National and Kapodistrian
University of Athens

Department of Physics
MSc on Control and Computing

Diploma Thesis

Title: User Mobility Analysis in Mobile Communication Systems

By

Antonios Skarlatos

Supervised by Prof. Markos Anastasopoulos

Athens, February 2023

Table of Contents

1. Introduction

1.1. Purpose of Thesis

The main focus of this thesis is on using data analytics to extract information from telecommunication data, specifically using data from the Telecom Italia Big Data Challenge. Time and spatial analysis is applied to the data to gain a better understanding of user patterns of network usage. In addition to that, this thesis also employs Gaussian processes and correlation matrices as data analytics tools. These methods are used to predict network usage needs and the thesis includes a thorough examination of these tools and their application on the data. The thesis also includes an analysis of GPS data of vehicle traffic in the same areas in order to compare telecommunication and car traffic and investigate any correlations between them. The findings from this research can be useful in determining network handover needs in specific areas and optimizing network coverage. Sequences of scripts are developed for each part of the analysis, with multiple examples provided and with the aim of drawing conclusions about the performance of these methods. The overall goal is to use data analytics to extract actionable insights and improve network performance in telecommunication systems.

The thesis uses Python for data analytics and the development of scripts, and the data is stored in MySQL tables. SQL is used for managing the data and making it accessible for the analysis. The choice of Python and SQL as the primary tools for the analysis and management of the data, allows the development of efficient and easily-replicable scripts, while being able to handle big amount of data using MySQL. It also provides the capability of easy integration with other tools and making the results of the analysis more accessible and meaningful.

1.2. Theoretical Framework

1.2.1. Big Data

'Big Data' refers to a field that constructs ways to analyze and systemically extract information from datasets too big or too complicated to deal with, using traditional data processing software. Data with many fields (rows) offer greater statistic power, while data with higher complexity (columns/attributes) may lead to higher false rate. 'Big Data Analytics' includes tasks like harvesting data, data storage, data analysis, search, sharing, transferring, visualizing, updating and more. Current usage of the term tends to refer to the use of predictive analytics or certain other advanced data analytics methods that extract value from big datasets. The need of data analytics methods has grown rapidly hand by hand with the data availability itself. The size and number of available datasets has skyrocketed with the massive spread of data capturing devices. With it came the development of data analytics, since big datasets hold valuable information to extract from. What qualifies as 'big data' varied depending on the capabilities of those analyzing it and their tools. In this project we will try to deal with big datasets [1].

1.2.2. Anaconda Framework

Anaconda is a distribution of the Python programming language for scientific computing purposes (data science, machine learning, large-scale data processing, predictive analytics, etc.). It aims to simplify the package management and data handling, while it offers data-science packages for multiple environments (in this project it will be used in Windows environment). The package management system is called 'conda', and through it we can import the libraries we need. Anaconda Navigator is a desktop graphical user interface (GUI) included in Anaconda distribution that allows users to launch all of Anaconda's applications without using command-line commands.

There are more than 7,000 available open-source packages for Anaconda distribution. They can be installed using either PyPI as well as the conda package. Furthermore, there is a virtual environment manager that gives the ability to create different environments with different packages installed (depending on the data processing challenges you are facing). We will use this feature later on.

1.2.3. Project Jupyter

Jupyter Notebook (also known as IPython Notebook) is a web-based interactive computational environment for creating notebook documents. A Jupyter Notebook document is a browser-based language shell containing an ordered list of input/output cells which can contain code, text, plots or mathematics. A Jupyter Notebook is a JSON document, usually ending with the '.ipynb' extension. What makes Jupyter Notebook really useful is the local server-like attributes. Multiple code blocks can be compiled in any order, with data saving (remain saved while Jupyter Notebook is active) [2].

JupyterLab is a newer user interface for Jupyter Notebook, offering a flexible user interface and more features than the classic notebook UI [3].

1.2.4. MySQL

MySQL is a free open-source relational database management system. MySQL has stand-alone clients that allow users to interact directly with a MySQL database using SQL, but more often, MySQL is used with other programs to implement applications that need the relational database capability. We will use it to store our data through Jupyter using a suitable library [4].

MySQL Workbench is a visual database design tool that integrates SQL development, administration, database design, creation and maintenance into a single integrated development environment for the MySQL database system.

1.2.5. GeoJSON

GeoJSON is an open standard format designed for representing simple geographical features, along with their non-spatial attributes [5]. It is based on the JSON format. It is a format for encoding a variety of geographical data structures. The feature may include points (addresses), line strings or even polygons. It can also be used to describe a whole route or even the entire service coverage for navigation apps. GeoJSON is widely supported across a range of geographical information systems (GIS) and web mapping tools, making it a popular and interoperable format for exchanging and visualizing geospatial data.

1.2.6. Gaussian Process

Gaussian Processes (GP) are a generic supervised learning method designed to solve regression and probabilistic classification problems. GP are useful in statistical modelling, benefiting from properties inherited from the normal distribution. While exact models scale poorly as the amount of data increases, multiple approximation methods have been developed which often retain good accuracy while drastically reducing computation time. Some Gaussian processes advantages include:

- The prediction interpolates the observations (at least for regular kernels).
- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals and decide based on those if one should refit the prediction in some region.
- Versatile: different kernels can be specified. Common kernels are provided, but it is also possible to specify custom kernels.

The most common application for the multi-output prediction problem is the Gaussian process regression. It is a non-parametric, Bayesian approach designed to solve regression and classification problems. We will the Python 'sklearn' package to implement it [6].

1.2.7. Correlation Matrix

When data are about aligned, we claim that the variables have linear relationship. In most cases though, data deviate significantly from following a linear tendency. A whole-round measure to describe the potential of linear relationship is correlation. Correlation sums up the strength and direction of the linear relationship between two quantitative variables. Correlation values range between -1 and 1, with the positive sign representing a positive correlation, while the opposite a negative one. The closer the correlation is to 1, the more of a linear relationship the data have (the data points fall closer to having linear variables correlating them). Accordingly, the closer the correlation is to 0, the weaker linear relationship is.

A correlation matrix is a matrix that gives the correlation coefficients between different variables we want to investigate. Every cell in the matrix represents the correlation between the variables crossing axis x and y [7].

There are 3 broad reasons to calculate a correlation matrix.

- To sum up big data, when the goal is to identify a pattern
- Introduction to new analysis
- Diagnostic measure to double check different data analysis procedures

In this project we will use ready-to-use libraries to calculate and plot correlation matrices.

1.3. Python Libraries

1.3.1. NumPy

'Numpy' is a library for the Python programming language. It is the fundamental package for scientific computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate these arrays. At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. NumPy arrays have some limitations, since their size is fixes at creation (unlike Python lists which can grow dynamically), and the array elements are required to be of the same data type. But despite these, they offer a strong, fast tool for advanced mathematical operations [8].

1.3.2. Mysql.connector

MySQL provides standards-based drivers for multiple languages enabling developers to build database applications in their language of choice. For our case, there is a Python Driver for MySQL (developed and maintained by the MySQL community) named 'mysql.connector'. It is a Python library for connecting to and interacting with MySQL databases. It provides a low-level API for sending and receiving data from a MySQL database, as well as tools for working with transactions, handling errors, and more. It also provides the ability to execute multiple statements with a single call, using batch execution. The above make it an essential tool for developers and data scientists working with large datasets and databases on MySQL [9].

1.3.3. Shapely.geometry

'Shapely' is a BSD-licensed Python package for manipulation and analysis of planar geometric objects. It is based on the widely deployed GEOS and JTS libraries. Shapely is not concerned with data formats or coordinate systems, but can be readily integrated with packages that are. Shapely geometry classes, such as shapely.Point, are the central data types in Shapely. Each geometry class extends the shapely.Geometry base class, which is a container of the underlying GEOS geometry object, to provide geometry type-specific attributes and behavior.

1.3.4. Matplotlib

'Matplotlib' is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits. It is designed to closely resemble the MATLAB interface of plotting. Since its release, it has been actively developed by the community and it is widely used in data analysis, scientific computing, and many other fields [10].

1.3.5. Requests

'Requests' is a popular Python library for making HTTP requests, including GET, POST, PUT, DELETE, and more. It abstracts the complexities of making requests behind a simple API, allowing

developers to send HTTP/1.1 requests. It provides automatic decompression of gzip and deflates encoded responses. It is an essential tool for web scraping, interacting with APIs, and working with web services, which is needed for this thesis.

1.3.6. Folium

'Folium' is a Python library for creating interactive maps for web browsers using the Leaflet JavaScript library. It is a particularly useful for visualizing geospatial data and creating interactive maps for data exploration and analysis. It has easy-to-use API for creating maps with markers, circles, polylines and other features, which we will use later on.

1.3.7. Pandas

'Pandas' is a widely-used, open-source data analysis and data manipulation library for Python. It provides data strictures for efficiently storing large datasets and tools for working with them. It is particularly useful for data analysis, cleaning, and preparation [11]. Some of the key features of 'pandas' include:

- A 'Dataframe' object for representing and manipulating tabular data, similar to a spreadsheet or SQL table.
- Methods for reading and writing data from a variety of sources, including CSV, Excel, SQL and more.
- Tools for cleaning and transforming data, including handling missing values, merging and joining datasets, and pivoting data.
- Efficient handling of large datasets with support for handling missing data and dealing with duplicate data.
- Built-in plotting and visualization capabilities using 'matplotlib'.

1.3.8. Seaborn

'Seaborn' is a Python data visualization library based on 'matplotlib'. It provides a high-level interface for creating attractive and informative statistical graphics. It is particularly well suited for exploring complex datasets and for visualizing relationships between multiple variables. Some of its key features include a simple and intuitive API, support for multiple plot types, including heatmaps (which we will use for the correlation visualization), violin plots, box plots, and built-in themes for making plots look polished and professional. Additionally, 'seaborn' has integrated support for working with 'pandas' dataframes, making it a popular choice among data scientists for data exploration and analysis. Overall, 'seaborn' is a powerful library for creating visualization of complex datasets and for exploring relationships between multiple variables in an intuitive and aesthetically pleasing way [10].

1.3.9. Sklearn

'Scikit-learn' (often abbreviated as 'sklearn') is a Python library for machine learning. It provides a wide range of algorithms for tasks such as classification, regression, clustering, and dimensionality reduction, as well as tools for model evaluation and selection. The library has a consistent API and a

focus on practical, real-world applications. It is well-documented and has a large and active community, making it a popular choice for building machine learning models in Python [12]. Some of the key features of 'sklearn' include:

- A variety of algorithms for supervised and unsupervised learning
- Easy-to-use APIs for training and evaluating models
- Built-in tools for preprocessing data and feature extraction
- Methods for model selection and evaluation, including cross-validation

1.3.10. Ipywidgets

'Ipywidgets' is a Python library for creating interactive, web-based widgets in Jupyter notebooks. These widgets allow users to interact with and manipulate data within Jupyter notebooks in real-time, making the process of data analysis more interactive and user-friendly. It includes a wide range of widgets, such as sliders, buttons, and text boxes, as well as more specialized widgets for displaying data such as graphs and tables. The library is easy to use and enables data scientists to create interactive visualizations and dashboards within Jupyter notebooks, making it a valuable tool for data exploration and analysis [13].

2. Analyzing and Presenting the Telecommunication Activity on user-input Routes in Milan province

2.1. Introduction

In this part of the thesis we will analyze and process telecommunication activity data about some public network usage. We will apply many different data analysis techniques, with the goal being to draw conclusions about the general network activity and its characteristics. The purpose is for the user to choose a desired route in the Milan province and the tool, which we are going to create, will calculate and analyze the telecommunication activity in that route. The (also) user chosen output options, will give useful information about the specific area's activity and its characteristics. We then could use these results to find ways to improve the network itself (e.g. an optimization problem) or we could apply them to help us in a different analysis project (which will happen in the second part of the thesis). The long term purpose of a tool like this is the overall improvement of the network and the optimization towards 5th generation networks.

In order to be able to perform such an analysis, we would need a big data collection from a public network. Data like this are difficult to acquire since they are confidential and there are legal difficulties. Companies and institutes that own such datasets only share them with a selected few research teams, which usually sign non-disclosure agreements (NDAs). This lack of data limits the volume of public research from the wide scientific community.

In this context, the supply of such datasets to a large number of research teams is a structural problem in the backbone of technological advancements in this field. An original example was given by Telecom Italia in cooperation with other Italian institutions (EIT ICT Labs, SpazioDati, MIT Media Lab, Northeastern University, Polytechnic University of Milan, Fondazione Bruno Kessler and University of Trent), organizing the 'Telecom Italia Big Data Challenge' [14]. With this initiative many anonymous datasets were made public. These datasets are unique in their kind, as they provide an, open source, rich accumulation of data from many sources of many kinds (such as telecommunication, weather, electricity data and others). These datasets have measurements over long periods of time in the Milan and Trento provinces. We will only deal with the Milan data because they are larger in size and so easier to analyze (and apply data analytics techniques). The only datasets we will need for this project are the ones about the geographical grid and the telecommunication activity. In the chapters to follow we will present and analyze them thoroughly.

2.2. Presenting the Data

The first dataset we will need for the analysis is the one containing the geographical grid of the areas we will work on. These are the areas for which we have the telecommunication data. The measurements are provided from different companies and institutions which use different systems to record the spatial activity of their users. The variance of the geographical distribution from all the different companies is taken into account and an accumulative grid of square cells is created. We have the total telecommunication activity throughout this square-cell grid in Milan for a long period of time. This adjustment provides us the ability to easily compare the telecommunication activity of different Milan areas.

We end up with a grid of 10,000 square cells around the Milan province (every cell covers an area of about 235m^2). The link reference of the geographical grid data (http://dx.doi.org/10.7910/dvn/QJWLFU) leads us to a 'geojson' file for the Milan province. We can upload it to see the province distribution into square cells (Image 1).

Image #1: Milan Grid

Every one of the small squares composes a cell. Every cell is numbered and is characterized by a unique Cell-ID (Image 2). Additional information is stored in the 'geojson' file, like the exact

coordinates of the 4 corners of the square cell. This information is necessary and will be further analyzed later on.

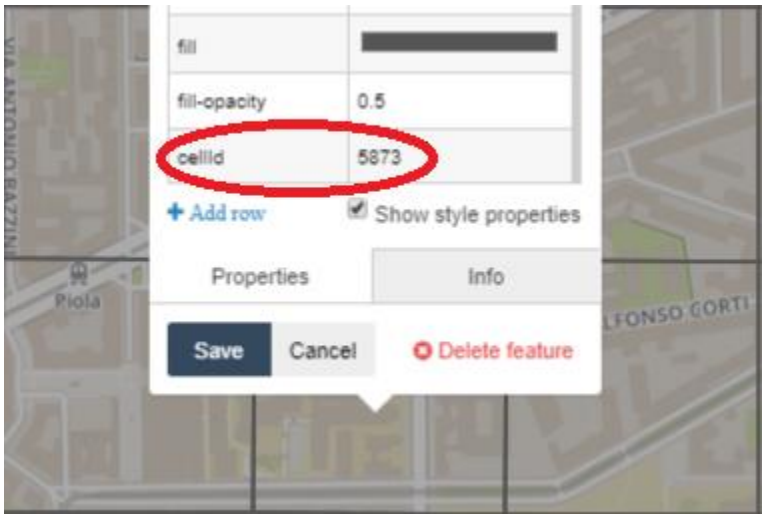


Image #2: Cell ID

The second dataset needed, is the one describing the telecommunication activity. The purpose of this dataset is to represent all kinds of telecommunication traffic that took place on the grid above over a long period of time. So, information, about all the ways users interacted with the network, is provided. The link reference of the telecommunication dataset (http://dx.doi.org/10.7910/dvn/EGZHFV), leads us to a list of 'text' files containing all the necessary information. 62 of these files are provided for the Milan province over the period from 1/11/2013 to 1/1/2014; 1 file for each day. That means that every file contains the telecommunication activity for the whole province over the course of 24 hours. The content of these files, opened in an excel table, can be seen in Image 3.

The shape of the information provided is simple and easily comprehensible. We will explain shortly what each one of the elements represents, and how we can translate them into useful information.

In the first column we have the Cell ID which was introduced above. It is a unique key that constitutes the identity of a specific cell. Using the geographical grid that was described above, we can identify the exact spatial area in which the telecommunication data (in the rest of the dataset) refer to.

1564	1.38E+12	39	0.04652	0.100905	0.030778	0.061217	3.496653
1564	1.38E+12	0	0.00173				
1564	1.38E+12	39	0.02499	0.049927	0.01163	0.038297	3.454974
1564	1.38E+12	0	0.038297				
1564	1.38E+12	39	0.046949	0.074635		0.100905	3.235227
1564	1.38E+12	0	0.00173				
1564	1.38E+12	39	0.169043	0.126975	0.01163	0.032509	2.646128
1564	1.38E+12	39		0.102378	0.019149	0.057563	4.413893
1564	1.38E+12	0	0.01163				
1564	1.38E+12	39	0.148762	0.061024	0.068736		3.964203
1564	1.38E+12	39	0.249667	0.107033	0.019149	0.019149	3.454893
1564	1.38E+12	0				0.055833	
1564	1.38E+12	39	0.196361	0.044139	0.038297	0.038297	2.397392
1564	1.38E+12	39	0.125895	0.030778	0.006921	0.00173	2.477659
1564	1.38E+12	0	0.00173				
1564	1.38E+12	39	0.104366	0.005191	0.055833	0.01163	2.326744
1564	1.38E+12	39	0.111144	0.087884			2.084419
1564	1.38E+12	39	0.178267	0.003461			2.952914
1564	1.38E+12	39	0.019149	0.04652		0.049587	2.359235

Image #3: Dataset in a 2D table

The second column contains the period of time in which the telecommunication data refer to. It is a natural number that represents milliseconds, and the time step is 600,000 milliseconds or 10 minutes. The telecommunication data describe the network usage from the users in the specific geographical area in the specific 10 minutes time (which starts in the moment the second column mentions). To calculate the ending moment of the measurements, all we have to do is add 600,000 milliseconds to that interval. It is obvious that, since every text file contains the entire daily data for the entire province, it will have 144 time measurements (144 10-minute periods in 24 hours). This means that the time column (milliseconds) will advance by 86,400,000 each day (or each file). So, every file contains the telecommunication data for all the 144 time periods in the day and for the entire province (all the square cells).

The column that contains the telecommunication data that interests us is the last one. To describe this column we will introduce the concept of Call Detail Records (CDR). It is a measurement unit to record the telecommunication traffic that increases in size every time a corresponding action takes place in the specific 10 minute period in the specific area. The number in the last column is the CDR measurement of the Internet usage activity. CDR increases every time a user is connected to the wireless network in the given area and time. Also, CDR increases when a specific connection (user-network) lasts more than 15 minutes or the user consumes more than 5MB of Internet data.

Columns 4 to 7 describe the CDR of the incoming and outgoing phone calls and messages. We can see that there are many black spaces in these columns. Since the data are broken down to very small areas and time periods, there is a possible scenario that no such activity takes place in the specific area and time (especially if the area has low population density and the time period is of less traffic).

That is the reason we will focus only on the Internet usage data. This kind of activity is the most common service users rely on (compared to phone calls and SMSs). So, Internet usage CDR records

are significantly higher in size than the rest (which are often blank) and that will make the data analytics procedures easier to implement. We ignore the rest of the telecommunication data, and we will remove them later on.

The thirds column is just the phone code of the country to which the data are addressed to. We only care about the rows in which the phone code is 39 (Italy phone code). To boil it down, this column is only useful if we want to find out how many calls/SMSs where addressed abroad. There is no such separation for the Internet data. The rows that contain the Internet usage CDR are the ones where phone code is 39.

Regarding the size of the data, each of the daily files has a size of about 80MB, and contains more than 2 million rows of raw data. So, we are facing a small scale 'Big Data' problem.

This was all the data we are going to need for analyzing and presenting the telecommunication traffic in the Milan province, according to user input.

2.3. Preprocessing and Saving to Databases

Having presented the shape of the data, we can now move on to the next step. Before starting up with the analytics procedures on the data, we have to transform them in a way that fits the needs of the analysis. That means, recognizing which of the data in our disposal are needed for the results we seek to get, and which we have to get rid of. The goal of this chapter is to strip the data of everything unnecessary and fit them in volume efficient database tables. That way, the data analytics to follow will be fast and efficient. The procedure of obtaining data from the databases will easily be automated.

We work with big data for this project. The databases will be large and sometimes difficult to handle with. That is why in every such problem, it is of outmost importance to make the databases as light as they can get.

At this point we have to point out that Python (and some SQL for the database handling) is used for the entire programming taking place. We will use the Anaconda framework (mostly Jupyter Notebook), which provides helpful tools for data analysis such as easy library imports and clever documentation. More details will be provided.

For the storage and management of the databases we will use MySQL Workbench, and for creating, accessing and altering the databases in any way desirable, we will use 'mysql.connector' library through Jupyter Notebook. 'Numpy' library is also used to manipulate and transform the raw data.

- Database for Milan's Cells

In this database we will register everything we need to know about the given cells that form the surrounding area of Milan. For the (square in shape) cells, all we could ever need for further analyzing are the coordinates of the four squares, which give us the exact spatial distribution of the cell grid. If we add the unique Cell ID each one is characterized by, we have the complete set of information. This information can be found in the 'geojson' file described in the chapter before. The contents of this file can be seen in Code Sample 1.

```
1 import json
2 data = json.load(open(r"C:\Users\Antonis\OneDrive\Bachelor Thesis\grids\milano-grid.geojson"))
```

```
1 data
```

```
{'crs': {'type': 'name', 'properties': {'name': 'urn:ogc:def:crs:EPSG::4326'}},
 'type': 'FeatureCollection',
 'features': [{'geometry': {'type': 'Polygon',
    'coordinates': [[[9.0114910478323, 45.35880131440966],
      [9.014491488013135, 45.35880097314403],
      [9.0144909480813, 45.35668565341486],
      [9.011490619692509, 45.356685994655464],
      [9.0114910478323, 45.358801314409661]]]},
   'type': 'Feature',
   'id': 0,
   'properties': {'cellId': 1}},
  {'geometry': {'type': 'Polygon',
    'coordinates': [[[9.014491488013135, 45.35880097314403],
      [9.017491928134044, 45.358800553060284],
      [9.017491276410173, 45.35668523336193],
      [9.0144909480813, 45.35668565341486],
      [9.014491488013135, 45.35880097314403]]]},
   'type': 'Feature',
   'id': 1,
   'properties': {'cellId': 2}},
```

Code Sample #1

All we have to do now is isolate the necessary information and save them in a dedicated database. In the database table, we can fit this cell information in 9 columns, 8 of which will be assigned for longitude and latitude of the 4 square-cell corners. The last one will save the unique Cell ID of the corresponding cell.

Having this information saved, we can find where a random geographical point (longitude & latitude) lies in the Milan grid using simple geometry. That means finding through its coordinates the exact square cell in which it belongs. To perform this act in python we will use the 'shapely.geometry' library (more to follow).

The code for inserting the data above in the corresponding table can be seen in Code Sample 2.

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="antbestskadata313BASE!1!9!9!7!",
  database="celliddb"
)


mycursor = mydb.cursor(buffered=True)


#sql = "DROP TABLE celliddb.cellidscoords;"
#mycursor.execute(sql)
#mydb.commit()


sql = '''CREATE TABLE celliddb.cellidscoords (tl_lat double, tl_lon double, tr_lat double,
        tr_lon double, br_lat double, br_lon double, bl_lat double, bl_lon double, cellid int);'''
mycursor.execute(sql)
mydb.commit()


for i in data['features']:
    sql = '''INSERT INTO celliddb.cellidscoords (tl_lat, tl_lon, tr_lat, tr_lon, br_lat, br_lon, bl_lat, bl_lon, ce
           VALUES ({}, {}, {}, {}, {}, {}, {}, {}, {})".format(i['geometry']['coordinates'][0][0][1],
           i['geometry']['coordinates'][0][0][0], i['geometry']['coordinates'][0][1][1],
           i['geometry']['coordinates'][0][1][0], i['geometry']['coordinates'][0][2][1],
           i['geometry']['coordinates'][0][2][0], i['geometry']['coordinates'][0][3][1],
           i['geometry']['coordinates'][0][3][0], i['properties']['cellId'])'''
    mycursor.execute(sql)
    mydb.commit()
```

Code Sample #2

No further change will be needed on this data at any point of the analysis. We have all the cell information needed, stored in the database named 'celliddb'. A sample of the database can be seen in Database Sample 1.

tl_lat	tl_lon	tr_lat	tr_lon	br_lat	br_lon	bl_lat	bl_lon	cellid
45.35880131440966	9.0114910478323	45.35880097314403	9.014491488013135	45.35668565341486	9.0144909480813	45.356685994655464	9.011490619692509	1
45.35880097314403	9.014491488013135	45.358800553060284	9.017491928134044	45.35668523336193	9.017491276410173	45.35668565341486	9.0144909480813	2
45.358800553060284	9.017491928134044	45.35880005415845	9.02049236818262	45.356684734496675	9.020491604666724	45.35668523336193	9.017491276410173	3
45.35880005415845	9.02049236818262	45.35879947643852	9.023492808146456	45.35668415681913	9.023491932838542	45.356684734496675	9.020491604666724	4
45.35879947643852	9.023492808146456	45.35879881990051	9.026493248013145	45.3568350032926	9.02649226091323	45.35668415681913	9.023491932838542	5
45.35879881990051	9.026493248013145	45.35879808454441	9.029493687770275	45.35668276502711	9.029492588878375	45.3568350032926	9.02649226091323	6
45.35879808454441	9.029493687770275	45.35879727037025	9.032494127405446	45.356681950912666	9.032492916721573	45.35668276502711	9.029492588878375	7
45.35879727037025	9.032494127405446	45.358796377378034	9.035494566906245	45.356681057985945	9.035493244430421	45.356681950912666	9.032492916721573	8
45.358796377378034	9.035494566906245	45.35879540556776	9.038495006260266	45.356680086246946	9.03849357199251	45.356681057985945	9.035493244430421	9
45.35879540556776	9.038495006260266	45.35879435493945	9.041495445455103	45.3566790356957	9.041493899395435	45.356680086246946	9.03849357199251	10

Database Sample #1

- Database for the Telecommunication Data

First step is the shapeshifting of the files from '.txt' to '.csv' format. We need the files to be in 'csv' format because then, through a simple SQL command, we can directly import them in a dedicated MySQL database. The code which performs the procedure above can be seen in Code Sample 3.

```
import numpy as np
from numpy import genfromtxt

#nof = number of days in the month
alldata = []
for i in range(1, 4):
    if i == 1:
        tempstr = "2013-11-"
        nof = 30
    elif i == 2:
        tempstr = "2013-12-"
        nof = 31
    else:
        tempstr = "2014-01-"
        nof = 1

    for x in range(1, nof+1):
        loadPath = r"C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\Milano\sms-call-internet-mi-" + str(tempstr)
        savePath = loadPath
        if x < 10:
            loadPath = loadPath + "0" + str(x) + ".txt"
            savePath = savePath + "0" + str(x) + ".csv"
        else:
            loadPath = loadPath + str(x) + ".txt"
            savePath = savePath + str(x) + ".csv"

        print(loadPath)

        ##determine suitable loadpath (where the downloaded data are saved) and savepath and run it
        ##create a loop for inserting the osv data in testdb (currently in mysql)

        data = genfromtxt(loadPath, delimiter='*', filling_values=0)
        np.savetxt(savePath, data, delimiter=",", fmt='%f')
```

Code Sample #3

The logic behind the above is quite simple. Iterating over all the files using exploiting some string concatenation while using 'genfromtxt' and 'savetxt' commands to read and write the files in the desired format.

Then, using the same iterating logic, we will use the 'LOAD DATA INFILE' SQL command to import the 'csv' files directly into the dedicated database (Code Sample 4). The way to do this is creating a unique database table for each of the 62 days' worth of data (counting them from 1 to 62). In the next step, we remove from these tables the unnecessary rows (the ones with phone code different than 39) and after that, the unnecessary columns (phone code, and all the ones skipped in the previous chapter). Code Sample 5 contains the above.

```
        dumbstr = '''CREATE TABLE testdb.day{}
                    (Cell_ID int unsigned, Time_stamp bigint unsigned,
                    phonecode int, sms_in double NULL DEFAULT NULL, sms_out double NULL DEFAULT NULL,
                    call_in double NULL DEFAULT NULL, call_out double NULL DEFAULT NULL,
                    internet_traffic decimal(10,6));'''.format(day)

        mycursor = mydb2.cursor(buffered=True)
        sql = dumbstr
        mycursor.execute(sql)
        mydb2.commit()
        mycursor.close()


        dumbstr = '''LOAD DATA INFILE
                    '{}'
                    INTO TABLE testdb.day{}
                    FIELDS TERMINATED BY ','
                    LINES TERMINATED BY '\n'
                    (Cell_ID, Time_stamp, phonecode, sms_in, sms_out, call_in, call_out, internet_traffic)
                    ;'''.format(loadPath, day)

        mycursor = mydb2.cursor(buffered=True)
        sql = dumbstr
        mycursor.execute(sql)
        mydb2.commit()
        mycursor.close()
```

Code Sample #4

```
for i in range(1, 63):
    print("day{}".format(i))

    mycursor = mydb2.cursor(buffered=True)
    sql = "DELETE FROM testdb.day{} WHERE phonecode <> 39;".format(i)
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()

    dumbstr = '''ALTER TABLE testdb.day{}
                    DROP COLUMN phonecode,
                    DROP COLUMN sms_in,
                    DROP COLUMN sms_out,
                    DROP COLUMN call_in,
                    DROP COLUMN call_out;'''.format(i)

    mycursor = mydb2.cursor(buffered=True)
    sql = dumbstr
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()
```

Code Sample #5

We now have isolated the desired telecommunication data in our database. The database is named 'testdb' and 62 daily tables have been created inside it. They contain only the essential data, which is the Internet usage in every Milan cell, for every 10 minutes throughout the day, for 62 days straight (different tables). No further changes are needed on these tables. When we need some of the data, we will fetch them and process them accordingly. A sample of the first table (first day of data) is provided in Database Sample 2.

	Cell_ID	Time_stamp	internet_traffic
▶	1	1383260400000	11.028366
	1	1383261000000	11.100963
	1	1383261600000	10.892771
	1	1383262200000	8.622425
	1	1383262800000	8.009927
	1	1383263400000	8.118420
	1	1383264000000	8.026270
	1	1383264600000	8.514179
	1	1383265200000	6.833425
	1	1383265800000	6.554605
	1	1383266400000	7.338716
	1	1383267000000	6.779705
	1	1383267600000	7.192162
	1	1383268200000	7.503314
	1	1383268800000	6.169534
	1	1383269400000	7.605452
	1	1383270000000	6.569565

Database Sample #2

We now have gathered and stored all the data we need in MySQL tables. The process of analyzing them and presenting them according to user input follows.

2.4. Data Analysis Procedures

We are ready to continue with the development of an automated procedure that analyses and presents the telecommunication data of the user-chosen route. The purpose of this part of the project is as follows. The user inputs a desired start and end point in the Milan province, the route that connects these points is tracked, and the telecommunication activity along that route is calculated and analyzed according to user's desire. We can later use this information for further investigation of the network itself or combined with a different process altogether for a more complex analysis. We will divide the work in logical steps and proceed linearly.

The first step is to determine the exact route on which we will apply the data analytics. That is, finding the exact square cells from which the route traverses, and take advantage of the telecommunication data we have stored. User inputs the coordinates (longitude and latitude) of the pick-up and drop-off location. For the example procedure that will be presented along, we have set a random set of coordinates for starting and finishing. The user can set these values to anything he wants (inside the Milan province). We use an online tool called 'Project-OSRM', with the help of the 'requests' library (by requesting online using the correct URL). It is a navigation system, which gives the fastest route connecting two points (sets of coordinates, starting and finishing) anywhere in the world. By doing that, we get a set of spatial points that connect the starting and finishing points (including these two). We then have the route we need as a set of coordinates (Code Sample 6). A mapping plot of the route is included to help understanding it, using the 'folium' library (Code Sample 7).

We input the route points (sets of coordinates) in a suitable database table (Code Sample 8). The next step would be to convert the set of coordinate points, to a set of the corresponding Milan cells in which they belong. But there is the risk, that the distance between two consecutive spatial points is greater than the size of the square cells. That means, that an intermediate route cell could be skipped and we need to prevent that from happening. That is why the next step is to call an SQL Procedure ('fullroute' in MySQL) which will insert additional points in the space between consecutive geographical points, until the distance between all the neighbor points is less than the length of the square cell side (Code Sample 9). That is the only way to guarantee that no cells will be missed from the route. The reason we can do the above is the following. 'Project-OSRM' adds a point to the calculated route of points each time there is a change of direction. So there can be a large distance between consecutive points, if the road is straight. Therefore, simply adding points in between causes no error, since they will fall on that same straight piece of road. We add a point between two neighboring ones if and only if, their distance is bigger than a square cell side (otherwise, obviously no cell can be skipped).

Now we have come to acquire the full route with dense points from start to finish. The next step is to iterate over all the route points and find the square cell they belong to (the way this is implemented is by iterating over every square cell and checking if the point falls inside). We have the coordinates of the point we want to investigate, and the coordinates of the corners of all the square cells stored in the database (we fetch the cell data from the database table they are saved into). The 'shapely.geometry' library is used (taking advantage of the classes 'Polygon' & 'Point') to check if the point belongs in the square cell. When we find the cell we are looking for, we add it to an array (if it

is already in the array, we skip it to avoid duplicates). At the end of this procedure we come to know all the cells from which the route passes through, saving them in a database table created for this purpose. The above are executed in Code Sample 10.

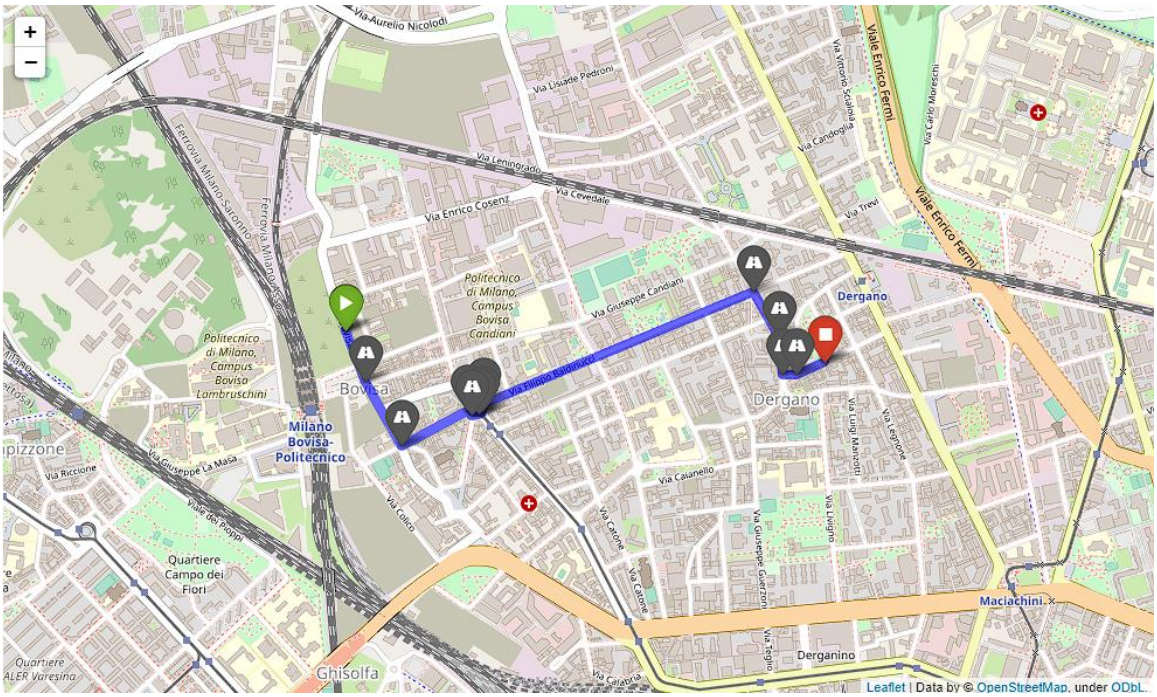
```
import requests
import folium
import polyline
import numpy as np


def get_route(pickup_lon, pickup_lat, dropoff_lon, dropoff_lat):

    url1 = "http://router.project-osrm.org/route/v1/driving/"
    url2 = "{},{};{},{}".format(pickup_lon, pickup_lat, dropoff_lon, dropoff_lat)
    url3 = "?alternatives=false&annotations=true"

    r = requests.get(url1 + url2 + url3)
    if r.status_code != 200:
        return {}

    res = r.json()
    routes = polyline.decode(res['routes'][0]['geometry'])
    start_point = [res['waypoints'][0]['location'][1], res['waypoints'][0]['location'][0]]
    end_point = [res['waypoints'][1]['location'][1], res['waypoints'][1]['location'][0]]
    distance = res['routes'][0]['distance']

    out = {'route':routes,
           'start_point':start_point,
           'end_point':end_point,
           'distance':distance
          }

    return out, routes


# We ask for the user to insert whatever coordinates he desires (in our grid)
# 45.504259, 9.160535
# 45.503461, 9.178174


print("Give me pickup longitude and latitude: ")
input_str = input()
tokens = input_str.split()
a = float(tokens[0])
b = float(tokens[1])

print("Give me dropoff longitude and latitude: ")
input_str = input()
tokens = input_str.split()
```

```
c = float(tokens[0])
d = float(tokens[1])

pickup_lon, pickup_lat, dropoff_lon, dropoff_lat =  a, b, c, d

###


whole_route, help_route = get_route(pickup_lon, pickup_lat, dropoff_lon, dropoff_lat)

print(whole_route)
print(help_route)
```

```
Give me pickup longitude and latitude:
9.160535 45.504259
Give me dropoff longitude and latitude:
9.178174 45.503461
{'route': [(45.50428, 9.16059), (45.50295, 9.16133), (45.50129, 9.16262), (45.50219, 9.16505), (45.50209, 9.16523),
(45.5021, 9.16546), (45.50218, 9.16559), (45.50238, 9.16565), (45.50527, 9.17554), (45.5041, 9.17649), (45.50313, 9.1
7661), (45.50313, 9.17716), (45.50346, 9.17818)], 'start_point': [45.504275, 9.160594], 'end_point': [45.503459, 9.17
8175], 'distance': 1882.3}
[(45.50428, 9.16059), (45.50295, 9.16133), (45.50129, 9.16262), (45.50219, 9.16505), (45.50209, 9.16523), (45.5021,
9.16546), (45.50218, 9.16559), (45.50238, 9.16565), (45.50527, 9.17554), (45.5041, 9.17649), (45.50313, 9.17661), (4
5.50313, 9.17716), (45.50346, 9.17818)]
```

Code Sample #6

```python
def get_map(whole_route, help_route):

    m = folium.Map(location = [(whole_route['start_point'][0] + whole_route['end_point'][0])/2,
                               (whole_route['start_point'][1] + whole_route['end_point'][1])/2],
                   zoom_start = 12)

    folium.PolyLine(
        whole_route['route'],
        weight = 8,
        color = 'blue',
        opacity = 0.6
    ).add_to(m)

    for i in help_route:
        folium.Marker(
            location = i,
            icon = folium.Icon(icon = 'road', color = 'gray', prefix='fa')
        ).add_to(m)

    folium.Marker(
        location = whole_route['start_point'],
        icon = folium.Icon(icon = 'play', color = 'green')
    ).add_to(m)

    folium.Marker(
        location = whole_route['end_point'],
        icon = folium.Icon(icon = 'stop', color = 'red')
    ).add_to(m)


    return m


#del help_route[1]
#del help_route[-1]

get_map(whole_route, help_route)
```



Code Sample #7

```python
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="antbestskadata313BASE!1!9!9!7!",
  database="celliddb"
)


mycursor = mydb.cursor(buffered=True)
sql = "TRUNCATE TABLE celliddb.routecoords;"
mycursor.execute(sql)
mydb.commit()
mycursor.close()



#mycursor = mydb.cursor(buffered=True)
#sql = "CREATE TABLE celliddb.routecoords (uniqid int, latitude double, longitude double);"
#mycursor.execute(sql)
#mydb.commit()
#mycursor.close()

a = 0

for i in help_route:
    mycursor = mydb.cursor(buffered=True)
    sql = "INSERT INTO celliddb.routecoords (uniqid, latitude, longitude) VALUES ({}, {}, {})".format(a, i[0], i[1])
    mycursor.execute(sql)
    mydb.commit()
    a = a + 1
```

Code Sample #8

```python
mycursor = mydb.cursor(buffered=True)
sql = "CALL celliddb.fullroute;"
mycursor.execute(sql)
mydb.commit()
mycursor.close()
```

```sql
DELIMITER //

CREATE PROCEDURE celliddb.fullroute()
BEGIN
DECLARE n INT;
DECLARE i INT;
DECLARE flag TINYINT;
DECLARE y1 DOUBLE;
DECLARE x1 DOUBLE;
DECLARE y2 DOUBLE;
DECLARE x2 DOUBLE;
DECLARE ystep DOUBLE;
DECLARE xstep DOUBLE;

SELECT COUNT(*) FROM celliddb.routecoords INTO n;
SET i = 0;
WHILE i < (n-1) DO
    SET flag=0;

    SELECT latitude FROM celliddb.routecoords WHERE uniqid = i INTO y1;
    SELECT longitude FROM celliddb.routecoords WHERE uniqid = i INTO x1;
    SELECT latitude FROM celliddb.routecoords WHERE uniqid = i+1 INTO y2;
    SELECT longitude FROM celliddb.routecoords WHERE uniqid = i+1 INTO x2;

    SET ystep = ABS(y2-y1);
    SET xstep = ABS(x2-x1);

    CASE
        WHEN (ystep > 0.00210700766) OR (xstep > 0.00299976684)
        THEN UPDATE celliddb.routecoords SET uniqid = uniqid + 1 WHERE uniqid > i;
        ELSE BEGIN END;
```

```
    END CASE;

    CASE
        WHEN (ystep > 0.00210700766) OR (xstep > 0.00299976684)
        THEN INSERT INTO celliddb.routecoords (uniqid, latitude, longitude) VALUES (i+1, (y1+y2)/2, (x1+x2)/2);
        ELSE BEGIN END;
    END CASE;

    CASE
        WHEN (ystep > 0.00210700766) OR (xstep > 0.00299976684)
        THEN SET flag=1;
        ELSE BEGIN END;
    END CASE;




    CASE
        WHEN (flag = 0)
        THEN SET i = i + 1;
        ELSE BEGIN END;
    END CASE;

    SELECT COUNT(*) FROM celliddb.routecoords INTO n;
END WHILE;

END //

#SET SQL_SAFE_UPDATES = 0;
#CALL celliddb.fullroute;
```

Code Sample #9

```
mycursor = mydb.cursor(buffered=True)
sql = "SELECT * FROM celliddb.routecoords ORDER BY uniqid;"
mycursor.execute(sql)
myresult = mycursor.fetchall()
mycursor.close()

```

```
mycursor = mydb.cursor(buffered=True)
sql = "SELECT * FROM celliddb.cellidscoords;"
mycursor.execute(sql)
cellids = mycursor.fetchall()
mycursor.close()
```

```
True
```

```
from shapely.geometry import Point
from shapely.geometry.polygon import Polygon

routecells = []

for i in myresult:
    #print(i)
    point = Point(i[1], i[2])

    for j in cellids:
        polygon = Polygon([(j[0], j[1]), (j[2], j[3]), (j[4], j[5]), (j[6], j[7])])
        checker = polygon.contains(point)

        if (checker == True):
            if j[8] not in routecells:
                routecells.append(j[8])
            break

print(routecells)
```

```
[6950, 6851, 6852, 6953, 6954, 7055, 6955, 6956]
```

```
mycursor = mydb.cursor(buffered=True)
sql = "TRUNCATE TABLE celliddb.routecellids;"
mycursor.execute(sql)
mydb.commit()
mycursor.close()

#mycursor = mydb.cursor(buffered=True)
#sql = "CREATE TABLE celliddb.routecellids (cellid int);"
#mycursor.execute(sql)
#mydb.commit()
#mycursor.close()

for i in routecells:
    mycursor = mydb.cursor(buffered=True)
    sql = "INSERT INTO celliddb.routecellids (cellid) VALUES ({});".format(i)
    mycursor.execute(sql)
    mydb.commit()
    mycursor.close()
```

Code Sample #10

At this point, we have saved in the database the cells from which the user-selected route traverses through. Next step is to isolate the telecommunication data of these cells, since we only need those to describe the network demand around the road we analyze. To carry this task out, we need the daily telecommunication traffic data we have already saved. Using a repeating loop, we will iterate over every daily traffic table, and forward the data we need (which are the ones of the route cells) to dedicated temporary daily database tables. That way, we will have 62 new temporary tables (one for each day) for the daily telecommunication traffic of the cells we are interested on (route cells). Each time this algorithm is rerun, the temporary tables will be truncated, and refilled with the new useful data we want to analyze (Code Sample 11).

We take an extra step, to calculate the average daily telecommunication traffic (the time step is 10-minutes as explained before) from the 62 days of data for the route cells. We also calculate the standard deviation for every value and accordingly the coefficient of variation. We concatenate these values in a single 'average daily values' table (Code Sample 12). We will use these data for plotting and further processing.

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="antbestskadata313BASE!1!9!9!7!",
  database="celliddb"
)


mycursor = mydb.cursor(buffered=True)
sql = "SELECT * FROM celliddb.routecellids;"
mycursor.execute(sql)
myresult = mycursor.fetchall()
mycursor.close()

route = []

for x in myresult:
    route.append(x[0])
```

```
mydb2 = mysql.connector.connect(
  host="localhost",
  user="root",
  password="antbestskadata313BASE!1!9!9!7!",
  database="testdb"
)

dumbstr = ", ".join("{:d}".format(i) for i in route)
print("Cell IDs: " + dumbstr)

for day in range(1, 63):

    tempstr = "TRUNCATE TABLE testdb.tempday" + str(day) + ";"

    mycursor = mydb2.cursor(buffered=True)
    sql = tempstr
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()

    tempstr = "INSERT INTO testdb.tempday{} SELECT * FROM testdb.day{} WHERE Cell_ID in ({});".format(day, day, dumb

    mycursor = mydb2.cursor(buffered=True)
    sql = tempstr
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()

    print("Created temp database for day {}".format(day))

#Inserting the needed data (with the specific cellids) in temp day table (not sorted in cellids).
```

Code Sample #11

```
alldata = []

for day in range(1, 63):

    tempstr = "SELECT * FROM testdb.tempday" + str(day) + ";"

    mycursor = mydb2.cursor(buffered=True)
    sql = tempstr
    mycursor.execute(sql)
    data = mycursor.fetchall()
    mycursor.close()

    data = np.array(data)

    alldata.append(data)
    #print(np.shape(temp))

avtraff = alldata[0]

for i in range(np.shape(alldata)[1]):
    isum = 0
    for x in range(np.shape(alldata)[0]):
        isum += alldata[x][i][2]

    iav = isum/np.shape(alldata)[0]
    avtraff[i][2] = iav

for i in range(np.shape(avtraff)[0]):
    avtraff[i][1] -= 1383260400000.0
    avtraff[i][1] /= 600000.0
    avtraff[i][1] += 1



temp = avtraff

avtraff = np.zeros((np.shape(temp)[0],np.shape(temp)[1]+2))
avtraff[:,:-2] = temp
```

```
for i in range(np.shape(alldata)[1]):
    sd = 0
    for x in range(np.shape(alldata)[0]):
        sd += pow(avtraff[i][2]-alldata[x][i][2], 2)

    sd = math.sqrt(sd/np.shape(alldata)[0])
    avtraff[i][3] = sd

for i in range(np.shape(avtraff)[0]):
    Cv = avtraff[i][3]/avtraff[i][2]
    avtraff[i][4] = Cv


mycursor = mydb2.cursor(buffered=True)
sql = "TRUNCATE TABLE testdb.temptimeav;"
mycursor.execute(sql)
mydb2.commit()
mycursor.close()

for row in avtraff:
    mycursor = mydb2.cursor(buffered=True)
    sql = "INSERT INTO testdb.temptimeav (Cell_ID, Time_stamp, int_time_av, sd, cv) VALUES ({}, {}, {}, {}, {});".f
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()


print("Created temp database for average daily internet.")
```

Code Sample #12

We have now configured the data in a handy way to process them and plot the results. The route-specific daily tables and the corresponding average daily traffic table have all the information we need to continue with the analysis of the specific route. One last procedure takes place before moving on. We want the data in the average daily traffic table to be sorted by the cell sequence in the route (data of the first route cell go first, etc.). So we sort the average daily traffic table using the cell sequence and re-store the data in the same table. The algorithm for that is in Code Sample 13.

```
import numpy as np
import mysql.connector
import math

#THIS PROGRAM REARRANGES THE (TIME) AVERAGE INTERNET TRAFFIC IN THE CELL ORDER THAT WE WANT

mydb2 = mysql.connector.connect(
  host="localhost",
  user="root",
  password="antbestskadata313BASE!1!9!9!7!",
  database="testdb"
)


print(np.shape(data))

data = np.array(data)
datanew = []

#correct order for cells is in route

for i in range(len(route)):
    flag = route[i]

    for row in data:
        if row[0] == flag:
            datanew.append(row)

```

```
mycursor = mydb2.cursor(buffered=True)
sql = "TRUNCATE TABLE testdb.temptimeav;"
mycursor.execute(sql)
mydb2.commit()
mycursor.close()

for row in datanew:
    mycursor = mydb2.cursor(buffered=True)
    sql = "INSERT INTO testdb.temptimeav (Cell_ID, Time_stamp, int_time_av, sd, cv) VALUES ({}, {}, {}, {}, {});".f
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()

print("Rearranged average daily internet in route order.")
```

Code Sample #13

We are ready for the next part of the analysis. The first step for visualizing the information we have isolated, is a time dependent plot. We will plot the average daily traffic of the route cells for the 24-hour duration starting at 12 am (the day is divided in 10-minutes, so we have 144 traffic values in the day for each cell). The code that performs this can be seen in Code Sample 14.

Another plot that can give us information about the uncertainty of the telecommunication traffic on the road is a time dependent telecommunication traffic plot of a single cell (so it can be easier understood) with its standard deviation as well as the coefficient of variation against time plot. These values have already been calculated and stored in the average daily traffic database, so we just fetch the data and plot them in Code Samples 15 & 16.

One last time dependent plot we will try is a bar plot of specific cells of the route in order to easier identify a pattern of telecommunication traffic moving from one cell to another throughout the day (Code Sample 17).

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

data = np.array(data)

i = 0
#data refer to the average telecom data of the road (sorted by cell order)
plt.figure(figsize=(14,10))
plt.title("Time Dependence (Temp Milano Road)")
plt.xlabel("# of 10-minutes in the day (144 in total, starting at 12 am)")
plt.ylabel("Average Internet traffic")

while i < np.shape(data)[0]:
    label = data[i][0]
    plt.plot(data[i:i+144,1], data[i:i+144,2], label= "{}".format(label))
    i+=144

plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0.)
plt.show()
```

Code Sample #14

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

# i here will represent what cell of the road we want to see
i = 0

plt.figure(figsize=(16,10))
plt.title("Time Dependence with Standard Deviation (Temp Milano Road)")
plt.xlabel("# of 10-minutes in the day (144 in total, starting at 12 am)")
plt.ylabel("Average Internet traffic")

label = data[i*144][0]
plt.errorbar(x=data[i*144:i*144+144,1], y=data[i*144:i*144+144,2], yerr=data[i*144:i*144+144,3], color='blue', label
plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0.)
plt.show()
```

Code Sample #15

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

# i here will represent what cell of the road we want to see
i = 0

plt.figure(figsize=(16,10))
plt.title("Coefficient of Variation (Temp Milano Road)")
plt.xlabel("# of 10-minutes in the day (144 in total, starting at 12 am)")
plt.ylabel("Cv")

label = data[i*144][0]
plt.plot(data[i*144:i*144+144,1], data[i*144:i*144+144,4], color='blue', label= "{}".format(label))

plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0.)
plt.show()
```

Code Sample #16

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

n = 0
temparray = []

s = 1  ##
k = 3  ##
# s and k will represent how many different cells we want to plot (starting and ending of the road, s < k)
# figure size might be important here

plt.figure(figsize=(15,5))

i = (s-1)*144
while i < k*144:
    label = data[i][0]

    stime = 50  ## this variable will represent the starting 10-minute of the plotting of the day
    etime = 60  ## same as above but ending 10-minute of the day (must be bigger than the above and up to 144)

# n variable will help us at separating the traffic of different cells at the same time spot, by altering x axis a

    plt.bar(data[i+stime:i+etime,1]+((n-1)*0.1), data[i+stime:i+etime,2], width=0.1, label= "{}".format(label))
    i+=144
    n+=1
    temparray.append(label)

plt.title("Time Dependence, Bar plot (Temp Milano Road, cells = {})".format(temparray))
plt.xlabel("# of 10-minutes in the day (144 in total, starting at 12 am)")
plt.ylabel("Average Internet traffic")
plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0.)
plt.show()
```

Code Sample #17

The next plot attempt will aim to find space dependency patterns. But first, we need to calculate the average spatial telecommunication traffic. The way to do it is, take the telecommunication data of a cell for an average day and calculate the average telecommunication traffic of that cell throughout the day (or a specific time window). Since there is a large deviation of the traffic throughout the day (network usage peaks around rush hours, and is greatly reduced during the night), it is better to choose time windows that minimize the deviation as much as possible, so that the results are reliable.

The spatial distribution of the telecommunication traffic will be plotted with two different approaches. Firstly, we calculate the average telecommunication traffic of each cell of the route during a period we define as network rush hour. That period is approximately 10am-6pm. The standard deviation is significantly lower when talking about that time period alone. We calculate the average telecom traffic for each cell during that time window as well as the standard deviation and the coefficient of variation and save the results in a dedicated table. The above are performed in Code Sample 18.

A different approach based on the same idea is, choosing multiple shorter time periods through the day to calculate the average cell traffic. That way, the precision of the results is increased since we the variation is further decreased with smaller time windows. Another advantage is that we can plot all the different time windows together and detect patterns of traffic movement from cell to cell for different time periods. The code for calculating the average spatial traffic for short time periods is identical with the last application. We will display one more example for the time period 11:30am-2pm (only the time limits change) in Code Sample 19, and the code for more time windows is easily derived from the previous ones (by only changing the time limits).

Code Samples 20 and 21 are the ones for the plot themselves. On Code Sample 20, we plot the average spatial telecom traffic for the route during network rush hours together with the standard deviation, as calculated before. On Code Sample 21, we do an identical plot for the average spatial telecom traffic of 3 different time periods in the day, for which we did the calculations during the previous steps.

```
temp = np.array(temp)
spav = []

# automatically sets order by the fixed prefix (prefix contains the route cells ordered)

for i in range(len(prefix)):
    flag = prefix[i]

    isum = 0
    n = 0
    for row in temp:
        if row[0] == flag and row[1] >= 60 and row[1] <= 110:
            isum += row[2]
            n += 1

    ispav = isum/n
    # We calculate the the average from time slots 60 to 110 which is approximately 10am - 6pm,
    # where we consider it to be the rush hour with much less standard deviation.

    k = 0
    sd = 0
    for row in temp:
        if row[0] == flag and row[1] >= 60 and row[1] <= 110:
            sd += pow(ispav - row[2], 2)
            k += 1

    sd = math.sqrt(sd/k)
    Cv = sd/ispav

    temprow = [prefix[i], ispav, sd, Cv]
    spav.append(temprow)



mycursor = mydb2.cursor(buffered=True)
sql = "TRUNCATE TABLE testdb.tempspaceav"
mycursor.execute(sql)
mydb2.commit()
mycursor.close()


for row in spav:
    mycursor = mydb2.cursor(buffered=True)
    sql = "INSERT INTO testdb.tempspaceav (Cell_ID, int_space_av, sd, cv) VALUES ({}, {}, {}, {});".format(row[0], r
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()
```

Code Sample #18

```
spav = []

# automatically sets order by the fixed prefix

for i in range(len(prefix)):
    flag = prefix[i]

    isum = 0
    n = 0
    for row in temp:
        if row[0] == flag and row[1] >= 70 and row[1] <= 85:
            isum += row[2]
            n += 1
    ispav = isum/n
    # We calculate the the average from time slots 70 to 85 which is approximately 11:30 - 14:00

    k = 0
    sd = 0
    for row in temp:
        if row[0] == flag and row[1] >= 70 and row[1] <= 85:
            sd += pow(ispav - row[2], 2)
            k += 1

    sd = math.sqrt(sd/k)
    Cv = sd/ispav

    temprow = [prefix[i], ispav, sd, Cv]
    spav.append(temprow)
```

```
mycursor = mydb2.cursor(buffered=True)
sql = "TRUNCATE TABLE testdb.tempspaceavtest1"
mycursor.execute(sql)
mydb2.commit()
mycursor.close()


for row in spav:
    mycursor = mydb2.cursor(buffered=True)
    sql = "INSERT INTO testdb.tempspaceavtest1 (Cell_ID, int_space_av, sd, cv) VALUES ({}, {}, {}, {});".format(row
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()
```

Code Sample #19

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

mycursor = mydb2.cursor(buffered=True)
sql = "SELECT * FROM testdb.tempspaceav;"
mycursor.execute(sql)
temp = mycursor.fetchall()
mycursor.close()

temp = np.array(temp)

my_xticks = prefix

plt.figure(figsize=(14,10))
plt.title("Space Dependence with Standard Deviation (Temp Milano Road, during rush hour, 10am - 6pm)")
#plt.ylim(0, 600)
plt.xlabel("# of Cell-id")
plt.ylabel("Average Internet traffic")
xt = np.linspace(1, np.shape(temp)[0], np.shape(temp)[0])
plt.errorbar(x=xt, y=temp[:,1], yerr=temp[:,2], color='blue', marker='x')
plt.xticks(xt, my_xticks)
plt.figtext(.2, .8, "[The sequence of cell-id's is based on their geographical position so to represent the road!]")
plt.show()
```

Code Sample #20

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

mycursor = mydb2.cursor(buffered=True)
sql = "SELECT * FROM testdb.tempspaceavtest1;"
mycursor.execute(sql)
temp1 = mycursor.fetchall()
mycursor.close()

temp1 = np.array(temp1)

mycursor = mydb2.cursor(buffered=True)
sql = "SELECT * FROM testdb.tempspaceavtest2;"
mycursor.execute(sql)
temp2 = mycursor.fetchall()
mycursor.close()

temp2 = np.array(temp2)

mycursor = mydb2.cursor(buffered=True)
sql = "SELECT * FROM testdb.tempspaceavtest3;"
mycursor.execute(sql)
temp3 = mycursor.fetchall()
mycursor.close()

temp3 = np.array(temp3)


my_xticks = prefix

plt.figure(figsize=(14,10))
plt.title("Space Dependence with Standard Deviation (Temp Milano Road, during different timestamps)")
#plt.ylim(100, 550)
```

```
plt.xlabel("# of Cell-id")
plt.ylabel("Average Internet traffic")
xt = np.linspace(1, np.shape(temp1)[0], np.shape(temp1)[0])
plt.xticks(xt, my_xticks)
plt.figtext(.2, .8, "[The sequence of cell-id's is based on their geographical position so to represent the road!]")

label1 = '11:30 - 14:00'
plt.errorbar(x=xt, y=temp1[:,1], yerr=temp1[:,2], color='blue', marker='x', label= "{}".format(label1))

label2 = '14:00 - 16:30'
plt.errorbar(x=xt, y=temp2[:,1], yerr=temp2[:,2], color='red', marker='x', label= "{}".format(label2))

label3 = '16:30 - 19:00'
plt.errorbar(x=xt, y=temp3[:,1], yerr=temp3[:,2], color='black', marker='x', label= "{}".format(label3))

plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0.)
plt.show()
```

Code Sample #21

One more data analysis approach we will try is with Gaussian Processes. In the Code Sample 22, we will plot the average daily telecom traffic for a random cell in the route as dots so we can have a broader image of its form. Afterwards, in Code Sample 23, we apply Gaussian Process Regressors, to fit the discrete data we have with a (predicted) continuous function, which will satisfactorily describe the shape of the telecommunication traffic through the day.

```
mycursor = mydb2.cursor(buffered=True)
sql = "SELECT * FROM testdb.temptimeav;"
mycursor.execute(sql)
temp = mycursor.fetchall()
mycursor.close()

temp = np.array(temp)

# i here will represent what cell of the road we want to see
i = 0

#n = 0

label = temp[i*144][0]
#color = prefix[n]

plt.figure(figsize=(16,10))
plt.title("Time Dependence (Temp Milano Road)")
plt.xlabel("# of 10-minutes in the day (144 in total, starting at 12 am)")
plt.ylabel("Average Internet traffic")
plt.plot(temp[i*144:i*144+144,1], temp[i*144:i*144+144,2], marker=".", linestyle='None', label= "{}".format(label))
plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0.)
plt.show()
```

Code Sample #22

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, RationalQuadratic, ExpSineSquared, ConstantKernel, Ma

# i here will represent what cell of the road we want to see
i = 0

#kernels = [166* Matern(length_scale=14, nu=1.5) + WhiteKernel(noise_level=1e+02)]
kernels = [166* Matern(length_scale=14, nu=1.5) + WhiteKernel(noise_level=1e+01)]

for kernel in kernels:

    gp = GaussianProcessRegressor(kernel=kernel, alpha=0, copy_X_train=True, n_restarts_optimizer=0,
                                  optimizer=None, normalize_y=True, random_state=None)

```

```
    X = temp[i*144:i*144+144,1]
    y = temp[i*144:i*144+144,2]

    X = np.atleast_2d(X).T
    y = np.atleast_2d(y).T

    gp.fit(X, y)

x1 = np.atleast_2d(np.linspace(0, 144, 30)).T

y_pred, sigma = gp.predict(x1, return_std=True)

plt.figure(figsize=(14,10))
plt.plot(X, y, 'r.', markersize=10, linestyle='None', label='Observations')
plt.plot(x1, y_pred - sigma, 'b-', label='Prediction',  lw=2, zorder=2)


#plt.fill(np.concatenate([x1, x1[::-1]]),
#         np.concatenate([y_pred - 5 * sigma,
#                         (y_pred + 5 * sigma)[::-1]]),
#          alpha=0.1, fc='b', ec='None', label='95% confidence interval')

plt.title("Time Dependence with predicted distribution (Temp Milano Road)")
plt.xlabel('time')
plt.ylabel('internet traffic')
#plt.legend(loc='upper left')
plt.show()
```

Code Sample #23

Lastly, we will try and create a Correlation Matrix for the telecommunication traffic of all the different cells in the route. This way, we can find a connection for the network usage between different cells and draw conclusions about correlations. To begin with, we have to convert the data in a form applicable for the correlation function. The data have to be in a table with the first row containing the Cell IDs (identity of the data below), and below each ID, the list of its 144 telecommunication traffic values. We save the result in a dedicated 'csv' file (we need to, in order to read it later using Pandas Dataframe). The above can be seen in Code Sample 24.

Code Sample 25 focuses on plotting the Correlation Matrix. The data (in the correct form) are fetched from the csv file using the Pandas library. This is because, Pandas gives the capability of using the '.corr()' function. This function automatically calculates the values of the correlation matrix, leaving us with only the plotting. The linear correlation matrix (in route cells sequence) is easily displayed using the 'heatmap' function of Seaborn library. One more option is added, called 'clustermap'. This display, organizes the data categories (cells) in correlation order, and groups them accordingly.

```
mycursor = mydb2.cursor(buffered=True)
sql = "SELECT * FROM testdb.temptimeav;"
mycursor.execute(sql)
temp = mycursor.fetchall()
mycursor.close()

temp = np.array(temp)

print(np.shape(temp))

olddata = []

i = 0

while i < np.shape(temp)[0]:
    temparray = [float(temp[i,0])]
```

```
    for k in range(144):
        temparray.append(float(temp[i+k,2]))

    #print(np.shape(temparray))

    olddata.append(temparray)
    i+=144

print(np.shape(olddata))

#print(olddata)

data = np.zeros((np.shape(olddata)[1], np.shape(olddata)[0]), dtype = float)

for i in range(np.shape(olddata)[0]):
    for j in range(np.shape(olddata)[1]):
        data[j][i] = float(olddata[i][j])

print(np.shape(data))

np.savetxt(r"C:\Users\Antonis\Documents\Jupyter Notebooks\Thesis\Test\CorrMatTest\TempMiRoad2.csv", data, delimiter=
```

Code Sample #24

```
# Milano Road chosen by the user
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from scipy.stats import norm


### Manually creating the axis

temparray = []
i = 0

while i < np.shape(temp)[0]:
    temparray.append(int(temp[i,0]))
    i+=144

###

data = pd.read_csv(r"C:\Users\Antonis\Documents\Jupyter Notebooks\Thesis\Test\CorrMatTest\TempMiRoad2.csv")
print(np.shape(data))

###

corrmat = data.corr()

f, ax = plt.subplots(figsize =(9, 8))
sns.heatmap(corrmat, ax = ax ,cmap ="YlGnBu", linewidths = 0.1, annot=True, xticklabels = temparray, yticklabels = t

###

corrmat = data.corr()

cg = sns.clustermap(corrmat, cmap ="YlGnBu", linewidths = 0.1, annot=True, xticklabels = temparray, yticklabels = t
plt.setp(cg.ax_heatmap.yaxis.get_majorticklabels(), rotation = 0)

cg
```

Code Sample #25

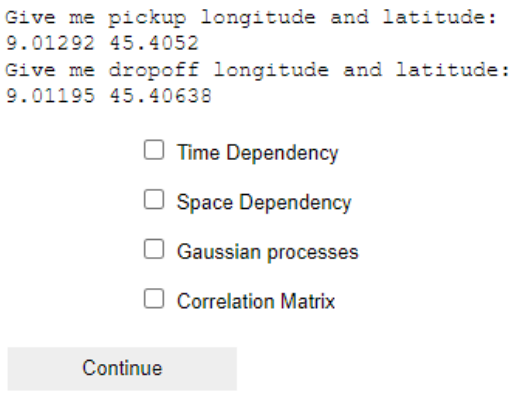
An additional interesting task we can develop is a type of user interface for the user so that they can input the needed coordinates and choose which of the outputs they want to be displayed. For this task, we will use the Ipywidgets library, create a list of checkboxes (the desired outputs for the user) and add a 'Continue' button which will start the analysis procedure and calculations according to the user's choice. To perform this, we will use a different Jupyter Notebook, and run the notebooks we have already developed remotely, depending on if the user has checked the according checkbox. The code that performs the above is the following (Code Sample 26).

```
import ipywidgets as widgets


print("Give me pickup longitude and latitude: ")
input_str = input()
tokens = input_str.split()
a = float(tokens[0])
b = float(tokens[1])

print("Give me dropoff longitude and latitude: ")
input_str = input()
tokens = input_str.split()
c = float(tokens[0])
d = float(tokens[1])

pickup_lon, pickup_lat, dropoff_lon, dropoff_lat =  a, b, c, d



data = ['Time Dependency', 'Space Dependency', 'Gaussian processes', 'Correlation Matrix']
checkboxes = [widgets.Checkbox(value=False, description=label) for label in data]
output = widgets.VBox(children=checkboxes)
display(output)


selected_data = []


button = widgets.Button(description='Continue')
out = widgets.Output()


def on_button_clicked(_):
    global selected_data
    # "linking function with output"
    with out:
          # what happens when we press the button
        #clear_output()
        for i in range(0, len(checkboxes)):
            if checkboxes[i].value == True:
                selected_data = selected_data + [checkboxes[i].description]
        print(selected_data)

        if selected_data == []:
            print("Error: No choices checked")
        else:
            #we need to give the scanned coords to coordsnew like a function, then data go to database and
            #rest of notebooks are the same

            %run ./UITempCoordsnew[FULL].ipynb

            %run ./TempCreateFull.ipynb

            if 'Time Dependency' in selected_data:
                %run ./TempTimeDependency.ipynb
            if 'Space Dependency' in selected_data:
                %run ./TempSpaceDependency.ipynb
            if 'Gaussian processes' in selected_data:
                %run ./TempGaussian.ipynb
            if 'Correlation Matrix' in selected_data:
                %run ./TempCorrMat.ipynb



# linking button and function together using a button's method
button.on_click(on_button_clicked)
# displaying button and its output together
widgets.VBox([button,out])
```

Code Sample #26

The UI resulting from the code above can be seen below (User Interface). We have already inserted pick-up and drop-off longitude and latitude and then the checkboxes appear. User then picks the desired ones and presses 'Continue'. Then the program runs the first two codes we developed, required for the initial pre-processing and cleansing of the data associated with the user-chosen route.

After that, all the notebooks according to the user's choices are executed and the results are displayed on the result pane, below the notebook. This is also handier, because we get all the results concentrated and it is easier to study them.



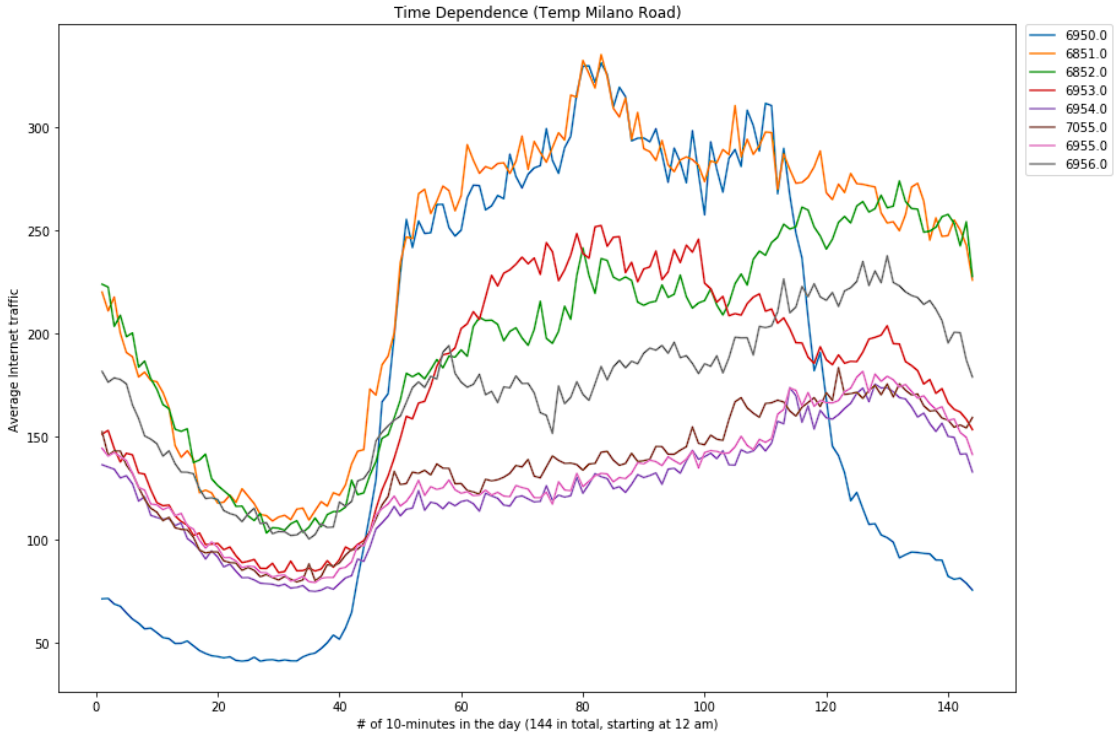
User Interface

This concludes the full data analysis procedure. We have all the data needed saved in dedicated databases, and every time, according to users input, the same automated sequence of codes is executed, printing all the results we need.

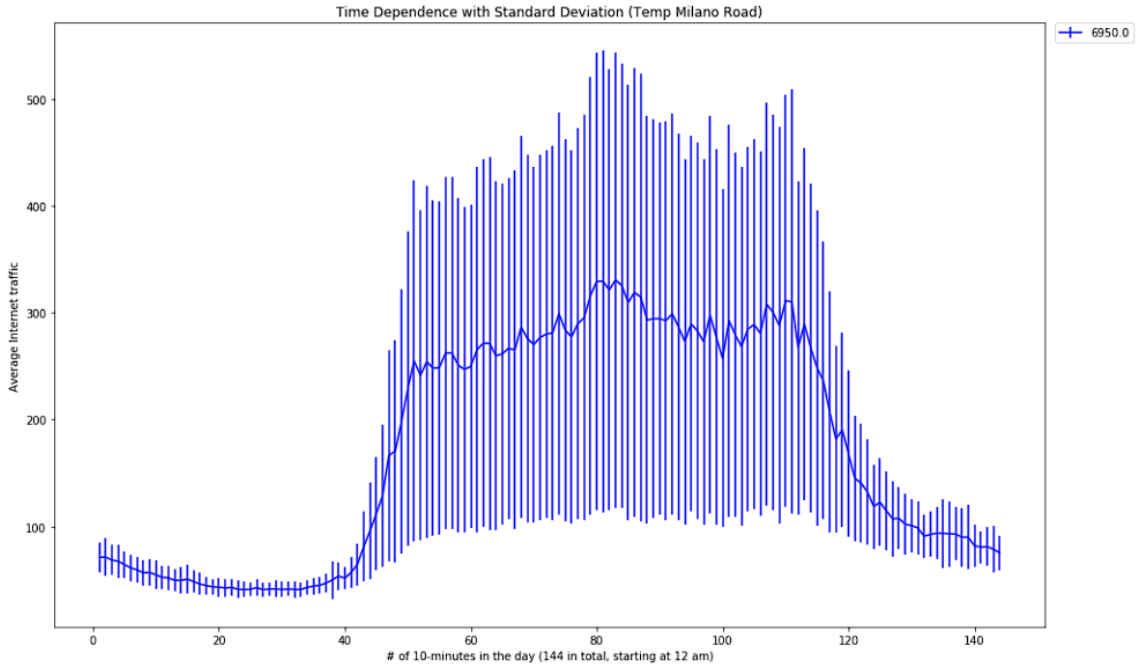
2.5. Results and Conclusions

The plot results, that will be included as an example of the procedure, belong to the route that we randomly chose at the start (user inputs the coordinates). All the detailed plots of specific parts of the route/data are also randomly chosen and can be altered easily because of the generalization of the codes (when a specific part is plotted, the variables that describe that part, are visibly separated at the top of the code). The results will be displayed and commented on.

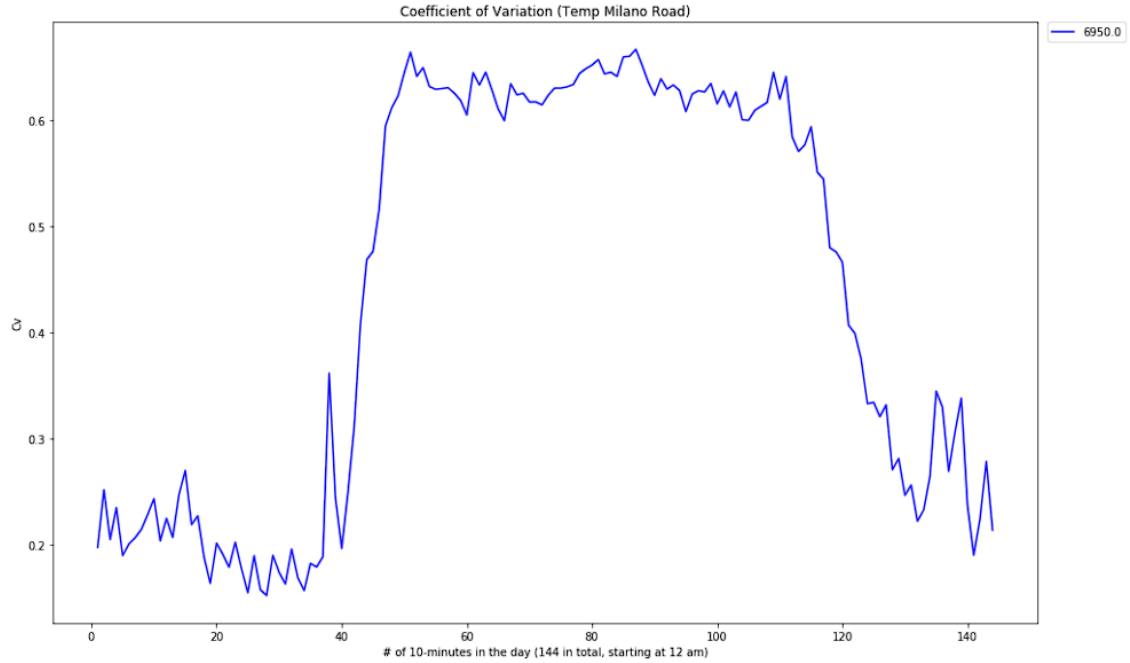
Time Dependency Results 1-4 (corresponding to Code Samples 23-26):



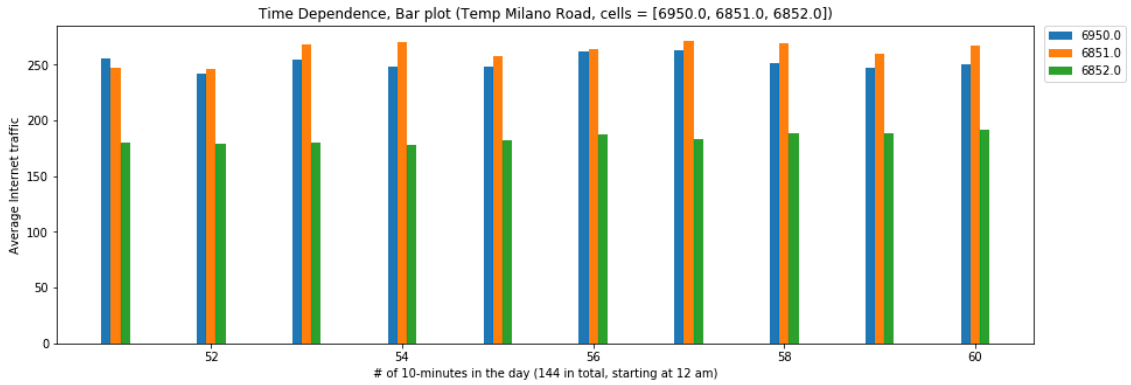
Time Dependency Result #1



Time Dependency Result #2



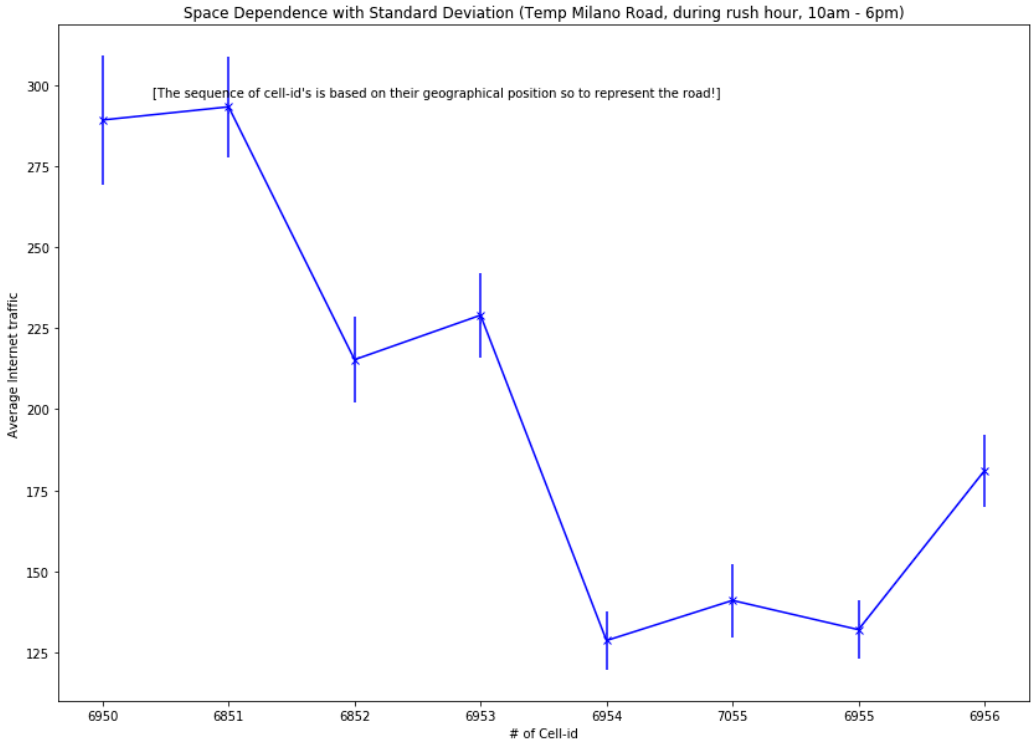
Time Dependency Result #3



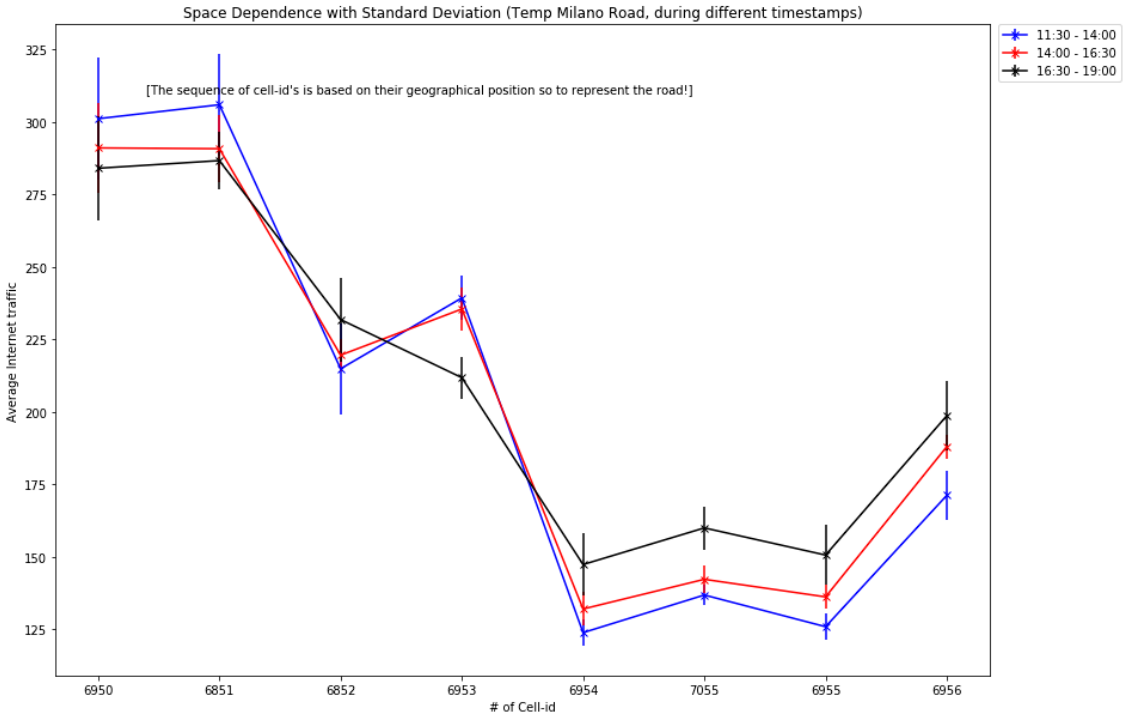
Time Dependency Result #4

The results are as expected. Traffic heats the lowest early in the morning and peak network traffic falls together with regular traffic. The pattern is the same for every cell, but the value of the traffic deviates depending on the density of the geographical area. Standard deviation is insignificant in the early morning hours, meaning that every day the network needs during these hours decrease. On the other hand, during rush hours, the standard deviation is quite significant compared to the total telecommunication traffic. This means that network usage during the day is highly dependent on the day itself. For example, weekday traffic is quite larger than weekends or holidays. Coefficient of variation also gives us the same information. On the bar plot, we can recognize different traffic growth patterns during a part of the day. That means, network usage is spatially dependent and moves geographically during the day in our route cells. Multiple conclusions like these can be drawn, depending our needs and desires.

Space Dependency Results 1-2 (corresponding to Code Samples 29-30):



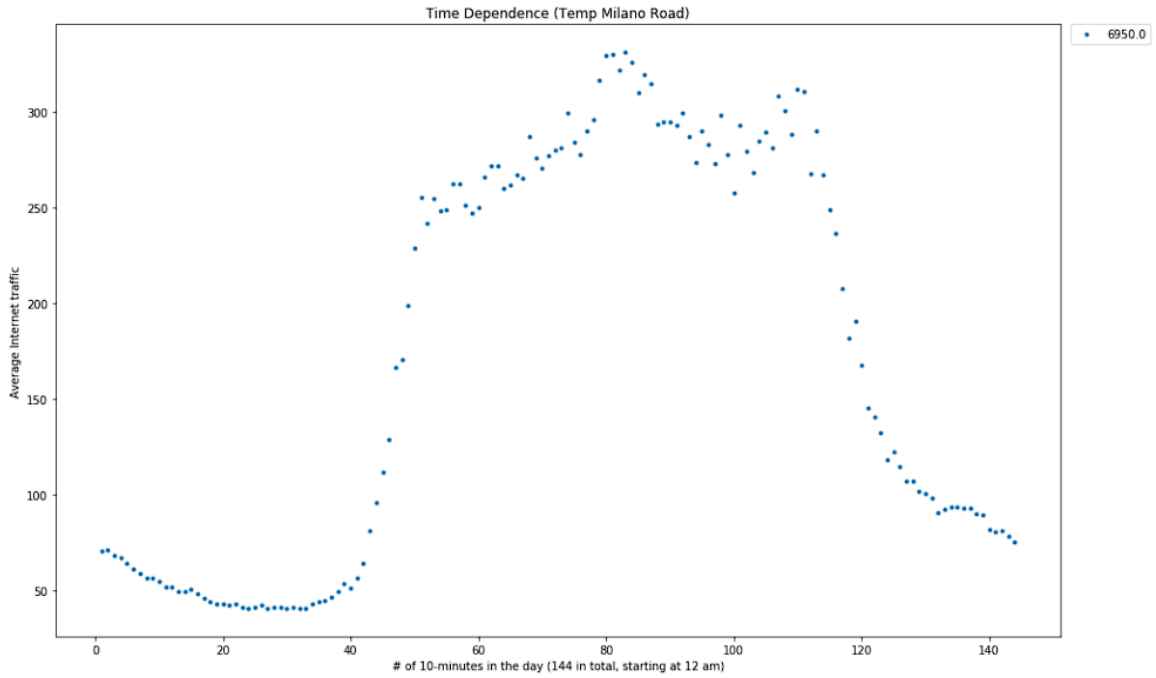
Space Dependency Result #1



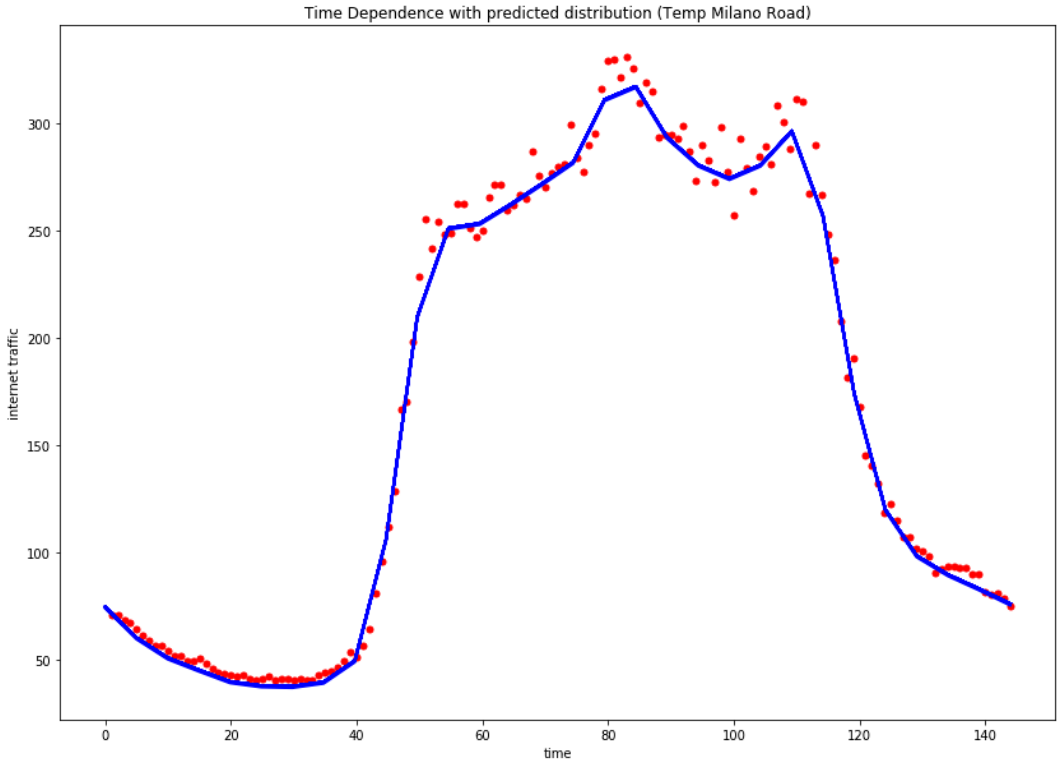
Space Dependency Result #2

Different telecommunication traffic values depending on the geographical area which is expected. Low standard deviation shows that spatial deviation is almost time constant and dependent to the population density or even business/recreational areas. The plot with the 3 different time periods on the other hand, shows that the route telecom traffic (which we considered time constant on spatial dependency) geographically moves on different times of the day. Telecommunication traffic is time dependent and it can move from cell to cell depending on the local human activity during the day.

Gaussian Processes Results 1-2 (corresponding to Code Samples 31-32):



Gaussian Processes Result #1



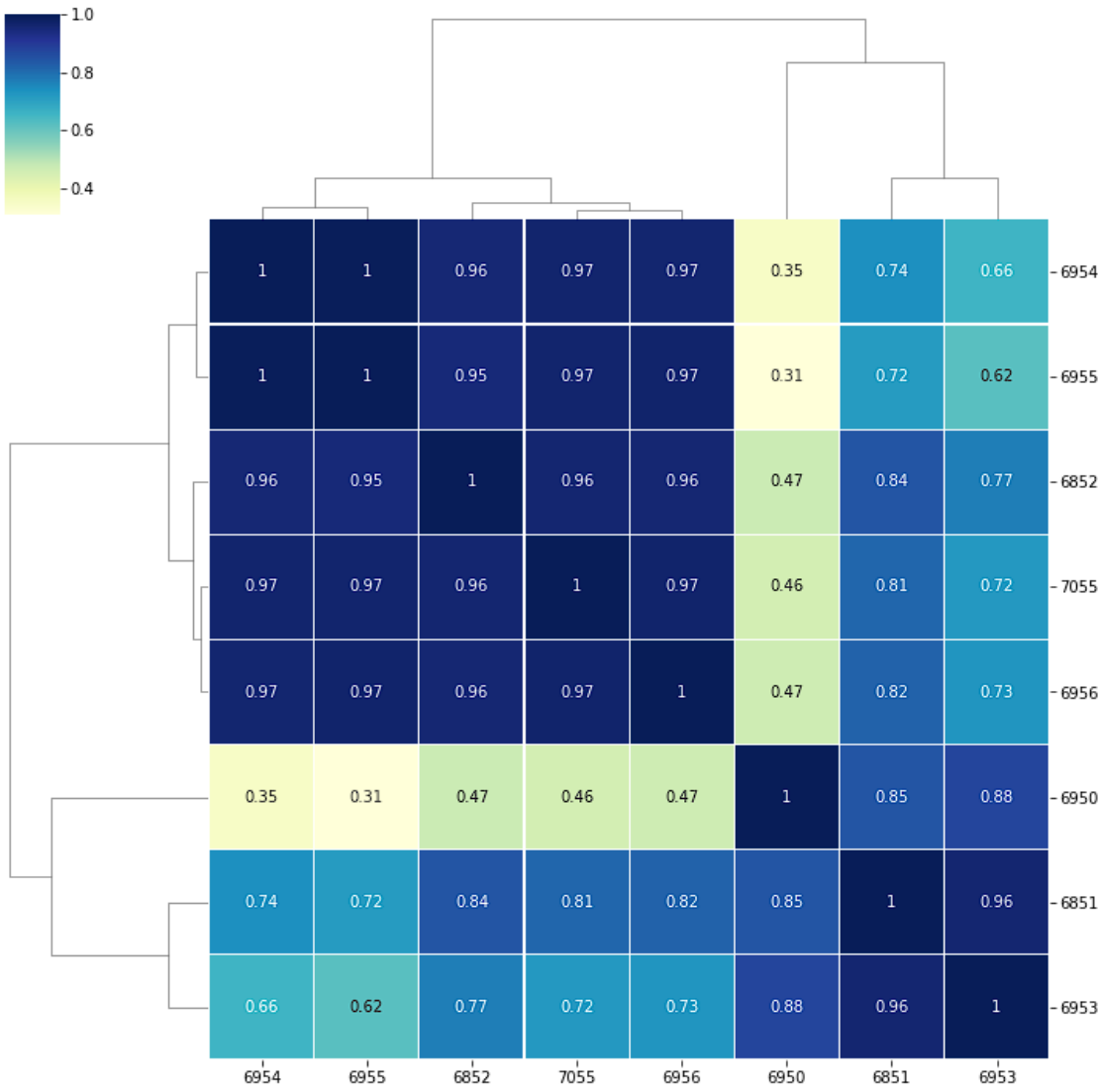
Gaussian Processes Result #2

A continuous function to approximately represent the average daily traffic is successfully calculated. An application like that (assuming constant day to day telecom traffic) can give us an approximation for the desired value (telecommunication traffic) in any cell of the route anytime in the day. The result is aligned with what we have seen already and confirms the difference in network usage through the day.

Correlation Matrices Results 1-2 (corresponding to Code Sample 34):

Correlation Matrices Result #1

The correlation matrix above portrays the exact relationship between all the different cells of the route. We can see that neighboring cells (the closer we are to the main diagonal) have the highest values for the correlation, which is expected (we expect small deviation between neighboring cells).



Correlation Matrices Result #2

One more display that can provide useful information is the clustering correlation matrix. In this type of plotting, as we can see, the cells are organized in a way to group cells with high correlation. That means that the cell sequence is sorted in a correlation way, to find patterns about the geographical dependency of the telecommunication traffic.

These were the results the code sequence produced for the example route. Users choose the exact coordinates they want, as well as the specific variables when the route is split to smaller parts.

The entire procedure lasts about 2-3 minutes on average (depending on the route length), which is significantly faster compared to using numpy arrays instead of database tables for the data storage. Considering the amount of data the tool has to process and calculate to get the results, we can consider it efficient.

The results this project produces can give useful information about the general network usage. They can also give indications on the needs of the network, and ways to improve and develop it. Apart from that, the results and the codes in general can be used as a stepping stone in different projects, so that, combined with additional analysis, they can offer necessary insights for the network betterment.

An example like that will be given in the second part of the thesis. For that project we will use the work done here, and combined with additional data and processes we will take it a step further.

3. Car Traffic – Telecommunication Activity Correlation

3.1. Introduction

In this part of the thesis we will deal with calculating the correlation of telecommunication activity and car traffic in the Milan province. We will use published data about car traffic on the roads of Milan as well as the data used in the previous project. They will be handled and processed according to the current's job needs. It is one more task that requires database handling and big data analytics. The previous task (calculating the telecommunication traffic on any Milan route) will be taken advantage of. Combined with the new data analysis procedure, it will produce entirely different conclusions for an entirely different project.

The ultimate goal is to create an automated procedure which, given the data of the road we want to study, will give a result about the car traffic – telecom activity correlation of that specific road. In other words, by what factor is the network usage determined by mobile users. This information can give us an insight about the network resource needs for servicing mobile users. This category of users is special, because their ongoing call or data session needs to be transferred to the next base station of the cellular network. This is a process known as handover, and its application needs extra network functionality in comparison to servicing spatially stable users. Knowing beforehand, which parts of the Milan province have greater handover handling needs, can be an important beneficial factor towards strengthening and optimizing the network itself (could be work for a different project).

We will present the data, create fitting database tables and import them, and create an automated sequence of codes which will fetch the specific data we need all the way to calculating the desired correlation. After that, we will validate the tool's results, comparing them to what we would expect for specific roads according to their attributes.

3.2. Presenting the Data

The extra dataset we are going to need for this work is about car traffic on Milan roads. It has been provided by the mapping and location technology company 'TomTom'. The format the data were published was in excel tables. One table for every hour of the day (12:00-01:00, etc.), every one of which presents the number of cars passing from every road of the Milan province during that specific hour. The data we have in our disposal are the average cars passing from each road during that specific hour for the duration of a month. These data are more recent compared to the telecommunication activity ones, but, given the fact that both the datasets are extended on long periods of time, we make the assumption that they both describe the general patterns of network and road usage, regardless of the day.

We mentioned the Milan province roads. There is one more table dedicated on the specific information of these roads. This table we have available is describing more than 150.000 different road parts throughout the Milan region, for which we have the starting and finishing coordinates, the road name etc. Furthermore, every road chunk is characterized by a unique key (BS_Id). This key is used in the 24 tables representing the automobile traffic of the roads for every hour of the day. A sample of the car traffic data and the road attributes can be seen in Image 4 and Image 5 respectively.

In Image 4 (car traffic data during 00:00-01:00) we can notice that there are data missing for some road parts. The roads that have no traffic (0 'BS_Hits') during the specific time window are skipped completely from the report. We will need to fix the missing data issue later on.

These data combined with the ones used before will be used for this new analysis project.

BS_Id	BS_Hits
2	3
3	57
4	7
5	2
6	3
7	1
8	4
9	1
10	6
11	34
12	51
13	50
14	29
16	6
17	6
20	2
23	6
25	7
27	7
29	7
30	2
31	6
32	7
33	1
34	7
35	3
38	6
40	6
41	6
45	1
47	7
50	7
57	7
59	1
60	7

Image #4

BS_Id	Segment Id	NewSegId	Length	FRC	SpeedLimit	StreetName	Coordinates(start)	Coordinates(finish)
1	-1.38E+13	-00004931-	151.65	7	35	Strada Comunale Gaggia	9.01292,45.4052	9.01195,45.40638
2	-1.38E+13	-00004931-	11.23	3	90	Via Manzoni	9.02239,45.40609	9.02241,45.40599
3	-1.38E+13	-00004931-	32.87	2	50	Strada Vigevanese	9.02279,45.40617	9.02262,45.40614
4	-1.38E+13	-00004931-	11.1	3	90	Via Manzoni	9.02284,45.40607	9.02279,45.40617
5	-1.38E+13	-00004931-	117.56	4	90	Strada Provinciale 38	9.03011,45.39619	9.02988,45.39515
6	-1.38E+13	-00004931-	60.24	3	50	SP139	9.10972,45.36605	9.10927,45.36639
7	-1.38E+13	-00004931-	20.72	3	50	SP139	9.10972,45.36605	9.10965,45.36604
8	-1.38E+13	-00004931-	20.88	3	50	SP139	9.10996,45.36597	9.10985,45.36603
9	-1.38E+13	-00004931-	22.01	3	50	SP139	9.11001,45.36578	9.11001,45.36585
10	-1.38E+13	-00004931-	15.31	3	50	Viale Longarone	9.1235,45.36761	9.12333,45.36768
11	-1.38E+13	-00004931-	217.27	2	70	Via dei Giovi	9.13428,45.36496	9.13367,45.36388
12	-1.38E+13	-00004931-	98.68	2	50	Strada Vigevanese	9.03177,45.41088	9.03084,45.41058
13	-1.38E+13	-00004931-	114.56	2	50	Strada Vigevanese	9.02992,45.41035	9.02984,45.41028
14	-1.38E+13	-00004931-	7.8	3	50	Viale Enrico Fermi	9.27183,45.56185	9.27173,45.56186
15	-1.38E+13	-00004931-	91.63	7	20		9.27156,45.54843	9.27164,45.54925
16	-1.38E+13	-00004931-	24.19	4	50	Via San Maurizio al Lam	9.27591,45.54831	9.2756,45.54832
17	-1.38E+13	-00004931-	34.8	4	50	Via San Maurizio al Lam	9.27635,45.54829	9.27591,45.54831
18	-1.38E+13	-00004931-	76.7	7	20	Via San Maurizio al Lam	9.27596,45.549	9.27591,45.54831
19	-1.38E+13	-00004931-	148.78	7	20	Via Ghisallo	9.2793,45.54819	9.2792,45.54685
20	-1.38E+13	-00004931-	202.61	6	35	Via Monte Cervino	9.27944,45.55001	9.2793,45.54819
21	-1.38E+13	-00004931-	96.37	7	20	Via San Maurizio al Lam	9.28089,45.54723	9.28089,45.5481
22	-1.38E+13	-00004931-	124.33	7	20	Via San Cristoforo	9.28368,45.55335	9.28357,45.55224
23	-1.38E+13	-00004931-	13.45	4	50	Via San Maurizio al Lam	9.28387,45.54795	9.28369,45.54797
24	-1.38E+13	-00004931-	198.38	7	20	Via Gransasso	9.28424,45.54971	9.28369,45.54797
25	-1.38E+13	-00004931-	28.88	4	50	Via San Maurizio al Lam	9.28423,45.54793	9.28387,45.54795
26	-1.38E+13	-00004931-	62.61	7	20	Via Torazza	9.28491,45.55218	9.28411,45.55225
27	-1.38E+13	-00004931-	36.61	4	50	Via San Maurizio al Lam	9.2847,45.54789	9.28423,45.54793
28	-1.38E+13	-00004931-	232.96	7	20	Via Moncenisio	9.28471,45.54579	9.2847,45.54789
29	-1.38E+13	-00004931-	13.39	4	50	Via San Maurizio al Lam	9.28487,45.54789	9.2847,45.54789
30	-1.38E+13	-00004931-	167.45	7	20	Via Resegone	9.28537,45.54936	9.28527,45.54911
31	-1.38E+13	-00004931-	45.78	4	50	Via San Maurizio al Lam	9.28546,45.54788	9.28487,45.54789
32	-1.38E+13	-00004931-	57.79	4	50	Via Torazza	9.28503,45.55309	9.28489,45.55278
33	-1.38E+13	-00004931-	148.53	7	20	Villaggio Brugherio	9.2862,45.54916	9.2859,45.54931
34	-1.38E+13	-00004931-	61.37	4	50	Via Torazza	9.28526,45.54953	9.28517,45.5497
35	-1.38E+13	-00004931-	24.99	7	20	Via Torazza	9.28552,45.55075	9.2852,45.55076
36	-1.38E+13	-00004931-	165.87	7	20		9.28546,45.54788	9.28544,45.54638
37	-1.38E+13	-00004931-	123.71	7	20	Via Grigna	9.28577,45.54896	9.28556,45.54843
38	-1.38E+13	-00004931-	43.03	4	50	Via San Maurizio al Lam	9.286,45.54781	9.28546,45.54788
39	-1.38E+13	-00004931-	70.45	7	20	Via Torazza	9.28559,45.55139	9.28552,45.55075

Image #5

3.3. Preprocessing and Saving to Databases

Having presented the shape of the data, we can now move on to the next step. We will use the database tables created already in MySQL, together with the ones we will create now. As before, the databases will be managed through Jupyter Notebook, using the 'mysql.connector' library.

In this database, we are going to store all the data related to the Milan automobile traffic. The necessary information for our job is the unique ID, its name, the coordinates (the starting and ending longitude and latitude is provided) as well as the car traffic over it for every hour of the day.

The first issue we need to resolve is the missing data problem. Rows with zero traffic are not included in the traffic tables so we need to add them with value 0. We solve this like following. First of all, we move all the data in hourly tables in the database in their exact form. This procedure can be seen in Code Sample 27. We then create a temporary table which contains the key values that have been given to all the roads (that means an increasing value from 1 to 150924, the number of the roads), which is displayed in Code Sample 28. Lastly, we create an iteration loop, which resorts to every hourly traffic table and finds which key values (road IDs) are missing from every table, with the help of the table that contains all the key values. It then inserts records with the missing keys and zero value for the car counter. To complete the result we create a new table for every hour of the day, and re-insert the already existing car traffic data in an increasing ID order while dropping the unnecessary tables, so that we have the desired data sorted. This iteration loop can be seen in Code Sample 29.

```
for hour in range(24):

    loadPath = r"C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/Milano_Car_Stats/April-2019_{}_00-{}_00.csv".format(

    print(loadPath)

    dumbstr = "CREATE TABLE cardb.hour{}_{} (road_id int unsigned, car_hits int unsigned);".format(hour, hour+1)

    mycursor = mydb2.cursor(buffered=True)
    sql = dumbstr
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()


    dumbstr = '''LOAD DATA INFILE
                '{}'
                INTO TABLE cardb.hour{}_{}
                FIELDS TERMINATED BY ','
                LINES TERMINATED BY '\n'
                IGNORE 1 LINES
                (road_id, car_hits)
                ;'''.format(loadPath, hour, hour+1)

    mycursor = mydb2.cursor(buffered=True)
    sql = dumbstr
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()
```

Code Sample #27

```
#one-time, auxillary table with all road id's (will be deleted later)
import mysql.connector

mydb2 = mysql.connector.connect(
  host="localhost",
  user="root",
  password="antbestskadata313BASE!1!9!9!7!",
  database="cardb"
)

mycursor = mydb2.cursor(buffered=True)
sql = 'CREATE TABLE cardb.hour0_1_temp (id int NOT NULL AUTO_INCREMENT, primary key (id));'
mycursor.execute(sql)
mydb2.commit()
mycursor.close()

for i in range(1, 150925):
    mycursor = mydb2.cursor(buffered=True)
    sql = 'INSERT INTO cardb.hour0_1_temp VALUES ();'
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()
```

Code Sample #28

```
for hour in range(1, 24):

    mycursor = mydb2.cursor(buffered=True)
    sql = "SELECT id FROM cardb.hour0_1_temp WHERE cardb.hour0_1_temp.id NOT IN (SELECT road_id FROM cardb.hour{}_{
    mycursor.execute(sql)
    data = mycursor.fetchall()
    mycursor.close()

    print("Hour: {}:00 - {}:00".format(hour, hour+1))
    for i in range(len(data)):
        mycursor = mydb2.cursor(buffered=True)
        sql = "INSERT INTO cardb.hour{}_{} VALUES ({}, {});".format(hour, hour+1, data[i][0], 0)
        mycursor.execute(sql)
        mydb2.commit()
        mycursor.close()

    mycursor = mydb2.cursor(buffered=True)
    sql ='CREATE TABLE cardb.hour{}_{}_fin LIKE cardb.hour{}_{};'.format(hour, hour+1, hour, hour+1)
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()

    mycursor = mydb2.cursor(buffered=True)
    sql = 'INSERT INTO cardb.hour{}_{}_fin (SELECT * FROM cardb.hour{}_{} ORDER BY road_id);'.format(hour, hour+1, h
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()

    mycursor = mydb2.cursor(buffered=True)
    sql = 'DROP TABLE cardb.hour{}_{};'.format(hour, hour+1)
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()

    mycursor = mydb2.cursor(buffered=True)
    sql = 'RENAME TABLE cardb.hour{}_{}_fin TO cardb.hour{}_{};'.format(hour, hour+1, hour, hour+1)
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()
```

Code Sample #29

At this point we have created 24 hourly car traffic tables containing the data for all the road parts we have available.

What we want to do next, in order to have all the data grouped, is unite the hourly traffic tables together with the road attributes into one table, which will contain columns for the road ID, the starting and finishing longitude and latitude as well as the 24 columns for the hourly car traffic that passes through it. These road parts are short and straight, so the edge coordinates provide all the information we need about their geographical position. We carry this task out using Microsoft Excel. We concatenate the data of Image 5 (while keeping only the ID and the coordinates) together with 24 Excel tables extracted from the 24 database tables containing the average hourly car traffic. We now have all the information we need grouped in a single table and store it in a dedicated database

table containing all the necessary car traffic information, while deleting the unnecessary tables we created before. The final procedure that executes the information storing can be seen in Code Sample 30.

```
import mysql.connector

mydb2 = mysql.connector.connect(
  host="localhost",
  user="root",
  password="antbestskadata313BASE!1!9!9!7!",
  database="cardb"
)


loadPath = r"C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/Milano_Car_Stats/fulldata.csv"

#print(loadPath)

dumbstr = "CREATE TABLE cardb.fulldata (road_id int unsigned, coords_long_s varchar(20), coords_lat_s varchar(20),

mycursor = mydb2.cursor(buffered=True)
sql = dumbstr
mycursor.execute(sql)
mydb2.commit()
mycursor.close()


dumbstr = '''LOAD DATA INFILE
            '{}'
            INTO TABLE cardb.fulldata
            FIELDS TERMINATED BY ','
            OPTIONALLY ENCLOSED BY '"'
            LINES TERMINATED BY '\n'
            IGNORE 1 LINES
            (road_id, coords_long_s, coords_lat_s, coords_long_f, coords_lat_f, car_hits_0_1, car_hits_1_2, car_hit
            ;'''.format(loadPath)

mycursor = mydb2.cursor(buffered=True)
sql = dumbstr
mycursor.execute(sql)
mydb2.commit()
mycursor.close()
```

Code Sample #30

The resulting database table has the following shape (Database Sample 2)

road_id	coords_long_s	coords_lat_s	coords_long_f	coords_lat_f	car_hits_0_1	car_hits_1_2	car_hits_2_3	car_hits_3_4	car_hits_4_5	car_hits_5_6	car_hits_6_7	car_hits_7_8	car_hits_8_9	car_hits_9_10	car_hits_10_11	car_hit
1	9.01292	45.4052	9.01195	45.4064	0	0	0	0	0	0	0	0	0	0	0	0
2	9.02239	45.4061	9.02241	45.406	3	3	0	1	1	4	13	18	59	33	19	18
3	9.02279	45.4062	9.02262	45.4061	57	37	27	8	8	23	68	145	184	181	234	270
4	9.02284	45.4061	9.02279	45.4062	7	7	8	1	0	0	7	23	17	31	33	39
5	9.03011	45.3962	9.02988	45.3951	2	3	0	1	1	4	8	26	51	32	21	24
6	9.10972	45.3661	9.10927	45.3664	3	0	6	1	2	4	33	132	181	94	53	55
7	9.10972	45.3661	9.10965	45.366	1	5	0	2	0	0	1	4	20	6	17	11
8	9.10996	45.366	9.10985	45.366	4	5	6	3	2	4	34	135	201	100	70	66

Database Sample #2

We create one more database table that looks exactly like the one in Image 5, with the road's name, ID and coordinates stored inside. We will need the road name for a procedure later on.

All the necessary road and car traffic information are now stored in MySQL tables, so we are ready for the next part of the project.

3.4. Data Analysis Procedures

At this point we will start creating the automated procedure we explained before. The goal is to automatically calculate the telecommunication – car traffic correlation of a desired road by inserting its attributes as starting input. This procedure will also be developed using the jupyter notebook, creating a sequence of codes running linear.

First of all, to make the tool easier to use, we will create two different ways to input the desired road. The first one (easier to implement), is for the user to directly input the 'Road ID' of the corresponding road he want to study. In that case, we can directly find from the road database which road that is, its coordinates and how many cars pass through in average. This method however, is not ideal for the user because he will need to have the road data available, and then search for the desired road.

Due to this we add a second choice, in which the user can input the name and the coordinates of the road he wants. In that case, we need to add an extra step in the procedure. We search in the road attributes database table for a road that fits the data given and keep the most fitting choice (that means the Road ID, if there is a fitting choice). If we can find such a road, then we match the user input to a corresponding Road ID and the rest of the procedure continues normally. If no such road can be found in the data, an error message is printed and the procedure stops. That extra piece of code can be seen in Code Sample 31.

```
#works only as part of the FullCarTelecom procedure
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="antbestskadata313BASE!1!9!9!7!",
  database="cardb"
)


mycursor = mydb.cursor(buffered=True)
sql = "SELECT coords_long_s, coords_lat_s, coords_long_f, coords_lat_f, road_id FROM cardb.roadinfo WHERE name='{}'
mycursor.execute(sql)
data = mycursor.fetchall()
mycursor.close()
```

```
class StopExecution(Exception):
    def _render_traceback_(self):
        pass

if (data == []):
    print('No road with these data found!')
    road = None
    raise StopExecution
```

```
import math

min=100 #big starting min

for row in data:
    x = rlong - (row[0]+row[2])/2
    y = rlat - (row[1]+row[3])/2

    if( math.sqrt(x*x + y*y) < min ):
        min = math.sqrt(x*x + y*y)
        road = row[4]
        slong = row[0]
        slat = row[1]
        flong = row[2]
        flat = row[3]

```

```
x = rlong - (slong+flong)/2
y = rlat - (slat+flat)/2

diserror = math.sqrt(x*x + y*y) #deviance from the center of the chosen road

if diserror > 0.005:    #arbitrary, if the road found is far away from the wanted one
    print('No road with these data found!')
    road = None
    raise StopExecution

print("Road ID: "+str(road))
#print(slong, slat, flong, flat)
```

Code Sample #31

What it does is fetch from the data from the database that correspond the road name user selected. If there is no such data the procedure is terminated with a suitable output. Otherwise, it goes through all the data fetched, to spot the piece of road that is closer to the coordinates input. When that piece of road is spotted an extra check is conducted, in which we find out if the chosen road is close enough (arbitrary close) to the desired road. If not the procedure terminates with the same message, since no road was found matching enough. The program stores the Road ID which we ended up with, in the same variable it would be saved if the user would directly input the desired Road ID. With the end of this code chunk, we have completed the Road ID search successfully, and we have the Road ID we needed to continue with. The rest of the procedure goes exactly like we had been given the Road ID from the start (the 2 methods overlap from now on). We will analyze the rest of the procedure once, and it is used in both cases.

The next step, having acquired the Road ID, is to calculate the telecommunication traffic on that piece of road. The car traffic is stored in the corresponding database, ready to be fetched for any road. For the telecom traffic, on the other hand, it is a bit more complicated, because we have the telecom traffic data for every Milan cell stored. That means, that the first thing we need to do is match the specific route (piece of road), with the cells it traverses through. We need to be careful, so that the exact piece of road for which we have the automobile data, is matched perfectly with a sequence of cells, so that the results are valid.

But we have already created a code sequence that executes the above, during the previous part of the thesis. We can use it exactly as it is with a few changes. The first change is about the input. We do not ask the user for starting and ending latitude and longitude, as we did before. We have the Road ID, so we just need to fetch the corresponding starting and finishing longitude and latitude from the database that stores the road attributes. We then use the 'Project-OSRM' tool exactly as we have already done. The second change, is that we test both ways of the route (from start to finish and from finish to start) to determine which of the two is the shortest path (roads can be single direction, so a wrong direction can produce entirely wrong routes). We then have come to know the shortest path from start to finish in coordinate points, which we store in the dedicated database table (about the path coordinates). We call again the 'fullroute' procedure to fill the distance gaps in between the coordinate points. Finally we match the route coordinates with the corresponding cells, by checking in which specific cell each one of the points belong, and save the route cells in a dedicated MySQL database table. The above can be seen in Code Sample 32. We have tested multiple times that the

sequence of cells which is produced is matching the desired road perfectly. An example will be provided.

```
def get_route(pickup_lon, pickup_lat, dropoff_lon, dropoff_lat):

    url1 = "http://router.project-osrm.org/route/v1/driving/"
    url2 = "{},{};{},{}".format(pickup_lon, pickup_lat, dropoff_lon, dropoff_lat)
    url3 = "?alternatives=false&annotations=true"

    r = requests.get(url1 + url2 + url3)
    if r.status_code != 200:
        return {}

    res = r.json()
    routes = polyline.decode(res['routes'][0]['geometry'])
    start_point = [res['waypoints'][0]['location'][1], res['waypoints'][0]['location'][0]]
    end_point = [res['waypoints'][1]['location'][1], res['waypoints'][1]['location'][0]]
    distance = res['routes'][0]['distance']

    out = {'route':routes,
           'start_point':start_point,
           'end_point':end_point,
           'distance':distance
          }

    return out, routes



mycursor = mydb2.cursor(buffered=True)
sql = "SELECT coords_long_s FROM newfulldata WHERE road_id = {}".format(road)
mycursor.execute(sql)
long_s = mycursor.fetchall()
mycursor.close()

mycursor = mydb2.cursor(buffered=True)
sql = "SELECT coords_lat_s FROM newfulldata WHERE road_id = {}".format(road)
mycursor.execute(sql)
lat_s = mycursor.fetchall()
mycursor.close()

mycursor = mydb2.cursor(buffered=True)
sql = "SELECT coords_long_f FROM newfulldata WHERE road_id = {}".format(road)
mycursor.execute(sql)
long_f = mycursor.fetchall()
mycursor.close()
```

```

mycursor = mydb2.cursor(buffered=True)
sql = "SELECT coords_lat_f FROM newfulldata WHERE road_id = {}".format(road)
mycursor.execute(sql)
lat_f = mycursor.fetchall()
mycursor.close()


### Sometimes the direction of the road is wrong, so we try both and choose the smallest distance (works!)

pickup_lon1, pickup_lat1, dropoff_lon1, dropoff_lat1 =  long_s[0][0], lat_s[0][0], long_f[0][0], lat_f[0][0]  #norm
dropoff_lon2, dropoff_lat2, pickup_lon2, pickup_lat2 =  long_s[0][0], lat_s[0][0], long_f[0][0], lat_f[0][0]  #oppo

###


whole_route1, help_route1 = get_route(pickup_lon1, pickup_lat1, dropoff_lon1, dropoff_lat1)
whole_route2, help_route2 = get_route(pickup_lon2, pickup_lat2, dropoff_lon2, dropoff_lat2)

if whole_route1['distance'] < whole_route2['distance']:
    whole_route = whole_route1
    help_route = help_route1
else:
    whole_route = whole_route2
    help_route = help_route2
```

```
mycursor = mydb.cursor(buffered=True)
sql = "TRUNCATE TABLE celliddb.routecoords;"
mycursor.execute(sql)
mydb.commit()
mycursor.close()

a = 0

for i in help_route:
    mycursor = mydb.cursor(buffered=True)
    sql = "INSERT INTO celliddb.routecoords (uniqid, latitude, longitude) VALUES ({}, {}, {})".format(a, i[0], i[1])
    mycursor.execute(sql)
    mydb.commit()
    mycursor.close()
    a = a + 1
```

```
mycursor = mydb.cursor(buffered=True)
sql = "CALL celliddb.fullroute;"
mycursor.execute(sql)
mydb.commit()
mycursor.close()
```

```
mycursor = mydb.cursor(buffered=True)
sql = "SELECT * FROM celliddb.routecoords ORDER BY uniqid;"
mycursor.execute(sql)
myresult = mycursor.fetchall()
mycursor.close()
```

```
mycursor = mydb.cursor(buffered=True)
sql = "SELECT * FROM celliddb.cellidscoords;"
mycursor.execute(sql)
cellids = mycursor.fetchall()
mycursor.close()
```

```
from shapely.geometry import Point
from shapely.geometry.polygon import Polygon

routecells = []

for i in myresult:
    #print(i)
    point = Point(i[1], i[2])

    for j in cellids:
        polygon = Polygon([(j[0], j[1]), (j[2], j[3]), (j[4], j[5]), (j[6], j[7])])
        checker = polygon.contains(point)

        if (checker == True):
            if j[8] not in routecells:
                routecells.append(j[8])
            break

#print(routecells)
```

```
mycursor = mydb.cursor(buffered=True)
sql = "TRUNCATE TABLE celliddb.routecellids;"
mycursor.execute(sql)
mydb.commit()
mycursor.close()

for i in routecells:
    mycursor = mydb.cursor(buffered=True)
    sql = "INSERT INTO celliddb.routecellids (cellid) VALUES ({});".format(i)
    mycursor.execute(sql)
    mydb.commit()
    mycursor.close()
```

Code Sample #32

At this point, we have stored in the database the cells through which the route traverses. Next step is to isolate the telecommunication data of these cells, which describe the network demand in the area of the road we want to study. To perform this, we will need the daily telecommunication traffic data

stored in the database. Exactly as we did in the previous part of the thesis, we will create an iterating loop going through all the daily traffic tables and storing only the specific data we want (cells of the route) in temporary database tables. That way, we will have 62 temporary tables (one for each day, truncated before starting the procedure again) with the telecom traffic of the cells that interest us. We do an extra step, and calculate the average daily telecom traffic from the 62 days. We save that new average daily traffic in a dedicated database table. The execution of the above can be seen in Code Sample 33.

```
mycursor = mydb.cursor(buffered=True)
sql = "SELECT * FROM celliddb.routecellids;"
mycursor.execute(sql)
myresult = mycursor.fetchall()
mycursor.close()

route = []

for x in myresult:
    route.append(x[0])
```

```
mydb2 = mysql.connector.connect(
  host="localhost",
  user="root",
  password="antbestskadata313BASE!1!9!9!7!",
  database="testdb"
)

dumbstr = ", ".join("{:d}".format(i) for i in route)
print("Cell IDs: " + dumbstr)

for day in range(1, 63):

    tempstr = "TRUNCATE TABLE testdb.tempday" + str(day) + ";"

    mycursor = mydb2.cursor(buffered=True)
    sql = tempstr
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()

    tempstr = "INSERT INTO testdb.tempday{} SELECT * FROM testdb.day{} WHERE Cell_ID in ({});".format(day, day, dumb

    mycursor = mydb2.cursor(buffered=True)
    sql = tempstr
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()

    print("Created temp database for day {}".format(day))
```

```
alldata = []

for day in range(1, 63):

    tempstr = "SELECT * FROM testdb.tempday" + str(day) + ";"

    mycursor = mydb2.cursor(buffered=True)
    sql = tempstr
    mycursor.execute(sql)
    data = mycursor.fetchall()
    mycursor.close()

    data = np.array(data)

    alldata.append(data)
    #print(np.shape(temp))

avtraff = alldata[0]

for i in range(np.shape(alldata)[1]):
    isum = 0
    for x in range(np.shape(alldata)[0]):
        isum += alldata[x][i][2]

    iav = isum/np.shape(alldata)[0]
    avtraff[i][2] = iav

for i in range(np.shape(avtraff)[0]):
    avtraff[i][1] -= 1383260400000.0
    avtraff[i][1] /= 600000.0
    avtraff[i][1] += 1



temp = avtraff

avtraff = np.zeros((np.shape(temp)[0],np.shape(temp)[1]+2))
avtraff[:,:-2] = temp
```

```
for i in range(np.shape(alldata)[1]):
    sd = 0
    for x in range(np.shape(alldata)[0]):
        sd += pow(avtraff[i][2]-alldata[x][i][2], 2)

    sd = math.sqrt(sd/np.shape(alldata)[0])
    avtraff[i][3] = sd

for i in range(np.shape(avtraff)[0]):
    Cv = avtraff[i][3]/avtraff[i][2]
    avtraff[i][4] = Cv


mycursor = mydb2.cursor(buffered=True)
sql = "TRUNCATE TABLE testdb.temptimeav;"
mycursor.execute(sql)
mydb2.commit()
mycursor.close()

for row in avtraff:
    mycursor = mydb2.cursor(buffered=True)
    sql = "INSERT INTO testdb.temptimeav (Cell_ID, Time_stamp, int_time_av, sd, cv) VALUES ({}, {}, {}, {}, {});".fo
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()


print("Created temp database for average daily internet.")
```

Code Sample #33

We have now processed and saved all the telecom data needed to describe the user chosen road. Specifically, we have calculated the average daily telecommunication traffic in the desired cells. Furthermore, fetching the automobile data from the corresponding database tables is trivial, so we have come to acquire all the needed data to move forward.

Next step is to plot our data. We fetch the average daily telecommunication traffic as well as the cells the road is comprised from. We do an extra step to make sure that the sequence of the cells saved in the average daily traffic table is the same with the geographical topology on the road (Code Sample 34).

```
datanew = []

#correct order for cells is in route
#route is the sorted array of cells, and data refers to the telecom data of the road
for i in range(len(route)):
    flag = route[i]

    for row in data:
        if row[0] == flag:
            datanew.append(row)

mycursor = mydb2.cursor(buffered=True)
sql = "TRUNCATE TABLE testdb.temptimeav;"
mycursor.execute(sql)
mydb2.commit()
mycursor.close()

for row in datanew:
    mycursor = mydb2.cursor(buffered=True)
    sql = "INSERT INTO testdb.temptimeav (Cell_ID, Time_stamp, int_time_av, sd, cv) VALUES ({}, {}, {}, {}, {});".fo
    mycursor.execute(sql)
    mydb2.commit()
    mycursor.close()

print("Rearranged average daily internet in route order.")
```

Code Sample #34

We are now ready to plot the data we have calculated and isolated. In the first script (Code Sample 35), we plot the average daily traffic of every cell of the road. That is not enough though, because the data for the telecommunication traffic are given in 10-minute intervals through the day, while for the automobile traffic we have average hourly data. So for the next step, we calculate the hourly average telecom traffic of each cell and proceed to plot it (Code Sample 36). For the next, and last plotting, we add the hourly average car traffic of the road we study. Furthermore, instead of drawing each cell's traffic separately, we combine them to create united average telecommunication traffic for the road (representative for the whole route). The code that performs that can be seen in Code Sample 37.

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

i = 0
#data refer to the average telecom data of the road (sorted by cell order)
plt.figure(figsize=(14,10))
plt.title("Time Dependence (Cells of the Milano Road)")
plt.xlabel("# of 10-minutes in the day (144 in total, starting at 12 am)")
plt.ylabel("Average Internet traffic")

while i < np.shape(data)[0]:
    label = data[i][0]
    plt.plot(data[i:i+144,1], data[i:i+144,2], label= "{}".format(label))
    i+=144

plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0.)
plt.show()
```

Code Sample #35

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

i = 0

plt.figure(figsize=(14,10))
plt.title("Time Dependence, hourly (Cells of the Milano Road)")
plt.xlabel("# of hours in the day (starting at 12 am)")
plt.ylabel("Average Internet traffic")

# i is for the day (for one cell), j for the hour, k for the 10-min
# mylist will contain the hoyrly daily telecom data for one cell, gets reset for every cell
while i < np.shape(data)[0]:
    label = data[i][0]

    mylist = []
    for j in range(24):
        itemp = i + j*6
        telesum = 0
        for k in range(6):
            telesum += data[itemp+k,2]
        teleav = telesum/6

        mylist.append([j, teleav])

    mylist = np.array(mylist)

    plt.plot(mylist[0:24,0], mylist[0:24,1], label= "{}".format(label))

    i+=144

plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0.)
plt.show()
```

Code Sample #36

```
# i is for the day (for one cell), j for the hour, k for the 10-min
# mylist will contain the hourly daily telecom data for every cell

i = 0
mylist = []
while i < np.shape(data)[0]:
    label = data[i][0]
    for j in range(24):
        itemp = i + j*6
        telesum = 0
        for k in range(6):
            telesum += data[itemp+k,2]
        teleav = telesum/6
        mylist.append([j, teleav])

    i+=144

#menewlist will contain the average of the hourly average telecom activity of each cell
mynewlist = []
for hour in range(24):
    sumhat = 0
    t = 0
    for row in mylist:
        if row[0] == hour:
            sumhat += row[1]
            t += 1
    avhat = sumhat/t
    mynewlist.append([hour, avhat])

mynewlist = np.array(mynewlist)

mycarlist = []
for hour in range(24):
    mycursor = mydb.cursor(buffered=True)
    sql = "SELECT car_hits_{}_{} FROM cardb.newfulldata WHERE road_id = {}".format(hour, hour+1, road)
    mycursor.execute(sql)
    car_hits = mycursor.fetchall()
    mycursor.close()
    mycarlist.append([hour, car_hits[0][0]])

mycarlist = np.array(mycarlist)


plt.plot(mynewlist[0:24,0], mynewlist[0:24,1], label= "Average Hourly Telecom Traffic for the Road")
plt.plot(mycarlist[0:24,0], mycarlist[0:24,1], label= "Car Hits")

plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0.)
plt.show()
```

Code Sample #37

The scripts above are about visualizing the data we desire. Apart from that, we will need to store these last data we formed (average hourly telecom and car traffic) in a new temporary database table, in order to use them for our next step (Code Sample 38). We perform this action in order to implement the main target of this part of the thesis, which is calculating the correlation between telecom and car traffic for the user chosen road.

```
mycursor = mydb.cursor(buffered=True)
sql = 'TRUNCATE TABLE testdb.24cartel;'
mycursor.execute(sql)
mydb.commit()
mycursor.close()

for i in range(24):
    x = mycarlist[i][1]
    y = mynewlist[i][1]

    mycursor = mydb.cursor(buffered=True)
    sql = 'INSERT INTO testdb.24cartel VALUES ({}, {});'.format(x, y)
    mycursor.execute(sql)
    mydb.commit()
    mycursor.close()
```

Code Sample #38

In order to calculate the correlation and plot it as a correlation matrix we will need to use the appropriate Python libraries (pandas and seaborn). These libraries provide tools that automatically calculate the desired correlation from appropriately shaped data. To give that appropriate shape, we make an extra step to save the table in a 'csv' type file, since that is the format accepted by the library functions. We can see in Code Sample 39 how this procedure is implemented, and the functions used for the correlation matrix calculation and plotting.

```
mycursor = mydb2.cursor(buffered=True)
sql = "SELECT * FROM testdb.24cartel;"
mycursor.execute(sql)
temp = mycursor.fetchall()
mycursor.close()

temp = np.array(temp)
print(np.shape(temp))

np.savetxt(r"C:\Users\Antonis\Documents\Jupyter Notebooks\Thesis\Test\CorrMatTest\CarTel.csv",
           temp, delimiter=",", fmt='%f', header='Car, Telecom')
```

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from scipy.stats import norm

###
data = pd.read_csv(r"C:\Users\Antonis\Documents\Jupyter Notebooks\Thesis\Test\CorrMatTest\CarTel.csv")

###
corrmat = data.corr()
f, ax = plt.subplots(figsize =(9, 8))
sns.heatmap(corrmat, ax = ax ,cmap ="YlGnBu", linewidths = 0.1, annot=True)
```

Code Sample #39

The script sequence thoroughly presented and explained in this chapter is the whole procedure that calculates the telecom – car traffic correlation for the user desired road, starting from the basic data that have been provided. In the next chapter we will present results for some roads and comment on them.

3.5. Results and Conclusions

In this chapter, we will run the code sequence we developed on some Milan roads. The goal is to verify the validity of the tool. This will occur in two steps.

Firstly, by inserting the data of the road we want (name & coordinates) we will validate that the script chooses the correct cells, in which the road belongs to. Essentially, we check if the telecommunication traffic is chosen correctly from the database. The second validation is a bit more intuitive. We will choose road parts from big roadways which means, cells in which the telecommunication traffic is almost entirely dependent to the passing vehicles. In such road parts we expect much larger telecom-car traffic correlation, since the car traffic will highly affect the network usage in that area. Respectively, we can pick more secluded roads, next to residences, universities, recreation areas etc. In such spots we expect significant lower telecom-car traffic correlation, since the network usage is mostly due to spatially stable users. So, the target is to confirm that the expected correlation coincides with the tool's result. We will perform the procedure on roads from both categories mentioned.

The first road for which we will run the scripts can be seen in Images 6 and 7. In the first one we can see the cell in which the road belongs to (ID: 4154).

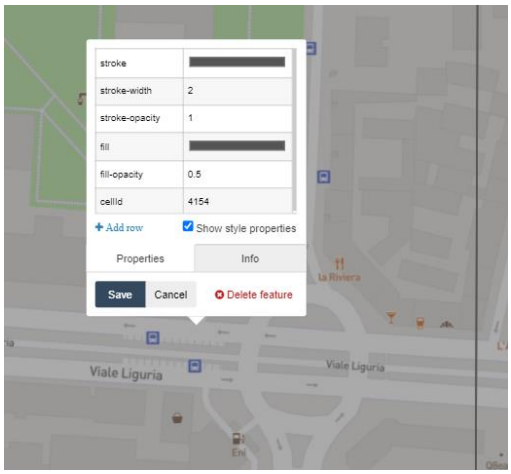


Image #6

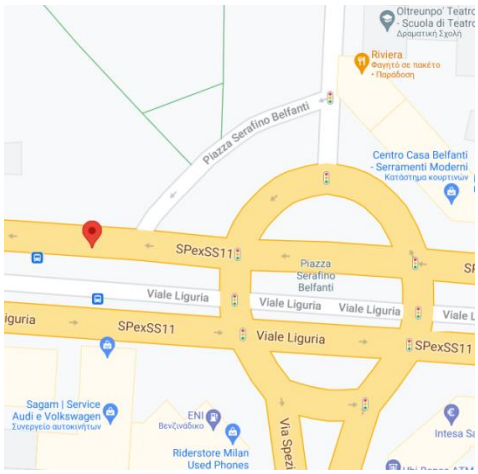
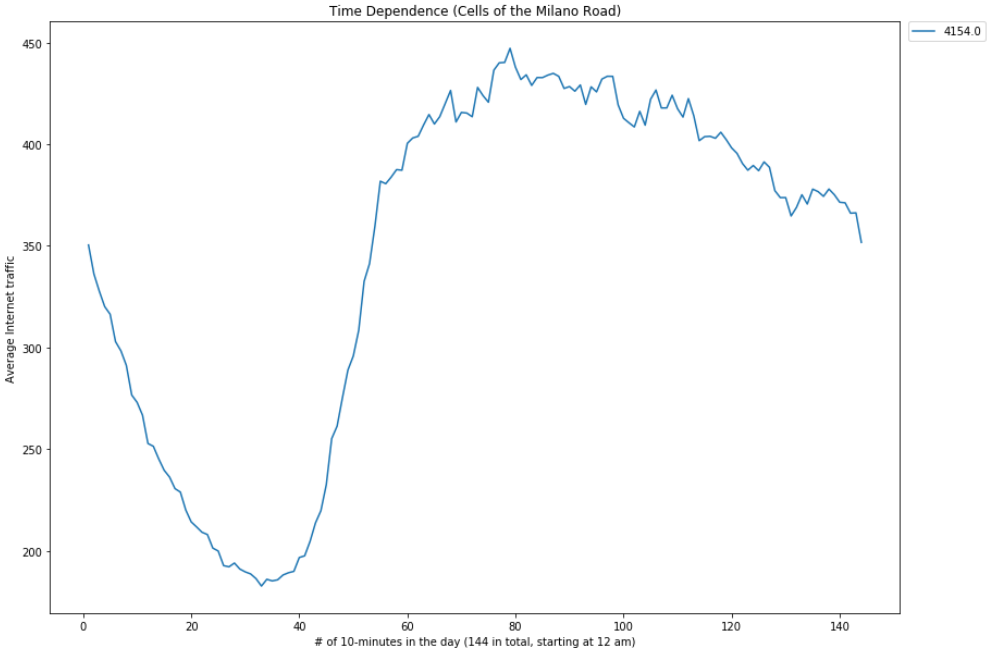
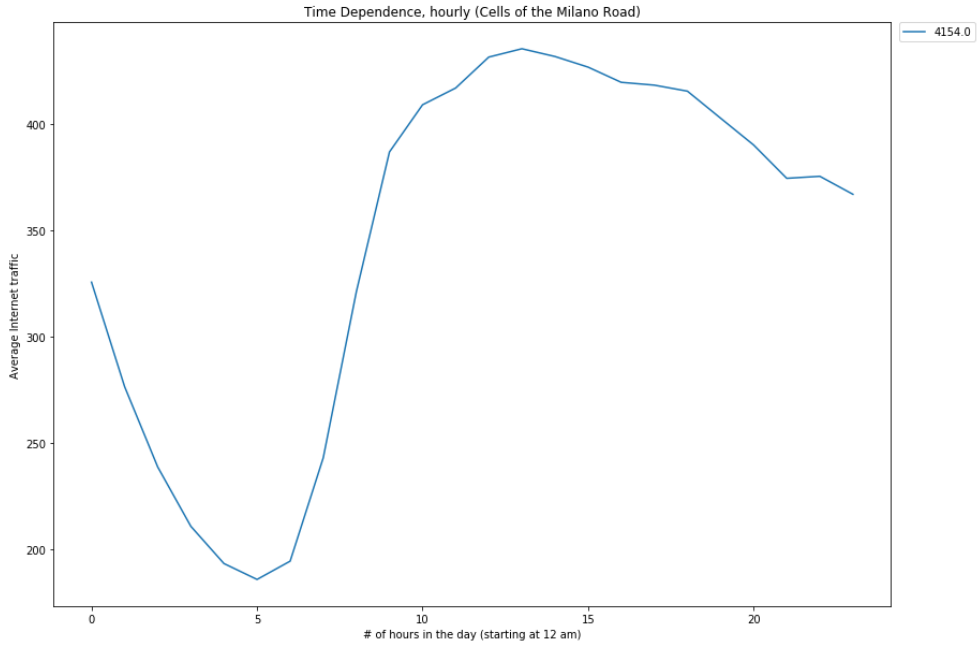


Image #7

It is a main roadway passing through the outskirts of the city center. We expect a high correlation between the telecommunication and car traffic, since it is a busy roadway comprising most of the cell it belongs to. We run the code sequence by inserting the road name and coordinates, and the outputs can be seen in Images 8 and 9.


Time Dependence (Cells of the Milano Road)
4154.0
Average internet traffic
# of 10-minutes in the day (144 in total, starting at 12 am)


Time Dependence, hourly (Cells of the Milano Road)
4154.0
Average internet traffic
# of hours in the day (starting at 12 am)

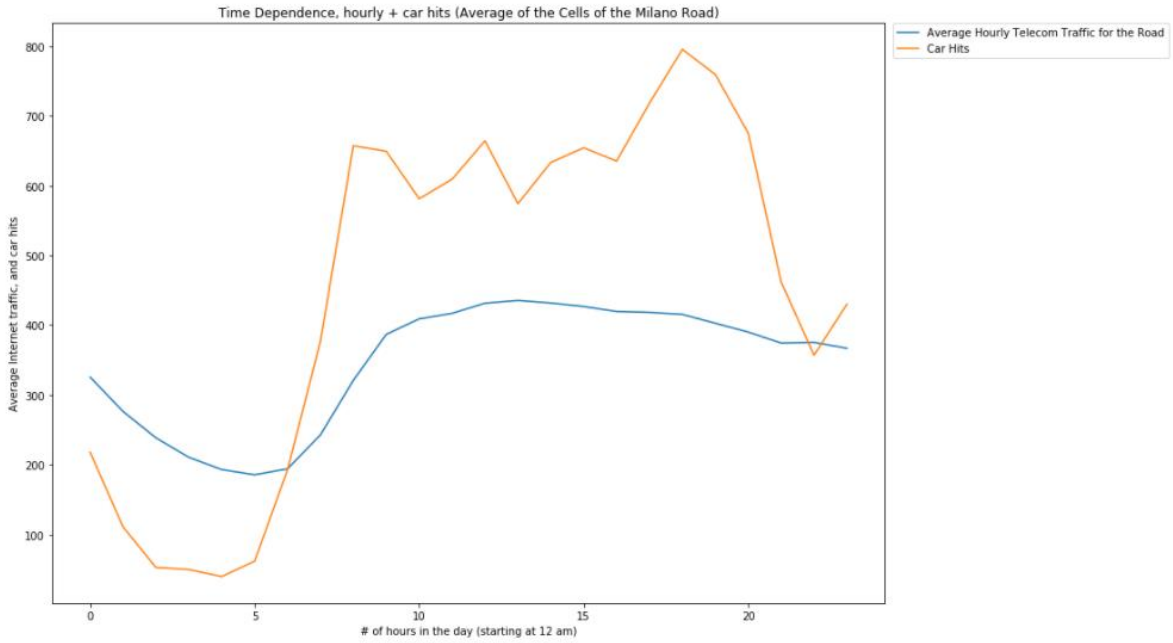


Image #8

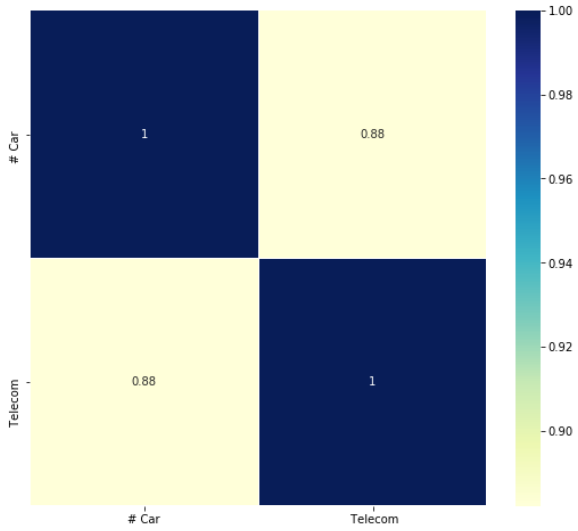


Image #9

As we can see, the program has chosen the correct cell (4154), and the correlation result is very high (0.88) as expected. That means that the result for this road input is successfully validated.

We will perform the procedure one more time. We can see the new road in Images 10 and 11.

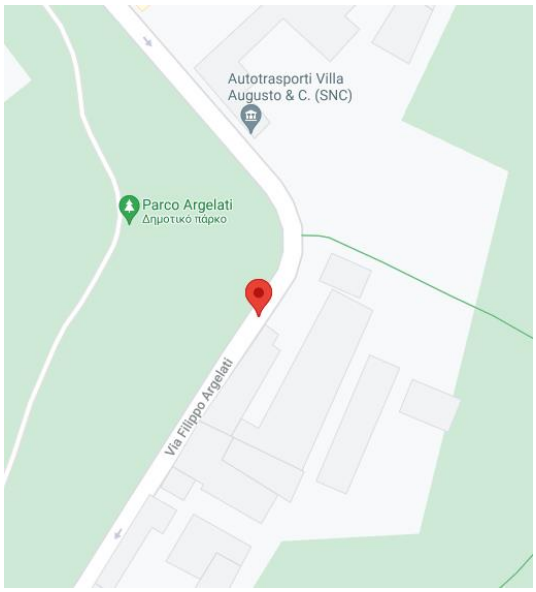


Image #10

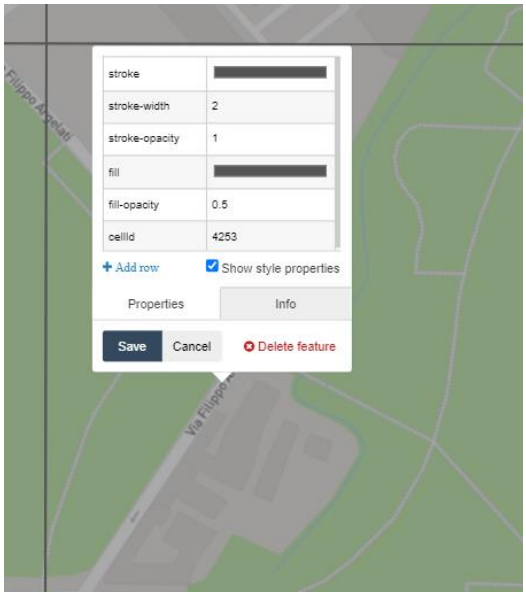
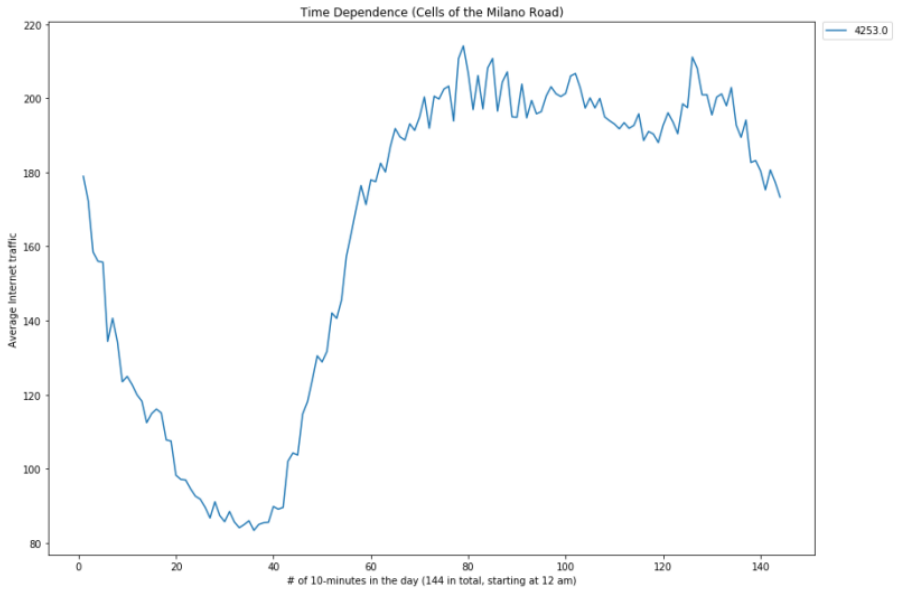
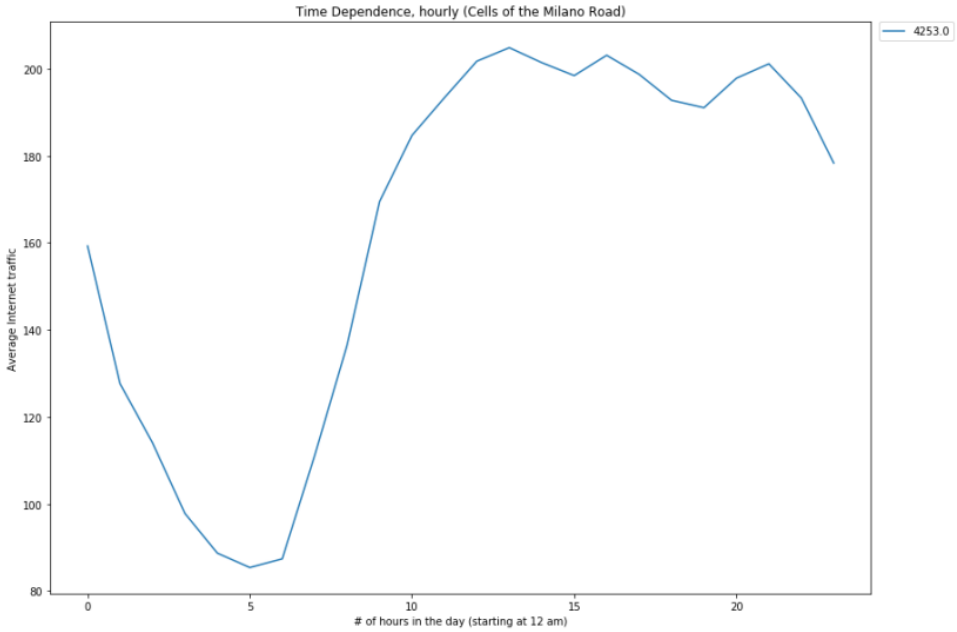


Image #11

It is an isolated road passing by park, industrial areas and public spaces. This means that the telecommunication traffic will be mostly due to the geographically stable users rather than passing vehicles. In other words, we expect a smaller telecom-car traffic correlation. The results of the code sequence can be seen in Images 12 and 13.




Time Dependence, hourly (Cells of the Milano Road)
4253.0
Average Internet traffic
# of hours in the day (starting at 12 am)

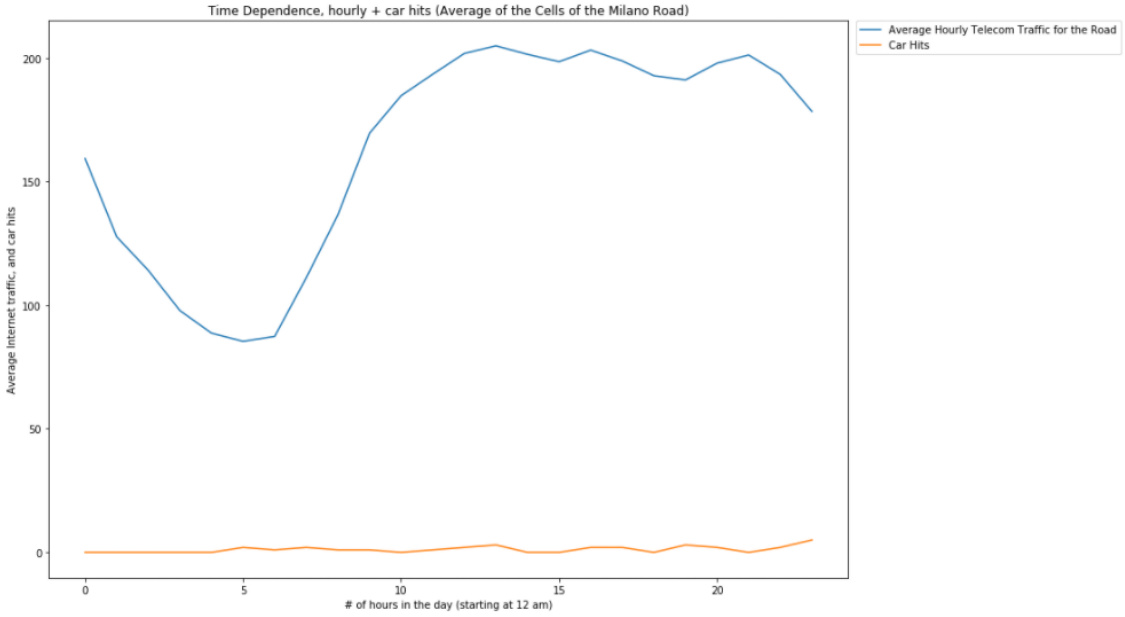

Time Dependence, hourly + car hits (Average of the Cells of the Milano Road)
Average Hourly Telecom Traffic for the Road
Car Hits
Average Internet traffic, and car hits
# of hours in the day (starting at 12 am)

Image #12

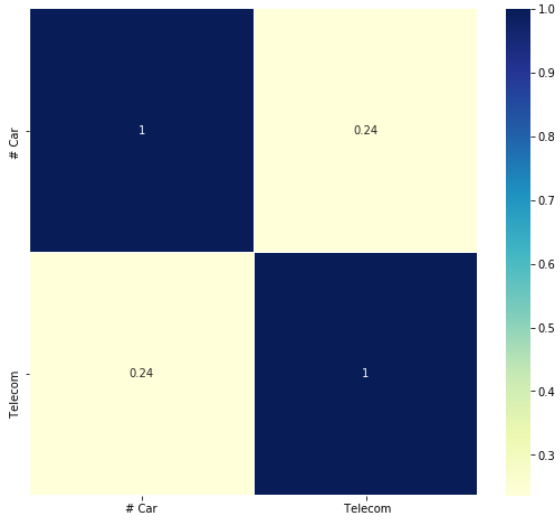


Image #13

The program has chosen the right cell (4253). Furthermore, the result of the correlation is low (0.24) as expected.

More test runs were implemented, and in accordance with the examples above, the validity of our tool has been verified. When given a random road part in the Milan province, it calculates the telecom-car traffic correlation of that specific road (if these data exist) and prints the output. This information can be helpful in order to better understand the network and its needs. Handover frequency is necessary information for optimizing the network and making it more robust.

In this part of the thesis, we had the chance to store and manipulate large scale datasets as well as extracting information from their processing.

4. Conclusion

At the center of our focus is the better understanding of the use of the telecommunications network to benefit society. All the different studies and approaches we have done in this project are aimed to draw conclusions towards that direction. By arriving at such conclusions, we can inform and improve the design and function of these networks. This thesis demonstrates the utility of data analytics in extracting actionable insights from telecommunication data. Through the use of time and spatial analysis, Gaussian processes, and correlation matrices, the research successfully gains a better understanding of user patterns of network usage, predicts network usage needs and investigates any correlations between telecommunication and vehicle traffic.

The use of Python and SQL as primary tools for the analysis and management of data allows for efficient and easily-replicable scripts, and the ability to handle large amounts of data using MySQL.

The findings from this research have the potential to improve network performance in telecommunication systems by determining network handover needs in user-chosen areas and optimizing network coverage. This thesis is an important step in leveraging data analytics to gain a deeper understanding of telecommunication systems and improve their performance.

References

1. *Big data: A review,* S. Sagiroglu, D. Sinanc, 2013
2. *Project Jupyter: A Computer Code that Transformed Science*, Linda Vu, 2021.
3. *Data Science Notebooks get real: JupyterLab releases to users*, Andrew Brust, 2018.
4. *"What is MySQL?" MySQL 8.0 Reference Manual*, Oracle Corporation, 2020.
5. *Geojson*, Howard Butler, Martin Daly, Allan Doyle, Sean Gillies, Stefan Hagen, Tim Schaub, Erik Wilde, 2014.
6. *Gaussian Process Regression Analysis for Functional Data*, Jian Qing Shi, Taeryon Choi, 2011.
7. *A correlation-matrix-based hierarchical clustering method for functional connectivity analysis*, Xiao Liu, Xiao-Hong Zhu, Peihua Qiu, Wei Chen, 2012.
8. *A guide to NumPy,* Travis E. Oliphant, 2006.
9. *MySQL Connector/Python Revealed,* JW Krogh, G Krogh, Gennick, 2018.
10. *Matplotlib and seaborn,* E. Bisong, 2019.
11. *Pandas, python data analysis library,* W. McKinney, 2015.
12. *Scikit-learn: Machine learning in Python,* F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, 2011.
13. *"Jupyter Widgets" documentation,* Project Jupyter, 2022.
14. *A multi-source dataset of urban life in the city of Milan and the Province of Trentino*, Gianni Barlacchi, Marco De Nadai, Roberto Larcher, Antonio Casella, Cristiana Chitic, Giovanni Torrisi, Fabrizio Antonelli, Alessandro Vespignani, Alex Pentland & Bruno Lepri, 2015.