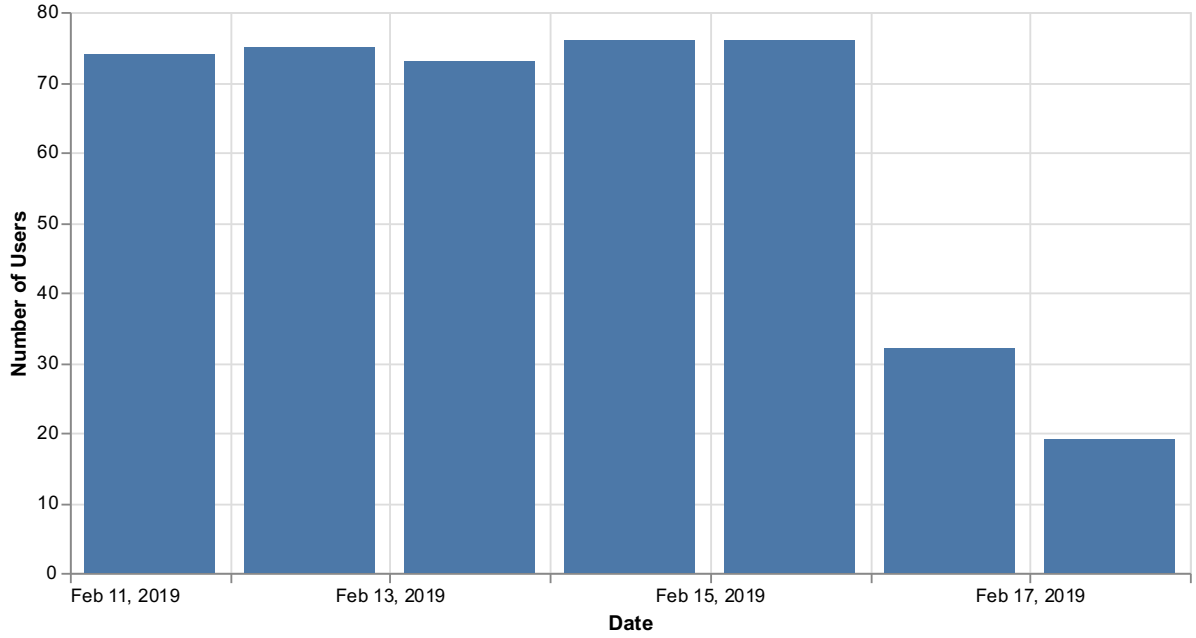


Figure 20: Distribution of Days for Drivers that Use the Firenze-Lastra a Signa 5 Times a Week

Commuter Behaviour: What Data Tell Us

In mobility studies, commuters play **a pivotal role** given their significant impact on peak-hour congestion. Therefore, let us focus on this specific group of drivers to explore the inferences that can be drawn from analysing their behaviour in greater depth.

By examining **commuters' individual behaviours** thanks to the charts below (Figures 21- 24), we can draw a few general insights.

- **Rush Hours Dominate**. Weekday early mornings and late afternoons see the highest traffic volumes, as commuters travel to and from work.
- **Short Distances**. Most commuters tend to travel relatively small distances, suggesting a car use within a confined area. These might be individuals living and working in close proximity.
- **Predictable Mobility**. Commuters are likely to travel between a small set of familiar locations (e.g., home, work, or nearby services) thus showing moderately predictable mobility patterns.

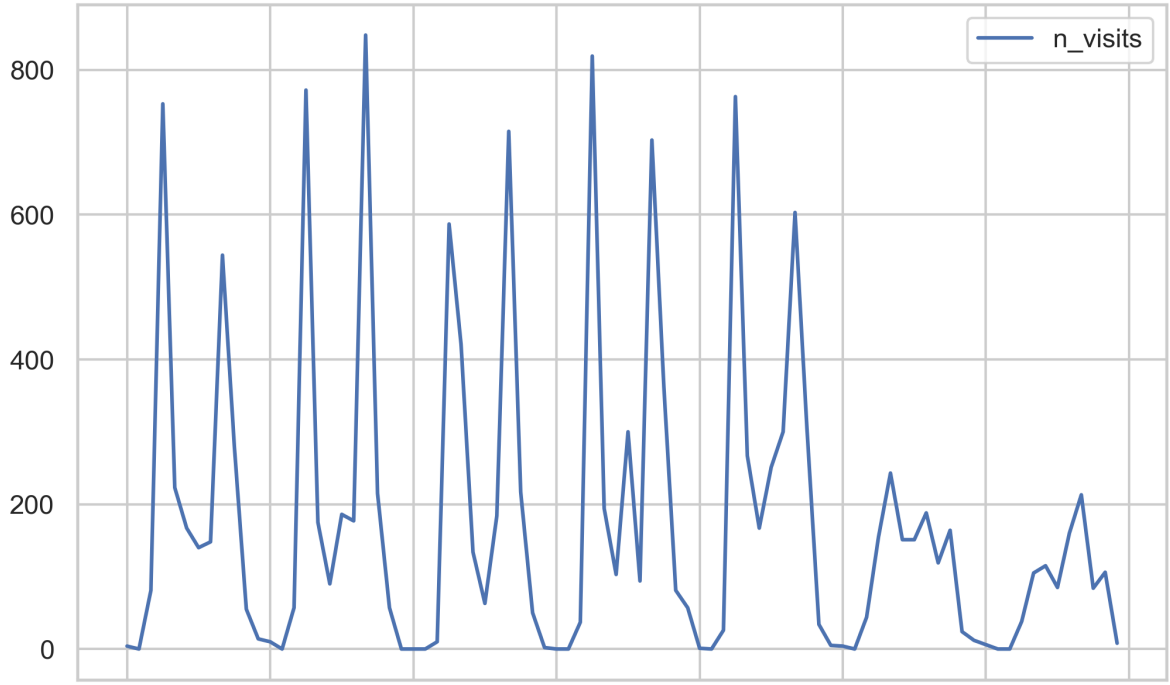


Figure 21: Commuters' Travel Behaviour over Time (Monday- Sunday in 1st Week of February)

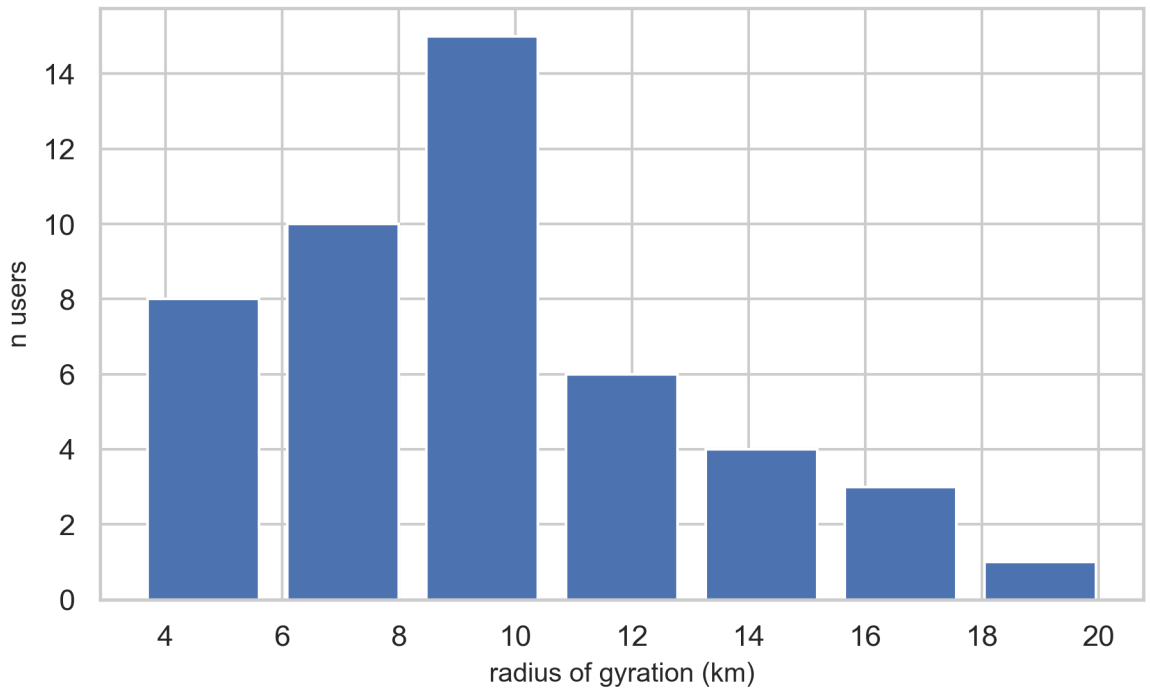


Figure 22: Characteristic Average Distance Travelled by Commuters

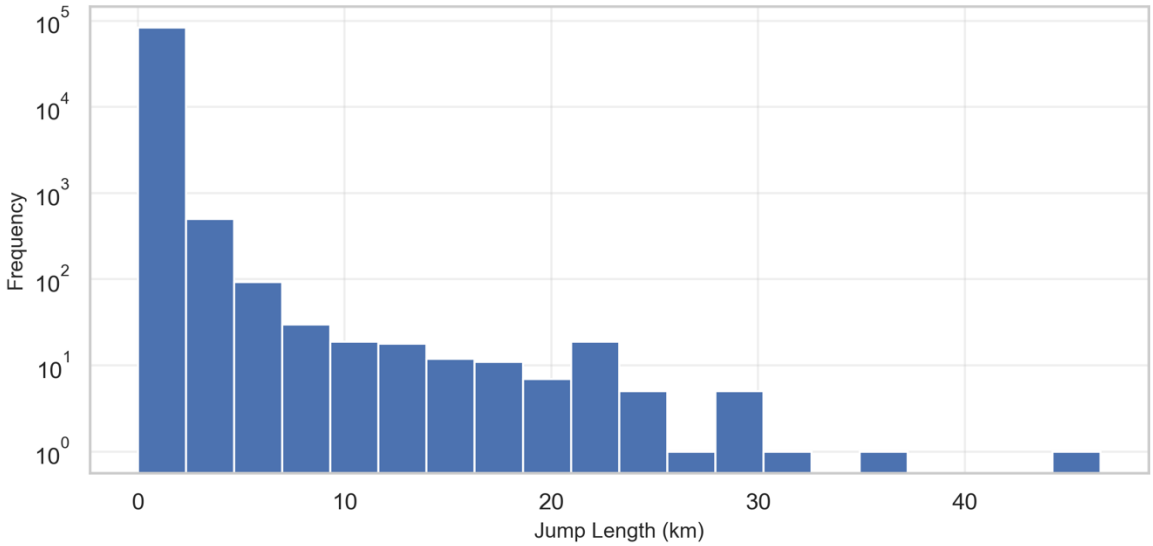


Figure 23: Distribution of Jump Lengths Among Commuters

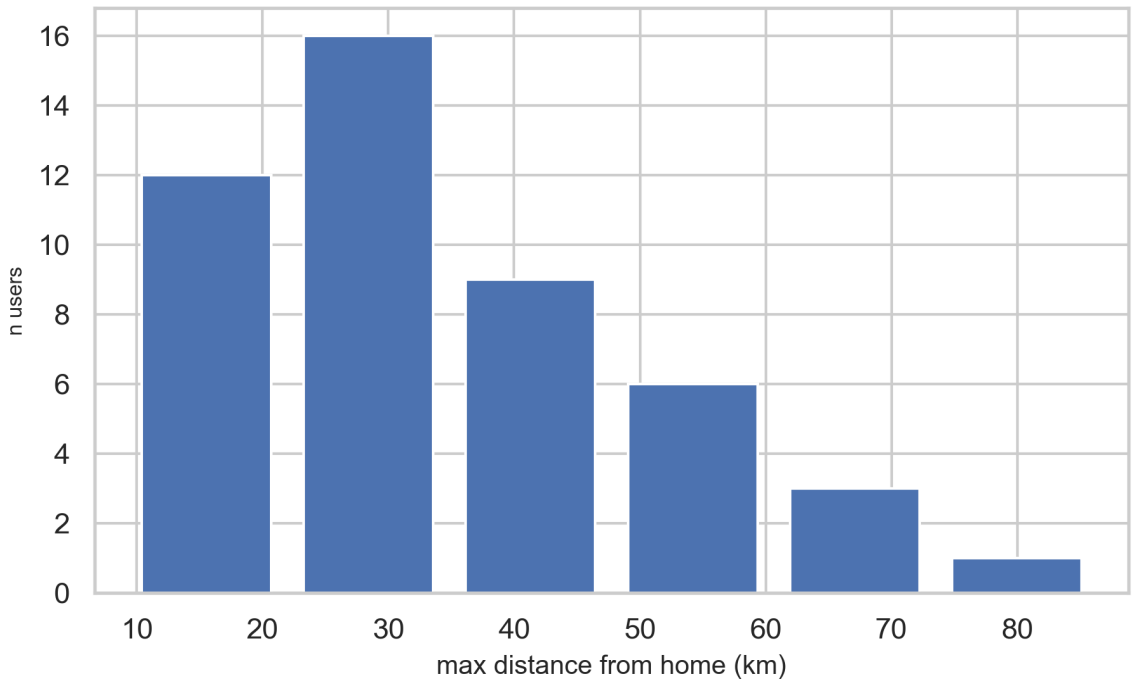


Figure 24: Maximum Distance Travelled from Home by Commuters

Because commuters often follow **short, predictable travel patterns**, they are an ideal target for mobility planners and administrators who could recommend the following:

- **Enhancing Public Transport**. Improving bus and rail connections along commuter routes could reduce reliance on private vehicles.
- **Introducing Congestion Charges**. Introducing fees during peak hours may encourage drivers to seek an alternative mode of transport.
- **Adopting flexible Work Schedules**. Encouraging businesses to adopt variable or hybrid work hours could help spread out traffic beyond peak times.

Mapping Stops and Homes

By studying commuters' trajectories over the course of a month, we can identify **their stops** (Figure 25) [29] and **home locations** (Figure 26) [30]. This information reveals **the catchment and service areas of the Firenze–Lastra a Signa section**, providing a foundation for targeted interventions to enhance public transport quality and frequency, offering viable alternatives to car use.

The maps below suggest that many commuters travel between **Empoli and Florence**, suggesting that enhanced train or bus services on this corridor could encourage people to shift from private vehicles to public transport, thereby easing traffic volumes on the Firenze-Lastra a Signa segment.

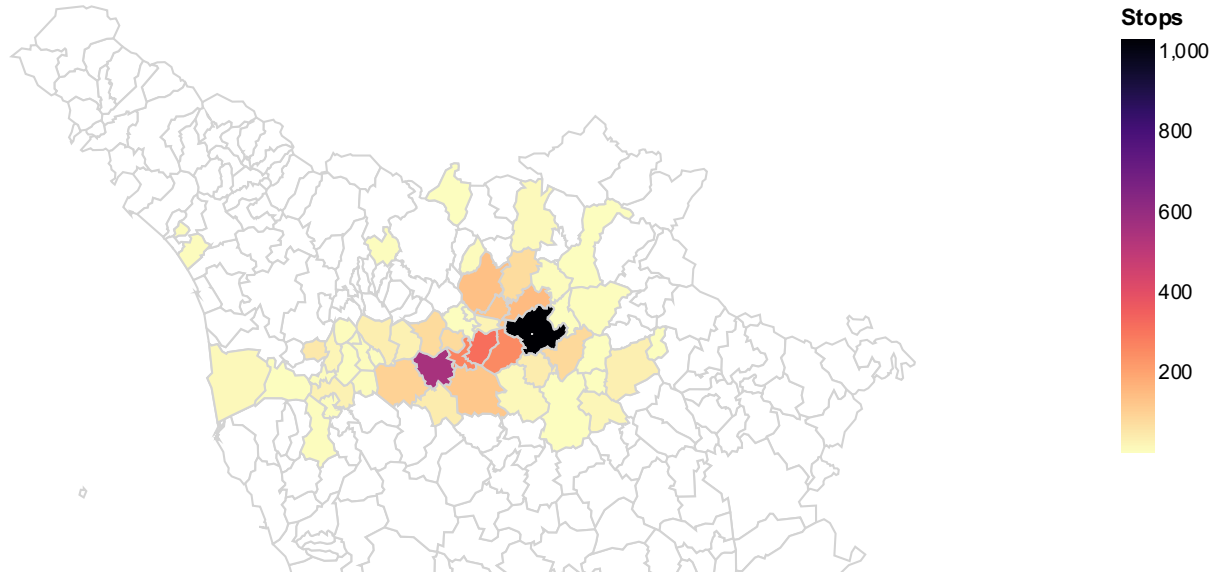


Figure 25: Firenze-Lastra a Signa's Commuters' Stops over the Course of February 2019

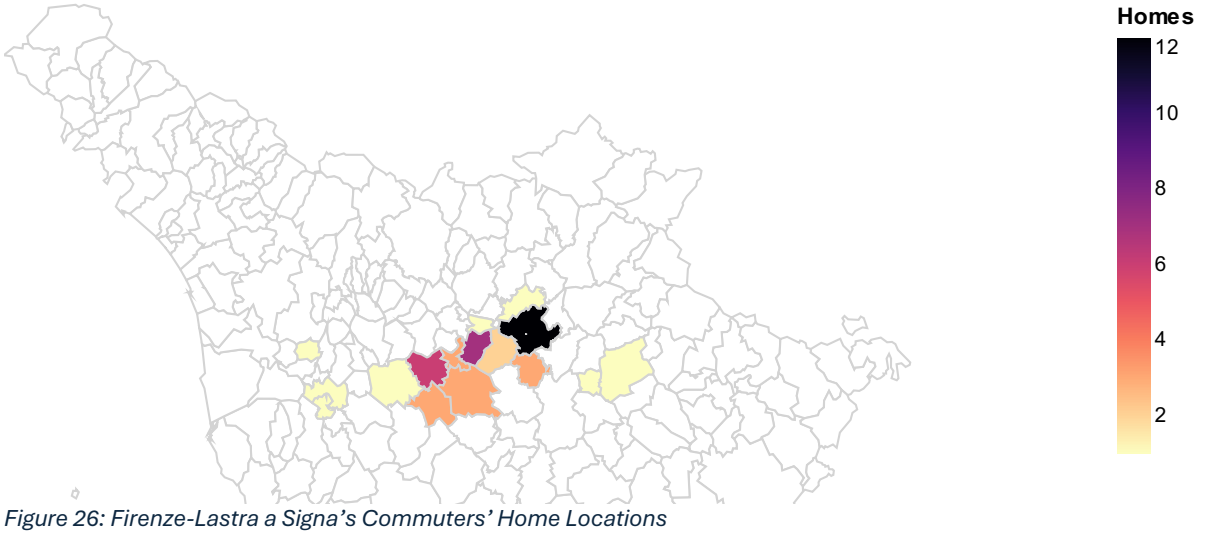
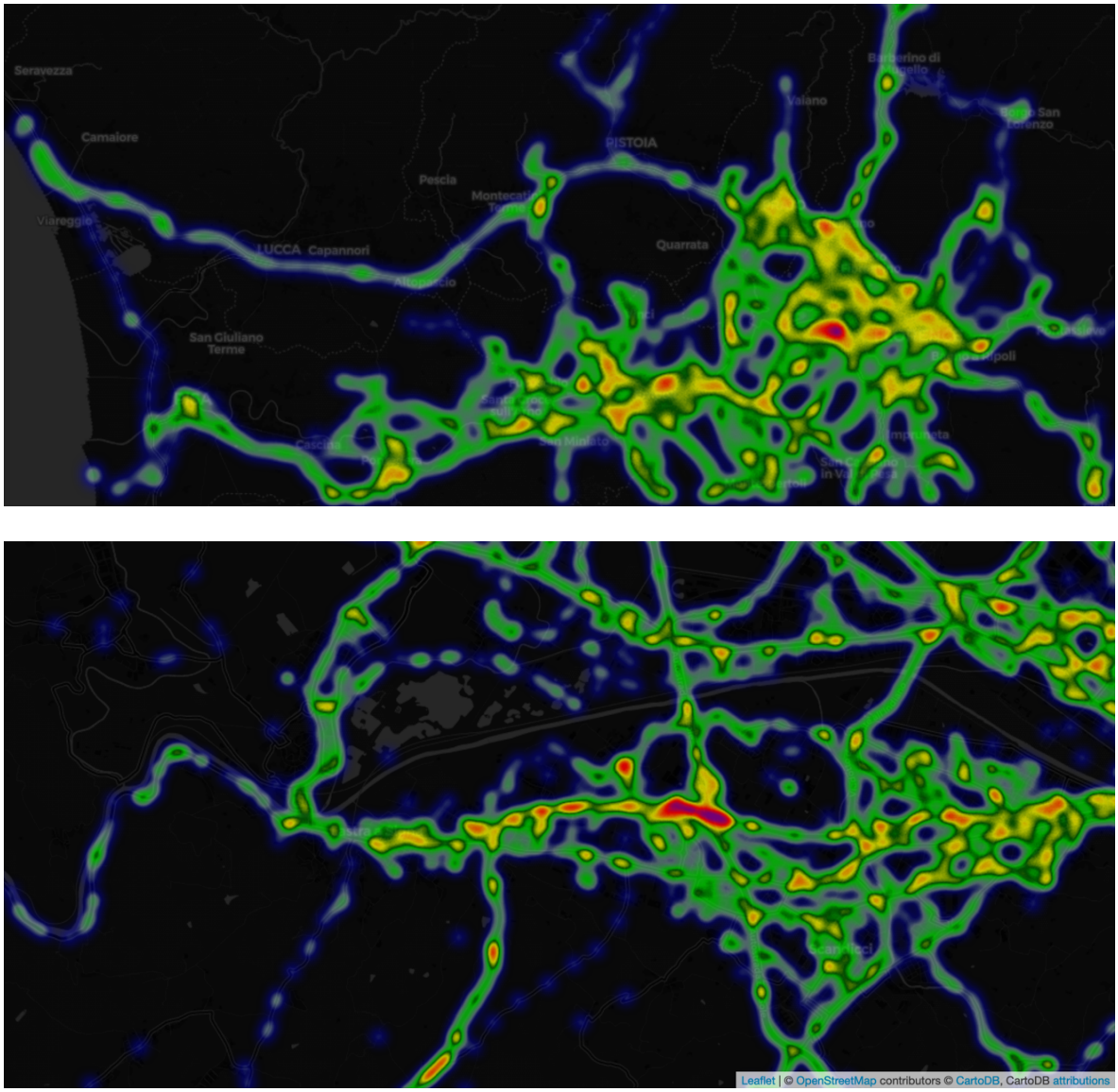


Figure 26: Firenze-Lastra a Signa's Commuters' Home Locations

Visualising Traffic: Dynamic Insights

Static analyses are only one piece of the puzzle as they cannot fully capture how traffic conditions evolve over time. By mapping commuters' flows and trajectories over a full month, planners gain a more **dynamic perspective on traffic patterns**, making it easier to spot recurring bottlenecks, high-density corridors, and potential areas of intervention. Below, we review **three types of visualisations** that can offer particularly valuable and actionable insights to policymakers.

- **Heatmaps of Traffic Intensity**. Creating a heatmap (Figures 27 and 28) [31] that highlights areas with the highest traffic intensity can pinpoint where congestions are most likely to occur. Unsurprisingly, **the junction between the FI-PI-LI and the A1 Milano-Napoli** emerges as a critical hotspot. Implementing targeted interventions there, such as widening lanes or improving connecting ramps, could provide immediate and lasting relief to the network.



Figures 27-28: Heatmaps of Traffic Intensity and Inset Zoom on the Junction between FIPILI and A1 MILANO-NAPOLI

- **Traffic Flow Visualisations**. By plotting trajectories [32] and flows [33] like in Figures 29 and 30, planners can swiftly identify areas more likely to see heavy traffic, junctions and segments most prone to congestion, and principal entry/exit points for cities. The information provided by these charts can therefore help transport planners to **prioritise interventions and investments**.

  Traffic flows could also be overlaid with the region's road network to pinpoint **underused or overused infrastructures**. For instance, Figure 31 highlights Florence's main entering (in red) and exiting (in blue) routes [34] and seems to indicate that some paths are mainly used for **inbound traffic only**. A deeper investigation on this imbalance might reveal some root issues like an inadequate public transport scheduling or poor drivers' awareness of certain roads. Administrators could then address these imbalances through awareness campaigns or improved transport services.

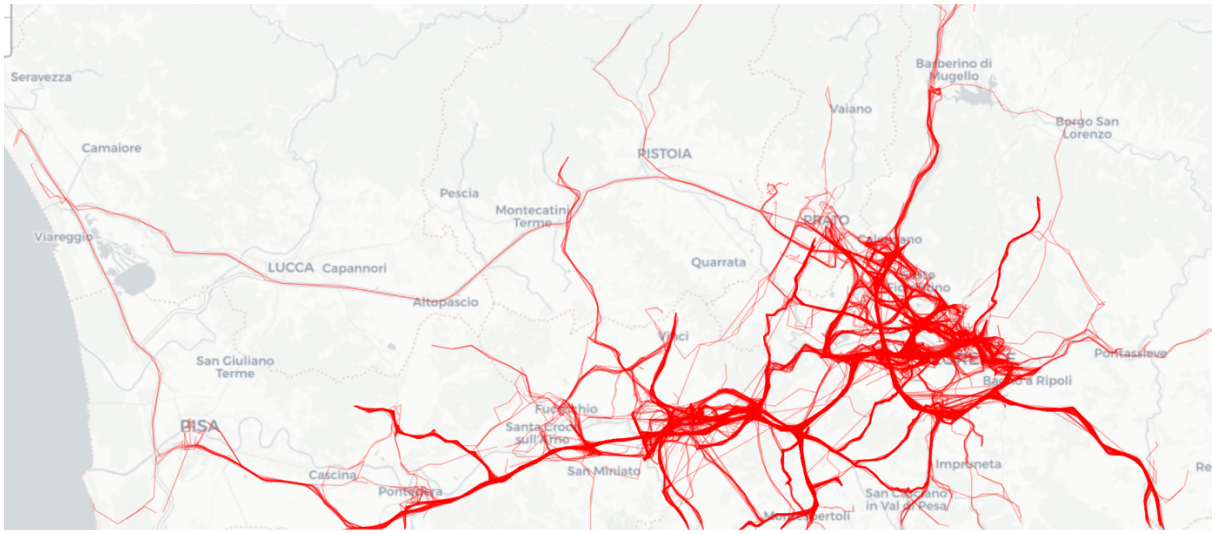


Figure 29: Commuters' Trajectories in February 2019

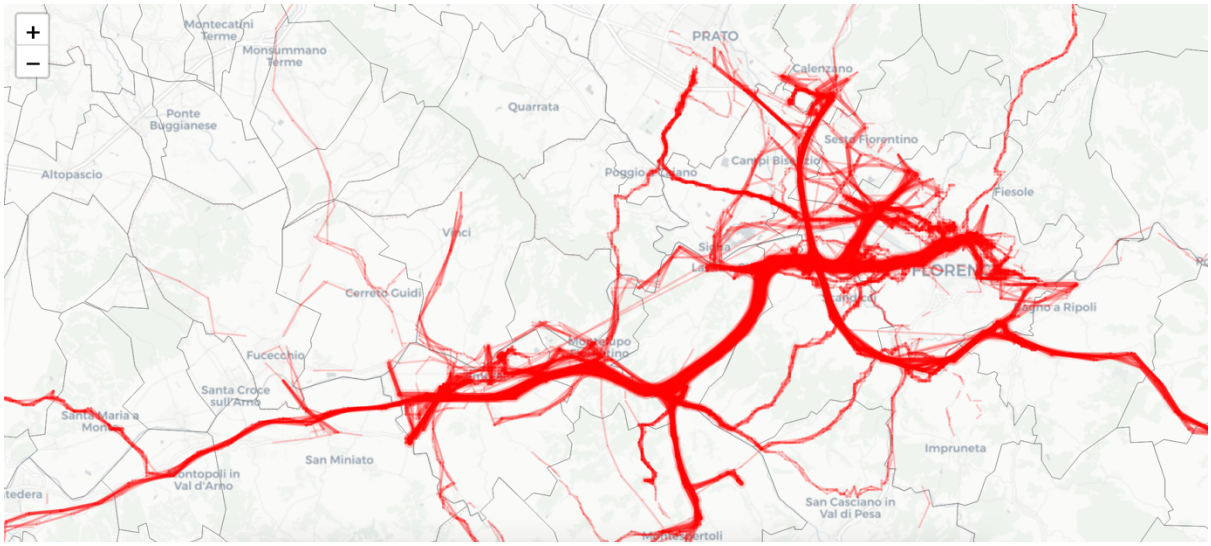


Figure 30: Commuters' Traffic Flows over February 2019

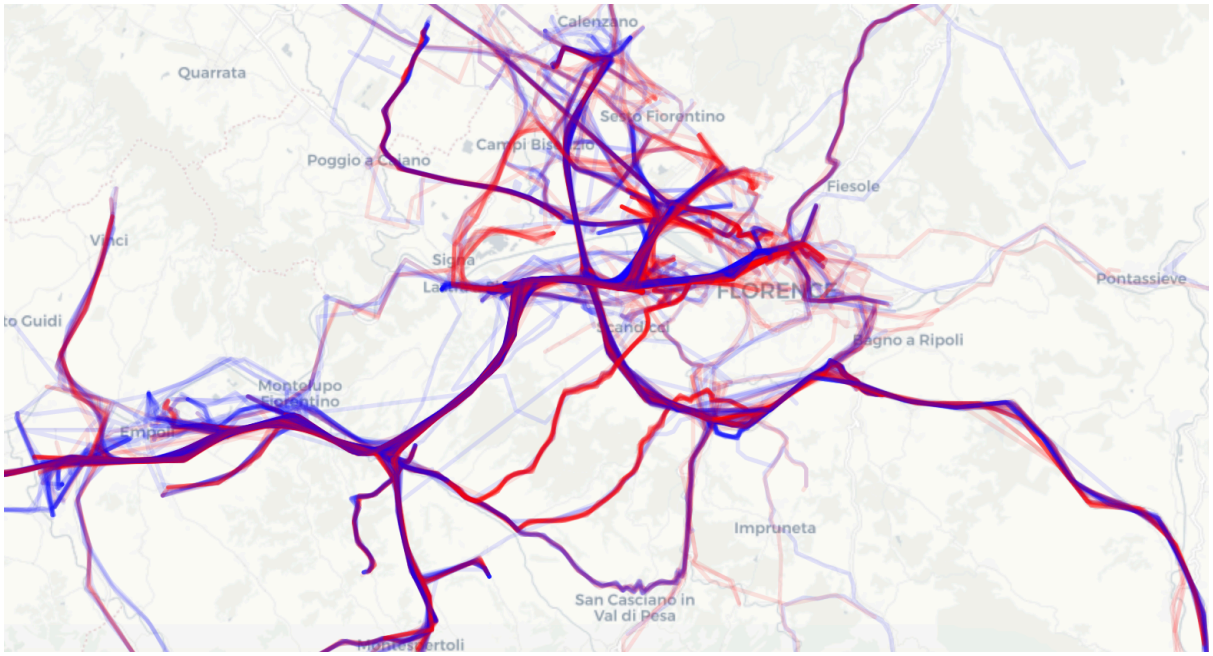


Figure 31: Inbound vs. Outbound. Inbound Trajectories in red. Outbound Trajectories in blue.

- **Clustering of trajectories**. This more advanced technique **groups individual trips** according to their origins, destinations, and other spatial distances. The resulting clusters can highlight opportunities for **alternative forms of transport** (e.g., park-and-ride facilities, new public transport lines, and carpooling initiatives), especially if many journeys start in the same general area.

  The map below (Figure 32) [35] shows trajectories into Florence grouped by their origin, destination, and spatial distance. These clusters could provide valuable input for **smart traffic management systems** helping authorities divert drivers to the best alternative routes efficiently in the event of accidents or planned closures, thereby minimising disruptions.

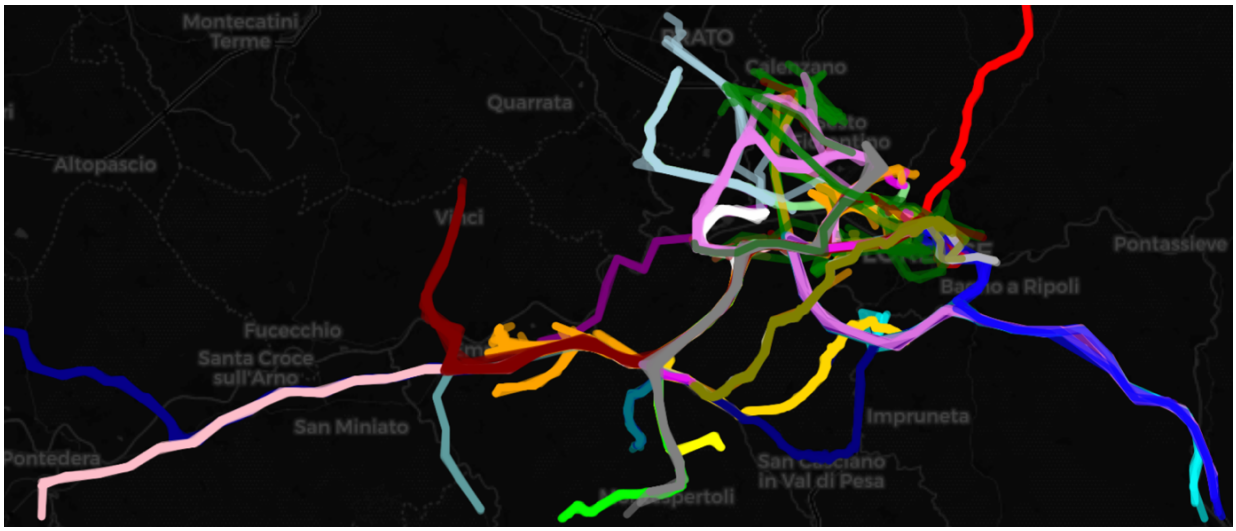


Figure 32: Clustering of Trajectories Entering Florence

1.6 Toward a Data-Driven Mobility

Once dismissed by some as a passing fad, the Big Data revolution has matured into a **transformative toolkit** for addressing real-world challenges and improving people's lives. The mobility sector is no exception.

Just a couple of decades ago, the idea of tracking regional mobility patterns in near-real time, and using these insights to optimise entire transport networks, would have seemed like science fiction. Today, not only this is feasible, but it is becoming increasingly vital for modern transport authorities. By integrating geographic information system (GIS) technologies, mobile phone analytics, GPS tracking, and machine learning, we can now pinpoint inefficiencies and allocate resources more precisely. As demonstrated in this project, evidence-based policymaking can **revitalise existing mobility systems** while paving the way for more efficient, sustainable future networks.

Future research

While the examples presented in this project underscore the transformative potential of data-driven transport planning, much remains to be explored, especially given a longer research horizon. Future investigations into the FIPILI motorway could concentrate on several key areas:

- **Improving the identification of commuters**
  To identify commuters, we selected drivers who travelled on the FIPILI motorway on weekdays (Monday to Friday) but not at the weekend. This approach is somehow simplistic, as we still do not know whether these drivers followed a typical commuter pattern during the day. Refining this identification could involve additional variables, such as time-of-day travel or consistency in the origin-destination patterns.
- **Refining Clustering Techniques**
  Incorporating additional spatial data, temporal details, and direction information could reveal how travel routes vary by time of day, weather conditions, and accident occurrences. This refined clustering might highlight different patterns of motorway use, including how traffic distribution changes when a particular segment is blocked or congested.
- **Analysing Speeds and Disruptions**
  Measuring average speeds on specific road segments could show how traffic flow evolves during the day, week, and year. Linking speed data to weather conditions or unplanned disruptions could further clarify how congestion propagates across the network.
- **Integrating Sensor Data**
  Combining GPS-based trajectories with real-time readings from road sensors would provide a more complete picture of daily traffic volumes, average speeds, and how

drivers adapt to changing weather or emergencies. Such integration might be crucial for pinpointing the onset of congestion and assessing the accident impact on broader travel corridors.

- **Using Predictive Modelling with AI**
  By leveraging machine learning, researchers could develop models incorporating factors such as time of day, traffic volumes, and weather to forecast the likelihood of congestion and accidents. Combined with mobility and routing predicting tools, this approach would let cities and regions anticipate bottlenecks before they even arise and then plan and manage the detours that diverted drivers are predicted to take.

- **Conducting Cost-Benefit Analysis**
  From solution design to implementation, policymakers must weigh the costs against expected benefits. Standard approaches might include calculating travel time savings, decreased fuel consumption, and improved safety records. Such valuations could offer a structured way to compare different projects.

- **Combining Different Datasets**
  Beyond analysing each dataset independently, a key step toward more holistic transportation planning could involve merging multiple data sources into a single analytical framework. For instance, combining public transport timetables, mobility data, and socio-economic indicators could expose whether certain communities are systematically underserved and penalised by existing infrastructure and guide strategic decisions to ensure equitable access across the region.

In short, the **full potential of data-driven transport planning** is yet to be fully unlocked. By fostering continued research, collaboration, and technological innovation, Tuscany (and other regions) could transform its mobility network into a more robust and predictive system with **great advantages for its economy and residents' quality of life**. Over time, these approaches will not only reduce congestion, accidents and travel times, but could also bring **more social equality** by giving rural communities fairer access to jobs, services, and economic opportunities.

Ultimately, the possibility of creating a smarter, more sustainable transport system, and its far-reaching social benefits, showcases the **unparalleled transformative potential of Big Data**, a powerful force that, when harnessed responsibly, can truly reshape and improve the way we live.

Part 2

Technical Report

2.1 Introduction

This technical report provides a detailed account of the data sources, processing techniques, and analytical methods employed to uncover the patterns and insights described in Part 1. It documents how raw GPS signals and anonymised mobile phone records were transformed into meaningful metrics, visualizations, and models, ultimately guiding the policy recommendations discussed earlier.

2.1.1 Data

The present dissertation draws upon several distinct datasets, each contributing critical information about mobility patterns and socio-economic contexts in Tuscany:

1. A mobility dataset based on **mobile phones records** with the following structure:

	StartId	EndId	PartOfWeek	Mot	tot
0	1	1	Mon-Fri	Bus	640.61
1	1	1	Mon-Fri	Road	228.45
2	1	1	Mon-Fri	SlowModes	5527.05
3	1	1	Saturday	Bus	963.81
4	1	1	Saturday	Road	333.20
...	...	...	...	...	...
672218	100007000	100007000	Saturday	Road	475.88
672219	100007000	100007000	Saturday	SlowModes	140.31
672220	100007000	100007000	Sunday	Bus	1.28
672221	100007000	100007000	Sunday	Road	417.13
672222	100007000	100007000	Sunday	SlowModes	74.11

   Each row represents **aggregated daily travel flows** between broader municipalities or sub-areas within municipalities, categorized by transport mode (*Mot*). The data is also split by time periods (*Mon-Fri, Saturday, Sunday*) but **lacks** more granular temporal information (e.g., specific hours). Because locations are approximated to the nearest cell tower, the dataset does not include precise coordinates or timestamps.

2. A series of logs created using the **GPS signals** from "black boxes" installed on cars for insurance purposes with this structure:

	id	Date	DeltaPos	Heading	ID	Latitude	Longitude	PanelSession	Speed	Time	Timestamp	geometry
0	0	2019-02-01	78	356	5514	43.768072	11.287648	1	4	06:48:47	2019-02-01 06:48:47	POINT (11.28765 43.76807)
1	1	2019-02-01	43	22	5514	43.768419	11.287818	1	22	06:49:35	2019-02-01 06:49:35	POINT (11.28782 43.76842)
2	2	2019-02-01	430	98	5514	43.768434	11.292807	1	52	06:50:11	2019-02-01 06:50:11	POINT (11.29281 43.76843)
3	3	2019-02-01	303	0	5514	43.768000	11.296633	1	0	06:50:47	2019-02-01 06:50:47	POINT (11.29663 43.76800)
4	4	2019-02-01	0	352	5514	43.768058	11.296517	1	4	06:51:24	2019-02-01 06:51:24	POINT (11.29652 43.76806)

   Data resolution depends on signal availability; certain stretches of road or times of day may have sparse coverage. The column *DeltaPos* represents the distance in metres from the previous point, *Heading* represents the direction of movement and *PanelSession* the state of the vehicle (0 = ignition, 1 = in-motion, 2 = off). N.B.: the original logs had also a column called *Quality* that indicated the strength of the GPS signal. Only data rows with the strongest signal (Quality=3) were retained, as per supplier's recommendation.

3. A shapefile (.shp) with the **administrative boundaries** of Tuscan having the following structure:

	NO	CODE	NAME	COD_CM	COD_COMUNE	COD_PROV~1	COD_REGI~2	NOME_CM	NOME_COM~3	NOME_PRO~4	NOME_REG~5	geometry
0	1	ZTL	Firenze Duomo	248.0	48017.0	48.0	9.0	Firenze	Firenze	Firenze	Toscana	POLYGON ((681714.437 4849091.117, 681715.668 4...
1	2	ZTL	Firenze Bargello	248.0	48017.0	48.0	9.0	Firenze	Firenze	Firenze	Toscana	POLYGON ((682066.911 4848443.429, 681831.643 4...
2	3	ZTL	Firenze SS. Annunziata	248.0	48017.0	48.0	9.0	Firenze	Firenze	Firenze	Toscana	POLYGON ((682247.640 4849356.491, 682028.527 4...
3	4	ZTL	Firenze Mercato Centrale	248.0	48017.0	48.0	9.0	Firenze	Firenze	Firenze	Toscana	POLYGON ((681417.547 4849875.979, 681800.793 4...
4	5	ZTL	Firenze Borgo Ognissanti	248.0	48017.0	48.0	9.0	Firenze	Firenze	Firenze	Toscana	POLYGON ((681060.600 4849370.300, 681077.282 4...

   This dataset allows for spatial joins and merges for visualisation purposes (e.g., linking each GPS record or municipality-based phone record to an official boundary polygon for mapping and analysis).

4. A series of shapefiles (.shp) of **Tuscany's transport infrastructures** [36] elaborated with *QGIS* [37], a geographic information system (GIS) software. These were used to visualise transport systems, identify motorway segments, and intersect with GPS points.

5. A CSV dataset of **socio-economic indicators** of municipalities in Tuscany [6]. These indicators have been merged with the administrative GeoDataFrame to explore correlations between mobility and socio-economic factors.

2.1.2 Structure of Technical Report: Linking to Part 1's Chapters

Having outlined the primary data sources, the sections that follow will provide a chapter-by-chapter breakdown of how these data were processed and analysed to yield the results discussed in **Part 1**. The following sections (2.2, 2.3, 2.4) correlate directly with the chapters of Part 1, namely 1.2, 1.4, and 1.5, that required distinct data manipulations techniques. Each subsection will describe the workflows, preprocessing steps, code snippets and analytical methods relevant to that particular chapter's focus, providing a look at how raw data were translated into the findings and insights discussed earlier.
The notebooks with the full code are available [38, 41, 42, 44, 47, 48, 50].

2.2 Methods for Chapter 1.2 (The Importance of Mobility Data)

2.2.1 Data Preprocessing and Cleaning

1. Shapefile of Tuscany administrate boundaries

This subsection details the workflow followed to import, clean, and prepare the shapefile of Tuscan administrative boundaries for subsequent analysis.

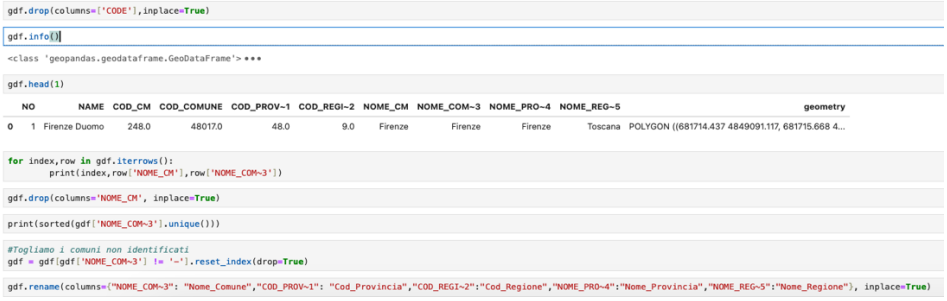
- **Importing the Shapefile**

  We first loaded the shapefile as a GeoDataFrame using the Python library **GeoPandas** [52]. A quick inspection via the `info()` method revealed relevant columns (e.g., `COD_CM`, `NOME_CM`, `geometry`) along with some fields not required for our study (e.g. `CODE`).

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 963 entries, 0 to 962
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   NO          963 non-null    int64
 1   CODE        20 non-null     object
 2   NAME        963 non-null    object
 3   COD_CM      963 non-null    float64
 4   COD_COMUNE  963 non-null    float64
 5   COD_PROV~1  963 non-null    float64
 6   COD_REGI~2  963 non-null    float64
 7   NOME_CM     923 non-null    object
 8   NOME_COM~3  963 non-null    object
 9   NOME_PRO~4  963 non-null    object
 10  NOME_REG~5  963 non-null    object
 11  geometry    963 non-null    geometry
dtypes: float64(4), geometry(1), int64(1), object(6)
```

- **Column Filtering and Renaming**

  Columns deemed irrelevant (e.g., those with excessive null values or attributes outside our research scope) were dropped to streamline the dataset. Remaining fields were renamed to more intuitive labels.



```
gdf.drop(columns=['CODE'],inplace=True)

gdf.info()

<class 'geopandas.geodataframe.GeoDataFrame'>

gdf.head(1)

   NO  NAME  COD_CM  COD_COMUNE  COD_PROV~1  COD_REGI~2  NOME_CM  NOME_COM~3  NOME_PRO~4  NOME_REG~5  geometry
0   1  Firenze Duomo  248.0  48017.0  48.0  9.0  Firenze  Firenze  Firenze  Toscana  POLYGON ((681714.437 4849091.117, 681715.668 4...

for index,row in gdf.iterrows():
        print(index,row['NOME_CM'],row['NOME_COM~3'])

gdf.drop(columns='NOME_CM', inplace=True)

print(sorted(gdf['NOME_COM~3'].unique()))

#Togliamo i comuni non identificati
gdf = gdf[gdf['NOME_COM~3'] != '~'].reset_index(drop=True)

gdf.rename(columns={"NOME_COM~3": "Nome_Comune","COD_PROV~1": "Cod_Provincia","COD_REGI~2":"Cod_Regione","NOME_PRO~4":"Nome_Provincia","NOME_REG~5":"Nome_Regione"}, inplace=True)
```

- **Duplicate and Null Handling & Saving of ZonePulite.geojson**

  We checked for duplicates across key identifiers (e.g., municipal codes) to ensure each municipality was uniquely represented. Rows with not identifiable values (such

as “- “) in crucial fields were filtered out. We got a GeoDataFrame with the following structure that is then saved for later use:

```
for index,row in gdf.iterrows():
        print(index,row['NOME_CM'],row['NOME_COM~3'])
```

```
gdf.drop(columns='NOME_CM', inplace=True)
```

```
print(sorted(gdf['NOME_COM~3'].unique()))
```

```
#Togliamo i comuni non identificati
gdf = gdf[gdf['NOME_COM~3'] != '-'].reset_index(drop=True)
```

```
gdf.rename(columns={"NOME_COM~3": "Nome_Comune","COD_PROV~1": "Cod_Provincia","COD_REGI~2":"Cod_Regione","NOME_PRO~4":"Nome_Provincia","NOME_REG~5":"Nome_Regione"}, inplace=True)
```

```
gdf['COD_CM'].unique()
```

```
array([248.,   0.])
```

```
gdf.drop(columns='COD_CM',inplace=True)
```

```
gdf.head()
```

```
<div> ***
```

```
gdf.to_file('ZonePulite.geojson', driver='GeoJSON')
```

```
cod_comune_check = gdf.groupby('Nome_Comune')['COD_COMUNE'].nunique()


inconsistent_comuni = cod_comune_check[cod_comune_check > 1]


if not inconsistent_comuni.empty:
    print("Inconsistencies found:")
    display(inconsistent_comuni)
else:
    print("All comuni have consistent COD_COMUNE values.")
```

```
All comuni have consistent COD_COMUNE values.
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 866 entries, 0 to 865
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   NO              866 non-null    int64
 1   NAME            866 non-null    object
 2   COD_COMUNE      866 non-null    float64
 3   Cod_Provincia   866 non-null    float64
 4   Cod_Regione     866 non-null    float64
 5   Nome_Comune     866 non-null    object
 6   Nome_Provincia  866 non-null    object
 7   Nome_Regione    866 non-null    object
 8   geometry        866 non-null    geometry
dtypes: float64(3), geometry(1), int64(1), object(4)
memory usage: 61.0+ KB
```

- **Merging Intra-municipal Areas**
  Certain municipalities were subdivided into multiple polygons. Since we decided to frame our analysis at municipal level, we **merged** these sub-polygons into a single polygon per municipality using the `dissolve()` function:

```
aggregated_gdf = gdf.dissolve(by='COD_COMUNE')

aggregated_gdf.head(5)
```

COD_COMUNE	geometry	NO	NAME	Cod_Provincia	Cod_Regione	Nome_Comune	Nome_Provincia	Nome_Regione
11020.0	POLYGON ((582703.141 4879659.469, 581618.329 4...	11020000	A12 - Liguria	11.0	7.0	Luni	La Spezia	Liguria
34004.0	POLYGON ((573515.003 4924755.433, 572321.122 4...	34004000	A15 - Emilia Romagna	34.0	8.0	Berceto	Parma	Emilia-Romagna
35046.0	POLYGON ((593115.003 4909179.549, 592619.258 4...	35046000	SS63	35.0	8.0	Ventasso	Reggio nell'Emilia	Emilia-Romagna
37001.0	POLYGON ((673057.771 4933672.770, 673326.958 4...	37001000	Anzola dell'Emilia	37.0	8.0	Anzola dell'Emilia	Bologna	Emilia-Romagna
37005.0	POLYGON ((690516.891 4939753.776, 690403.517 4...	37005000	A13 - Bologna	37.0	8.0	Bentivoglio	Bologna	Emilia-Romagna

- **Reprojecting to EPSG:4326**
  The resulting GeoDataFrame has EPSG:32632 coordinate reference system (CRS) which is a UTM (Universal Transverse Mercator) projection in metres. However, to ensure compatibility with mapping libraries (e.g., Folium) and maintain consistency across datasets, the GeoDataFrame was reprojected to **EPSG:4326** what uses latitude and longitude (WGS84):

```
aggregated_gdf = aggregated_gdf.to_crs(epsg=4326)
```

- **Filtering for Tuscany & Saving of TessellationToscana.geojson**
  Finally, the dataset was **restricted** to polygons belonging to Tuscany by verifying administrative codes or region fields. This final GeoDataFrame was saved as a GeoJSON and plotted to confirm that our dataset accurately represented Tuscany's municipalities.

[48]:
```
aggregated_gdf=aggregated_gdf[aggregated_gdf['Nome_Regione'] == 'Toscana']
```

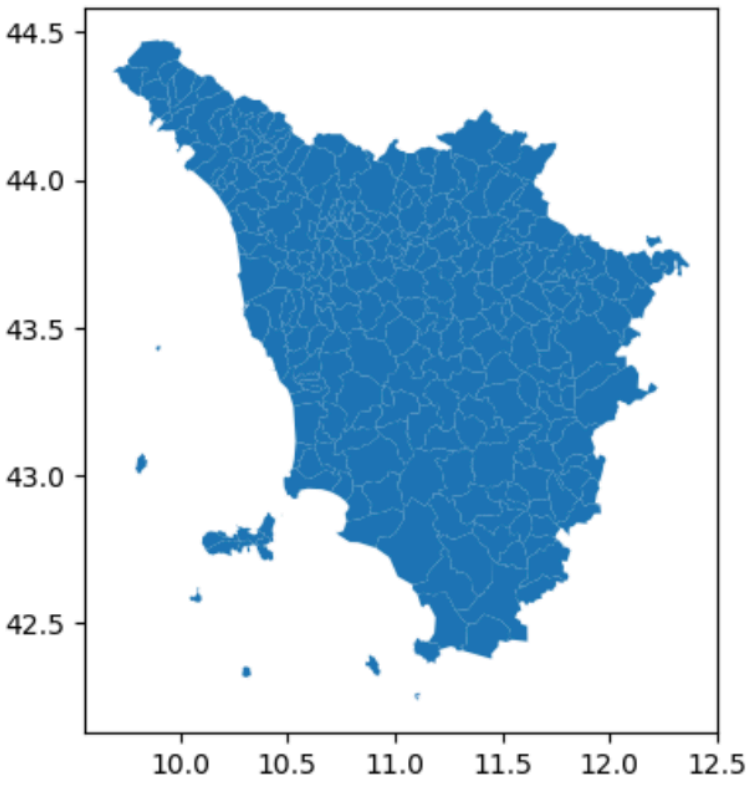
[49]:
```
aggregated_gdf.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Float64Index: 276 entries, 45001.0 to 100007.0
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   geometry        276 non-null    geometry
 1   NO              276 non-null    int64
 2   NAME            276 non-null    object
 3   Cod_Provincia   276 non-null    float64
 4   Cod_Regione     276 non-null    float64
 5   Nome_Comune     276 non-null    object
 6   Nome_Provincia  276 non-null    object
 7   Nome_Regione    276 non-null    object
dtypes: float64(2), geometry(1), int64(1), object(4)
memory usage: 19.4+ KB
```

[78]:
```
aggregated_gdf.to_file("/Users/Andre/Desktop/Internship/TessellationToscana.geojson", driver="GeoJSON")
```

]:
```
aggregated_gdf.plot()
```

]: <Axes: >



2. CSV file of socio-economic indicators for Tuscany

This subsection describes how we imported, cleaned, and prepared the CSV containing socio-economic indicators for Tuscan municipalities.

- **Importing the CSV file and naming the columns**

  The CSV file is read with the function:

```
indicatori=pd.read_csv('Internship/indicatori/Indicatori_2022.csv',encoding='ISO-8859-1',sep=';')
```

One row had an invalid name for comune (“*Toscana”)* and was filtered out. We then renamed columns based on the dataset’s provided legend, resulting in a DataFrame with clearer field names and with the following structure:

```
indicatori.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 273 entries, 0 to 272
Data columns (total 22 columns):
 #   Column                                                    Non-Null Count  Dtype
---  ------                                                    --------------  -----
 0   comune                                                    273 non-null    object
 1   cod_istat_comune                                          273 non-null    float64
 2   Percentage Change in Resident Population (3 years)        273 non-null    object
 3   Percentage of Population Over 80 Years Old                273 non-null    object
 4   Natural Balance (Relative)                                273 non-null    object
 5   Percentage of Young Foreigners (0-19 Years)               273 non-null    object
 6   Percentage of Youth (15-24) in Other Professional Conditions  273 non-null  object
 7   Percentage of Individuals (25-49) with Tertiary Education  273 non-null    object
 8   Percentage of Taxpayers with Income Below €10,000         273 non-null    object
 9   Average Taxable Income (Euros) Per Taxpayer               273 non-null    int64
 10  Percentage of Job Seekers in the Labor Force              273 non-null    object
 11  Percentage of Active Sole Proprietorships with Foreign-Born Owner  273 non-null  object
 12  Environmental Risk Index                                  273 non-null    object
 13  Percentage of Land Consumed                               273 non-null    object
 14  Tourism Pressure (per 1,000 Inhabitants)                  273 non-null    object
 15  Organizations Registered in Regional Registers Per 10,000 Residents  273 non-null  object
 16  Individuals (0-64) with Disabilities Per 1,000 Residents  273 non-null    object
 17  Local Units Providing Health Assistance Per 1,000 Residents  273 non-null  object
 18  Percentage of Services Offered Online at the Highest Level by Municipalities  273 non-null  object
 19  Emergency Response Time Target Indicator (75th Percentile)  273 non-null  object
 20  Percentage of Active Enterprises in Innovation Sectors    273 non-null    object
 21  Percentage of Agricultural Area Cultivated Using Organic Methods  273 non-null  object
dtypes: float64(1), int64(1), object(20)
memory usage: 49.1+ KB
```

- **Handling Mismatched Municipalities**

  A discrepancy emerged between this DataFrame and our administrative boundaries: **three pairs of municipalities** had been merged into three larger ones in the indicators’ dataset, while in the shapefile they remained as six distinct municipalities. This would have caused an issue when merging. To resolve this, we duplicated the relevant rows and assigned the correct unique codes to each split municipality:

```
rows_to_duplicate = indicatori.loc[[37, 49, 71]]

indicatori = pd.concat([indicatori, rows_to_duplicate], ignore_index=True)
```

```
#siccome non c'è corrispondenza tra tessellation e indicatori perché 6 comuni sono stati fusi in 3, li ricreiamo prima del merge
indicatori.loc[49, 'cod_istat_comune'] = 48016.0
indicatori.loc[274, 'cod_istat_comune'] = 48023.0
indicatori.loc[37, 'cod_istat_comune'] = 48003.0
indicatori.loc[273, 'cod_istat_comune'] = 48045.0
indicatori.loc[71, 'cod_istat_comune'] = 48042.0
indicatori.loc[275, 'cod_istat_comune'] = 48040.0
```

2.2.2 Creating the GeoDataframe by Merging Indicators and Tessellation

- **Merging**
  The two datasets are merged on two univocal values that they have in common. The resulting DataFrame is checked for null values:

```
tessellation=gpd.read_file("/Users/Andre/Desktop/Internship/TessellationToscana.geojson", driver="GeoJSON")
```

```
indicatori_df = pd.merge(indicatori, tessellation, left_on='cod_istat_comune', right_on='COD_COMUNE', how='left')
```

```
indicatori_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>•••
```

```
nan_rows = indicatori_df[indicatori_df.isna().any(axis=1)]
print(nan_rows)
```

- **Converting the DataFrame into GeoDataFrame**
  We converted the DataFrame into a GeoDataFrame using GeoPandas specifying the column to be used for the geometry.

```
indicatori_gdf=gpd.GeoDataFrame(indicatori_df,  geometry=indicatori_df["geometry"])
```

- **Preparing the GeoDataFrame**
  At first, we encoutered issues during the visualisation. To resolve them, we used regular expressions (regex) to clean the column names by removing spaces and replacing special characters. Additionally, we converted decimal numbers by replacing commas with full stops and transformed object data types into floats.

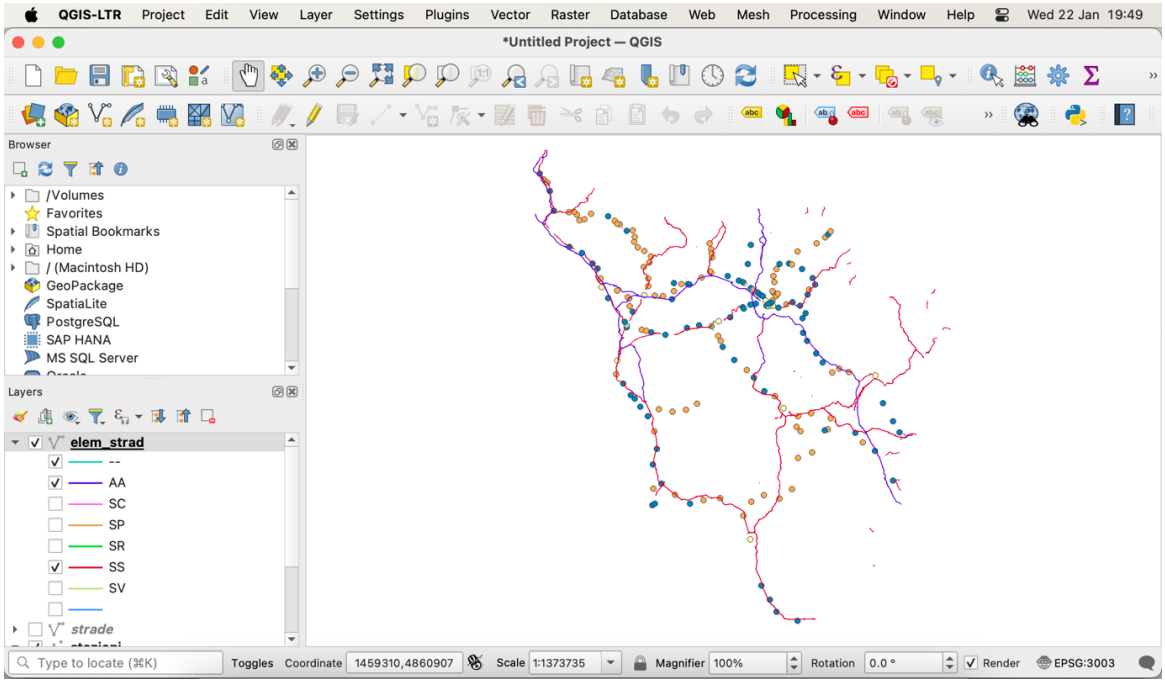
```
indicatori_gdf.columns = (
    indicatori_gdf.columns
    .str.strip()  # Toglie spazi prima e dopo
    .str.replace(r"[^a-zA-Z0-9_]", "_", regex=True)  # Rimpiazza con underscore i valori speciali
)
```

```
for col in indicatori_gdf.select_dtypes(include='object').columns:
    indicatori_gdf[col] = (
        indicatori_gdf[col]
        .str.replace(",", ".", regex=False)  #Se indicatori hanno virgole nei decimali si cambiano in punti
        .astype(float, errors="ignore")  # Cambia in float se possibile
    )
```

2.2.3 Preparing the GeoDataFrames of Main Transport Infrastructures

We next turned our attention to Tuscany's main transport infrastructure files provided as shapefiles [36]

Reading, filtering and exporting with QGIS
Each shapefile (representing railway stations, motorways, or key roads) was opened in QGIS-LTR. We then filtered the features of interested. Finally, the resulting filtered layers were re-saved as shapefiles for consistency.



Resulting Shapefiles:

- **Railway Stations**: Classified by category (e.g., silver, gold, platinum).
- **Motorways**
- **Other Important Main Roads**

2.2.4 Creation of Interactive Maps with Altair

To visualize the socio-economic indicators (Figures 1–4) alongside Tuscany's main transport infrastructures (motorways, important roads, railway stations), we used **Vega-Altair** [53], a Python library that creates interactive and visually appealing charts. This section describes how the map is generated, including toggles for showing or hiding each type of infrastructure.

- **Setting Up Toggle Parameters**
  We introduced three Altair parameters—toggle_autostrade, toggle_main, and toggle_stazioni—to allow users to switch motorways, major roads, and railway stations on or off.

- **Preparing Shapefiles**
  Each infrastructure shapefile (motorways, main roads, railway stations) is reprojected to EPSG:4326 for consistency. The shapefiles are then converted to a GeoInterface object (`__geo_interface__`) so that Altair can interpret them as geospatial features.

- **Plotting Transport Layers**
  Motorways and main roads are drawn as geoshapes with orange outlines (thicker for motorways). Railway stations appear as circles, color-coded by category (silver, gold, platinum). The script also orders the stations by rank (silver, gold, platinum) so the most important stations stand out.

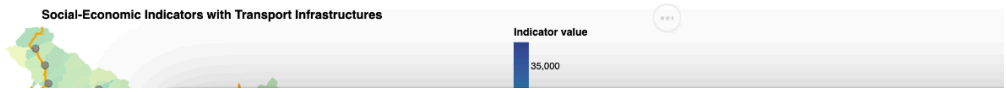
- **Incorporating Socio-Economic Indicators**
  We define `selected_indicator`, a parameter bound to a dropdown listing all possible numeric fields in the indicators' dataset (e.g., average income, population over 80, etc.). We then transform each feature by extracting the "`selected_value`" from `datum.properties[indicator]`. This controls which numeric field is displayed on the map at any given time thereby colouring the polygons according to the currently selected indicator.

- **Combining Layers**
  The base layer (municipality polygons) is drawn with the chosen indicator using a Mercator projection. We then add the transport layers (motorways, main roads, stations) to this base chart. The final composite chart includes interactive toggles for each infrastructure layer and a dropdown for choosing different socio-economic indicators.

  Below is the full code that implements this logic:

```
import altair as alt
## Autostrade
toggle_autostrade = alt.param(
    name='toggle_autostrade',
    bind=alt.binding_radio(options=['Show Motorways', 'Hide Motorways'], name='Motorways:'),
    value='Show Motorways')
motorways = motorways.to_crs(epsg=4326)
geojson_motor = motorways.__geo_interface__


autostrade=alt.Chart(alt.Data(values=geojson_motor["features"])).mark_geoshape(stroke="orange",
    strokeWidth=2, fill=None).encode(opacity=alt.condition(
            "toggle_autostrade == 'Show Motorways'",
            alt.value(0.7),
            alt.value(0)
        )).project(
    type="mercator"
).properties(
    width=600,
    height=500,
    title=""
).add_params(
    toggle_autostrade
)

## Main Roads
toggle_main = alt.param(
    name='toggle_main',
    bind=alt.binding_radio(options=['Show Main Roads', 'Hide Main Roads'], name='Main Roads:'),
    value='Show Main Roads')
roads = roads.to_crs(epsg=4326)
geojson_roads = roads.__geo_interface__

main=alt.Chart(alt.Data(values=geojson_roads["features"])).mark_geoshape(stroke="orange",
    strokeWidth=0.8, fill=None).encode(opacity=alt.condition(
            "toggle_main == 'Show Main Roads'",
            alt.value(0.7),
            alt.value(0)
        )).project(
    type="mercator"
).properties(
    width=600,
    height=500,
    title=""
).add_params(
    toggle_main
)
```

```
## Stazioni
toggle_stazioni = alt.param(
    name='toggle_stazioni',
    bind=alt.binding_radio(options=['Show Stations', 'Hide Stations'], name='Stations:'),
    value='Show Stations')

color_mapping = {
    "silver": "grey",
    "gold": "yellow",
    "platinum":"red"
}

stazioni_=stazioni_gdf[stazioni_gdf['categoria']!='bronze']
category_order = {'silver': 0, 'gold': 1, 'platinum': 2}
stazioni_['rank'] = stazioni_['categoria'].map(category_order)
stazioni = alt.Chart(stazioni_).mark_circle(size=60,stroke='black', strokeWidth=0.3).encode(
    longitude='longitude:Q',
    latitude='latitude:Q',
    color=alt.Color("categoria:N", title="Station Category",scale=alt.Scale(
            domain=list(color_mapping.keys()),
            range=list(color_mapping.values())
        ),
        legend=alt.Legend(
            title="Railway Station Categories",
            orient="right"
        ) if toggle_stazioni.value == 'Show Stations' else None
    ),order=alt.Order('rank:Q', sort='ascending'),#prima disegna i silver, poi i gold e infine i platinum
    opacity=alt.condition("toggle_stazioni == 'Show Stations'",
            alt.value(0.8),
            alt.value(0)
        ),
    tooltip=[
        alt.Tooltip("comune:N", title="Comune"),
        alt.Tooltip("categoria:N", title="Category")
    ]
).project(
    type="mercator"
).properties(
    width=600,
    height=500,
    title=""
).add_params(
    toggle_stazioni
)
## Mappa indicatori
indicators=list(indicatori_gdf.columns)

options = sorted(ind for ind in indicators if ind not in['comune','cod_istat_comune','COD_COMUNE', 'NO', 'NAME', 'Cod_Provincia', 'Cod_Regione',
       'Nome_Comune', 'Nome_Provincia', 'Nome_Regione', 'geometry'])
selected_indicator = alt.param(
    name='indicator',
    bind=alt.binding_select(options=options, name="Select Indicator"),
    value=options[0]
)


indicatori_gdf = indicatori_gdf.set_crs(epsg=4326).to_crs(epsg=4326)


geojson = indicatori_gdf.__geo_interface__

base = alt.Chart(alt.Data(values=geojson["features"])).mark_geoshape().encode(
    color=alt.Color(
        "selected_value:Q",
        title='Indicator value'
    ),
    tooltip=[
        alt.Tooltip("properties.comune:N", title="Municipality"),
        alt.Tooltip("selected_value:Q", title="'Indicator value:")
    ]
).add_params(
    selected_indicator
).transform_calculate(
    selected_value='datum.properties[indicator]',
    selected_title='indicator'
).project(
    type="mercator"
).properties(
    width=600,
    height=500,
    title='Social-Economic Indicators with Transport Infrastructures'
)



base = base.encode(
    opacity=alt.value(0.9)
)
combined_chart = (base +autostrade+main+stazioni)
combined_chart
```



2.3 Methods for Chapter 1.4 (A Region-wide Perspective)

2.3.1. Data Preprocessing and Cleaning

This subsection details the workflow followed to import, clean, and prepare the DataFrames, and GeoDataFrames for subsequent analysis and visualisations starting from the mobility dataset based on **mobile phones records** that reports aggregated daily travel flows between broader municipalities or sub-areas within municipalities.

- **Importing and inspecting the Excel file**

We read the Excel file into a pandas DataFrame using Python's `pandas.read_excel()` function. A quick check via `info()` revealed 5 columns and 672,223 rows, with no null values in any field. This indicates a relatively clean dataset to start with.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 672223 entries, 0 to 672222
Data columns (total 5 columns):
 #   Column      Non-Null Count   Dtype 
---  ------      --------------   ----- 
 0   StartId     672223 non-null  int64 
 1   EndId       672223 non-null  int64 
 2   PartOfWeek  672223 non-null  object
 3   Mot         672223 non-null  object
 4   tot         672223 non-null  float64
dtypes: float64(1), int64(2), object(2)
memory usage: 25.6+ MB
```

- **Filtering by Valid StartId and EndId**

The dataset includes `StartId` and `EndId` values corresponding to the origin and destination zones. We retained only rows with zones that can be mapped to valid areas in our initial GeoDataFrames of administrative boundaries (see section 2.2.1). This step ensures that subsequent merges and spatial operations do not encounter mismatched or invalid zone IDs.

```
zone_pulite=gpd.read_file('ZonePulite.geojson', driver='GeoJSON')
```

```
lista_zone=(zone_pulite['NO'].unique()).tolist()
```

```
##prendiamo solo i movimenti che sono interni alle zone del geodataframe di zonizzazione iniziale
movements_tos = movements_df[(movements_df["StartId"].isin(lista_zone))&(movements_df["EndId"].isin(lista_zone))]
print(movements_tos)
```

```
          StartId      EndId PartOfWeek        Mot      tot
0               1          1    Mon-Fri        Bus   640.61
1               1          1    Mon-Fri       Road   228.45
2               1          1    Mon-Fri  SlowModes  5527.05
3               1          1   Saturday        Bus   963.81
4               1          1   Saturday       Road   333.20
...           ...        ...        ...        ...      ...
672218  100007000  100007000   Saturday       Road   475.88
672219  100007000  100007000   Saturday  SlowModes   140.31
672220  100007000  100007000     Sunday        Bus     1.28
672221  100007000  100007000     Sunday       Road   417.13
672222  100007000  100007000     Sunday  SlowModes    74.11

[587707 rows x 5 columns]
```

- **Merging of the Filtered DataFrame with the Zones' DataFrame**

  After filtering, we merged the phone-based flow records with the zones' attribute data to incorporate details about each origin and destination (municipality name, province, geometry etc.). To achieved this, we followed these steps:

  1. Merged on `StartId` to retrieve origin details, renamed columns to distinguish new fields
  2. Merged the resulting DataFrame with the Zones' DataFrames by this time on `EndId` to retrieve destination attributes, we renamed columns to clarify which fields refer to the destination, and we removed any extra columns that we did not need.

The code used is the following:

```
#Questo codice esegue due merge tra il dataframe movements_tos e il zone_pulite:
#prima sullo "StartId" per ottenere i dettagli della località di partenza (nome, provincia, geometria),
#e poi sull'"EndId" per ottenere i dettagli della località di arrivo.
#Infine, rinomina le colonne per distinguere i campi di partenza/arrivo, elimina quelle superflue e stampa la struttura del dataframe risultante.
movements_with_start = movements_tos.merge(zone_pulite[['NO', 'Nome_Comune','Nome_Provincia','geometry']], left_on='StartId', right_on='NO', how='left')
movements_with_start = movements_with_start.rename(columns={'Nome_Comune': 'Start_Nome_Comune','geometry':'Start_Geometry','Nome_Provincia':'Start_Nome_Provincia'}).drop(columns=['NO'])

movements_with_names = movements_with_start.merge(zone_pulite[['NO', 'Nome_Comune','geometry','Nome_Provincia']], left_on='EndId', right_on='NO', how='left')
movements_with_names = movements_with_names.rename(columns={'Nome_Comune': 'End_Nome_Comune','geometry':'End_Geometry','Nome_Provincia':'End_Nome_Provincia'}).drop(columns=['NO'])

movements_with_names.head(2)
```

	StartId	EndId	PartOfWeek	Mot	tot	Start_Nome_Comune	Start_Nome_Provincia	Start_Geometry	End_Nome_Comune	End_Geometry	End_Nome_Provincia
0	1	1	Mon-Fri	Bus	640.61	Firenze	Firenze	POLYGON ((681714.437 4849091.117, 681715.668 4...	Firenze	POLYGON ((681714.437 4849091.117, 681715.668 4...	Firenze
1	1	1	Mon-Fri	Road	228.45	Firenze	Firenze	POLYGON ((681714.437 4849091.117, 681715.668 4...	Firenze	POLYGON ((681714.437 4849091.117, 681715.668 4...	Firenze

And the resulting DataFrame has this structure:

```
movements_with_names.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 587707 entries, 0 to 587706
Data columns (total 11 columns):
 #   Column                Non-Null Count   Dtype
---  ------                --------------   -----
 0   StartId               587707 non-null  int64
 1   EndId                 587707 non-null  int64
 2   PartOfWeek            587707 non-null  object
 3   Mot                   587707 non-null  object
 4   tot                   587707 non-null  float64
 5   Start_Nome_Comune     587707 non-null  object
 6   Start_Nome_Provincia  587707 non-null  object
 7   Start_Geometry        587707 non-null  geometry
 8   End_Nome_Comune       587707 non-null  object
 9   End_Geometry          587707 non-null  geometry
 10  End_Nome_Provincia    587707 non-null  object
dtypes: float64(1), geometry(2), int64(2), object(6)
memory usage: 53.8+ MB
```

- **Filtering only the Inter-Municipality movements within Tuscany**

1. Movements with the same origin and destination municipality were excluded to filter out intra-municipality movements. This ensured that only inter-municipality movements were kept.

```
## Creaimo intercomune movements, togliamo cioè i movimenti intracomunali
intercomune_movements=movements_with_names[movements_with_names['Start_Nome_Comune']!=movements_with_names['End_Nome_Comune']]
```

2. Movements were further filtered to retain only those originating and terminating within provinces of Tuscany.

```
tuscany_provinces = ["Firenze","Massa-Carrara",
    "Lucca",
    "Pistoia",
    "Livorno",
    "Pisa",
    "Arezzo",
    "Siena",
    "Grosseto",
    "Prato"]
```

```
intercomune_movements=intercomune_movements[
(intercomune_movements['Start_Nome_Provincia'].isin(tuscany_provinces)) &
(intercomune_movements['End_Nome_Provincia'].isin(tuscany_provinces))]
```

```
intercomune_movements.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 426748 entries, 1129 to 587697
Data columns (total 11 columns):
 #   Column                Non-Null Count   Dtype
---  ------                --------------   -----
 0   StartId               426748 non-null  int64
 1   EndId                 426748 non-null  int64
 2   PartOfWeek            426748 non-null  object
 3   Mot                   426748 non-null  object
 4   tot                   426748 non-null  float64
 5   Start_Nome_Comune     426748 non-null  object
 6   Start_Nome_Provincia  426748 non-null  object
 7   Start_Geometry        426748 non-null  geometry
 8   End_Nome_Comune       426748 non-null  object
 9   End_Geometry          426748 non-null  geometry
 10  End_Nome_Provincia    426748 non-null  object
dtypes: float64(1), geometry(2), int64(2), object(6)
memory usage: 39.1+ MB
```

- **Creating a GeoDataFrame with Total Outgoing movements between municipalities**
  1. We aggregated the movements in the `intercomune_movements` DataFrame by using a `groupby()` function on `Start_Nome_Comune` and `End_Nome_Comune` so that we have a DataFrame with the total movements between municipalities.

[92]:
```
#Se facciamo un groupby per start e end, allora abbiamo quello che ci interessa (movimenti totali da un comune all'altro)
out_movements_summary=intercomune_movements.groupby(['Start_Nome_Comune', 'End_Nome_Comune']).agg({'tot': 'sum'}).reset_index()


out_movements_summary=out_movements_summary.rename(columns={
    'tot': 'Total_Out_Movement',
})


out_movements_summary.head(2)
```

[92]:

	Start_Nome_Comune	End_Nome_Comune	Total_Out_Movement
0	Abbadia San Salvatore	Abetone Cutigliano	0.37
1	Abbadia San Salvatore	Arcidosso	918.64

[93]:
```
out_movements_summary.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31136 entries, 0 to 31135
Data columns (total 3 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Start_Nome_Comune   31136 non-null  object
 1   End_Nome_Comune     31136 non-null  object
 2   Total_Out_Movement  31136 non-null  float64
dtypes: float64(1), object(2)
memory usage: 729.9+ KB
```

  2. We merged the resulting DataFrame with tessellation to first on `Start_Nome_Comune` and then on `End_Nome_Comune` so at end we have a GeoDataFrame with information of both origin and destination

```
tessellation_renamed = tessellation.reset_index().rename(columns={'Nome_Comune': 'Start_Nome_Comune', 'geometry': 'Start_Geometry'})
```

```
Tot_InterMunic_Mov_gdf = tessellation_renamed.merge(out_movements_summary, on='Start_Nome_Comune', how='left')
```

```
tessellation_renamed = tessellation.reset_index().rename(columns={'Nome_Comune': 'End_Nome_Comune', 'geometry': 'End_Geometry'})
```

```
Tot_InterMunic_Mov_gdf=tessellation_renamed.merge(Tot_InterMunic_Mov_gdf, on='End_Nome_Comune', how='left')
```

```
Tot_InterMunic_Mov_gdf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 31136 entries, 0 to 31135
Data columns (total 21 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   index_x             31136 non-null  int64
 1   COD_COMUNE_x        31136 non-null  float64
 2   NO_x                31136 non-null  int64
 3   NAME_x              31136 non-null  object
 4   Cod_Provincia_x     31136 non-null  float64
 5   Cod_Regione_x       31136 non-null  float64
 6   End_Nome_Comune     31136 non-null  object
 7   Nome_Provincia_x    31136 non-null  object
 8   Nome_Regione_x      31136 non-null  object
 9   End_Geometry        31136 non-null  geometry
 10  index_y             31136 non-null  int64
 11  COD_COMUNE_y        31136 non-null  float64
 12  NO_y                31136 non-null  int64
 13  NAME_y              31136 non-null  object
 14  Cod_Provincia_y     31136 non-null  float64
 15  Cod_Regione_y       31136 non-null  float64
 16  Start_Nome_Comune   31136 non-null  object
 17  Nome_Provincia_y    31136 non-null  object
 18  Nome_Regione_y      31136 non-null  object
 19  Start_Geometry      31136 non-null  geometry
 20  Total_Out_Movement  31136 non-null  float64
dtypes: float64(7), geometry(2), int64(4), object(8)
memory usage: 5.2+ MB
```

3. We renamed the columns of the resulting DataFrame for better readability and then we save the file. The file cannot be saved as a GeoJSON as it has two geometry columns. To address this problem, we convert the geometry columns into WKT (Well-Known Text), a text markup language for representing vector geometry objects such as points, lines, and polygons. After this operation, the GeoDataFrame could be saved as a CSV file.

```
end_columns = {col: 'End_' + col.replace('_x', '') for col in Tot_InterMunic_Mov_gdf.columns if col.endswith('_x')}
start_columns = {col: 'Start_' + col.replace('_y', '') for col in Tot_InterMunic_Mov_gdf.columns if col.endswith('_y')}
renaming_dict = {**end_columns, **start_columns}
Tot_InterMunic_Mov_gdf = Tot_InterMunic_Mov_gdf.rename(columns=renaming_dict)
```

```
Tot_InterMunic_Mov_gdf.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Int64Index: 31136 entries, 0 to 31135
Data columns (total 20 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   index                 31136 non-null  int64
 1   End_COD_COMUNE        31136 non-null  float64
 2   End_NO                31136 non-null  int64
 3   End_NAME              31136 non-null  object
 4   End_Cod_Provincia     31136 non-null  float64
 5   End_Cod_Regione       31136 non-null  float64
 6   End_Nome_Comune       31136 non-null  object
 7   End_Nome_Provincia    31136 non-null  object
 8   End_Nome_Regione      31136 non-null  object
 9   End_Geometry          31136 non-null  geometry
 10  Start_COD_COMUNE      31136 non-null  float64
 11  Start_NO              31136 non-null  int64
 12  Start_NAME            31136 non-null  object
 13  Start_Cod_Provincia   31136 non-null  float64
 14  Start_Cod_Regione     31136 non-null  float64
 15  Start_Nome_Comune     31136 non-null  object
 16  Start_Nome_Provincia  31136 non-null  object
 17  Start_Nome_Regione    31136 non-null  object
 18  Start_Geometry        31136 non-null  geometry
 19  Total_Out_Movement    31136 non-null  float64
dtypes: float64(7), geometry(2), int64(3), object(8)
memory usage: 5.0+ MB
```

```
Tot_InterMunic_Mov_gdf['Start_Geometry'] = Tot_InterMunic_Mov_gdf['Start_Geometry'].apply(lambda geom: geom.wkt)
Tot_InterMunic_Mov_gdf['End_Geometry'] = Tot_InterMunic_Mov_gdf['End_Geometry'].apply(lambda geom: geom.wkt)
# a ogni geometria viene applicato una funzione lambda che converte la geometria in wkt che riesce a storare meglio in csv

Tot_InterMunic_Mov_gdf.to_csv("MovimentiUscita_InterMunic_Out.csv")
```

- **Creating and saving a reduced GeoDataFrame with Total Outgoing movements between municipalities**
  We created and saved a lighter version of the GeoDataFrame by dropping columns of less useful information.

```
Tot_Mov_ridotta_gdf=Tot_InterMunic_Mov_gdf.drop(columns=['End_index','End_NO','End_NAME','End_Cod_Provincia',
                                                         'End_Cod_Regione','End_Nome_Regione',
                                                        'Start_index','Start_Nome_Regione','Start_NAME','Start_Cod_Regione','Start_Cod_Provincia',
                                                        'Start_NO'])
```

```
Tot_Mov_ridotta_gdf.info()
```

```
<class 'pandas.core.frame.DataFrame'>•••
```

```
Tot_Mov_ridotta_gdf['Start_Geometry'] = Tot_Mov_ridotta_gdf['Start_Geometry'].apply(lambda geom: geom.wkt)
Tot_Mov_ridotta_gdf['End_Geometry'] = Tot_Mov_ridotta_gdf['End_Geometry'].apply(lambda geom: geom.wkt)

Tot_Mov_ridotta_gdf.to_csv("MovimentiUscitaRidotta_StartComune_EndComune.csv", index=False)
```

- **Using the function `load()` to convert a WKT back into a Shapely geometry object**
  When reading the file of GeoDataFrame stored as a CSV file, we used this code to parse through WKT strings and returns the corresponding Shapely geometry objects.

```
from shapely.wkt import loads

Tot_Mov_ridotta_gdf = pd.read_csv("MovimentiUscitaRidotta_StartComune_EndComune.csv")
Tot_Mov_ridotta_gdf['Start_Geometry'] = Tot_Mov_ridotta_gdf['Start_Geometry'].apply(loads)
Tot_Mov_ridotta_gdf['End_Geometry'] = Tot_Mov_ridotta_gdf['End_Geometry'].apply(loads)
```

2.3.2 Visualisations of Inbound and Outbound Movements (Figures 5 & 6)

1. From the GeoDataFrame with Total Outgoing movements between municipalities we grouped by the `Total_Out_Movement` on `Start_Nome_Comune` for Outbound movements.
2. We merged the resulting DataFrame with the original GeoDataFrame.
3. We reprojected the Coordinate Reference System in EPSG:4326 which is the WGS 84 system (latitude/longitude in degrees).
4. We converted the GeoDataFrame into a GeoJSON compatible format required by Altair for its `mark_geoshape` method to render geospatial shapes. The `geojson["features"]` contains the list of features (geometries and their properties) that Altair uses to build the visualization.
5. The map is encoded with colours that vary according to the magnitude of total outbound movements
6. The map is saved as an `html` file and displayed.
7. To create the map for inbound movements we followed the same procedure but the `groupby()` was done on `End_Nome_Comune`.

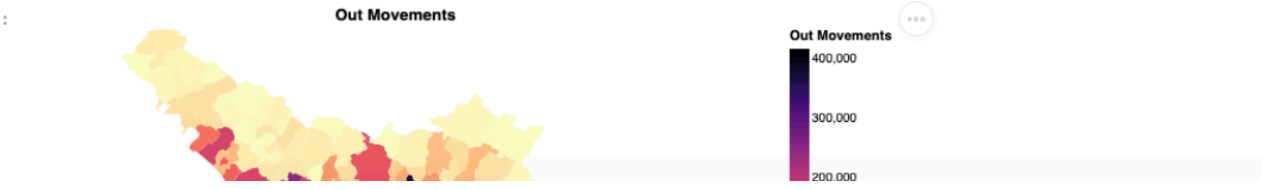
```
df_out=Tot_Mov_ridotta_gdf.groupby("Start_Nome_Comune")["Total_Out_Movement"].sum().reset_index()
```

```
df_merged = Tot_Mov_ridotta_gdf.drop_duplicates(subset=["Start_Nome_Comune"]).merge(
    df_out,
    left_on="Start_Nome_Comune",
    right_on="Start_Nome_Comune",
    how="left"
)
```

```
# Settiamo la geometria col suo crs
gdf_out = gpd.GeoDataFrame(df_merged, geometry="Start_Geometry", crs="EPSG:32632")
```

```
gdf_out=gdf_out[['Start_Nome_Comune','Start_Geometry','Total_Out_Movement_y']]
```

```
import altair as alt

gdf_out =gdf_out.to_crs(epsg=4326) #riproiettiamo in EPSG:4326 (WGS 84 – lat/lon)


geojson = gdf_out.__geo_interface__


chart = alt.Chart(alt.Data(values=geojson["features"])).mark_geoshape().encode(
    color=alt.Color("properties.Total_Out_Movement_y:Q", title="Out Movements",scale=alt.Scale(scheme='magma', reverse=True)),
    tooltip=[alt.Tooltip("properties.Start_Nome_Comune:N", title="Municipality"),
             alt.Tooltip("properties.Total_Out_Movement_y:Q", title="Out Movements")]
).properties(
    width=600,
    height=500,
    title="Out Movements"
)
chart.save('movimenti_uscita_comuni.html')
chart
```



2.3.3 Map of Flows between Municipalities (Figure 7)

To create this map, we used Altair and we followed these steps:

1. A **slider was added** to enable interactive filtering of movements by total volume. Using `alt.binding_range`, we set a range between 0 and 40,000, stepping in increments of 500. The slider was bound to a parameter named `movement_threshold`.

2. The dataset `Tot_Mov_ridotta_gdf` was **filtered** to exclude rows where the `Total_Out_Movement` value was below 500 and its geometries were reprojected to the EPSG:4326 CRS to ensure compatibility with web-based mapping tools.

3. **Centroids** for both `Start_Geometry` and `End_Geometry` were calculated with the `.centroid` property to represent origin and destination municipalities as single points.

4. A **new DataFrame**, `flow_data`, was created with relevant details for visualization.

5. Using Altair, **flow lines were plotted** with `mark_line()` between `start_lon`, `start_lat` and `end_lon`, `end_lat`. The line colour and thickness encode the `Total_Out_Movement` values, with higher volumes appearing more red and thicker. The movement slider was incorporated into the visualisation using `add_selection` and `transform_filter` so that flows were filtered based on user input.

6. **The tessellation was plotted** with `mark_geoshape()` and the two charts are combined

The full code for these operations is the following:

```python
movement_slider = alt.binding_range(min=0, max=40000, step=500, name='Movement Threshold:')
movement_selection = alt.param(name='movement_threshold',
    bind=movement_slider,
    value=3000)

Tot_Mov_ridotta_gdf_fil = Tot_Mov_ridotta_gdf[Tot_Mov_ridotta_gdf['Total_Out_Movement'] > 500]
Tot_Mov_ridotta_gdf_fil = Tot_Mov_ridotta_gdf_fil.set_geometry("End_Geometry")
Tot_Mov_ridotta_gdf_fil["End_Geometry"] = Tot_Mov_ridotta_gdf_fil["End_Geometry"].to_crs(epsg=4326)
Tot_Mov_ridotta_gdf_fil = Tot_Mov_ridotta_gdf_fil.set_geometry("Start_Geometry")
Tot_Mov_ridotta_gdf_fil["Start_Geometry"] = Tot_Mov_ridotta_gdf_fil["Start_Geometry"].to_crs(epsg=4326)

# Centroidi
Tot_Mov_ridotta_gdf_fil["Start_Centroid"] = Tot_Mov_ridotta_gdf_fil["Start_Geometry"].centroid
Tot_Mov_ridotta_gdf_fil["End_Centroid"] = Tot_Mov_ridotta_gdf_fil["End_Geometry"].centroid

# Prepariamo i dati dei flussi
flow_data = pd.DataFrame({
    "start_lon": Tot_Mov_ridotta_gdf_fil["Start_Geometry"].centroid.x,
    "start_lat": Tot_Mov_ridotta_gdf_fil["Start_Geometry"].centroid.y,
    "end_lon": Tot_Mov_ridotta_gdf_fil["End_Geometry"].centroid.x,
    "end_lat": Tot_Mov_ridotta_gdf_fil["End_Geometry"].centroid.y,
    "Total_Out_Movement": Tot_Mov_ridotta_gdf_fil["Total_Out_Movement"],
    "Start_Nome_Comune": Tot_Mov_ridotta_gdf_fil["Start_Nome_Comune"],
    "End_Nome_Comune": Tot_Mov_ridotta_gdf_fil["End_Nome_Comune"]
})

# Creiamo la mappa di Altair
flussi = alt.Chart(flow_data).mark_line().encode(
    longitude="start_lon:Q",
    latitude="start_lat:Q",
    longitude2="end_lon:Q",
    latitude2="end_lat:Q",
    color=alt.Color("Total_Out_Movement:Q", title="Movement Volume", scale=alt.Scale(scheme='reds')),
    size=alt.Size("Total_Out_Movement:Q", title="Line Thickness", scale=alt.Scale(range=[1, 8])),
    order=alt.Order("Total_Out_Movement:Q", sort="ascending"),
    tooltip=[
        alt.Tooltip("Start_Nome_Comune:N", title="From"),
        alt.Tooltip("End_Nome_Comune:N", title="To"),
        alt.Tooltip("Total_Out_Movement:Q", title="Movement Volume")
    ]
).add_selection(
    movement_selection
).transform_filter(
    alt.datum.Total_Out_Movement > movement_selection
).properties(
    width=600,
    height=450,
    title=f"Out Movement Flows"
)

# Convert la tessellation in EPSG:4326 nel GeoJSON
tessellation = tessellation.to_crs(epsg=4326)
geojson = tessellation.__geo_interface__

# Creiamo la Tessellazione
tess = alt.Chart(alt.Data(values=geojson["features"])).mark_geoshape(fill='white', stroke='lightgrey').encode(
    tooltip=alt.Tooltip("properties.Nome_Comune:N", title="Municipality")
).project(
    type="mercator"
).properties(
    width=600,
    height=450,
    title='Average Daily Movement flows'
)

# Combiniamo Flussi e Tessellazione
flows = (tess + flussi)

flows
```

Average Daily Movement flows

2.3.4 Map of Flows between Municipalities with Network Graph (Figure 8)

In this subchapter, we outline the steps to create a map of flows between municipalities using a network graph. To achieve this, we used the **NetworkX** library for creating the graph structure and the **Folium** library for the geospatial visualisation. The steps are as follows:

1. The dataset `Tot_Mov_ridotta_gdf` was filtered to retain only rows where `Total_Out_Movement` exceeded 5,000. The value of this threshold would depend on the scope of the analysis. In this case, we chose a relatively high number to avoid excessive cluttering of the map.
2. Using the `nx.DiGraph()` method of NetworkX, a directed graph structure was initialised to represent flows. We then iterated the rows in the filtered dataset to:
   - Add the start (`Start_Nome_Comune`) and end (`End_Nome_Comune`) municipalities as graph nodes.
   - Assign geometries to the respective nodes, and the centroids of this geometries were calculated using the `.centroid` property of the **Shapely** library
   - Add directed edges between nodes with weights corresponding to the `Total_Out_Movement` values. These edges represent the flow volumes between municipalities.
3. A Folium map was created, centred on Tuscany with the *CartoDB Positron* tile layer. The tessellation of municipalities was added with the `GeoJson()` function.
4. We drew the flows with the `PolyLine()` function of Folium using the properties `.x` and `.y` of Shapely to extract latitude and longitude of the centroids. The colormap is scaled by calculating the minimum and maximum weights and the line thickness is normalised between 20 and 1 for better visualisation.
5. We finally added municipality markers located on the centroids of the nodes using the function `CircleMarker()` of Folium.

The complete code for this visualization is as follows:

```
Tot_Mov_ridotta_gdf=Tot_Mov_ridotta_gdf[Tot_Mov_ridotta_gdf['Total_Out_Movement']>5000]
```

```
#fa un grafo diretto
import networkx as nx
G = nx.DiGraph()
for _, row in Tot_Mov_ridotta_gdf.iterrows():
    #aggiungo nodi
    G.add_node(row["Start_Nome_Comune"])
    G.add_node(row["End_Nome_Comune"])

    # a ogni nodo aggiungo geometria e centroidi
    G.nodes[row["Start_Nome_Comune"]]["geometry"] = row["Start_Geometry"]
    G.nodes[row["Start_Nome_Comune"]]["centroid"] = row["Start_Geometry"].centroid

    G.nodes[row["End_Nome_Comune"]]["geometry"] = row["End_Geometry"]
    G.nodes[row["End_Nome_Comune"]]["centroid"] = row["End_Geometry"].centroid

    #aggiungo gli archi
    G.add_edge(row["Start_Nome_Comune"], row["End_Nome_Comune"], weight=row["Total_Out_Movement"])
```

```
tessellation=gpd.read_file("/Users/Andre/Desktop/Internship/TessellationToscana.geojson", driver="GeoJSON")
```

```
import folium

latitude = 43.7696
longitude = 11.2558
m = folium.Map(location=[latitude, longitude], zoom_start=8, tiles='CartoDB positron')


folium.GeoJson(
    tessellation,
    style_function=lambda feature: {
        "color": "grey",
        "weight": 0.5,
        "fillColor": "white",
        "fillOpacity": 0.5
    },
    tooltip=folium.GeoJsonTooltip(
        fields=['NAME', 'Nome_Provincia'],
        aliases=['Name:', 'Nome_Provincia:'],
        localize=True
    )
).add_to(m)
```

```


all_weights = [G[u][v]["weight"] for u, v in G.edges()]
min_weight, max_weight = min(all_weights), max(all_weights)

colormap = cm.LinearColormap(['green', 'red'], vmin=min_weight, vmax=max_weight)
colormap.caption = "Flow Volume"
m.add_child(colormap)

for u, v in G.edges():
    start_centroid = G.nodes[u]["centroid"]
    end_centroid = G.nodes[v]["centroid"]
    weight = G[u][v]["weight"]


    if max_weight != min_weight:
        line_width = 1 + (weight - min_weight) / (max_weight - min_weight) * (20 - 1)
    else:
        line_width = 10


    line_color = colormap(weight)

    folium.PolyLine(
        locations=[
            (start_centroid.y, start_centroid.x),
            (end_centroid.y, end_centroid.x)
        ],
        color=line_color,
        weight=line_width,
        popup=f"{u} → {v}: {weight}",
    ).add_to(m)
```

```
for node in G.nodes():
    c = G.nodes[node]["centroid"]
    folium.CircleMarker(
        location=[c.y, c.x],
        radius=0.3,
        color="black",
        fill=True,
        fill_color="black",
        popup=node
    ).add_to(m)
```

2.3.5 Interactive Map of Outgoing Movements (Figure 9)

This visualisation integrates an interactive dropdown menu and combines a map and bar chart to explore outgoing movements patterns of each municipality. The steps to create the visualisation are as follows:

1. **A dropdown menu** was created to allow the user to select a municipality interactively. Using `alt.binding_select`, a dropdown was bound to a selection parameter named `click`, which is tied to the `Start_Nome_Comune` field. This enables filtering of data related to the selected municipality.

2. The dataset `Tot_Mov_ridotta_gdf` was filtered to include only the relevant fields: `Start_Nome_Comune`, `End_Nome_Comune`, and `Total_Out_Movement`. For each municipality, the top 10 destinations by total outgoing movement were identified using `nlargest(10,"Total_Out_Movement")`. This aggregated data was then used for the bar chart.

3. We created **a bar chart** that displays the top 10 destinations for outgoing movements from the selected municipality. The dropdown selection (`click`) was applied to filter the data dynamically.

4. A tessellation map was created using `mark_geoshape` to plot the boundaries of the municipalities with a **background map** that displays all municipalities with a white fill and grey border. We also added a **highlight layer:** When a municipality is selected from the dropdown, it is highlighted in black with a red border. The selection filter (`click`) makes sure that only the selected municipality is highlighted. The dataset was pre-processed by reprojecting it to EPSG:4326, and by renaming the municipality field to `Start_Nome_Comune` to ensure consistency with the dropdown selection.

5. A separate map of outgoing movments was created using `Tot_Mov_ridotta_gdf`, which was made interactive with `transform_filter(click)`.

6. The tessellation background map, highlight layer, and outgoing movements map were overlaid to create a **composite map**. Then the resulting map was vertically **concatenated** with the bar chart using `alt.vconcat`.

The full code is below:

```
7…  #Selettori
    options = sorted(Tot_Mov_ridotta_gdf["Start_Nome_Comune"].dropna().unique())
    dropdown = alt.binding_select(options=options, name="Select Comune: ")

    click = alt.selection_point(
        fields=["properties.Start_Nome_Comune"],
        bind=dropdown,
        value=[{"properties.Start_Nome_Comune": "Pisa"}]
    )

    #barchart
    df_flow = Tot_Mov_ridotta_gdf[["Start_Nome_Comune", "End_Nome_Comune", "Total_Out_Movement"]]
    filtered_data = df_flow.groupby("Start_Nome_Comune").apply(
        lambda x: x.nlargest(10, "Total_Out_Movement")).reset_index(drop=True)

    bars = (
        alt.Chart(filtered_data)
        .transform_calculate(**{
            "properties.Start_Nome_Comune": "datum.Start_Nome_Comune"
        })
        .transform_filter(click).add_selection(click)
        .mark_bar()
        .encode(
            x=alt.X("Total_Out_Movement:Q", title="Total Movement"),
            y=alt.Y("End_Nome_Comune:N", sort='-x', title="Ending Location"),
            tooltip=[
                alt.Tooltip("Total_Out_Movement:Q", title="Total Movement"),
                alt.Tooltip("properties.Start_Nome_Comune:N", title="Selected Comune")
            ]
        ).properties(
        width=600,
        height=400,
        title="Top 10 Ending Locations by Total Movement"
    ))

    # Background map & Highlights
    comuni=tessellation[['Nome_Comune','geometry']]
    comuni = comuni.rename(columns={"Nome_Comune": "Start_Nome_Comune"})
    comuni.to_crs(epsg=4326)
    comuni_geojson=comuni.__geo_interface__

    background = alt.Chart(alt.Data(values=comuni_geojson["features"])).mark_geoshape(
        fill="white",
        stroke="grey",
        strokeWidth=0.5
    ).project(
        type="mercator"
    ).properties(
        width=600,
        height=500
    )

    highlight = alt.Chart(alt.Data(values=comuni_geojson["features"])).mark_geoshape(
        fill="black",
        stroke="red",
        strokeWidth=0.5
    ).project(
        type="mercator"
    ).transform_filter(click).add_selection(click).properties(
        width=600,
        height=500
    )
```

```
# Mappa Movimenti
Tot_Mov_ridotta_gdf =Tot_Mov_ridotta_gdf.set_geometry("End_Geometry")
Tot_Mov_ridotta_gdf =Tot_Mov_ridotta_gdf.to_crs(epsg=4326)
Tot_Mov_ridotta_gdf=Tot_Mov_ridotta_gdf.drop(columns='Start_Geometry') #NB se non droppiamo non riesce a serializzare

geojson=json.loads(Tot_Mov_ridotta_gdf.to_json())

chart=alt.Chart(alt.Data(values=geojson["features"])).mark_geoshape().transform_filter(
    click
).encode(
    color=alt.Color(
        "properties.Total_Out_Movement:Q",
        title="Movimenti Uscita",
        scale=alt.Scale(scheme="spectral", reverse=True),
        legend=alt.Legend(title="Total movements", orient="right")
    ),
    tooltip=[
        alt.Tooltip("properties.End_Nome_Comune:N", title="Comune"),
        alt.Tooltip("properties.Total_Out_Movement:Q", title="Incoming movements"),
        alt.Tooltip("properties.Start_Nome_Comune:N", title="Comune of origin")
    ]
).project(
    type="mercator"
).properties(
    width=600,
    height=500,
    title="Total Out intercomune Movements"
)
#Sovrappone i tre
display = (background + chart + highlight)
#Concatena verticalmente con la barchart
final_display = alt.vconcat(
    display,
    bars
).resolve_legend(
    color="independent"
)

final_display
```

]:

2.3.6 Sankey of Outbounds Movements from Florence (Figure 10)

Sankey diagrams provide another powerful way to visualize flows between nodes. In this visualisation, a Sankey diagram was used to represent outgoing movements from Florence to other municipalities, with the thickness of the connections indicating the volume of the flows. The steps to create this diagram are as follows:

1. First, we filtered the Tot_Mov_ridotta_gdf GeoDataFrame to include only the movements from Florence with a magnitude of at least 3000 daily movements a day. The resulting `Out_Florence_Fil` is the gdf used to create the Sankey

```
Out_Florence=Tot_Mov_ridotta_gdf[Tot_Mov_ridotta_gdf['Start_Nome_Comune']=='Firenze']
```

```
Out_Florence_Fil=Out_Florence[Out_Florence['Total_Out_Movement']>3000]
```

2. A unique list of nodes (municipalities) was created by combining the unique values from both `Start_Nome_Comune` and `End_Nome_Comune`.

3. Each node was assigned a unique index to match the Sankey diagram's requirements by mapping the node names to their corresponding indices.

4. The source nodes and target nodes were mapped from the nodes' indices. The flow values were taken directly from the `Total_Out_Movement` column.

5. We used the library **Plotly** [39] to create a Sankey diagram:
   - The `go.Figure` function initialises a Plotly figure object.
   - The `go.Sankey` function defines the Sankey diagram and has two main components: `node` and `link`.
   - `Node` defines the properties of the nodes (municipalities) in the Sankey diagram.
   - `Link` defines the connections (flows) between nodes and uses the previously created `source`, `target`, and `value` mappings to represent the flows between nodes

Full code:

```python
import plotly.graph_objects as go

df=Out_Florence_Fil


nodes = list(set(df['Start_Nome_Comune']).union(set(df['End_Nome_Comune'])))
node_indices = {node: idx for idx, node in enumerate(nodes)}


sources = df['Start_Nome_Comune'].map(node_indices)
targets = df['End_Nome_Comune'].map(node_indices)
values = df['Total_Out_Movement']


fig = go.Figure(data=[go.Sankey(
    node=dict(
        pad=15,
        thickness=20,
        line=dict(color="black", width=0.5),
        label=nodes,
        color="blue"
    ),
    link=dict(
        source=sources,
        target=targets,
        value=values
    )
)])

fig.update_layout(title_text="Average Out Movements from Florence >3000", font_size=10)
fig.show()

fig.write_html("sankey_diagram.html")
```

2.6.7 Chord of Movements between Firenze, Pisa and Livorno (Figure 11)

Chord diagrams are an effective way to represent relationships and flows between entities, making them especially useful to quickly visualise movement patterns. We used the library **Holoviews** with the **Bokeh backend** enabled [40]. All steps are detailed below:

1. We generated the filtered dataset to be used for the diagram with this code:

```
Out_Florence_Pisa_Livorno=Tot_Mov_ridotta_gdf[(Tot_Mov_ridotta_gdf['Start_Nome_Comune']=='Firenze')
|(Tot_Mov_ridotta_gdf['Start_Nome_Comune']=='Pisa')
|(Tot_Mov_ridotta_gdf['Start_Nome_Comune']=='Livorno')]
Out_Florence_Pisa_Livorno_Fil=Out_Florence_Pisa_Livorno[Out_Florence_Pisa_Livorno['Total_Out_Movement']>5000]
```

2. We created a node DataFrame called `nodes` with unique names of the origin and destination municipality, and we indexed it.

3. We converted `nodes` into a **HoloViews Dataset**. The `kdims='index'` specifies the key dimension, which is used for plotting or linking data points. The `vdims='name'` specifies the value dimension, which contains descriptive information (in this case, the municipality names).

4. We mapped each municipality name in the `Start_Nome_Comune` and `End_Nome_Comune` columns of `Out_Florence_Pisa_Livorno_Fil` to its corresponding index in `nodes`.

5. The `hv.Chord()` function of Holoviews expects two main inputs:
   - **Edges**: A DataFrame (or similar structure) containing a *source*, a *target*, and a *value*. We had them all stored in our df as `start_idx`, `end_idx`, and `Total_Out_Movement`.
   - **Nodes**: A dataset with all nodes, specifying a unique identifier for each node and a descriptive label or attribute. This information was stored in `nodes`.

6. The diagram's style was customised with the function `chord.opts()`

The full code used is below:

```
import holoviews as hv

hv.extension('bokeh')


df = Out_Florence_Pisa_Livorno_Fil


nodes = pd.DataFrame({'name': pd.concat([df['Start_Nome_Comune'], df['End_Nome_Comune']]).unique()})
nodes['index'] = nodes.index
nodes = hv.Dataset(nodes, kdims='index', vdims='name')


df['start_idx'] = df['Start_Nome_Comune'].map(nodes.data.set_index('name')['index'])
df['end_idx'] = df['End_Nome_Comune'].map(nodes.data.set_index('name')['index'])


chord = hv.Chord((df[['start_idx', 'end_idx', 'Total_Out_Movement']], nodes))


chord.opts(
    labels='name',
    edge_color='Total_Out_Movement',
    node_color='name',
    cmap='Category20',
    edge_cmap='reds',
    width=700,
    height=700,
    tools=['hover'],
    title='Out Movements from Florence, Pisa, and Livorno'
)


hv.save(chord, 'chord_diagram_FIPILI.html', backend='bokeh')
```

2.6.8 Bar Chart of Modes of Transports (Figure 12)

Bar charts can be quickly created and customised using **Altair**. To generate the chart displayed in Figure 12 we followed this straightforward procedure:

1. We aggregated the `Mot` columns in `intercomune_movements` by summing the by summing the values in the column `tot` for each mode of transport and we reset the index. We then filtered out the modes of transport that are not widely available in Tuscany or are not identified:

```
Mot_movements= intercomune_movements.groupby('Mot').agg({'tot': 'sum'}).reset_index()
Mot_movements=Mot_movements[(Mot_movements['Mot']!='NotClassified') & (Mot_movements['Mot']!='Plane') & (Mot_movements['Mot'] != 'Tram')]
```

2. We created and customised the bar chart with the function `mark_bar()` of Altair:

```
import altair as alt

data = Mot_movements

chart = alt.Chart(data).mark_bar().encode(
    y=alt.Y('Mot:N',sort='-x'),
    x=alt.X('tot:Q'),
    color=alt.Color('tot:Q', scale=alt.Scale(scheme='reds'), title="Movement Intensity"),
    tooltip=[alt.Tooltip('tot:Q', title='Total Movement')]
).properties(
    width=500,
    height=300,
    title="Movements by Means of Transport"
)

chart.show()
chart.save('movements_by_MOT.html')
```

2.6.9 Bar Chart of Percentage Mot Usage (Figure 13)

To visualize the percentage distribution of movements by mode of transport for the top 10 destinations, we used Altair to create a bar chart. These are the steps that we followed including the precursory data manipulation

1. We grouped the dataset `intercomune_movements` by `End_Nome_Comune` (destination municipality) and `Mot` (mode of transport), and the total movements (`tot`) were summed for each combination.
2. Movements labelled as `Plane` and `NotClassified` were removed to focus on relevant modes of transport.
3. The total movements (irrespectively of the Mode of Transport) were calculated for each destination municipality with a `groupby('End_Nome_Comune')['tot']` and by using the `transform('sum')` to retain the original structure of the original DataFrame
4. The percentage of movements for each mode of transport within each destination was calculated and added as a new column.
5. The dataset was grouped again by `End_Nome_Comune` to sum all movements for each destination. This was used to rank and select the top 10 destinations.
6. The dataset was filtered to include only the top 10 destinations identified in the previous step.
7. The percentage of movements by mode of transport for each destination was visualised using Altair.

Full code below:

```
EndComune_movements= intercomune_movements.groupby(by=['End_Nome_Comune','Mot']).agg({'tot': 'sum'}).reset_index()
```

```
EndComune_movements=EndComune_movements[(EndComune_movements['Mot']!='Plane')&(EndComune_movements['Mot']!='NotClassified')]
```

```
total_movements = EndComune_movements.groupby('End_Nome_Comune')['tot'].transform('sum')
total_movements
```

```
0 14074.52 •••
```

```
EndComune_movements['percent'] = (EndComune_movements['tot'] / total_movements) * 100
```

```
EndComune_movements
```

```
<div> •••
```

```
top_destinations = EndComune_movements.groupby('End_Nome_Comune', as_index=False)['tot'].sum()
top_destinations = top_destinations.sort_values(by='tot', ascending=False).head(10)
top_destination_list = top_destinations['End_Nome_Comune'].tolist()
```

```
data = EndComune_movements[EndComune_movements['End_Nome_Comune'].isin(top_destination_list)]


chart = alt.Chart(data).mark_bar().encode(
    x=alt.X('percent:Q', title="% Movements"),
    y=alt.Y('End_Nome_Comune:N', sort=top_destination_list, title="Destination"),
    color=alt.Color('Mot:N', title="Mode of Transport"),
    order=alt.Order('percent:Q', sort='descending'),
    tooltip=[alt.Tooltip('percent:Q', title="% Movements"), alt.Tooltip('Mot:N', title="Mode of Transport")]
).properties(
    width=600,
    height=800,
    title="Percentage of Movements by Destination and Mode of Transport"
)

chart.show()
chart.save('Mot%_Top10Municipalities.html')
```

2.6.10 Maps of Percentage of Train Usage (Figures 14-15)

To create these two maps, we used Folium, following this procedure:

1. We followed steps 1–4 outlined in Section 2.6.9. The dataset was then filtered to include only train movements, and subsequently merged with the tessellation of Tuscany. The resulting DataFrame was converted into a GeoDataFrame. The code for this preparatory stage is below:

```
EndComune_movements= intercomune_movements.groupby(by=['End_Nome_Comune','Mot']).agg({'tot': 'sum'}).reset_index()
```

```
EndComune_movements=EndComune_movements[(EndComune_movements['Mot']!='Plane')&(EndComune_movements['Mot']!='NotClassified')]
```

```
total_movements = EndComune_movements.groupby('End_Nome_Comune')['tot'].transform('sum')
total_movements
```

```
0 14074.52 •••
```

```
EndComune_movements['percent'] = (EndComune_movements['tot'] / total_movements) * 100
```

```
EndComune_movementsTrain=EndComune_movements[EndComune_movements['Mot']=='Train']
```

```
tessellation=gpd.read_file("/Users/Andre/Desktop/Internship/TessellationToscana.geojson")
```

```
EndComune_movementsTrain_gdf = EndComune_movementsTrain.merge(tessellation, left_on='End_Nome_Comune', right_on='Nome_Comune', how='inner')
```

```
EndComune_movementsTrain_gdf= gpd.GeoDataFrame(
    EndComune_movementsTrain_gdf,
    geometry=EndComune_movementsTrain_gdf["geometry"]
)
```

```
EndComune_movementsTrain_gdf = EndComune_movementsTrain_gdf.set_crs(epsg=4326)
```

2. The Shapefile of the rail tracks and the train stations [36] were imported directly into Python. The Stations' GeoDataFrame had 77 null values in the category column, the one needed for the visualisation. Further analysis revealed that 54 these stations were disused and 23 of them were managed by the local train company *TFT*. To ensure consistency, we decided to manually reassign the nominal value "bronze" (i.e. minor stations) to the latter. The following code was used for this manipulation:

```
LineeFerr_gdf = gpd.read_file("internship/ferrovie/Ferrovie.shp")
```

```
TutteStazioni=gpd.read_file("internship/ferrovie/TutteStazioni.shp")
```

```
TutteStazioni.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 258 entries, 0 to 257
Data columns (total 19 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   pk_uid     258 non-null    int64
 1   cod_stz    258 non-null    object
 2   den_uff    258 non-null    object
 3   cod_gnz    258 non-null    object
 4   x_stz      258 non-null    float64
 5   y_stz      258 non-null    float64
 6   tip_gnz    258 non-null    object
 7   gnz        258 non-null    object
 8   org        258 non-null    object
 9   origine    258 non-null    object
 10  indirizzo  253 non-null    object
 11  comune     258 non-null    object
 12  provincia  258 non-null    object
 13  gestore    204 non-null    object
 14  categoria  181 non-null    object
 15  cod_sta    258 non-null    object
 16  data_agg   258 non-null    object
 17  data_elab  258 non-null    object
 18  geometry   258 non-null    geometry
dtypes: float64(2), geometry(1), int64(1), object(15)
memory usage: 38.4+ KB
```

```
TutteStazioni['gestore'].unique()
```

```
array(['RFI', None, 'TFT', 'Centostazioni', 'GrandiStazioni'],
      dtype=object)
```

```
TutteStazioni.loc[TutteStazioni['gestore'] == 'TFT', 'categoria'] = 'bronze'
```

```
TutteStazioni=TutteStazioni.to_crs(epsg=4326)
LineeFerr_gdf=LineeFerr_gdf.to_crs(epsg=4326)
```

3. The final map that combines multiple layers was created with Folium:

   - A base map centred on Tuscany was initialised using **Folium** with the `CartoDB positron` tile set.
   - We added the tessellation layer representing the municipalities in Tuscany. A `style_function` was applied to render all polygons in green and low opacity, creating a subtle overlay. Tooltips were added to display the name of the municipality and its province when hovered over.
   - A `LinearColormap()` was created to represent the percentage of train usage (percent field in `EndComune_movementsTrain_gdf`) across municipalities. A `movement_style_function` dynamically applied colours to municipalities based on their train usage percentage.
   - A GeoJSON layer of railways (`LineeFerr_gdf`) was added, styled with a black colour line.
   - Train stations were filtered to include only those categorised as bronze, silver, gold, or platinum. These categories were sorted in ascending order of importance using a mapping. A helper function, `get_color`, assigned colours to stations based on their category. With a `for` cycle that iterates through the rows of Stations' GeoDataFrame, `CircleMarkers` were added for each station, sized appropriately and styled with colours based on their category. Tooltips displayed the station's name and category.

Full code below:

```python
import folium
import branca.colormap as cm

latitude = 43.7696
longitude = 11.2558
map_f = folium.Map(location=[latitude, longitude], zoom_start=8, tiles='CartoDB positron')

#Tessellazione
def style_function(feature):
        return {
            'fillColor': 'green',
            'color': 'green',
            'weight': 0.5,
            'fillOpacity': 0.1,
        }
folium.GeoJson(
    tessellation,
    style_function=style_function,
    tooltip=folium.GeoJsonTooltip(
        fields=['NAME', 'Nome_Provincia'],
        aliases=['Name:', 'Province:'],
        localize=True
    )
).add_to(map_f)

#Movimenti in Treno
colormap = cm.LinearColormap(
    colors=['lightblue', 'yellow', 'orange', 'red'],
    vmin=EndComune_movementsTrain_gdf['percent'].min(),
    vmax=EndComune_movementsTrain_gdf['percent'].max()
)

colormap.caption = "Train Usage in % of Total Movements"
colormap.add_to(map_f)

def movement_style_function(feature):
    out_movement = feature['properties']['percent']
    color = colormap(out_movement)
    return {
        'fillColor': color,
        'color': 'gray',
        'weight': 0.5,
        'fillOpacity': 0.6,
    }

folium.GeoJson(
    EndComune_movementsTrain_gdf,
    style_function=movement_style_function,
    tooltip=folium.GeoJsonTooltip(
        fields=['Nome_Comune', 'percent'],
        aliases=['Comune:', '% Train'],
        localize=True
    )
).add_to(map_f)
```

```
#Linee Ferroviarie
folium.GeoJson(
    LineeFerr_gdf,
    tooltip=folium.GeoJsonTooltip(
        fields=['pk_uid', 'den_uff'],
        aliases=['ID:', 'Type:']
    ),
    style_function=lambda x: {
        'weight': 1.5,
        'color': 'black',
        'opacity': 0.7
    }
).add_to(map_f)

#Stazioni
TutteStazioni_fil = TutteStazioni[
    TutteStazioni['categoria'].isin(['bronze', 'silver', 'gold', 'platinum'])
].sort_values(by='categoria', key=lambda x: x.map({'bronze': 0, 'silver': 1, 'gold': 2, 'platinum': 3}))
def get_color(category):
    color_map = {
        'bronze': 'brown',
        'silver': 'gray',
        'gold': 'lime',
        'platinum': 'white',
    }
    return color_map.get(category.lower())

for idx, row in TutteStazioni_fil.iterrows():
    lat = row.geometry.y
    lon = row.geometry.x
    Name= row.den_uff.capitalize()
    category=row.categoria
    color = get_color(category)
    if not color:
        continue

    tooltip= f"Stazione: {Name}   Categoria: {category}"

    folium.CircleMarker(
        location=[lat, lon],
        radius=2.5,
        opacity=0.6,
        color=color,
        fill=True,
        fill_color=color,
        fill_opacity=1,
        tooltip=tooltip
    ).add_to(map_f)


map_f.save("Stazioni+Ferrovie.html")


map_f
```

2.4 Methods for Chapter 1.5 (A more Detailed Perspective: the FI-PI-LI motorway)

The methods used to obtain the results discusses in Chapter 1.5 can be divided into 5 main subsections.

2.4.1 Preparation of the Shapefile of the *Firenze-Lastra a Signa* Segment

This subsection describes the filtering steps of the FI-PI-LI motorway shapefile with the goal to isolate a specific road section for further analysis, namely the *Firenze-Lastra a Signa* segment.

1. A Shapefile of the whole FI-PI-LI motorway was prepared using *QGIS* from open Datasets provided by the Tuscany regional administration [36].

2. The Shapefile was read in Python as a GeoDataFrame and then filtered to retain only carriageways while excluding elements such as junctions and ramps. We visualised the resulting GeoDataFrame using Folium (this visualisation was used to create Figure 16). For this step, we used the following code:

```
fipili_area_gdf = gpd.read_file("internship/ArealeRistretto_fipili/fipili/fipili_.shp")
```

```
fipili_area_gdf['tipo_ele'].unique()
```

```
array(['di tronco carreggiata', 'raccordo, bretella, svincolo',
       'di rotatoria'], dtype=object)
```

```
fipili_carreggiata_gdf=fipili_area_gdf[fipili_area_gdf['tipo_ele']=='di tronco carreggiata']
```

```
import folium
latitude = 43.7696
longitude = 11.2558
map_f = folium.Map(location=[latitude, longitude], zoom_start=8, tiles='CartoDB positron')
def style_function(feature):
    return {
        'fillColor': 'black',
        'color': 'darkred',
        'weight': 4,
        'fillOpacity': 0.8,
    }
folium.GeoJson(
    fipili_carreggiata_gdf,
    style_function=style_function,
    tooltip=folium.GeoJsonTooltip(
        fields=['pk_uid', 'tipo_ele'],
        aliases=['ID:', 'Type:']
    )
).add_to(map_f)

map_f.save('FIPILI.html')
map_f
```

3. A buffer of 10 metres was applied to all the geometries of the GeoDataFrame to account for potential inaccuracies of GPS signals. Junctions within the area of interest that were still present despite step 2 were then manually removed.

```
fipili_carreggiata_gdf['geometry'] = fipili_carreggiata_gdf['geometry'].buffer(10)
```

```
latitude = 43.7696
longitude = 11.2558
map_f = folium.Map(location=[latitude, longitude], zoom_start=8, tiles='CartoDB positron')

folium.GeoJson(
    fipili_carreggiata_gdf,
    tooltip=folium.GeoJsonTooltip(
        fields=['pk_uid', 'tipo_ele'],
        aliases=['ID:', 'Type:']
    )
).add_to(map_f)


map_f
```

```
<div style="width:100%;"><div style="position:relative;width:100%;height:0;padding-bottom:60%;"><span s
```

```
svincoli=[382792,382793,383348,383860,223649,223602]
fipili_carreggiata_gdf=fipili_carreggiata_gdf[~fipili_carreggiata_gdf['pk_uid'].isin(svincoli)]
```

```
fipili_carreggiata_gdf.to_file('internship/ArealeRistretto_FIPILI/Buffer10m/FIPILI_Buffer_10m.shp')
```

```
fipili_buffer_gdf=gpd.read_file('internship/ArealeRistretto_FIPILI/Buffer10m/FIPILI_Buffer_10m.shp')
```

4. From the buffered dataset we manually selected the geometries at the northern, southern, eastern, and western extremes of the *Firenze-Lastra a Signa* section based on their unique identifiers (`pk_uid`). These geometries were then merged into a single shape using the `unary_union` function from Shapely. Finally, a bounding rectangular box was created to include all the selected geometry.

```
from shapely.ops import unary_union
from shapely.geometry import box
selected_geometries = fipili_buffer_gdf[
    fipili_buffer_gdf['pk_uid'].isin([382845, 382844,223583,223636,392711])]['geometry']

combined_geometry = unary_union(selected_geometries) #uniamo tutte le geometrie selezionate sopra in un'unica geometria

bounding_box = combined_geometry.bounds #questo fa i confini del box (rettangolare) delle geometrie selezionate a mano col min e max e unite
bounding_box_geom = box(*bounding_box)  # Converte il box in un poligono di shapely
```

5. All the geometries in the FI-PI-LI included in the *Firenze-Lastra a Signa* bounding box segments were selected using the `.within()` method. The filtered geometries were further simplified by dissolving boundaries within the same segment using the `.dissolve()` method. Each filtered segment was saved to a separate shapefile for further analysis. The resulting GeoDataFrame is displayed using Folium (used to generate Figure 17) and saved as a Shapefile.

```
fipili_Firenze_Lastra = fipili_buffer_gdf[fipili_buffer_gdf['geometry'].within(bounding_box_geom)]
```

```
latitude = 43.7696 •••
```

```
fipili_Firenze_Lastra=fipili_Firenze_Lastra.dissolve()
```

```
latitude = 43.7696
longitude = 11.2558
map_f = folium.Map(location=[latitude, longitude], zoom_start=8, tiles='CartoDB positron')

def style_function(feature):
    return {
        'fillColor': 'black',
        'color': 'darkred',
        'weight': 4,
        'fillOpacity': 0.8,
    }
folium.GeoJson(
    fipili_Firenze_Lastra,
    style_function=style_function,
    tooltip=folium.GeoJsonTooltip(
        fields=['pk_uid', 'tipo_ele'],
        aliases=['ID:', 'Type:']
    )
).add_to(map_f)


map_f
```

2.4.2 Reading and Preprocessing GPS data

The available GPS datasets consisted of a year's worth of signals from 2019 of northern Tuscany, divided into daily sets. Each one of these datasets contained between 650000 and 1100000 records. Due to the computational expense and time required to process such a large volume of data, we focused our analysis on a smaller subset. Specifically, we focussed our attention on a group of commuters of the *Firenze-Lastra a Signa* segment and examined their patterns and behaviour over a one-month period. This analysis is therefore intended to serve as a proof of concept for a more comprehensive study to be conducted when resources allow.

This section outlines the methods employed to prepare the GPS dataset that was subsequently used to identify the aforementioned group of commuters.

1. To ensure the data was representative and not influenced by anomalies, we decided to analyse GPS data from a standard week without national holidays, public transport strikes, and major accidents on the road segment of interest. The second week of February 2019 was selected for this purpose.

2. The files were stored in external server so a secure SSH connection to the remote server was established using the **Paramiko** library that provides both client and server functionality [43].

    - For each date in the list, the corresponding file path was constructed and accessed via SFTP. The file was then opened and decompressed using `gzip` to read its contents line by line.
    - The decompressed data was processed in chunks of 100,000 lines for efficiency and scalability. Once a chunk reached 100,000 lines, it was converted into a Pandas DataFrame and stored in a list for further processing. After all chunks for a specific file were processed, they were concatenated into a single DataFrame for the day.
    - These daily DataFrames were stored in a dictionary (`dataframes`) with dates as keys for easy access. This allowed a parallel processing in the following steps

```python
import paramiko
import gzip
import io
import traceback

dates = ['20190211', '20190212', '20190213', '20190214', '20190215', '20190216', '20190217']
hostname =
username = "avitali"

dataframes = {}

try:
    # Apriamo le connessioni SSH & SFTP
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh.connect(hostname, username=username)

    sftp = ssh.open_sftp()

    for date in dates:
        remote_file_path =

        try:
            with sftp.open(remote_file_path, "rb") as remote_file:
                with gzip.GzipFile(fileobj=remote_file) as decompressed_file:

                    chunks = []
                    all_dataframes = []  # Processiamo per chunks e li salviamo qua
                    print(f"Reading lines from the file for {date}...")

                    for i, line in enumerate(decompressed_file):
                        chunks.append(line.decode("utf-8").strip())

                        # Processiamo in chunks di 100000
                        if len(chunks) >= 100000:
                            print(f"Processing chunk at line {i + 1} for {date}...")
                            dataframe_chunk = pd.read_csv(io.StringIO("\n".join(chunks)), header=None)
                            all_dataframes.append(dataframe_chunk)  # Salvo il chunk
                            chunks = []  # Pulisce buffer

                    # Processa le linee rimanenti se le linee non sono divisibili per 100000
                    if chunks:
                        print(f"Processing the final chunk for {date}...")
                        dataframe_chunk = pd.read_csv(io.StringIO("\n".join(chunks)), header=None)
                        all_dataframes.append(dataframe_chunk)  # Salva il chunk finale

                    # Combina tutti i chunk in in un df
                    print(f"Combining all chunks into a final DataFrame for {date}...")
                    final_dataframe = pd.concat(all_dataframes, ignore_index=True)

                    print(f"Final DataFrame for {date} (showing first 3 rows):")
                    print(final_dataframe.head(3))
                    dataframes[date] = final_dataframe  # Mette il df in un dizionario

        except Exception as file_error:
            print(f"Error occurred while processing the file for {date}: {file_error}")
            traceback.print_exc()

    # Chiude le connessioni SFTP and SSH
    sftp.close()
    ssh.close()

except Exception as e:
    print(f"An error occurred: {e}")
    traceback.print_exc() #mi dice dove è venuto l'errore
```

3. We checked the structure of the DataFrames and we noticed that they did not have a geometry column and that their latitude and longitude columns had coordinates multiplied by a scaling factor of 1000000 maybe for storage purposes

```
Final DataFrame for 20190217 (showing first 3 rows):
    0           1         2         3         4   5    6  7  8     9
0  95  2019-02-16  22:08:53  43720798  10949104   0    0  3  2  1107
1  95  2019-02-17  09:37:30  43720987  10948929   0    0  1  0     0
2  95  2019-02-17  09:41:18  43720155  10958623  36  170  3  1  2017
```

```python
for date, df in dataframes.items():
    print(date, len(df))
```

```
20190211 1025094
20190212 1039055
20190213 1065370
20190214 1107852
20190215 1121731
20190216 1034149
20190217 859602
```

4. The latitude and longitude columns were divided by 1000000 to convert them back to decimal degrees. These values were then used to create a `geometry` column that contains `Point` objects. The DataFrames were converted into GeoDataFrames and their CRS is set to EPSG:4326, the standard for latitude/longitude coordinates. The resulting GeoDataFrames are stored in a new dictionary called `gdfs` with the dates as keys.

```
# Dividiamo le coordinate per 1 milione per tutto il dizionario
for date, df in dataframes.items():
    df[3] = df[3] / 1_000_000
    df[4] = df[4] / 1_000_000
    print(f"df for {date}:{df.head(2)}")
```

```
df for 20190211: 0 1 2 3 4 5 6 7 8 9 •••
```

```
## Creiamo il gdf

from shapely.geometry import Point
gdfs={}
for date, df in dataframes.items():
    geometry = gpd.points_from_xy(df[4], df[3])
    gdf = gpd.GeoDataFrame(df, geometry=geometry)
    gdf.set_crs(epsg=4326, inplace=True)
    gdfs[date]=gdf
    print(gdfs[date].head(2))
    print(f"{date,df} converted")
```

5. After resetting the index of each `GeoDataFrame` in the `gdfs` dictionary and removing the old index to make sure that the new index is sequential and starts from 0, we replace the numerical column headers with the descriptive names based on the metadata provided with the dataset for better clarity and readability.

```
for date, gdf in gdfs.items():
    gdfs[date] = gdf.reset_index(drop=True)
```

```
import matplotlib.pyplot as plt •••
```

```
for date, gdf in gdfs.items(): •••
```

```
#N.B. Non salva con colonne numeriche!

for date, gdf in gdfs.items():
    gdf.columns = ['ID', 'Date', 'Time', 'Latitude', 'Longitude', 'Speed', 'Heading', 'Quality', 'PanelSession', 'DeltaPos', 'geometry']
    gdfs[date]= gdf
```

6. Since the data's provider recommended to keep only the signals with a quality of 3, we filtered the GeoDataFrames accordingly and then dropped the `Quality` column.

```
for date, gdf in gdfs.items():
    print(date, gdf['Quality'].unique())
```

```
20190211 [3 1 2]
20190212 [3 1 2]
20190213 [1 3 2]
20190214 [3 1 2]
20190215 [3 1 2]
20190216 [3 1 2]
20190217 [3 1 2]
```

```
for date, gdf in gdfs.items():
    gdf2=gdf[gdf['Quality']==3]
    gdf2=gdf2.drop(columns=['Quality'])
    gdfs[date]=gdf2
```

```
for date, gdf in gdfs.items():
    print(len(gdf))
```

```
890180
901239
925858
963612
977355
904101
764563
```

7. By analysing the dates within each daily GeoDataFrame, we discovered that approximately 10-12% of the records belonged to other days. To ensure consistency, we filtered each dataset to retain only the records corresponding to the specific day it represents.

```
for date, gdf in gdfs.items():
    unique_dates = gdf['Date'].unique()
    print(f"Unique Dates: in {date}: {unique_dates}")
    data=f"{date[0:4]}-{date[4:6]}-{date[6:8]}"
    gdf1=gdf[gdf['Date']==data]
    gdfs[date]=gdf1
```

```
Unique Dates: in 20190211: ['2019-02-09' '2019-02-11' '2019-02-10' '2019-02-06' '2019-02-08' •••
```

```
for date, gdf in gdfs.items():
    unique_dates = gdf['Date'].unique()
    print(f"Unique Dates: in {date}: {unique_dates}")
```

```
Unique Dates: in 20190211: ['2019-02-11']
Unique Dates: in 20190212: ['2019-02-12']
Unique Dates: in 20190213: ['2019-02-13']
Unique Dates: in 20190214: ['2019-02-14']
Unique Dates: in 20190215: ['2019-02-15']
Unique Dates: in 20190216: ['2019-02-16']
Unique Dates: in 20190217: ['2019-02-17']
```

```
for date, gdf in gdfs.items():
    print(len(gdf))
```

```
817467
826184
849465
884131
890294
819406
679852
```

8. The next step was to process each daily GeoDataFrame to retain only the data of drivers that used the *Firenze-Lastra a Signa* on that specific day. This was achieved by writing a code that iterates through the `gdfs` dictionary and performs the following operations:

   - Converts the CRS of each daily GeoDataFrame `gdf` to match that of `fipili_area_FI_Lastra` to ensure spatial compatibility.
   - Performs a spatial join to identify the Geospatial points of `gdf` that fall fully within `fipili_area_FI_Lastra` area.
   - Aggregates the points retained in the previous step on the `ID` field, to count how many signals were recorded for each driver within our road section of interest.
   - Creates a list called `valid_ID` that includes only IDs with three or more GPS signals within the `fipili_area_FI_Lastra` area. This threshold was chosen to exclude drivers that might have passed near, below or above the road segment without using it.
   - Filters the original GeoDataFrame to include only rows where the `ID` is in the `valid_ID` list. This step ensures that only valid signals for the specific day are retained.
   - Saves the filtered GeoDataFrame back into the new dictionary `gdfs_FI_LASTRA`.

```
gdfs_FI_LASTRA={}
for date, gdf in gdfs.items():
    gdf = gdf.to_crs(fipili_area_FI_Lastra.crs)
    points_in_area = gpd.sjoin(gdf, fipili_area_FI_Lastra, how="inner", predicate="within")
    uid_counts = points_in_area.groupby('ID').size()
    print(f"Number of points within FI-LASTRA on {date}: {len(points_in_area)}")
    print(f"Number of Unique IDs using FI-LASTRA on {date}:{points_in_area['ID'].nunique()}")
    valid_ID=uid_counts[uid_counts>=3].index
    gdf_FI_LASTRA = gdf[gdf['ID'].isin(valid_ID)]
    print(f"{date}: Filtered GeoDataFrame has {len(gdf_FI_LASTRA)} rows.")
    gdfs_FI_LASTRA[date]=gdf_FI_LASTRA
```

9. We decided to create the `Timestamp` column at this stage and not later to address a potential issue previously encountered. Occasionally, when downloading saved datasets from the server, an error would occur, resulting in the dataset being retrieved without the `Time` column, which is essential for analysing trajectories. By addressing this now, we made sure to avoid problems in later stages of analysis. To achieve this, we iterated through the dictionary `gdfs_FI_LASTRA` and for each GeoDataFrame we created a new column that combined the two columns `Date` and `Time`.

```
gdfs_FI_LASTRA_Timestamps = {}
for date, gdf in gdfs_FI_LASTRA.items():
    gdf['Timestamp'] = gdf['Date'] + ' ' + gdf['Time']
    print(f"{date} {gdf.head(1)}")
    gdfs_FI_LASTRA_Timestamps[date] = gdf
```

10. We saved the GeoDataFrames both as individual datasets and as a single concatenated weekly dataset on a remote server using the Paramiko library. After establishing an SSH connection with the server, we opened an SFTP session for file transfer. For the individual datasets, we iterated through the dictionary where the GeoDataFrames are stored and then we saved them as GeoJSON files. For the weekly dataset, all daily GeoDataFrames were concatenated into a single file before saving. We ensured that the SFTP and SSH connections were properly closed, even in the event of an error, with a `finally` block.

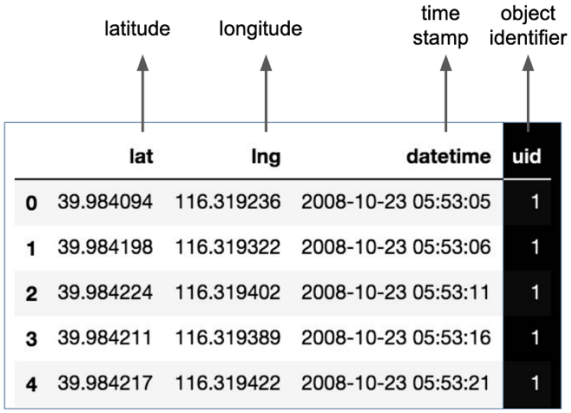
```python
# salviamo prima i gdfs singoli
import paramiko
import geopandas as gpd
import pandas as pd
import io
# Configurazione
hostname = 
username = "avitali"

try:
    #apro connessione ssh
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh.connect(hostname, username=username)

    sftp = ssh.open_sftp()


    for date, gdf in gdfs_FI_LASTRA_Timestamps.items():
        print(f"Saving gdf {date} with head: {gdf.head(1)}")
        with sftp.open(f"/home/avitali/Filtered_FI_LASTRA_{date}.geojson", "w") as file:
            file.write(gdf.to_json())

        print(f"GeoJSON file {date} saved successfully at /home/avitali/Filtered_FI_LASTRA_{date}.geojson")
        print("")


except Exception as e:
    print(f"An error occurred: {e}")

finally:
    #chiudo sdfp e ssh
    try:
        sftp.close()
        ssh.close()
    except Exception as cleanup_error:
        print(f"Error during cleanup: {cleanup_error}")
```

```python
## salvataggio del gdf concatenato della settimana

import paramiko
import geopandas as gpd
import pandas as pd
import io
# Configurazione
hostname = 
username = "avitali"
merged_output_path = "/home/avitali/Firenze_Lastra_filtered_2_Settimana_Febbraio.geojson"

try:
    #apro connessione ssh
    print("Connecting to SSH server...")
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh.connect(hostname, username=username)
    print("Connection to SSH server established")

    print("Opening sftp connection...")
    sftp = ssh.open_sftp()
    print("sftp connection opened")

    print("Concatenating gdfs...")
    merged_gdf = gpd.GeoDataFrame(pd.concat(gdfs_FI_LASTRA_Timestamps.values(), ignore_index=True))
    print(f"Gdfs concatenated and has {len(merged_gdf)} rows")

    print("Saving file...")
    with sftp.open(merged_output_path, "w") as merged_file:
        merged_file.write(merged_gdf.to_json())

    # Verifica della dimensione del file
    attrs = sftp.stat(merged_output_path)
    print(f"Merged GeoJSON file saved successfully at {merged_output_path}")
    print(f"File size: {attrs.st_size} bytes")


except Exception as e:
    print(f"An error occurred: {e}")

finally:
    #chiudo sdfp e ssh
    try:
        sftp.close()
        ssh.close()
    except Exception as cleanup_error:
        print(f"Error during cleanup: {cleanup_error}")
```

2.4.3 Analysis of Weekly Mobility Data of Firenze-Lastra a Signa's Users (Second Week of February 2019)

In this subsection we examine and discuss the methods used to analyse the dataset of GPS signals created for the second week of February 2019, as described in **2.4.2**.

1. After reading the weekly GeoDataFrame obtained in **2.4.2** using Paramiko, we converted the GeoDataFrame into a **Trajectory DataFrame** (`TrajDataFrame`), a data structure optimised for handling mobility data, using the `Scikit-mobility` (`Skmob`) library [45]. TrajDataFrames have the following structure [46]:



	lat	lng	datetime	uid
0	39.984094	116.319236	2008-10-23 05:53:05	1
1	39.984198	116.319322	2008-10-23 05:53:06	1
2	39.984224	116.319402	2008-10-23 05:53:11	1
3	39.984211	116.319389	2008-10-23 05:53:16	1
4	39.984217	116.319422	2008-10-23 05:53:21	1

Figure 1: Representation of a '**TrajDataFrame**' object. Each row represents a point of a moving object's trajectory, described by three mandatory columns (**lat**, **lng**, **datetime**) and by optional columns **uid** and **tid**, indicating the identifier of the moving object associated with the point and the trajectory identifier, respectively.

After transforming the `Timestamp` column of the GeoDataFrame into `datetime64` objects, the process of creating the TrajDataFrame is the quite straightforward:

```
gdf['Timestamp']=pd.to_datetime(gdf['Timestamp'])
```

```
gdf.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'> ...
```

```
from skmob import TrajDataFrame

tdf = TrajDataFrame(gdf, timestamp=True, latitude='Latitude', longitude='Longitude', datetime='Timestamp', user_id='ID')

print("\nTrajDataFrame:")
print(tdf.columns)
print(tdf.head())
```

```
TrajDataFrame:
Index(['id', 'uid', 'Date', 'Time', 'lat', 'lng', 'Speed', 'Heading',
       'PanelSession', 'DeltaPos', 'datetime', 'geometry'],
      dtype='object')
  id   uid        Date      Time        lat        lng  Speed  Heading  \
0  0  2564  2019-02-11  08:38:33  43.805330  11.115726     22      112
1  1  2564  2019-02-11  08:42:04  43.820942  11.121897     28      298
2  2  2564  2019-02-11  08:44:50  43.832805  11.111003     52      272
3  3  2564  2019-02-11  08:47:51  43.846563  11.108816     46      300
4  4  2564  2019-02-11  08:50:41  43.863286  11.109768     46       30

   PanelSession  DeltaPos            datetime                 geometry
0             1      2021 2019-02-11 08:38:33  POINT (11.11573 43.80533)
1             1      2017 2019-02-11 08:42:04  POINT (11.12190 43.82094)
2             1      2050 2019-02-11 08:44:50  POINT (11.11100 43.83281)
3             1      2040 2019-02-11 08:47:51  POINT (11.10882 43.84656)
4             1      2054 2019-02-11 08:50:41  POINT (11.10977 43.86329)
```

```
tdf = tdf.reset_index(drop=True)
```

```
tdf = tdf.sort_values(by=['uid', 'datetime'])
print('\nAnalysis:')
print('records:', len(tdf))
print('unique users:', len(tdf['uid'].unique()))
print('period:', tdf.datetime.min(), '-', tdf.datetime.max())
```

```
Analysis:
records: 319639
unique users: 1874
period: 2019-02-11 00:00:16 - 2019-02-17 22:56:51
```

2. We **identified the occasional and regular users** of the *Firenze-Lastra a Signa* segment by following these steps:
   - We aggregated the TrajectoryDataFrame by `uid` (user ID) user ID and calculated the number of unique dates each user appeared in the dataset. This resulted in a DataFrame called `usage_count` that tells us how many days a specific user drove on the road segment of interest.

   - We further aggregated `usage_count` by `days_used` to create a `grouped_counts` DataFrame. This returned 7 groups and allowed us to determine how many drivers used the road segment once, twice, and so on, up to seven days during the second week of February.

```
usage_counts = tdf.groupby('uid')['Date'].nunique().reset_index()
```

```
usage_counts.columns = ['uid', 'days_used']
```

```
usage_counts.sort_values(by='days_used',ascending=False).head(10)
```

```
<div> •••
```

```
usage_counts
```

```
<div> •••
```

```
grouped_counts=usage_counts.groupby('days_used').size().reset_index()
```

```
grouped_counts.columns=['days_used','uid']
```

```
grouped_counts
```

   - We used `grouped_counts` to create the Altair Bar and Pie charts for Figures 17 and 18.

```
import altair as alt
chart = alt.Chart(grouped_counts).mark_bar().encode(
    y=alt.Y('days_used:N', title='Days used'),
    x=alt.X('uid:Q'),
    color=alt.Color('uid:Q', scale=alt.Scale(scheme='reds'), title="Number of individual user"),
    tooltip=[alt.Tooltip('uid:Q', title='Number of Users')]
).properties(
    width=500,
    height=300,
    title="Frequency of use of the section Firenze-Lastra "
)
chart.save('Frequency of use of the section Firenze-Lastra.html')
chart.show()
```

```
<style> •••
```

```
pie_chart = alt.Chart(grouped_counts).mark_arc().encode(
    theta='uid:Q',
    color='days_used:N',
    tooltip=[
        alt.Tooltip('days_used:N', title='Days Used'),
        alt.Tooltip('uid:Q', title='Number of Users')
    ]
).properties(title="Proportions of Users for the Firenze-Lastra Section")
pie_chart.save('Pie Proportions of Users for the Firenze-Lastra Section.html')
pie_chart
```

   - We created a dictionary that organises users by the number of days they travelled on the *Firenze-Lastra a Signa* section. For each user ID we also created a list of unique dates when they were active on that road. We also wrote a short code to

easily access and analyse groups from the dictionary by dynamically creating a DataFrame for the selected group.

```
usage_dict = {}

for num_days in usage_counts_df['days_used'].unique():
    users_with_num_days = usage_counts_df[usage_counts_df['days_used'] == num_days]
    usage_dict[num_days] = {}
    for uid in users_with_num_days['uid']:
        user_data = tdf[tdf['uid'] == uid]
        unique_dates = user_data['Date'].unique()
        usage_dict[num_days][uid] = list(unique_dates)
```

```
print(usage_dict[7])
```

```
{35675: ['2019-02-11', '2019-02-12', '2019-02-13', '2019-02-14', '2019-02-15', '2019-02-16', '201
```

```
gruppo=input('Choose group:')
gruppo = int(gruppo)
rows=[]
for uid, dates in usage_dict[gruppo].items():
    for date in dates:
     rows.append({"uid": uid, "date": date})
dynamic_name = f"group_{gruppo}"
globals()[dynamic_name] = pd.DataFrame(rows)
```

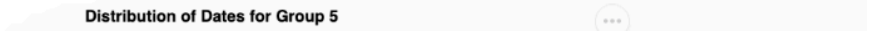
```
Choose group: 5
```

- After being aggregated by `date`, the groups in `usage_dict` were used to generate the bar charts displayed in Figure 19 and 20

```
import altair as alt


date_counts = group_1.groupby('date').size().reset_index(name='count')


chart = alt.Chart(date_counts).mark_bar(size=70).encode(
    x=alt.X('date:T', title='Date', timeUnit='yearmonthdate'),
    y=alt.Y('count:Q', title='Number of Users'),
    tooltip=[
        alt.Tooltip('date:T', title='Date'),
        alt.Tooltip('count:Q', title='Number of Users')
    ]
).properties(
    title="Distribution of Dates for Group 1",
    width=600,
    height=300
)
chart.save('Group5Usage.html')
chart.show()
```

<style> •••

```
import altair as alt


date_counts = group_5.groupby('date').size().reset_index(name='count')


chart = alt.Chart(date_counts).mark_bar(size=70).encode(
    x=alt.X('date:T', title='Date', timeUnit='yearmonthdate'),
    y=alt.Y('count:Q', title='Number of Users'),
    tooltip=[
        alt.Tooltip('date:T', title='Date'),
        alt.Tooltip('count:Q', title='Number of Users')
    ]
).properties(
    title="Distribution of Dates for Group 5",
    width=600,
    height=300
)
chart.save('Group5Usage.html')
chart.show()
```



3. We **identified the commuters** of the *Firenze-Lastra a Signa* section by following this procedure:

- From `usage_dict` we extract a list of `uid` of drivers that use the road segment 5 times a week.

```
gruppo = int(input("Choose group between 1 and 7: "))

users=[]
for uid, dates in usage_dict[gruppo].items():
    users.append(uid)

dynamic_name = f"users_group_{gruppo}"
globals()[dynamic_name] = users
print(f"users in group {gruppo}: {users}")
print(f"number of users in {gruppo}: {len(users)}")
```

```
Choose group between 1 and 7:  5
users in group 5: [5514, 16537, 37225, 37946, 59890, 62722, 69887, 89001, 89909, 89955, 120617, 12
, 277078, 289947, 294628, 304492, 307037, 321892, 324242, 352469, 354872, 355051, 355260, 361991,
85, 694607, 702805, 719688, 742889, 750084, 752564, 764720, 824618, 827335, 830176, 846562, 850911
number of users in 5: 85
```

- We filtered the original `TrajDataFrame` (`tdf`) to include only the users in group 5, creating a subset called `tdf_group_5`. Next, we further filtered this subset by

separating the data into two categories: weekdays (`weekdays_data`), which include specific dates from Monday to Friday, and weekend (`weekend_data`), which include specific Saturday and Sunday dates. This allows us to analyse user behaviour during weekdays and weekends separately.

```
tdf_group_5=tdf[(tdf['uid'].isin(users_group_5)) ]
```

```
weekdays=['2019-02-11','2019-02-12','2019-02-13','2019-02-14','2019-02-15']
weekend=['2019-02-16','2019-02-17']
weekdays_data = tdf_group_5[tdf_group_5['Date'].isin(weekdays)]
weekend_data = tdf_group_5[tdf_group_5['Date'].isin(weekend)]
```

- For the purpose of this study, we assumed that commuters were the drivers who used the road segment daily between Monday and Friday, but not at weekends. To isolate this group, we identified users who travelled during the weekend. These drivers were subsequently excluded from the dataset, leaving 47 unique users classified as weekday-only commuters.

```
weekend_ids=set(weekend_data['uid'])
```

```
commuters= tdf_group_5[~tdf_group_5['uid'].isin(weekend_ids)]
```

```
commuters['uid'].nunique()
```

```
47
```

```
commuters['uid'].unique()
```

```
array([  5514,  89001,  89909, 120617, 122711, 134643, 136295, 167834,
       186115, 188859, 194213, 241704, 253545, 259780, 261668, 269269,
```

4. Using Scikit-Mobility, we also performed some additional mobility analyses on this dataset, covering both individual measures (e.g. `max_distance_from_home`) and collective ones (e.g. `visits_per_time_unit`). However, we decided not to include these results in the report for Part 1. In case of interest, the full analysis is available [47].

2.4.4 Reading and Preprocessing Commuters' GPS data for February 2019

In this chapter we describe and explain the methods employed to create a GeoDataFrame that includes all GPS signals for the commuter group identified in **2.4.3** for the entire month of February 2019.

The code used to read and prepare the data replicates all the step done in **2.4.2,** apart from two main differences:

1. When reading the file, we filtered out the GPS signal that did not belong to IDs in the commuter group. Furthermore, since we had to process 28 large files remotely and run into some network issues, we included a retry mechanism that attempts multiple connections in case the initial one fails.

```
import paramiko
import gzip
import io
import traceback
import time

commuters = [
    5514, 69887, 89001, 89909, 120617, 122711, 134643, 136295,
    167834, 186115, 188859, 194213, 241704, 253545, 259780, 261668,
    269269, 270033, 272297, 289947, 304492, 324242, 352469, 354872,
    355260, 361991, 424643, 445765, 484424, 502875, 515060, 521426,
    556920, 569861, 569958, 570789, 584136, 642671, 694607, 702805,
    742889, 764720, 830176, 850911, 865983, 876930, 891366
]
dates = [
    '20190201', '20190202', '20190203', '20190204', '20190205',
    '20190206', '20190207', '20190208', '20190209', '20190210',
    '20190211', '20190212', '20190213', '20190214', '20190215',
    '20190216', '20190217', '20190218', '20190219', '20190220',
    '20190221', '20190222', '20190223', '20190224', '20190225',
    '20190226', '20190227', '20190228'
]
hostname = "xxxxxxxxxxxxxxxxxxxxxxxxxxxx"
username = "avitali"

dataframes = {}
RETRIES = 5

try:
    # apriamo le connessioni
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh.connect(hostname, username=username)
    sftp = ssh.open_sftp()

    for date in dates:
        remote_file_path = f"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx/MOB_SQUARE_EMILIA_{date}_0.csv.gz"

        attempt = 0
        while attempt < RETRIES:
            try:
                print(f"Attempt {attempt+1} to process {remote_file_path}...")

                with sftp.open(remote_file_path, "rb") as remote_file:
                    with gzip.GzipFile(fileobj=remote_file) as decompressed_file:
                        chunks = []
                        all_dataframes = []
                        print(f"Reading lines from the file for {date}...")

                        for i, line in enumerate(decompressed_file):
                            chunks.append(line.decode("utf-8").strip())
```

```
                        # chunks di 100,000
                        if len(chunks) >= 100000:
                            print(f"Processing chunk at line {i+1} for {date}...")
                            dataframe_chunk = pd.read_csv(io.StringIO("\n".join(chunks)), header=None)
                            filtered_chunk = dataframe_chunk[dataframe_chunk[0].isin(commuters)]
                            all_dataframes.append(filtered_chunk)
                            chunks = []

                    # Processa il chunk finale se non è divisibile per 100000
                    if chunks:
                        print(f"Processing final chunk for {date}...")
                        dataframe_chunk = pd.read_csv(io.StringIO("\n".join(chunks)), header=None)
                        filtered_chunk = dataframe_chunk[dataframe_chunk[0].isin(commuters)] #0 è l'ID
                        all_dataframes.append(filtered_chunk)

                    # concatena tutti i chunks
                    print(f"Combining all chunks into final DataFrame for {date}...")
                    final_dataframe = pd.concat(all_dataframes, ignore_index=True)

                    print(f"Final DataFrame for {date} (showing first 3 rows):")
                    print(final_dataframe.head(3))
                    print(f"Lunghezza dataframe: {len(final_dataframe)}")

                    dataframes[date] = final_dataframe

                break

            except (paramiko.SSHException, paramiko.ssh_exception.NoValidConnectionsError) as conn_err:
                print(f"Connection issue on attempt {attempt+1} for {date}: {conn_err}")
                traceback.print_exc()
                attempt += 1
                if attempt < RETRIES:
                    print("Retrying after 60 seconds...")
                    time.sleep(60)
                else:
                    print(f"Failed to process {date} after {RETRIES} attempts.")

            except Exception as file_error:
                print(f"An error occurred while processing the file for {date}: {file_error}")
                traceback.print_exc()

                break

    #chiude le connessioni
    sftp.close()
    ssh.close()

except Exception as e:
    print(f"An error occurred outside the loop: {e}")
    traceback.print_exc()
```

2. The GeoDataFrames were *not filtered* based on whether the drivers used the *Firenze-Lastra a Signa* segment on a specific day. Instead, we retained *all* the GPS signals from the commuter group for the *entire* month of February 2019, ensuring a comprehensive dataset for further analysis.

Ultimately, at the end of the process, we got a GeoDataframe with 84519 rows that included all the GPS signals of the commuter group for February 2019.

2.4.5 Analysis of Commuters' GPS data for February 2019

In this chapter, we illustrate and discuss the methods used to prepare and analyse the dataset created in **2.4.4** that includes all the GPS signals from February 2019 for the commuter group identified in **2.4.3.**

Data preprocessing

1. The file was read and transformed into a TrajDataFrame using the same procedure followed in **2.4.3.**

2. Using the `filter()` function of Scikit-mobility, we removed points that would imply speeds higher than 150km/h, as probable GPS errors.

```
%%time
f_tdf = filtering.filter(tdf, max_speed_kmh=150.0)
print('Number of records:\t%s'%len(f_tdf))
print('Filtered points:\t%s'%(len(tdf) - len(f_tdf)))
```

```
Number of records:      84466
Filtered points:        53
CPU times: user 959 ms, sys: 163 ms, total: 1.12 s
Wall time: 1.12 s
```

3. With the `compress()` function, we reduced the number of trajectory points by merging locations within a 100-meter radius in order to work with lighter dataset without losing too much granularity.

```
%%time
fc_tdf = compression.compress(f_tdf, spatial_radius_km=0.1) #punti entro 100 metri sono rimossi
print('Points of the filtered trajectory:\t%s'%len(f_tdf))
print('Points of the compressed trajectory:\t%s'%len(fc_tdf))
print('Compressed points:\t\t\t%s'%(len(f_tdf)-len(fc_tdf)))
```

```
Points of the filtered trajectory:      84150
Points of the compressed trajectory:    64769
Compressed points:                      19381
CPU times: user 3.99 s, sys: 23.4 ms, total: 4.02 s
Wall time: 4.02 s
```

Visits per Time

4. We applied the function `visits_per_time_unit()` to compute the number of GPS data points recorded every two hours during the first week of February. This analysis allowed us to visualise temporal mobility patterns and generate Figure 21 with **Seaborn** [49].

```
from skmob.measures.collective import visits_per_time_unit
```

```
import seaborn as sns
import matplotlib.pyplot as plt
start_date = '2019-02-04'
end_date = '2019-02-10'

filtered = fc_tdf[(fc_tdf['Date'] >= start_date) & (fc_tdf['Date'] <= end_date)]

sns.set(style="whitegrid", rc={"figure.figsize": (9, 5)})


sns.lineplot(data=visits_per_time_unit(filtered, time_unit='2h'))
plt.savefig("1weeksVisits.png", dpi=300, bbox_inches="tight")
plt.show()
```

Individual Measures

5. By applying the `radius_of_gyration()` function, we calculated the characteristic individual distances travelled by the drivers from the commuter group. We then visualized these distances using a histogram in Seaborn, creating Figure 22.

```
from skmob.measures.individual import radius_of_gyration
```

```
rg_df = radius_of_gyration(tdf)
rg_df.head(20)
```

```
100%|██████████| 47/47 [00:00<00:00, 145.10it/s]
```

Type Markdown and LaTeX: α^2

```
sns.set(style="whitegrid", rc={"figure.figsize": (7, 4)})
plt.hist(rg_df['radius_of_gyration'], bins=7, rwidth=0.8)
plt.xlabel('radius of gyration (km)', fontsize=10)
plt.ylabel('n users', fontsize=10)
plt.savefig("radius.png", dpi=300, bbox_inches="tight")
plt.show()
```

6. We measured the travelling predictabilities of the users from the commuter group with the `real_enthropy()` function, after sorting trajectory data in ascending order by datetime, as recommended in the documentation. We then visualised the results using a histogram in Seaborn, producing Figure 23.

```
from skmob.measures.individual import real_entropy, uncorrelated_entropy
```

```
#start_date = '2019-02-04'
#end_date = '2019-02-06'

#filtered = fc_tdf[(fc_tdf['Date'] >= start_date) & (fc_tdf['Date'] <= end_date)]
```

```
tdf_sort = fc_tdf.sort_values(by=['datetime'], ascending=True)
```

```
s_unc_df = real_entropy(tdf_sort)
s_unc_df.head()
```

```
100%|██████████| 44/44 [00:00<00:00, 136.12it/s]
```

```
plt.hist(s_unc_df['real_entropy'])
plt.ylabel('n users', fontsize=10)
plt.xlabel('real enthropy', fontsize=10)
plt.grid(alpha=0.2)
plt.savefig("enthropy.png", dpi=300, bbox_inches="tight")
plt.show()
```

7. With the `maximum_distance()` function, we computed the maximum distance (in kilometres) travelled by each individual in the TrajDataFrame and then we represented them with a histogram on Seaborn.

```
from skmob.measures.individual import maximum_distance
```

```
md_df = maximum_distance(tdf)
md_df.sort_values(by='maximum_distance',ascending=False)
```

```
100%|██████████| 47/47 [00:00<00:00, 117.59it/s]
```

```
sns.set(style="whitegrid", rc={"figure.figsize": (7, 5)})
plt.hist(md_df['maximum_distance'], bins=8, rwidth=0.8)
plt.xlabel('max $r$', fontsize=15)
plt.ylabel('n users', fontsize=15)
plt.savefig("max_distance.png", dpi=300, bbox_inches="tight")
plt.show()
```

8. We computed the jump lengths (in kilometres), defined as the geographic distances between two consecutive points visited by a user, using the `jump_lengths()` function. By plotting the distributions of these distances in a histogram, we generated Figure 23.

```
from skmob.measures.individual import jump_lengths
```

```
tdf_sort = fc_tdf.sort_values(by=['datetime'], ascending=True)
```

```
jl_df = jump_lengths(tdf_sort)
jl_df
```

```
100%|██████████| 47/47 [00:00<00:00, 85.25it/s]
```

```
import matplotlib.pyplot as plt
import numpy as np

all_jump_lengths = np.concatenate(jl_df['jump_lengths'].values)

plt.figure(figsize=(9, 4))
plt.hist(all_jump_lengths, bins=20)
plt.xlabel('Jump Length (km)', fontsize=10)
plt.ylabel('Frequency', fontsize=10)
plt.yscale('log')
plt.grid(alpha=0.3)
plt.savefig("jumps_log.png", dpi=300, bbox_inches="tight")
plt.show()
```

9. We determined and visualised the maximum distance travelled by the users from their home location with the function `max_distance_from_home()`. The results were displayed in Figure 24.

```
from skmob.measures.individual import max_distance_from_home
```

```
mdh_df = max_distance_from_home(tdf)
mdh_df.head()
```

```
100%|██████████| 47/47 [00:00<00:00, 57.36it/s]
```

```
#Vediamo quali sono i 10 utenti che hanno fatto la distanza maggiore da casa
ranked_df = mdh_df.sort_values(by='max_distance_from_home', ascending=False).reset_index(drop=True)
ranked_df[:10]
```

```
<div>
```

Type Markdown and LaTeX: α^2

```
sns.set(style="whitegrid", rc={"figure.figsize": (7, 4)})
plt.hist(mdh_df['max_distance_from_home'], bins=6, rwidth=0.8)
plt.xlabel('max distance from home (km)', fontsize=10)
plt.ylabel('n users', fontsize=8)
plt.savefig("max_dist_homedh.png", dpi=300, bbox_inches="tight")
plt.show()
```

Stops detection

10. We detected the stay locations (or stops) for each individual with `stay_locations()` and we mapped them on the tessellation of Tuscany:

- According to the parameters we set, a stop was identified when a user remains within a 200-meter radius for at least 20 minutes, indicating that they may have stopped at a meaningful place (e.g., home, work, a shop).

```
from skmob.preprocessing import detection
```

```
stays= skmob.preprocessing.detection.stay_locations(
    fc_tdf, minutes_for_a_stop=20.0, spatial_radius_km=0.2)
```

```
stays.head(1)
```

	id	Date	DeltaPos	Heading	uid	lat	lng	PanelSession	Speed	Time	datetime	geometry	leaving_datetime
0	58	2019-02-01	142	0	5514	43.772773	11.15268	1	0	07:25:37	2019-02-01 07:24:51	POINT (11.152065 43.772987)	2019-02-01 17:46:21

- We downloaded the tessellation map of Tuscany prepared in 2.1 and we assigned to each tile a unique numerical identifier, as required for the mapping process. To prevents errors in spatial mappings caused by the presence of MultiPolygons geometries in the tessellation we applied the `.explode()` function. This operation ensures that MultiPolygon tiles are transformed into individual polygons.

```
tessellation=gpd.read_file('Internship/TessellationToscana.geojson', driver='GeoJSON')
```

```
tessellation['tile_ID']=range(len(tessellation))
```

```
tessellation.plot() •••
```

```
from shapely.geometry import MultiPolygon, Polygon
```

```
t_exp=tessellation.explode(index_parts=True)
set(t_exp['geometry'])
```

```
{<POLYGON ((10.038 44.036, 10.041 44.027, 10.019 44.044, 10.033 44.065, 10.03...>, •••
```

```
t_exp.plot() •••
```

- Using the `mapping()` method, each stay location was assigned to a corresponding tile in the tessellation based on its spatial position.

```
help(stays.mapping)
```

```
Help on method mapping in module skmob.core.trajectorydataframe: •••
```

```
map_stays = stays.mapping(t_exp)
map_stays.head(2)
```

	id	Date	DeltaPos	Heading	uid	lat	lng	PanelSession	Speed	Time	datetime	geometry	leaving_datetime	tile_ID
0	58	2019-02-01	142	0	5514	43.772773	11.152680	1	0	07:25:37	2019-02-01 07:24:51	POINT (11.152065 43.772987)	2019-02-01 17:46:21	106
1	133	2019-02-01	186	358	5514	43.767858	11.287461	1	8	18:23:34	2019-02-01 18:23:34	POINT (11.287464 43.76785)	2019-02-02 08:03:17	84

- We aggregated `map_stays` by `tile_ID` and then merged the resulting DataFrame with the tessellation GeoDataFrame on `tile_ID`.

```
count=map_stays.groupby('tile_ID').count().reset_index()[['tile_ID', 'uid']].rename(columns={'uid': 'n_points'})
```

```
count
```

```
<div> •••
```

```
most_points = count.sort_values(by='n_points', ascending=False) •••
```

```
import folium •••
```

```
tessellation_stops = tessellation.merge(count, on='tile_ID', how='left')
```

```
tessellation_stops.head(1)
```

	COD_COMUNE	NO	NAME	Cod_Provincia	Cod_Regione	Nome_Comune	Nome_Provincia	Nome_Regione	geometry	tile_ID	n_points
0	45001.0	45001000	Aulla	45.0	9.0	Aulla	Massa-Carrara	Toscana	POLYGON ((9.97566 44.16410, 9.96647 44.16943, ...	0	NaN

- We visualised the number of stops on the tessellation map with Altair to produce Figure 25.

```
import altair as alt

tessellation_p =tessellation_stops.to_crs(epsg=4326)


geojson = json.loads(tessellation_stops.to_json())


chart = alt.Chart(alt.Data(values=geojson["features"])).mark_geoshape(stroke='lightgrey').encode(
    color=alt.Color("properties.n_points:Q", title="Stops",scale=alt.Scale(scheme='magma', reverse=True)),
    tooltip=[alt.Tooltip("properties.Nome_Comune:N", title="Comune"),
             alt.Tooltip("properties.n_points:Q", title="Stops")]
).properties(
    width=600,
    height=500,
    title="Stops per Tile"
)

geojson = tessellation.__geo_interface__

base_map = alt.Chart(alt.Data(values=geojson["features"])).mark_geoshape(fill='white',stroke='lightgrey').encode(
    tooltip=alt.Tooltip("properties.Nome_Comune:N", title="Comune")
).properties(
    width=600,
    height=500,
    title="Stops per comune"
)
map_stops=(base_map +chart)
map_stops.save('map_stops.html')
map_stops.show()
```

Home Locations

11. We located the home locations of the drivers in the commuter group and displayed them on the tessellation map:

- Using the `home_location()` function, we considered a home location as most frequently visited location during nighttime hours for each individual.

```
import skmob
from skmob.measures.individual import home_location
hl_df = home_location(fc_tdf, start_night='22:00', end_night='6:00')
```

```
100%|██████████| 47/47 [00:00<00:00, 139.94it/s]
```

```
hl_df
```

	uid	lat	lng
0	5514	43.765474	11.277748
1	69887	43.712375	11.058959
2	89001	43.744944	11.292254

- We converted the `hl_df` DataFrame into a GeoDataframe by creating a geometry column from the `lat` and `lng` columns. Thereafter, we spatially joined the tessellation GeoDataFrame on it.

```
tessellation=gpd.read_file('Internship/TessellationToscana.geojson', driver='GeoJSON')
```

```
hl_df
```

```
<div> •••
```

```
hl_gdf = gpd.GeoDataFrame(
    hl_df,
    geometry=gpd.points_from_xy(hl_df['lng'], hl_df['lat']),
    crs='EPSG:4326'
)
```

```
hl_tessellation=gpd.sjoin(hl_gdf,tessellation, how='left', predicate='within') #uso within perché voglio solo i punti DENTRO i plogoni
```

- We aggregated the number of homes in each tessellation tile by counting the number of home locations assigned to each tile (`tile_point_counts`). This count was then merged into the tessellation GeoDataFrame as a new column `n_points`. Tiles without any homes were removed, and the final tessellation

(`tessellation_h`) was reprojected to match the coordinate reference system of the original dataset.

```
tile_point_counts = hl_tessellation.groupby('index_right').size()
```

```
tile_point_counts
```

```
index_right •••
```

```
tessellation['n_points'] = tile_point_counts
```

```
tessellation_h = tessellation.dropna(subset=['n_points'])
```

```
tessellation_h = tessellation.to_crs(hl_gdf.crs)
```

- With Altair, we created a choropleth map where the number of homes in each tile is represented by a colour gradient based on the `n_points` column. This map was used to generate Figure 26.

```
import altair as alt

tessellation_h =tessellation_h.to_crs(epsg=4326)


geojson = json.loads(tessellation_h.to_json())


chart = alt.Chart(alt.Data(values=geojson["features"])).mark_geoshape(stroke='lightgrey').encode(
    color=alt.Color("properties.n_points:Q", title="Homes",scale=alt.Scale(scheme='magma', reverse=True)),
    tooltip=[alt.Tooltip("properties.Nome_Comune:N", title="Comune"),
             alt.Tooltip("properties.n_points:Q", title="Homes")]
).properties(
    width=600,
    height=500,
    title="Homes"
)


geojson = json.loads(tessellation.to_json())

base_map = alt.Chart(alt.Data(values=geojson["features"])).mark_geoshape(fill='white',stroke='lightgrey').encode(
    tooltip=alt.Tooltip("properties.Nome_Comune:N", title="Comune")
).properties(
    width=600,
    height=500,
    title="homes per comune"
)
map_h=(base_map +chart)

map_h.save('map_home.html')
map_h.show()
```

Traffic Flows

12. We represented on the map of Tuscany the traffic flows derived from the trajectories of the entire TrajDataFrame:
    - We loaded the tessellation GeoDataFrame of Tuscany and, to standardise the spatial structure, we generated a new square tessellation (`toscana_tess_squared`) with uniform 250m × 250m grid cells using `tiler.get()`.

```
import skmob
from skmob.utils.plot import plot_gdf
from skmob.tessellation.tilers import tiler
```

```
tessellation = gpd.read_file("/Users/Andre/Desktop/Internship/TessellationToscana.geojson")
```

```
toscana_tess_squared = tiler.get('squared', base_shape=tessellation, meters=250)
print("tiles = %s" %len(toscana_tess_squared))
toscana_tess_squared.head()
```

```
tiles = 703444 •••
```

- With the `to_flowdataframe()` function, we aggregated the TrajDataFrame into a FlowDataFrame applying `toscana_tess_squared` as the spatial tessellation to aggregate the points.

```
fdf=fc_tdf.to_flowdataframe(toscana_tess_squared)
fdf.head()
```

- We rendered the flows from the FlowDataFrame onto a Folium map with `plot_flows()`. This map was used for Figure 30.

```
def style_function(feature):
    return {
        'fillColor': 'green',
        'color': 'grey',
        'weight': 0.5,
        'fillOpacity': 0.0
    }
map_f = folium.Map(location=[tessellation.geometry.centroid.y.mean(), tessellation.geometry.centroid.x.mean()], zoom_start=9,tiles='cartodbpositron')
popup_features = ['Nome_Comune']
folium.GeoJson(
    tessellation,
    tooltip=folium.GeoJsonTooltip(fields=popup_features),
    style_function= style_function
).add_to(map_f)
map_f
```

```
fdf.plot_flows(map_f=map_f, min_flow=2, flow_weight=2, zoom=15, radius_origin_point=0, opacity=0.2)
```

Trajectory Visualisations

13. When trying to visualise all the trajectories on a map, we noticed that many were long straight lines, especially between Firenze and Empoli. This issue might be caused by GPS malfunctions or poor signal reception, leading to large time gaps between recorded points. As a result, some trajectories lose granularity and, instead of accurately reflecting movement patterns, they are displayed as straight-line connections between distant points. To mitigate this problem and have a better visualisation, we used the following approach:

- We split trajectories whenever there was a gap of more than 20 minutes between GPS signals. While this might artificially divide some continuous trips, it effectively reduced the problem of missing GPS data.
  To achieve this, we ordered the TrajDataFrame by `uid` and `datetime` and then for each user, we calculated the time differences in seconds between consecutive GPS points. We created a new column called `tid`, which served as a trajectory identifier. Using the `cumsum()` method on a Boolean condition, a new sequential `tid` was created each time the difference between two points was higher than 20 minutes.

```
fc2_tdf = fc2_tdf.sort_values(by=['uid', 'datetime'])

# Calcoliamo la timediffrence in secondi per ogni 'uid'
fc2_tdf['time_diff'] = fc2_tdf.groupby('uid')['datetime'].diff().dt.total_seconds()

#Definiamo una nuovo trajectory id ogni volta che il gap è sopra i 20 minuti
#cumsum() su un booleano dà un nuovo tid ogni volta che trova un True!
time_gap_threshold = 20 * 60  # 20 minuti
fc2_tdf['tid'] = (
    (fc2_tdf['time_diff'] > time_gap_threshold).cumsum()
)


fc2_tdf = fc2_tdf.drop(columns=['time_diff'])
```

- We plotted the trajectories with Folium using the `Polyline` which needs a list of coordinates tuples (lat, long) for each line. To achieve this, we iterated over each trajectory in the dataset by grouping the data by `uid` and trajectory `tid`. For each trajectory, we extracted the corresponding latitude and longitude values from the grouped data and zipped them into a list of coordinate pairs. If the trajectory contained at least two points, we added it to the Folium map using.

```
m = folium.Map(location=[fc2_tdf['lat'].mean(), fc2_tdf['lng'].mean()],
               zoom_start=9,
               tiles='CartoDB positron')

for (uid, tid), group in fc2_tdf.groupby(['uid', 'tid']):
    coords = list(zip(group['lat'], group['lng']))
    if len(coords) >= 2:
        folium.PolyLine(
            coords,
            color='red',
            weight=1,
            opacity=0.3
        ).add_to(m)

m.save('trajectories.html')
```

Visualisation of Florence's outbound and inbound trajectories

14. To differentiate and then visualise the trajectories that end inside or outside of Florence, we used the same approach employed above with Folium and `PolyLine()`, but with a small modification. Specifically, we calculated the distances between the first and last point of each trajectory and the centre of Florence. We drew in red the trajectories that started outside Florence and ended in it. The trajectories that began inside Florence and finished outside were instead drawn in blue. Finally, we did not draw the trajectories that did not end nor start inside a radius of 6 kilometres from the centre of Florence. The resulting map was displayed in Figure 31.

```
from geopy.distance import geodesic

location_florence = (43.7696, 11.2558)
DISTANCE_THRESHOLD_KM = 6.0

m = folium.Map(location=[43.7696, 11.2558], zoom_start=10, tiles='CartoDB positron')


for (uid, tid), group in fc2_tdf.groupby(['uid', 'tid']):
    coords = list(zip(group['lat'], group['lng']))
    if len(coords) >= 2:
        last_point = coords[-1]
        first_point=coords[0]
        dist_end = geodesic(last_point, location_florence).km
        dist_start=geodesic(first_point, location_florence).km
        if dist_end <= DISTANCE_THRESHOLD_KM and dist_start >= DISTANCE_THRESHOLD_KM:
            color='red'
        elif dist_end >= DISTANCE_THRESHOLD_KM and dist_start <= DISTANCE_THRESHOLD_KM:
            color='blue'
        else:
            continue
        folium.PolyLine(locations=coords, color=color, weight=3, opacity=0.1).add_to(m)

m.save('in_out_Florence.html')
m
```

Heatmap

15. To create a heatmap of all the points in out TrajDataFrame, we used the `HeatMap()` function of Folium. To achieve this, first we extracted the coordinates (in longitude-latitude order) from the `geometry` columns using `.coords`. Since Folium expects coordinates in latitude-longitude order, we swapped the values accordingly and created a list that was passed to `Heatmap()`, using a customised colour gradient to optimise visibility on the `CartoDB Dark Matter` tile layer. This map was used to create Figures 27 and 28.

```
from folium.plugins import HeatMap

heatmap_points = []
for point in fc2_tdf['geometry']:
    for coord in point.coords:
        heatmap_points.append([coord[1], coord[0]])

m = folium.Map(
    location=[fc2_tdf['lat'].mean(), fc2_tdf['lng'].mean()],
    zoom_start=10,
    tiles='CartoDB dark_matter'
)

HeatMap(
    data=heatmap_points,
    radius=8,
    blur=10,
    max_zoom=1,
    gradient={0.3: 'blue', 0.45: 'lightblue', 0.65: 'lime', 0.7: 'darkgreen', 0.77: 'yellow', 0.8: 'orange', 0.85: 'red', 0.9: 'purple'}
).add_to(m)

m.save('trajectory_heatmap.html')
m
```

Trajectory Clustering

16. Trajectory clustering has recently emerged as one of the most effective techniques in trajectory data mining for uncovering common patterns, achieved by grouping together trajectories that exhibit high similarity. Although a standard DBSCAN algorithm can be quickly and efficiently adapted for this task, the main challenge is to determine how “close” or “similar” two trajectories are [50]. Various similarity measures have been proposed in literature, including the Hausdorff distance, the Fréchet distance, and the average Euclidean distance, among others. However, given the large size of our TrajDataFrame and the high computational cost of comparing complete trajectories using these measures, we opted for a variation of a single-point measure. In our approach, three representative points from each trajectory were calculated and used to measure similarity: the centroid, the starting point (origin), and the ending point (destination). By doing so, we significantly reduced the computational overhead while still capturing the main characteristics of the trajectories, enabling effective clustering.

    - As described in point 12 of this subsection, we created a new GeoDataFrame containing all the trajectories as LineString geometris. To do this, we grouped the TrajDataFrame by `uid` and `tid`, extracted the coordinate pairs for each group and then used Shapely’s LineString class to create a line for each trajectory.

```
lines=[]
for (uid,tid), traj in fc2_tdf.groupby(['uid', 'tid']):
    coordinates=list(zip(traj['lng'], traj['lat']))
    if len(coordinates)>1:
        line=LineString(coordinates)
        lines.append({'uid': uid, 'tid': tid, 'geometry': line})
lines_gdf=gpd.GeoDataFrame(lines,crs="EPSG:4326")
```

```
lines_gdf.head()
```

	uid	tid	geometry
0	5514	0	LINESTRING (11.28773 43.76825, 11.29281 43.768...
1	5514	1	LINESTRING (11.15674 43.77238, 11.16028 43.770...
2	5514	2	LINESTRING (11.28217 43.76967, 11.28363 43.771...
3	5514	3	LINESTRING (11.25227 43.77762, 11.25112 43.776...

- We retrieved the geometries from the GeoDataFrame with `lines_gdf.geometry` and for each one of them we calculated its centroid. We then extracted the x and y coordinates (i.e. longitude and latitude, respectively) with `.x` and `.y`. Finally, with their values we created 2 new columns in the original GeoDataFrame called `cent_lng` and `cent_lat`.

```
lines_gdf['cent_lng'] = lines_gdf.geometry.centroid.x
lines_gdf['cent_lat'] = lines_gdf.geometry.centroid.y
```

- We created 4 additional columns to store the longitude and latitude coordinates of the first and last point of each trajectory (i.e. its origin and destination). To do this, we used a list comprehension on the geometries in the `geometry` column. For each geometry, we applied `line.coords[0]` to get the first coordinate tuple (the starting point) and `line.coords[-1]` for last one (the ending point). By selecting the appropriate element (`[0]` for longitude and `[1]` for latitude) from these tuples we populated the new the columns `start_lng`, `start_lat`, `end_lng`, and `end_lat`.

```
#aggiungiamo il punto di partenza e di arrivo per il clustering
lines_gdf['start_lng'] = [line.coords[0][0] for line in lines_gdf.geometry]
lines_gdf['start_lat'] = [line.coords[0][1] for line in lines_gdf.geometry]
lines_gdf['end_lng']   = [line.coords[-1][0] for line in lines_gdf.geometry]
lines_gdf['end_lat']   = [line.coords[-1][1] for line in lines_gdf.geometry]
```

- For the purpose of this study, we performed a clustering on Florence's inbound traffic (i.e. only on the trajectories that start outside Florence's metropolitan area and end within it).
  To achieve this, we first created a 15-kilometre buffer around the centre of Florence.

```
from shapely.geometry import Point
import geopandas as gpd

florence_centro = Point(11.2558, 43.7696)
florence_gdf = gpd.GeoDataFrame({'geometry': [florence_centro]}, crs='EPSG:4326') #deve essere 4326 visto che siamo in lat e lng


buffer_firenze_15 = (florence_gdf.to_crs(epsg=3857).buffer(15000).to_crs(epsg=4326)) #per fare il buffer in metri devo riproiettare in 3857 e poi riportarlo in 4326
```

  Next, we added two new columns, `end_points` and `start_points`, to the GeoDataFrame, representing the geometries of the endpoints and starting points of each trajectory. Finally, we generated a new GeoDataFrame that

includes only those trajectories that starts outside and end within the buffered area around Florence. The resulting GeoDataFrame contained 605 trajectories.

```
lines_gdf['end_points'] = gpd.points_from_xy(lines_gdf['end_lng'], lines_gdf['end_lat'], crs='EPSG:4326')
lines_gdf['start_points']= gpd.points_from_xy(lines_gdf['start_lng'], lines_gdf['start_lat'], crs='EPSG:4326')
```

```
print("CRS of lines_gdf:", lines_gdf.crs)
```

```
buffer_polygon = buffer_firenze_15.geometry.iloc[0]
```

```
lines_gdf_end_inFlorence_start_outFlorence = lines_gdf[
    lines_gdf['end_points'].within(buffer_polygon) &
    ~lines_gdf['start_points'].within(buffer_polygon)].copy().reset_index(drop=True)
```

```
print(len(lines_gdf_end_inFlorence_start_outFlorence))
print(len(lines_gdf))
```

```
605
3350
```

- We then performed a clustering on the filtered GeoDataFrame using the DBSCAN algorithm with the `fit_predict` method and the columns `start_lat, start_lng, end_lat, end_lng, cent_lng, cent_lat` as input. After some fine-tuning, the parameter `eps` (the maximum distance between two samples for them to be considered as part of the same neighbourhood) set at 0.035 and the `min_samples` at 3. By using these parameters, we obtained 30 clusters and 81 noise/outlier points.
  A new column named `cluster` was added to the GeoDataFrame to store the cluster labels for each trajectory.

```
feature_cols = ['start_lat', 'start_lng', 'end_lat', 'end_lng', 'cent_lng', 'cent_lat']
```

```
db = DBSCAN(eps=0.035, min_samples=3)
clusters = db.fit_predict(lines_gdf_end_inFlorence_start_outFlorence[feature_cols])
lines_gdf_end_inFlorence_start_outFlorence['cluster'] = clusters
lines_gdf_end_inFlorence_start_outFlorence['cluster'].unique()
```

```
array([ 0, -1,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
       16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
```

- We filtered out the cluster `-1` (i.e. noise or outliers), and we plotted the other clusters with Folium by creating a colour mapping that assigned a unique colour to the trajectories in the same cluster. The map was saved and used to generate Figure 32.

```
cluster_plot=lines_gdf_end_inFlorence_start_outFlorence[lines_gdf_end_inFlorence_start_outFlorence['cluster']>=1]

unique_clusters = end_cluster_plot['cluster'].unique()
color_list = [
    'red', 'blue', 'green', 'purple', 'orange', 'darkred', 'lightred',
    'white', 'darkblue', 'darkgreen', 'cadetblue', 'darkpurple',
    'pink', 'lightblue', 'lightgreen', 'pink', 'yellow', 'brown',
    'cyan', 'magenta', 'gold', 'lime', 'teal', 'navy', 'violet'
]
cluster_color_mapping = {cluster: color_list[i % len(color_list)] for i, cluster in enumerate(unique_clusters)}


def style_function(feature, color):
    return {
        'color': color,
        'weight': 4,
        'opacity': 0.6
    }

m = folium.Map(
    location=[43.7696, 11.2558],
    zoom_start=10,
    tiles='CartoDB dark_matter'
)

for _, row in end_cluster_plot.iterrows():
    cluster_id = row['cluster']
    color = cluster_color_mapping.get(cluster_id, 'black')
    folium.GeoJson(
        row['geometry'],
        style_function=lambda feature, color=color: style_function(feature, color),
        tooltip=f"UID: {row['uid']}, Segment: {row['tid']}, Cluster: {cluster_id}"
    ).add_to(m)

m.save('clustering_entering_Florence_starting_out_Florence.html')
```

- To evaluate the clustering results, we calculated the silhouette score, which was 0.548, indicating that the clusters are relatively well-separated.

```
from sklearn.metrics import silhouette_score
X = cluster_plot[['start_lat', 'start_lng', 'end_lat', 'end_lng', 'cent_lng', 'cent_lat']]
sil_score = silhouette_score(X, cluster_plot['cluster'])
print(f"Silhouette Score: {sil_score:.3f}")
```

```
Silhouette Score: 0.548
```

- To summarise the outcomes of the clustering, we computed the number of trajectories for each cluster and examined how the user IDs are distributed among the different clusters. The results indicate that **clusters vary in size**. Such differences might indicate that certain mobility patterns are much more common than others. They also show that **some clusters are homogeneous** and are exclusively associated with a single UID. These clusters likely represent highly regular behaviour for the associated user. Whereas **clusters with a more heterogeneous composition** may represent common corridors that attracts users with different overall travel routines. Another interesting result is that **some users are associated with multiple clusters**. This behaviour suggests that these users have multiple alternate routes that may be linked to route variations due to time-of-day effects, different travel purposes (e.g., work vs. leisure), or route deviations due to external factors (such as traffic or weather).

```
cluster_summary = cluster_plot.groupby('cluster')['uid'].agg(['count', list]).reset_index()
cluster_summary.rename(columns={'count': 'traj_count', 'list': 'uids'}, inplace=True)
cluster_summary
```

	cluster	traj_count	uids
0	0	6	[5514, 5514, 5514, 5514, 5514, 5514]
1	1	5	[5514, 186115, 830176, 830176, 830176]
2	2	39	[5514, 89909, 89909, 89909, 89909, 89909, 8990...
3	3	4	[69887, 69887, 69887, 304492]
4	4	106	[69887, 69887, 69887, 69887, 69887, 69887, 698...
5	5	17	[89909, 89909, 89909, 89909, 89909, 89909, 899...
6	6	81	[120617, 120617, 120617, 120617, 120617, 13464...
7	7	13	[120617, 569958, 569958, 569958, 569958, 56995...
8	8	13	[134643, 134643, 134643, 134643, 134643, 13464...
9	9	34	[186115, 324242, 352469, 445765, 445765, 44576...
10	10	5	[253545, 253545, 253545, 253545, 253545]
11	11	28	[259780, 259780, 259780, 259780, 259780, 25978...
12	12	13	[261668, 261668, 261668, 261668, 261668, 26166...
13	13	17	[304492, 304492, 304492, 304492, 304492, 30449...
14	14	3	[304492, 304492, 304492]
15	15	18	[352469, 352469, 352469, 352469, 352469, 35246...
16	16	5	[355260, 355260, 445765, 570789, 570789]
17	17	5	[424643, 424643, 424643, 424643, 424643]
18	18	5	[515060, 515060, 515060, 850911, 850911]
19	19	15	[515060, 515060, 515060, 515060, 515060, 51506...
20	20	9	[556920, 556920, 556920, 556920, 556920, 55692...
21	21	10	[694607, 694607, 694607, 694607, 694607, 69460...
22	22	5	[694607, 694607, 694607, 694607, 694607]
23	23	10	[702805, 702805, 702805, 702805, 702805, 70280...
24	24	4	[702805, 702805, 702805, 702805]
25	25	17	[764720, 764720, 764720, 764720, 764720, 76472...
26	26	15	[850911, 850911, 850911, 850911, 850911, 85091...
27	27	3	[850911, 850911, 850911]
28	28	13	[865983, 865983, 865983, 865983, 865983, 86598...
29	29	6	[876930, 876930, 876930, 876930, 876930, 876930]

```python
from collections import defaultdict

cluster_summary = cluster_plot.groupby('cluster')['uid'].agg(list).reset_index()

uid_cluster_map = defaultdict(set)

for _, row in cluster_summary.iterrows():
    cluster_id = row['cluster']
    for uid in row['uid']:
        uid_cluster_map[uid].add(cluster_id)

uid_cluster_map = {uid: sorted(list(clusters)) for uid, clusters in uid_cluster_map.items()}

for uid, clusters in uid_cluster_map.items():
    print(f"UID {uid}: Clusters {clusters}")
```

```
UID 5514: Clusters [0, 1, 2]
UID 186115: Clusters [1, 9]
UID 830176: Clusters [1, 6]
UID 89909: Clusters [2, 5]
UID 120617: Clusters [2, 6, 7]
UID 241704: Clusters [2]
UID 289947: Clusters [2]
UID 352469: Clusters [2, 9, 15]
UID 354872: Clusters [2, 4]
UID 484424: Clusters [2, 5]
UID 569958: Clusters [2, 6, 7, 13]
UID 570789: Clusters [2, 13, 16]
UID 865983: Clusters [2, 11, 28]
UID 891366: Clusters [2, 9]
UID 69887: Clusters [3, 4]
UID 304492: Clusters [3, 13, 14]
UID 122711: Clusters [4]
UID 136295: Clusters [4, 6]
UID 167834: Clusters [4]
UID 270033: Clusters [4]
UID 324242: Clusters [4, 9]
UID 361991: Clusters [4]
UID 521426: Clusters [4, 6]
UID 569861: Clusters [4]
UID 642671: Clusters [4, 9]
UID 134643: Clusters [6, 8]
UID 269269: Clusters [6]
UID 850911: Clusters [6, 18, 26, 27]
UID 445765: Clusters [9, 16]
UID 502875: Clusters [9]
UID 584136: Clusters [9]
UID 253545: Clusters [10]
UID 259780: Clusters [11]
UID 261668: Clusters [12]
UID 355260: Clusters [16]
UID 424643: Clusters [17]
UID 515060: Clusters [18, 19]
UID 556920: Clusters [20]
UID 694607: Clusters [21, 22]
UID 702805: Clusters [23, 24]
UID 764720: Clusters [25]
UID 876930: Clusters [29]
```

```python
from collections import defaultdict


cluster_uid_map = defaultdict(set)

for uid, clusters in uid_cluster_map.items():
    for cluster in clusters:
        cluster_uid_map[cluster].add(uid)

cluster_uid_map = {cluster: sorted(list(uids)) for cluster, uids in cluster_uid_map.items()}

for cluster, uids in sorted(cluster_uid_map.items()):
    print(f"Cluster {cluster}: UIDs {uids}")
```

```
Cluster 0: UIDs [5514]
Cluster 1: UIDs [5514, 186115, 830176]
Cluster 2: UIDs [5514, 89909, 120617, 241704, 289947, 352469, 354872, 484424, 569958, 570789, 865983, 891366]
Cluster 3: UIDs [69887, 304492]
Cluster 4: UIDs [69887, 122711, 136295, 167834, 270033, 324242, 354872, 361991, 521426, 569861, 642671]
Cluster 5: UIDs [89909, 484424]
Cluster 6: UIDs [120617, 134643, 136295, 269269, 521426, 569958, 830176, 850911]
Cluster 7: UIDs [120617, 569958]
Cluster 8: UIDs [134643]
Cluster 9: UIDs [186115, 324242, 352469, 445765, 502875, 584136, 642671, 891366]
Cluster 10: UIDs [253545]
Cluster 11: UIDs [259780, 865983]
Cluster 12: UIDs [261668]
Cluster 13: UIDs [304492, 569958, 570789]
Cluster 14: UIDs [304492]
Cluster 15: UIDs [352469]
Cluster 16: UIDs [355260, 445765, 570789]
Cluster 17: UIDs [424643]
Cluster 18: UIDs [515060, 850911]
Cluster 19: UIDs [515060]
Cluster 20: UIDs [556920]
Cluster 21: UIDs [694607]
Cluster 22: UIDs [694607]
Cluster 23: UIDs [702805]
Cluster 24: UIDs [702805]
Cluster 25: UIDs [764720]
Cluster 26: UIDs [850911]
Cluster 27: UIDs [850911]
Cluster 28: UIDs [865983]
Cluster 29: UIDs [876930]
```

Bibliography and References

1. Baporikar, N. (2016). Infrastructure development as a catalyst for social-economic advancement. *International Journal of System Dynamics Applications (IJSDA)*, *5*(4), 101-113.
2. Magazzino, C., & Mele, M. (2021). On the relationship between transportation infrastructure and economic development in China. *Research in Transportation Economics*, *88*, 100947.
3. Wang, L., Xue, X., Zhao, Z., & Wang, Z. (2018). The Impacts of Transportation Infrastructure on Sustainable Development: Emerging Trends and Challenges. *International Journal of Environmental Research and Public Health*, *15*(6).
4. Shi, J., Bai, T., Zhao, Z., & Tan, H. (2024). Driving Economic Growth through Transportation Infrastructure: An In-Depth Spatial Econometric Analysis. *Sustainability*, *16*(10), 4283.
5. Lucas, K. (2011). Making the connections between transport disadvantage and the social exclusion of low income populations in the Tshwane Region of South Africa. *Journal of transport geography*, *19*(6), 1320-1334.
6. **https://www.regione.toscana.it/statistiche/indicatori-comunali-per-le-politiche-locali**
7. **https://www.regione.toscana.it/documents/10180/400011/b4qcferrovie.pdf**
8. https://andrevitali.github.io/dissertation/charts/Indicators_chart+infrastructures.html
9. https://andrevitali.github.io/dissertation/charts/movimenti_entrata_comuni.html
10. https://andrevitali.github.io/dissertation/charts/movimenti_uscita_comuni.html
11. https://andrevitali.github.io/dissertation/charts/flussi.html
12. https://andrevitali.github.io/dissertation/charts/flussi_grafo.html
13. https://andrevitali.github.io/dissertation/charts/flussi_grafo.html
14. https://andrevitali.github.io/dissertation/charts/Out_Movements_Interactive+Bars.html
15. https://andrevitali.github.io/dissertation/charts/sankey_diagram.html
16. https://andrevitali.github.io/dissertation/charts/chord_diagram_FIPILI.html
17. https://andrevitali.github.io/dissertation/charts/Mot%_Top10Municipalities.html
18. https://andrevitali.github.io/dissertation/charts/movements_by_MOT.html
19. https://andrevitali.github.io/dissertation/charts/TrainUsage.html
20. https://andrevitali.github.io/dissertation/charts/Stazioni+Ferrovie1.html
21. https://www.asaps.it/80515-_fipili_incidenti_e_code_da_oltre_50_anni__storia_della_strada__maledetta__della.html
22. https://andrevitali.github.io/dissertation/charts/FIPILI.html
23. https://www.firenzepost.it/2023/07/28/fipili-giani-terza-corsia-da-scandicci-a-lastra-a-signa-e-da-pontedera-al-bivio-per-pisa-livorno/
24. https://andrevitali.github.io/dissertation/charts/Firenze_Lastra.html
25. https://andrevitali.github.io/dissertation/charts/Frequency of use of the section Firenze-Lastra.html
26. https://andrevitali.github.io/dissertation/charts/Pie Proportions of Users for the Firenze-Lastra Section.html
27. https://andrevitali.github.io/dissertation/charts/Group1Usage.html
28. https://andrevitali.github.io/dissertation/charts/Group5Usage.html
29. https://andrevitali.github.io/dissertation/charts/map_stops.html
30. https://andrevitali.github.io/dissertation/charts/map_home.html
31. https://andrevitali.github.io/dissertation/charts/trajectory_heatmap.html
32. https://andrevitali.github.io/dissertation/charts/trajectories.html
33. https://andrevitali.github.io/dissertation/charts/flows_map5.html
34. https://andrevitali.github.io/dissertation/charts/in_out_Florence.html
35. https://andrevitali.github.io/dissertation/charts/clustering_entering_Florence_starting_out_Florence.html
36. https://dati.toscana.it/dataset/grafo-civici

37. https://www.qgis.org/
38. https://andrevitali.github.io/dissertation/code/Chapter 1.2 (The Importance of Mobility Data).ipynb
39. https://plotly.com/python/sankey-diagram/
40. https://holoviews.org/reference/elements/bokeh/Chord.html
41. https://andrevitali.github.io/dissertation/code/Code for Chapter 1.4 (A Region-wide Perspective).ipynb
42. https://andrevitali.github.io/dissertation/code/Methods for Chapter 1.5. Part 1. Creation of Shapefile of FI-Lastra segment.ipynb
43. https://www.paramiko.org/
44. https://andrevitali.github.io/dissertation/code/Methods for Chapter 1.5. Part 2. Filtro e Salvataggio GPS 2 settimana Febbraio 2019 FI-Lastra & Lastra Ginestra_Val.ipynb
45. https://scikit-mobility.github.io/scikit-mobility/
46. Pappalardo, L., Simini, F., Barlacchi, G., & Pellungrini, R. (2022). scikit-mobility: A Python Library for the Analysis, Generation, and Risk Assessment of Mobility Data. *Journal of Statistical Software*, *103*(4), 1–38.
47. https://andrevitali.github.io/dissertation/code/Methods for Chapter 1.5. Part 3. Analisi_Settimana2_Febbraio2019.ipynb
48. https://andrevitali.github.io/dissertation/code/Methods for Chapter 1.5. Part 4. Filtro e Salvataggio Pendolari Tutto Febbraio 2019.ipynb
49. https://seaborn.pydata.org/
50. Hu, D., Chen, L., Fang, H., Fang, Z., Li, T., & Gao, Y. (2023). Spatio-temporal trajectory similarity measures: A comprehensive survey and quantitative study. *IEEE Transactions on Knowledge and Data Engineering*.
51. https://andrevitali.github.io/dissertation/code/Methods for Chapter 1.5. Part 5. Analisi_Pendolari_TuttoFebbraio.ipynb
52. https://geopandas.org/en/stable/
53. https://altair-viz.github.io/