00.Introduction

February 2, 2022

EPFL Extension School - 2020/2022

Capstone Project - Xavier Nicolovici

This capstone project will try to find a machine learning model to answer that question:

What are the Taxi travel speed prevision for New-York City (NYC) depending on the travel characteristics and weather forecast ?

The ML models will be trained to determine the speed in kilometers per hour a Taxi trip will take in New-York City, depending on travel characteristics like pickup and dropoff location, period of the day, of the week,... and weather characteristics like temperature of the day, rain, snow,...

This model should help to determine the time that would take to go from one location to another in New-York using a Taxi cab.

To achieve this, I will use a dataset taken from *Kaggle* and I'll extend it using weather dataset taken from the *National Centers for Environmental Information*.

Note: In the rest of this project, the *NYC* acronym will be used to name *Ney-York City*

1 Time spent on this project

As of the 1st of February 2022: 98 hours

2 Datasets

Two datasets will be used for this capstone project.

2.1 NYC Taxi Travel Dataset

This NYC Taxi Travel dataset has been found on Kaggle and was used in a competition where participants, based on individual trip attributes, should predict the duration of each trip in the test set. This dataset is based on the 2016 NYC Yellow Cab trip record data made available in Big Query on Google Cloud Platform. The data was sampled and cleaned for the purposes of this playground competition.

This dataset covers data from the 1st of January until the 30th of June 2016.

Most of the cleaning work have been done on this dataset, which is good news for me :-)

2.2 NYC Weeather informations

To extend the previous dataset with weather informations, I've used the Climate Data Search Engine from the *National Centers for Environmental Information*

This search engine allows me to grab multiple weather informations concerning NYC, covering the period of the NYC Taxi Dataset.

Merging those two datasets will lead to one dataset with travel informations along with weather ones.

3 Organisation

This project is organized in 4 big chapters:

- Data preparation: Where datasets will be prepared and cleaned
- Data Exploration: Where data will be explored and improved
- Machine learning models: Where ML models will be trained
- Results and Communication: Where ML models results will be presented and anlyzed

Those big chapters are contained in Notebooks which names starts by, respectively, 10, 20, 30 and 40. Any sub-Notebooks needed for one of this chapter will be numbered with the number of the chapter + 1. For example:

10 - Data preparation

11 - Data preparation - NYC Taxi dataset

12 - Data preparation - NYC Weather dataset

4 Technics

4.1 GeoPandas

At this time of writing, I'm not sure to have time to explore this wonderful library GeoPandas.

This library provides Pandas Dataframe and Series extension classes to draw maps along with datapoint. This might be very nice and usefull to represent pickup and dropoff point, weather stations zone and other geographic features.

> Note: Be sure to execute the following instruction to ensure that GeoPandas library is available in your Jupyter Notebook environment.

[1]:
```
!pip install shapely fiona pyproj geopandas
```

```
Requirement already satisfied: shapely in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (1.8.0)
Requirement already satisfied: fiona in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (1.8.20)
Requirement already satisfied: pyproj in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (3.0.1)
Requirement already satisfied: geopandas in /opt/anaconda3/envs/exts-
```

```
ml/lib/python3.6/site-packages (0.9.0)
Requirement already satisfied: click>=4.0 in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from fiona) (8.0.3)
Requirement already satisfied: munch in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from fiona) (2.5.0)
Requirement already satisfied: certifi in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from fiona) (2020.12.5)
Requirement already satisfied: cligj>=0.5 in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from fiona) (0.7.2)
Requirement already satisfied: click-plugins>=1.0 in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from fiona) (1.1.1)
Requirement already satisfied: attrs>=17 in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from fiona) (20.3.0)
Requirement already satisfied: six>=1.7 in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from fiona) (1.15.0)
Requirement already satisfied: setuptools in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from fiona) (49.6.0.post20210108)
Requirement already satisfied: pandas>=0.24.0 in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from geopandas) (0.24.2)
Requirement already satisfied: importlib-metadata; python_version < "3.8" in
/opt/anaconda3/envs/exts-ml/lib/python3.6/site-packages (from click>=4.0->fiona)
(3.10.1)
Requirement already satisfied: numpy>=1.12.0 in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from pandas>=0.24.0->geopandas) (1.15.4)
Requirement already satisfied: python-dateutil>=2.5.0 in
/opt/anaconda3/envs/exts-ml/lib/python3.6/site-packages (from
pandas>=0.24.0->geopandas) (2.8.1)
Requirement already satisfied: pytz>=2011k in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from pandas>=0.24.0->geopandas) (2021.1)
Requirement already satisfied: zipp>=0.5 in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (from importlib-metadata; python_version <
"3.8"->click>=4.0->fiona) (3.4.1)
Requirement already satisfied: typing-extensions>=3.6.4; python_version < "3.8"
in /opt/anaconda3/envs/exts-ml/lib/python3.6/site-packages (from importlib-
metadata; python_version < "3.8"->click>=4.0->fiona) (3.7.4.3)
```

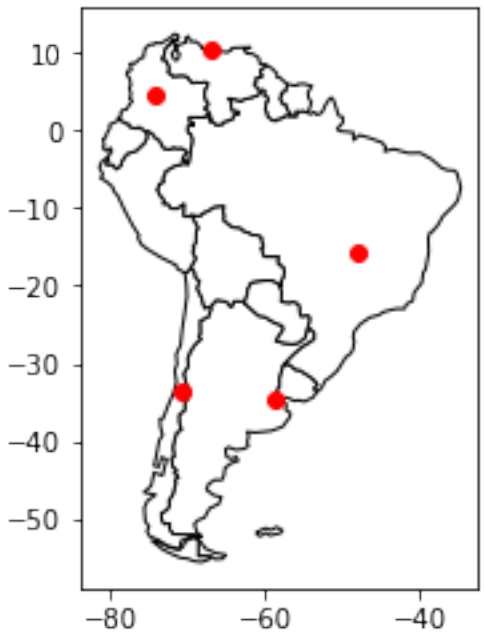
In order to validate that the GeoPandas library as been correctly installed, let's draw a small map of south America.

This code snippet has been copied from the GeoPandas website.

> Note: If you get any error here, then the GeoPandas feature has not been correctly installed. This might lead to errors while running Notebooks of this project

[2]:
```
import pandas as pd
import geopandas
import matplotlib.pyplot as plt

df = pd.DataFrame(
```

```
    {'City': ['Buenos Aires', 'Brasilia', 'Santiago', 'Bogota', 'Caracas'],
     'Country': ['Argentina', 'Brazil', 'Chile', 'Colombia', 'Venezuela'],
     'Latitude': [-34.58, -15.78, -33.45, 4.60, 10.48],
     'Longitude': [-58.66, -47.91, -70.66, -74.08, -66.86]})

gdf = geopandas.GeoDataFrame(
    df, geometry=geopandas.points_from_xy(df.Longitude, df.Latitude))

world = geopandas.read_file(geopandas.datasets.get_path('naturalearth_lowres'))

# We restrict to South America.
ax = world[world.continent == 'South America'].plot(
    color='white', edgecolor='black')

# We can now plot our ``GeoDataFrame``.
gdf.plot(ax=ax, color='red')

plt.show()
```



4.2 my_utils.ipynb

In order to simplify the code in the different Notebooks, many usefull functions will be coded into a Notebook named my_utils.py

Prior to using this import approach, be sure that the ipynb module is installed on your Python environment by running the following instruction.

[3]:
```
# Ensure that 'ipynb' module exists in the Python env
!pip install ipynb
```

```
Requirement already satisfied: ipynb in /opt/anaconda3/envs/exts-
ml/lib/python3.6/site-packages (0.5.1)
```

Then, in order to use any function defined in *my_utils.ipynb*, each Notebooks of this project will start with the following code:

[4]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

Details of the functions available in my_utils can be found in the file itself as Python embeded comments

4.3 Map of New-York City

In order to draw the map of New-York City, I've decided to use a shape definition available on the University of Texas at Austin website

The shape I've used is the 2010 New York City Boroughs and here is how it looks:

> Note: The function I use here is coded in the my_utils library and accept optional parameters like latitude/longitude series to be plotted along with the map

[5]:
```
# Draw NYC map with two optional locations
draw_nyc_map(latitude=[40.65, 40.6], longitude=[-74, -73.9])
```

5 Let's Go

Time to start my journey in this Capstone project with the next notebook: Data Preparation Introduction

10.Data Preparation Introduction

February 2, 2022

1 The datasets

This capstone project is based on two datasets, *NYC Taxi Travel* and *NYC Weather* datasets. You can find more details on their origin in the Introduction Notebook Notebook.

Here is a description of their structure and some quick cleanup actions.

1.1 NYC Taxi Travel Dataset

This dataset is made of 11 columns, which contains 10 independent variables and one dependent.

The 10 independent variables will be part of the final dataset features, and the dependent one will be used to create my result vector: *km_per_hour*

Here is a description of the variables.

1.1.1 Independent variables

- id - a unique identifier for each trip.
- vendorid - a code indicating the provider associated with the trip record.
- pickupdatetime - date and time when the meter was engaged.
- dropoffdatetime - date and time when the meter was disengaged.
- passengercount - the number of passengers in the vehicle (driver entered value).
- pickuplongitude - the longitude where the meter was engaged
- pickuplatitude - the latitude where the meter was engaged
- dropofflongitude - the longitude where the meter was disengaged
- dropofflatitude - the latitude where the meter was disengaged
- store_and_forward_flag - This flag indicates whether the trip record was held in vehicle memory before sending to the vendor because the vehicle did not have a connection to the serve.

1.1.2 Dependent variable

- trip_duration - duration of the trip in seconds.

1.2 NYC Weather Dataset

This dataset is made of 11'544 lines with 66 columns. Each line contains the daily values for a particular weather station regarding weather informations: temperature, rain, snow, wind, ...

What I will have to do first with this dataset is to extract the list of the different weather stations along with their locations. This weather station list will be merged with the NYC Travel one, using the weather stations coordinates to determine which one is the nearest from pickup and dropoff.

Here is a description of the variables.

1.2.1 Weather station static informations

- STATION - Identification of the weather station
- NAME - The name of the weather station
- LATITUDE - Latitude of the weather station
- LONGITUDE - Logitude of the weather station
- ELEVATION - Elevation

1.2.2 Independent variables

- DATE - Date when the measures where done (YYYY-MM-DD)
- AWND - Average wind speed
- DAPR - Number of days included in the multiday precipitation total (MDPR)
- DASF - Number of days included in the multiday snow fall total (MDSF)
- MDPR - Multiday precipitation total (use with DAPR and DWPR, if available)
- MDSF - Multiday snowfall total
- PGTM - Peak gust time
- PRCP - Precipitation
- PSUN - Daily percent of possible sunshine for the period
- SNOW - Snowfall
- SNWD - Snow depth
- TAVG - Average Temperature.
- TMAX - Maximum temperature
- TMIN - Minimum temperature
- TOBS - Temperature at the time of observation
- TSUN - Total sunshine for the period
- WDF2 - Direction of fastest 2-minute wind
- WDF5 - Direction of fastest 5-second wind
- WESD - Water equivalent of snow on the ground
- WESF - Water equivalent of snowfall
- WSF2 - Fastest 2-minute wind speed
- WSF5 - Fastest 5-second wind speed
- WT01 - Fog, ice fog, or freezing fog (may include heavy fog)
- WT02 - Heavy fog or heaving freezing fog (not always distinguished from fog)
- WT03 - Thunder
- WT04 - Ice pellets, sleet, snow pellets, or small hail"
- WT05 - Hail (may include small hail)
- WT06 - Glaze or rime
- WT08 - Smoke or haze
- WT09 - Blowing or drifting snow
- WT11 - High or damaging winds

Note: There is no dependent variable in this dataset as all the column will be used to

add features to the NYC Taxi Travel dataset.

Note 2: All the units in this dataset are in metric standard

2 Table of content

Data preparation is splitted into seven Notebooks

2.1 The 83 Weather Stations

In the NYC Weather Dataset, I've found what I've called *static weather station data*: - STATION - Identification of the weather station - NAME - The name of the weather station - LATITUDE - Latitude of the weather station - LONGITUDE - Logitude of the weather station - ELEVATION - Elevation

Those static data generates a lot of duplicated data into the dataset as there is a total of 83 different weather stations in the whole dataset.

The goal of this Notebook is to create a small dataset that contains the static features of the weather stations.

2.2 NYC Taxi Travel Data Preparation

This first Notebook prepares data grabbed from Kaggle, containing the Taxi Travel informations. Most of the work here will be to remove useless columns, outliers and prepare data for the rest of the project.

As this dataset has been found already cleaned, this work will be quite straight forward.

One question you may have reading this NYC Taxi Travel Data Preparation Notebook is how I've choosen the latitude/logitude values that I will use to remove some pickup and dropoff points from the project ? Well, looking at the weather stations locations available into the NYC Weather Datasets, I've decided to match the travel location with the weather station ones.

We'll see that this approach did not remove a lot of lines from NYC Taxi Travel Dataset (less than 6'000 over ~1'500'000), and will be more efficient when I will merge the two dataset. As I would like to match pickup and dropoff location to the nearest weather station, removing travel location far away from stations makes sense.

2.3 NYC Weather Data Preparation

This Notebook contains the data preparation process of the independent features of the NYC Weather Dataset.

After cleaning up the dataset (drop some useless columns), most of the work will be focused on creating two datasets from this one:

- Weather Categorical dataset

This dataset will be built using features that are defined as categories: Wind direction, fog, peak ust,...

We'll see later in this project that this dataset will be grouped by days instead of weather station locations.

- Weather Numerical dataset

This second dataset will contains all the numerical values of the measures taken by the stations: Temperature, precipitation, snow…

This dataset will be grouped by weather station and date, and we will discover that not all the data is available for each station/day. We will see how to solve this.

2.4 NYC Taxi Travel Dataset Feature Engineering

With this third dataset, which is the biggest one, I will try to engineer some interesting features, using static data informations from the 83 weather stations: - Distance from nearest weather station - Distance in kilometers between pickup and dropoff locations - …

It's in this Notebook that I will build my result vector: *km_per_hour*

2.5 NYC Weather Categorical Dataset Feature Engineering

The *Weather Categorical* dataset will be grouped by days in this Notebook. This will produce a dataset with 182 lines, which is the number of days between the 1st of January 2016 and the 30th of June 2016.

Reason of this approach will be explained in the notebook.

2.6 NYC Weather Numerical Dataset Feature Engineering

In this Notebook, the *Weather Numerical* dataset, before being grouped by days and by weather stations, will be extended and the missing values extrapolated.

- Extended: To add missing tuple of (days, weather stations)
- Extrapolated: To fill the weather stations NaN values using the average of the n nearest non null weather stations values.

The whole extension and extrapolation process will be described in this Notebook.

This will produce a dataset with 15'106 lines, which is the number of days (182) multiplied by the number of weather stations (83).

2.7 The global Dataset - Merging all the datasets into a big one

After all this cleaning and feature engineering process on the two original dataset, it will be time to merge all of the datasets produced in previous Notebooks:

- Taxi Travel dataset
- Weather Stations Dataset
- Weather Categorical Dataset
- Weather Numerical Dataset

This will result in a *full* dataset ready for ML training process

3 Overview of the created datasets

Af the end of data cleaning and feature engineering, I will obtain four datasets:

1. *stations* dataset

A dataset with all the static informations of the weather stations like elevation, latitude, name.

2. *travel* dataset

A *feature engineered* dataset of the *NYC Taxi Travel dataset* like pickup_datetime, dropoff_location

3. *weather categorical* dataset

Categorical feature per day, *engineered* from the weather stations dataset like snow fall, fog.

4. *weather numerical* dataset

Numerical feature per day and per weather stations, *engineered* from the weather stations dataset like temperature average, quantity of snow on the road

5. Full dataset

This dataset is a merge of the *numerical* and *categorical* features of the four previous dataset, ready to be used with ML training processes.

> Note: Construction and description of these dataset will be detailed in the following Notebooks

4 Technics

To perform the merge of the data, I've decided to use an *sqlite* database approach and play with *INNER JOIN* methods to merge datasets.

This technic will be used in the The global Dataset - Merging all the datasets into a big one to build the *Full* dataset

Stay tuned ;-)

5 Let's go

Time to go to the first data preparation Notebook: The 83 Weather Stations :-)

11.The 83 Weather Stations

February 2, 2022

1 Build the Weather Stations dataset

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

Looking at the NYC Weather Dataset, I've found that it contains measures for a total of 83 weather stations around New-York City.

Those weather stations are identified by their *NAME*, *STATION* (their id), *LATITUDE*, *LONGITUDE*, and *ELEVATION*

Looking at the *weather* dataset, I've found that one of those 83 station does not have any value in *ELEVATION.

[2]:
```
# Load 'weather' dataset, limit column to station, name, latitude, longitude and␣
 ↪elevation
nyc_stations=load_csv('weather')[['STATION', 'NAME', 'LATITUDE', 'LONGITUDE',␣
 ↪'ELEVATION']]

# Remove line with empty elevations (this column is not always filled in)
print("Numer of stations without elevation:")
nyc_stations[nyc_stations.ELEVATION.isna()].drop_duplicates()
```

```
Numer of stations without elevation:
```

[2]:

	STATION	NAME	LATITUDE	LONGITUDE	ELEVATION
2791	US1NJES0020	BLOOMFIELD 1.7 S, NJ US	40.785	-74.1885	NaN

To solve this, I've search on Elevation Finder website the value I should use in this missing *ELEVATION* feature. This gives me 40 meters for latitude 40.785 and longitude -741885.

Let's use the *fillna()* method on the *ELEVATION* columns to replace NaN values (which again are related to only one of the weather station) with the value *40*

> Note: Prior to the *fillna()* method call, I will drop duplicates from the dataset. The interesting thing here is to build a dataset with the weather stations static dataset.

[3]:
```
US1NJES0020_ELEVATION=40

# Drop duplicates from nyc_stations dataset
nyc_stations.drop_duplicates(inplace=True)

# fillna() values in ELEVATION with 40 (the missing elevation value)
nyc_stations['ELEVATION']=nyc_stations['ELEVATION'].fillna(US1NJES0020_ELEVATION)
print("List of stations without elevation:",nyc_stations[nyc_stations.ELEVATION.
 ↪isna()]['STATION'].count())
```

```
List of stations without elevation: 0
```

Now that our stations static dataset is cleaned, let's save it in our SQLite database, it will be usefull when we will need to reuse it.

> Note 1: now that this data will be made available in our database, I've added a new function in *my_utils* library: *get_stations()* which returns the *weather_stations* dataset stored in the Database.

[4]:
```
# Verify SQL tablename is defined in my_utils library
print("Table name used to save this dataset:", STATION_TABLENAME)
```

```
Table name used to save this dataset: stations
```

[5]:
```
# Save weather station dataset to SQL Database
save_sql(nyc_stations, STATION_TABLENAME)
```

```
Saving OK
```

[5]:
```
True
```

[6]:
```
# Display stations dataset information to validate that
# it contains data for the 83 stations, data taken from SQL Database
stations=load_sql(STATION_TABLENAME)
stations.describe()
```

```
Query: SELECT * FROM stations
```

[6]:

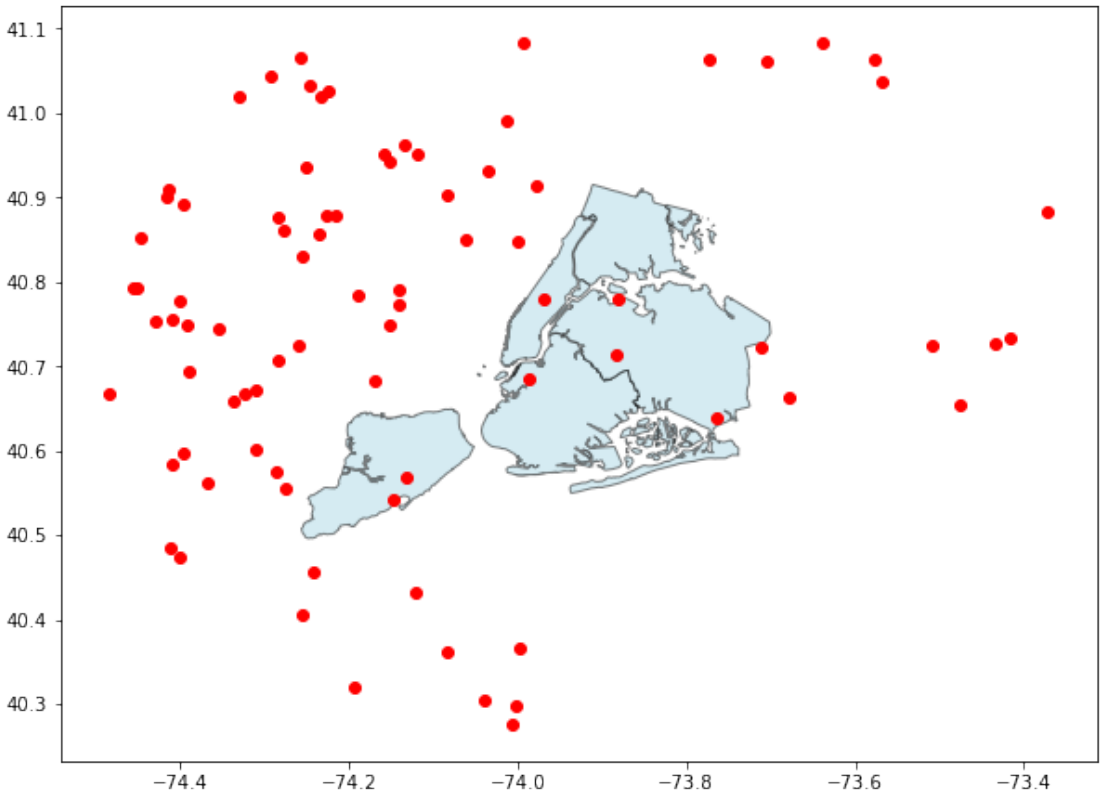
	LATITUDE	LONGITUDE	ELEVATION
count	83.000000	83.000000	83.000000
mean	40.758148	-74.122859	47.293976
std	0.207156	0.277533	44.306354
min	40.275368	-74.482746	2.100000
25%	40.656987	-74.316848	16.500000
50%	40.772892	-74.192680	28.000000
75%	40.906158	-74.000878	71.750000
max	41.083546	-73.373090	188.700000

2 A bit of fun :-)

What about trying to draw the weather stations locations on a New-York City map ?

I'll use the *draw_nyc_map()* function from *my_utils* library to draw the NYC map. This drawing function accept latitude and longitude datapoints to be plotted on the map.

[7]:
```
# Draw NYC map with weather stations
draw_nyc_map(latitude=stations['LATITUDE'], longitude=stations['LONGITUDE'])
```



3 That's it

Even if this Notebooks is very short, it let me build a dataset of static weather stations features.

Let's follow up with the next notebook: NYC Taxi Travel Data Preparation

12.NYC Taxi Travel Data Preparation

February 2, 2022

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

It's now time to load the *NYC Taxi Travel* Dataset into a Pandas dataframe. We load data using *load_csv* function from my_utils library.

[2]:
```
# Load dataset with function from my_utils library
nyc_travel = load_csv('travel')
```

[3]:
```
# Display first 5 lines
nyc_travel.head(5)
```

[3]:
```
          id  vendor_id      pickup_datetime     dropoff_datetime
passenger_count  \
0  id2875421          2  2016-03-14 17:24:55  2016-03-14 17:32:30
1
1  id2377394          1  2016-06-12 00:43:35  2016-06-12 00:54:38
1
2  id3858529          2  2016-01-19 11:35:24  2016-01-19 12:10:48
1
3  id3504673          2  2016-04-06 19:32:31  2016-04-06 19:39:40
1
4  id2181028          2  2016-03-26 13:30:55  2016-03-26 13:38:10
1

   pickup_longitude  pickup_latitude  dropoff_longitude  dropoff_latitude
store_and_fwd_flag  \
0        -73.982155        40.767937         -73.964630         40.765602
N
1        -73.980415        40.738564         -73.999481         40.731152
N
2        -73.979027        40.763939         -74.005333         40.710087
N
```

```
3         -74.010040         40.719971         -74.012268         40.706718
N
4         -73.973053         40.793209         -73.972923         40.782520
N

   trip_duration
0            455
1            663
2           2124
3            429
4            435
```

[4]:
```
# Display informations
nyc_travel.info(verbose=True)
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1458644 entries, 0 to 364660
Data columns (total 11 columns):
id                    1458644 non-null object
vendor_id             1458644 non-null int64
pickup_datetime       1458644 non-null object
dropoff_datetime      1458644 non-null object
passenger_count       1458644 non-null int64
pickup_longitude      1458644 non-null float64
pickup_latitude       1458644 non-null float64
dropoff_longitude     1458644 non-null float64
dropoff_latitude      1458644 non-null float64
store_and_fwd_flag    1458644 non-null object
trip_duration         1458644 non-null int64
dtypes: float64(4), int64(3), object(4)
memory usage: 133.5+ MB
```

Ok, this dataset is as claimed in the description, seems to have no missing values. That's good. It is made of around 1'500'000 lines with three columns made of int64 values, four columns of float64 values and the rest is made of strings.

[5]:
```
# Verify that we do not have missing values
nyc_travel.isnull().sum()
```

[5]:
```
id                    0
vendor_id             0
pickup_datetime       0
dropoff_datetime      0
passenger_count       0
pickup_longitude      0
pickup_latitude       0
dropoff_longitude     0
dropoff_latitude      0
```

```
store_and_fwd_flag    0
trip_duration         0
dtype: int64
```

1 Dataset cleaning process

As everything might not be useable to build my model, let's make some cleanup and reformat this dataset.

1.1 id

The *id* column should be dropped from the dataset. It contains a unique id of the trip recorded, completly useless to train models.

[6]:
```
nyc_travel.drop('id', axis=1, inplace=True)
```

1.2 vendor_id

This column identifies the *vendor* of the trip but, looking at it's content, we can see that the number of different vendors is two, and the number of trips seems to be equally distributed.

[7]:
```
# Get the number of unique values
print('Number of unique values within the whole dataset: {}'.
 ↪format(nyc_travel['vendor_id'].nunique()))
```

```
Number of unique values within the whole dataset: 2
```

According to this, dropping this column seems to be an evidence.

[8]:
```
nyc_travel.drop('vendor_id', axis=1, inplace=True)
```

1.3 store_and_fwd_flag

The name of this column helps to determine that it's a flag, a categorical column, which contains 'Y' and 'N' values.

Let replace Y values with a 1 and N values with 0 to be able to use this feature when training our model.

[9]:
```
# Confirm we have two values only in this column
nyc_travel['store_and_fwd_flag'].unique()
```

[9]:
```
array(['N', 'Y'], dtype=object)
```

[10]:
```
# Use pd.replace to create a categorical column
nyc_travel['store_and_fwd_flag']=nyc_travel['store_and_fwd_flag'].replace({'Y':␣
 ↪1, 'N': 0})
```

1.4 trip_duration

Looking deeper on this column, we can find some big outliers that should be removed from the dataset.

[11]:
```
nyc_travel[['trip_duration']].describe().astype('int')
```

[11]:

	trip_duration
count	1458644
mean	959
std	5237
min	1
25%	397
50%	662
75%	1075
max	3526282

As we can see on the table above, we have some incoherent values both on the minimum side (trip of 1 second long) and on the maximum side (the maximum time trip took more than 40 days !!).

Let's have a look at the dataset obtained if we get rid of travel time below 60 seconds (one minute) and above 7200 seconds (2 hours).

[12]:
```
number_of_outliers=nyc_travel[
    np.logical_not(
        np.logical_and(
            nyc_travel['trip_duration'] <= 7200,
            nyc_travel['trip_duration'] >= 60,
        )
    )
]['trip_duration'].count()

number_of_travel=nyc_travel['trip_duration'].count()

print("Number of outliers:", number_of_outliers)
print("Total number of travel in dataset:", number_of_travel)
print("Percentage of data removed from dataset using [60,7200] seconds limit: {:.
 ↪2f} %".format(number_of_outliers/number_of_travel*100))
```

```
Number of outliers: 10848
Total number of travel in dataset: 1458644
Percentage of data removed from dataset using [60,7200] seconds limit: 0.74 %
```

Removing trip below 60 seconds or above 7'200 seconds will remove less than 1% of our dataset.

I've decided to drop those lines to remove some incoherent values that might influence negatively the training process.

[13]:
```
# Drop lines that contains travel_time > 7200 or travel_time < 60
nyc_travel=nyc_travel[
    np.logical_and(
```

```
        nyc_travel['trip_duration'] <= 7200,
        nyc_travel['trip_duration'] >= 60,
    )
]

nyc_travel['trip_duration'].describe().astype('int').to_frame()
```

[13]:

	trip_duration
count	1447796
mean	840
std	653
min	60
25%	401
50%	665
75%	1076
max	7191

1.5 passenger_count

Passenger count is a column that specifies, for a particular trip, the number of person present in the Taxi.

I've found 19 trips concerning either 0 or more than 6 passengers. For that reason, I've decided to drop those lines which I consider as outliers.

[14]:
```
# Count passenger_count outliers (0 or more than 6)
invalid_passenger_count=nyc_travel[
    np.logical_or(
        nyc_travel['passenger_count'] <= 0,
        nyc_travel['passenger_count'] > 6
    )
]['passenger_count'].count()

print("Number of passenger invalid (0 or more than 6):", invalid_passenger_count)
```

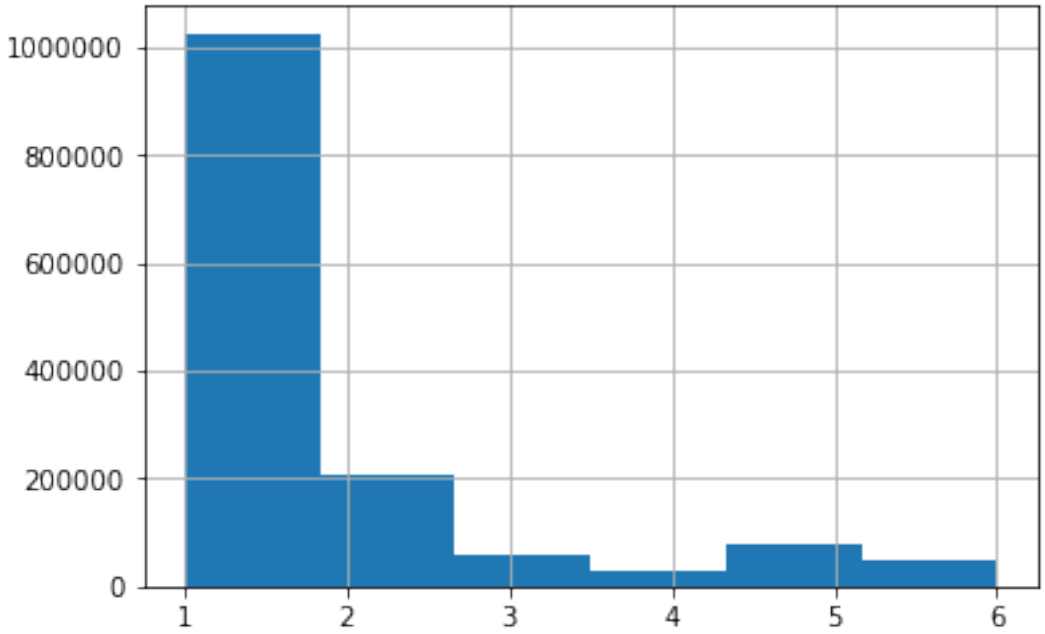
```
Number of passenger invalid (0 or more than 6): 19
```

Quickly drawing an histogram of the values shows that most of the travels are made with one and only one passenger.

> Note: This could lead to creating a new feature: Trips with one person only.

[15]:
```
nyc_travel = nyc_travel[
    np.logical_not(
        np.logical_or(
            nyc_travel['passenger_count'] <= 0,
            nyc_travel['passenger_count'] > 6
        )
    )
```

```
]

nyc_travel['passenger_count'].hist(bins=6)
plt.show()
```



1.6 Longitude and latitude

Same approach for the latitude and the longitude values, we should seek for outliers and removed them.

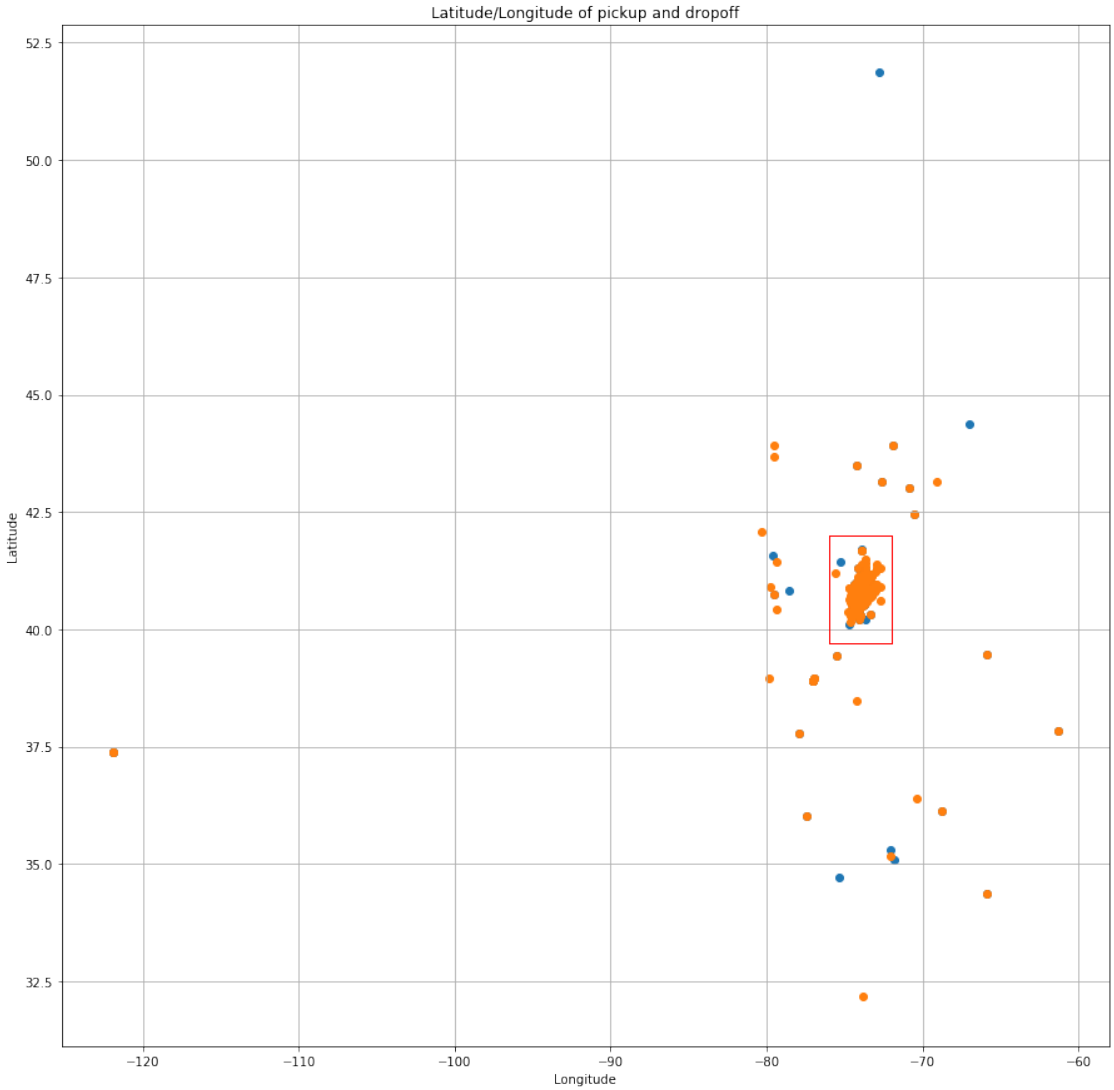
Let's draw a scatter plots using *latitude* and *longitude* as X and Y axis for each location: - pickup - dropoff

On this scatter plot, I'll draw a rectangle that will surround the data points that I consider as good data points. The one outside the rectangle will be considered as outliers.

> Note: I'm defining here a function to draw the scatter plots as I will reuse it several time

[16]:
```
# Set rectangle angle position
LOWER_LEFT=[-72, 39.7]
UPPER_RIGHT=[-76, 42]

def draw_scatter():
    # Set figure size
    plt.figure(figsize=(15, 15))
```

```
    # Set axis labels
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')

    # Scatter plot of pickup and dropoff positions
    plt.scatter(nyc_travel['pickup_longitude'], nyc_travel['pickup_latitude'])
    plt.scatter(nyc_travel['dropoff_longitude'], nyc_travel['dropoff_latitude'])

    # Draw rectangle with function from 'my_utils' library
    plot_rectangle(lower_left=LOWER_LEFT, upper_right=UPPER_RIGHT)

    # Display grid axis
    plt.grid()

    # set title
    plt.title("Latitude/Longitude of pickup and dropoff")

    # Show canvas
    plt.show()

draw_scatter()
```



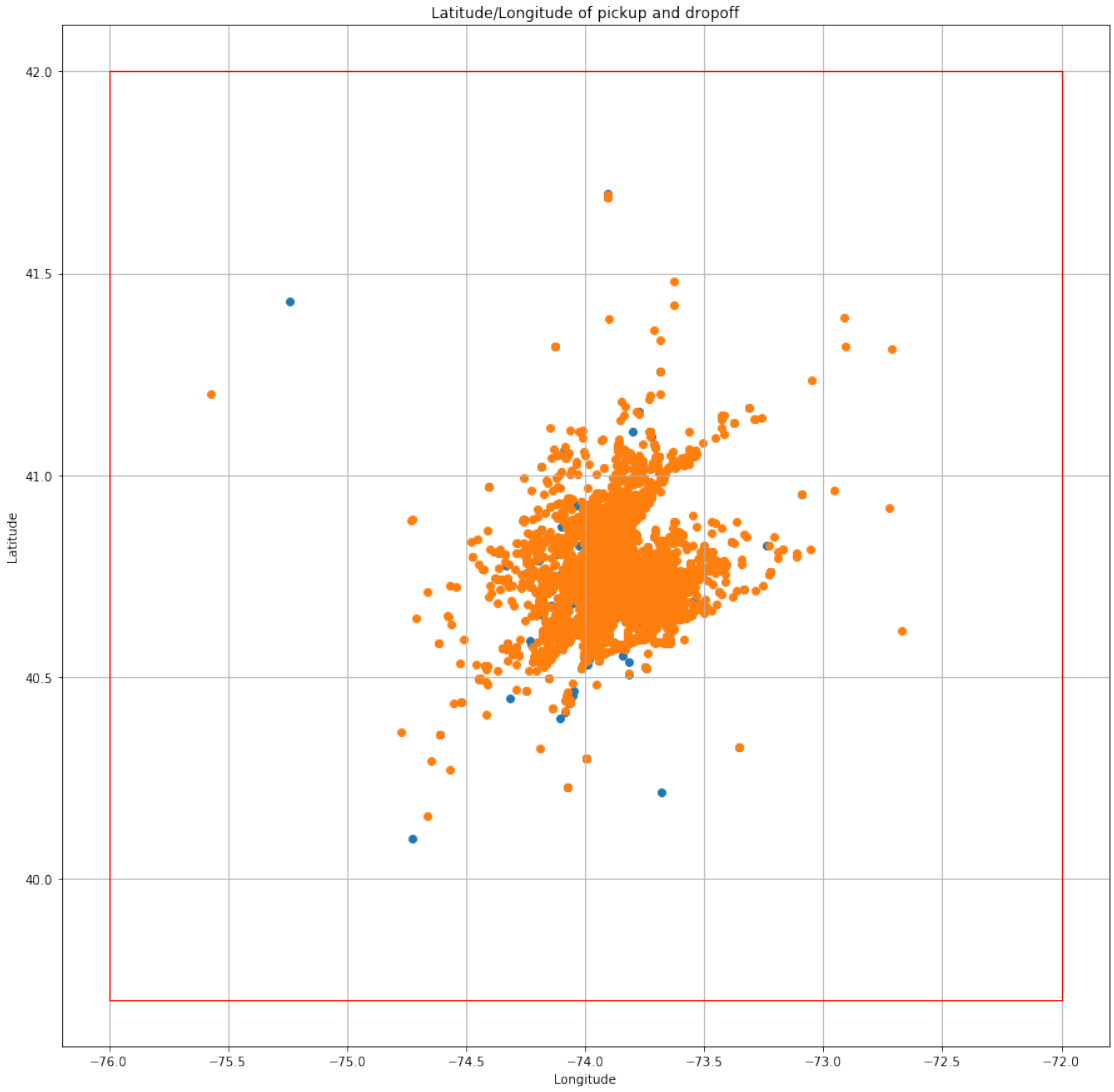
The rectangle seems to be good and exclude outliers. Let's build a filter based on the rectangle value, count line that will be removed, and draw a new graph with outliers dropped from the dataset.

Note: The global filter building process is defined as a function to be reused below in this Notebook

[17]:
```
def get_global_filter() -> pd.core.series.Series:
    # Build filter to remove outliers from dataset
    # !! Warning, longitude is a negative value

    # pickup latitude
    pickup_lat_filter=np.logical_and(
        nyc_travel['pickup_latitude'] >= LOWER_LEFT[1],
```

```
        nyc_travel['pickup_latitude'] <= UPPER_RIGHT[1]
    )

    # pickup longitude
    pickup_long_filter=np.logical_and(
        nyc_travel['pickup_longitude'] <= LOWER_LEFT[0],
        nyc_travel['pickup_longitude'] >= UPPER_RIGHT[0]
    )

    # dropoff latitude
    dropoff_lat_filter=np.logical_and(
        nyc_travel['dropoff_latitude'] >= LOWER_LEFT[1],
        nyc_travel['dropoff_latitude'] <= UPPER_RIGHT[1]
    )

    # pickup longitude
    dropoff_long_filter=np.logical_and(
        nyc_travel['dropoff_longitude'] <= LOWER_LEFT[0],
        nyc_travel['dropoff_longitude'] >= UPPER_RIGHT[0]
    )


    # Return global filter
    return np.logical_and(
        np.logical_and(pickup_lat_filter, pickup_long_filter),
        np.logical_and(dropoff_lat_filter, dropoff_long_filter)
    )

# get global filter
global_filter=get_global_filter()

# The try/except block is used when no False values exists in global_filter
try:
    print("Number of lines that will be removed from dataset:", global_filter.
 ↪value_counts()[False])
except:
    print("No lines found to be removed")


# Remove outliers
nyc_travel=nyc_travel[global_filter]

draw_scatter()
```

```
Number of lines that will be removed from dataset: 34
```



Fine, this datasets finally looks better, but now that I have a clearer view on it due to the first removal of outliers, I think I can do better be redifining the rectangle with closer values.

But, wich values should I use ?

Well, answer can come from the The 83 Weather Stations notebook, where I've plotted the weather stations on the NYC map.

Using a simple query, we can get the limits of the weather stations:

[18]:
```
# Get max/min of longitude/latitude from stations dataset
query="SELECT min(LONGITUDE) as min_long, max (LONGITUDE) as max_long,␣
 ↪min(latitude) as min_lat, max(latitude) as max_lat from stations"
stations_limit=load_sql(query=query)
stations_limit
```

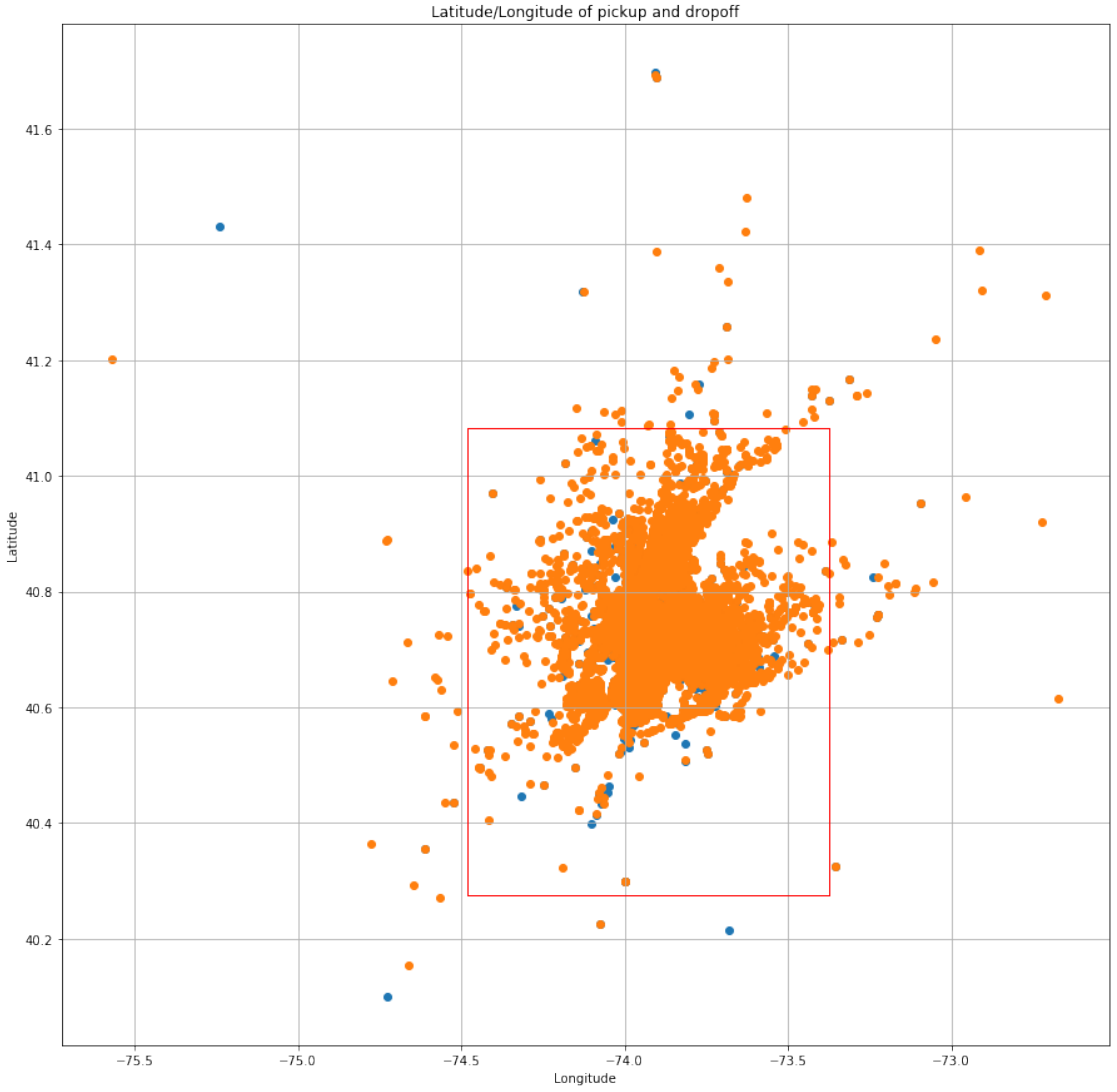
```
Query: SELECT min(LONGITUDE) as min_long, max (LONGITUDE) as max_long,
min(latitude) as min_lat, max(latitude) as max_lat from stations
```

[18]:

	min_long	max_long	min_lat	max_lat
0	-74.482746	-73.37309	40.275368	41.083546

Use those *stations* limit values to remap our rectangle to a value that will match our weather stations locations:

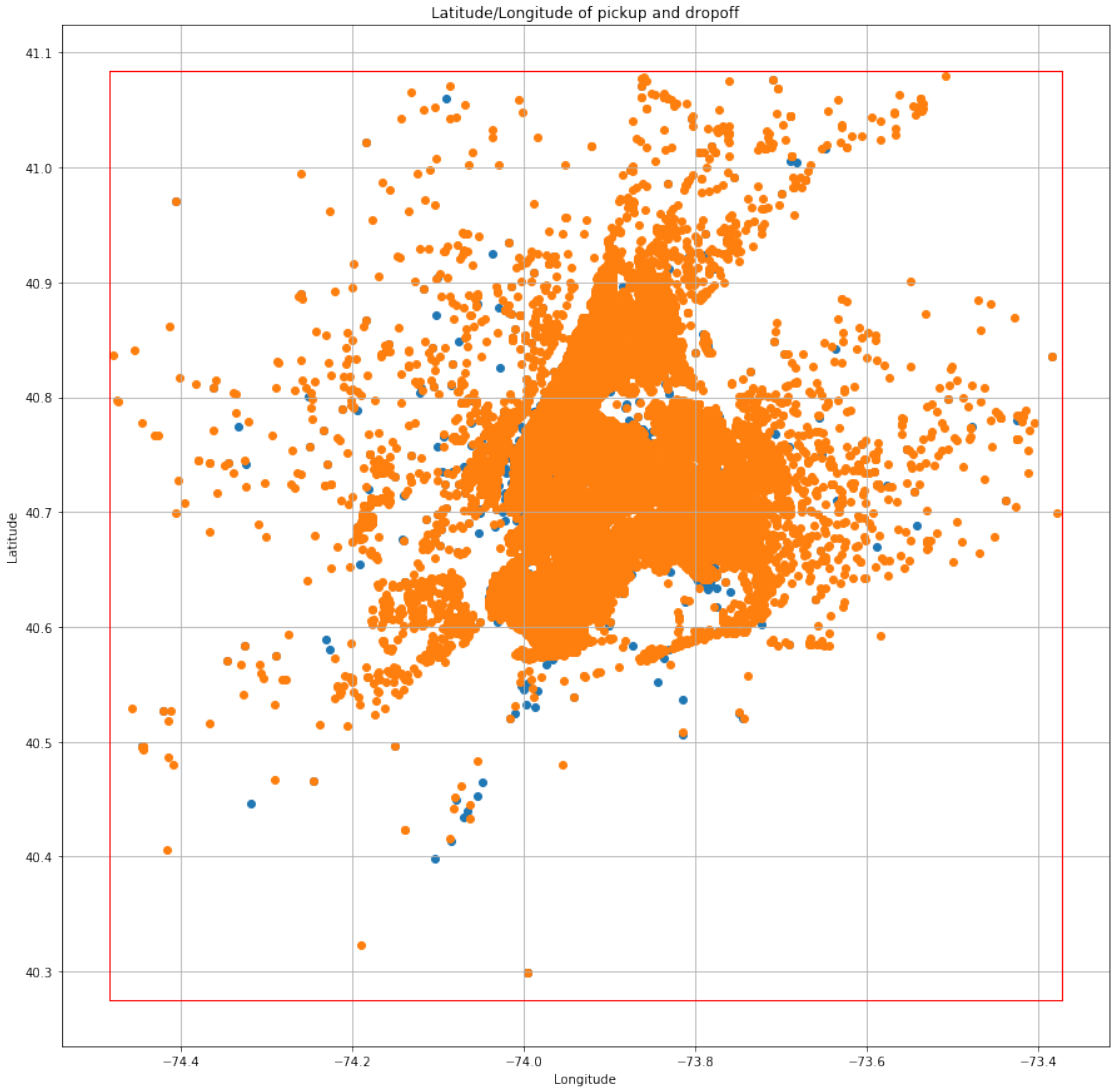
[19]:
```
# Set rectangle angle position
LOWER_LEFT=[float(stations_limit['max_long']), float(stations_limit['min_lat'])]
UPPER_RIGHT=[float(stations_limit['min_long']), float(stations_limit['max_lat'])]

draw_scatter()
```



This new rectangle value will remove a few lines, I think that our outliers removal is quite good now.

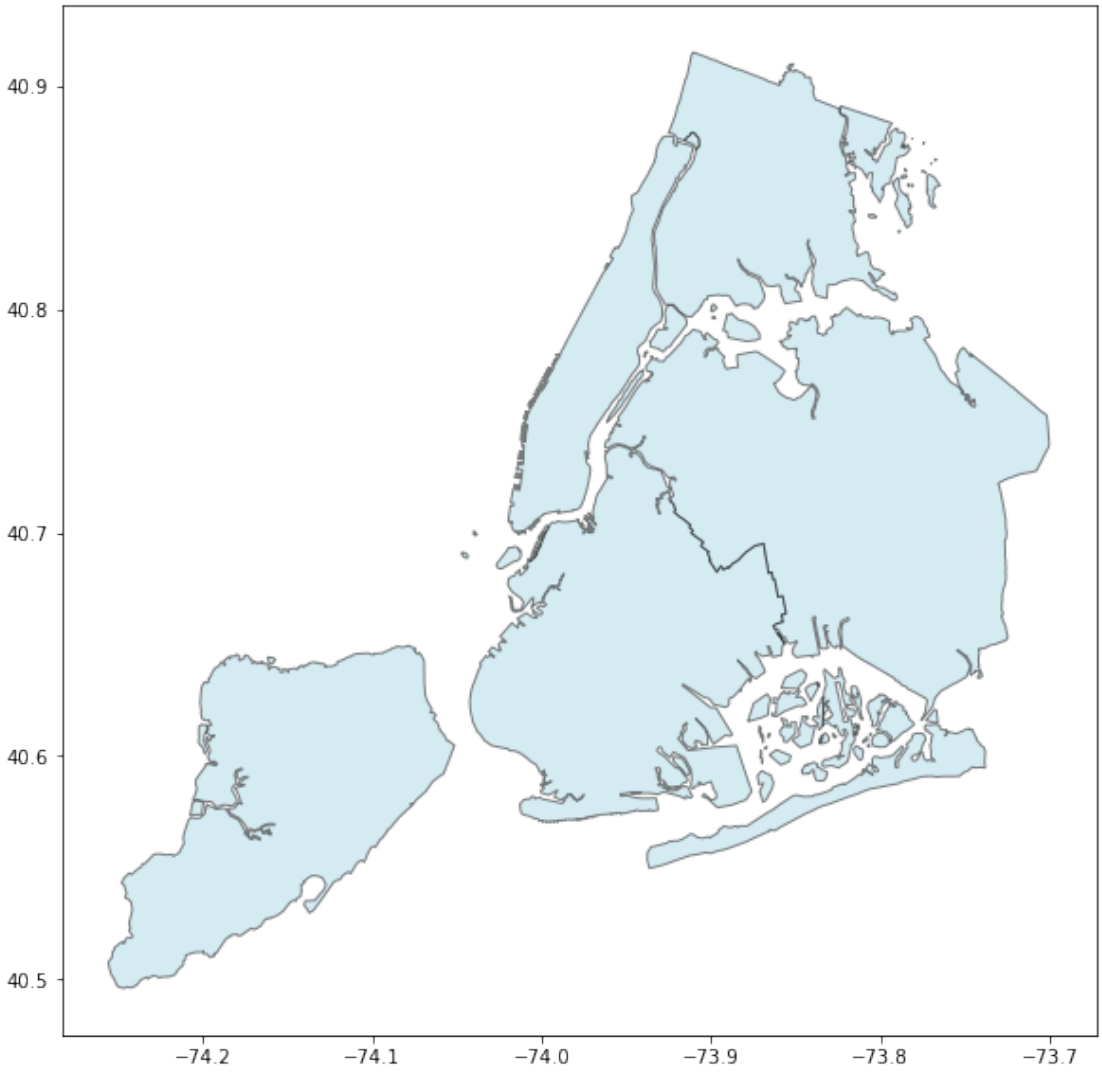
[20]:
```
# get global filter
global_filter=get_global_filter()

# The try/except block is used when no False values exists in global_filter
try:
    print("Number of lines that will be removed from dataset:", global_filter.
 ↪value_counts()[False])
except:
    print("No lines found to be removed")


# Remove outliers
nyc_travel=nyc_travel[global_filter]

draw_scatter()
```

```
Number of lines that will be removed from dataset: 100
```



Hey, that really starts looking like New-York City :-)

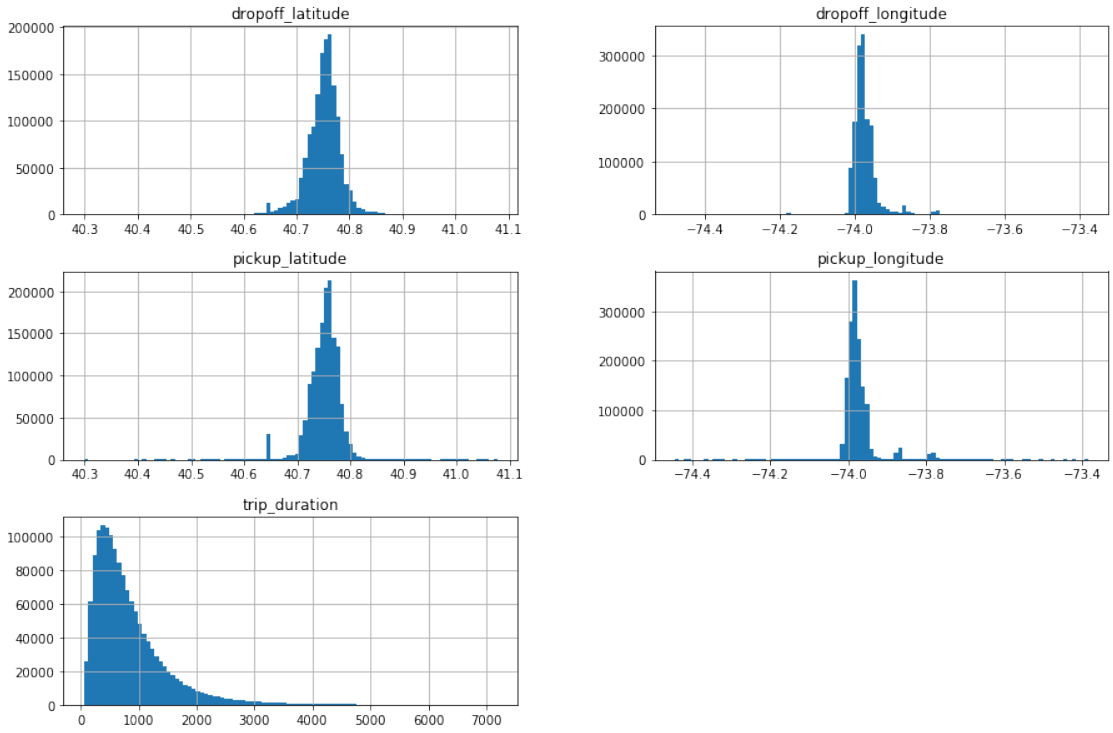
[21]:
```
# Function from 'my_utils' library
draw_nyc_map()
```



As a final control, let's display histogram of the numerical features of this dataset, this will shows how the cleaning process has provided good results.

[22]:
```
print("Number of lines in the dataset:", nyc_travel['trip_duration'].count())

nyc_travel.drop(['passenger_count', 'store_and_fwd_flag'], axis=1).
 ↪hist(bins=100, figsize=(15,10))
plt.show()
```

```
Number of lines in the dataset: 1447643
```



That'all folks for this Notebook.

> Note: As we can see on the histogram above, there's some Feature Engineering to do on the dataset, but that's another story.

2 Save dataset to SQL database

[23]:
```
save_sql(nyc_travel, 'travel')
```

```
Saving OK
```

[23]:
```
True
```

Control that the INSERT process into database worked by counting the number of lines present in the *travel* table:

[24]:
```
load_sql(query="SELECT count(*) FROM travel")
```

```
Query: SELECT count(*) FROM travel
```

[24]:

	count(*)
0	1447643

Let's continue on the next notebook: NYC Weather Data Preparation :-)

13.NYC Weather Data Preparation

February 2, 2022

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

1 Correlation Matrix and Heatmap

In this Notebook, I've decided to use the *pd.DataFrame.corr()* method to get a correlation matrix between features. This approach should help identify some correlations between features and determine which of them could be safely dropped without negatively influence model training performance.

Before starting working on this dataset,

To simplify the correlation matrix analysis, I've coded a function to draw the result of *pd.DataFrame.corr()* method using *Seaborn Heatmap* graph:

```
def draw_correlation_matrix(dataset, title='Correlation Matrix', figsize=(10,10), fontsize=10, s
    """
    Draw a correlation matrix using a Seaborn heatmap for better visual search of data correlati

    Expects a dataset as parameter, and optional features like title, figsize, fontsize, and sns

    Returns:
    --------
    None

    """
```

This function has been implemented in the my_utils library.

2 The _ATTRIBUTES columns

Each of the idependent variables described in the Data Preparation Notebook, but LOCATION and DATE, do have an *_ATTRIBUTES* supplementary columns that gives some more informa-

tions on the variable its related to.

As I do not have the description of the content of these *_ATTRIBUTES* columns, it's quite difficult to understand the meaning of each one (for example, when *PRCP* (precipitation) is set to 0.0, the *PRCP_ATTRIBUTES* contains „*N* which is not clear for me what is the sense of that value.

Anyway, the other columns do have enough interesting features, so I decide to drop the *_ATTRIBUTES* columns.

2.1 Load dataset and drop _ATTRIBUTES columns

[2]:
```
# Function from 'my_utils' library
nyc_weather=load_csv('weather')

# df = df[df.columns.drop(list(df.filter(regex='Test')))]

# Get the _ATTRIBUTES column name using a regex
attributes_columns=nyc_weather.filter(regex='_ATTRIBUTES').columns

print("Columns that will be dropped:")
print(list(attributes_columns))

nyc_weather.drop(attributes_columns, axis=1, inplace=True)
```

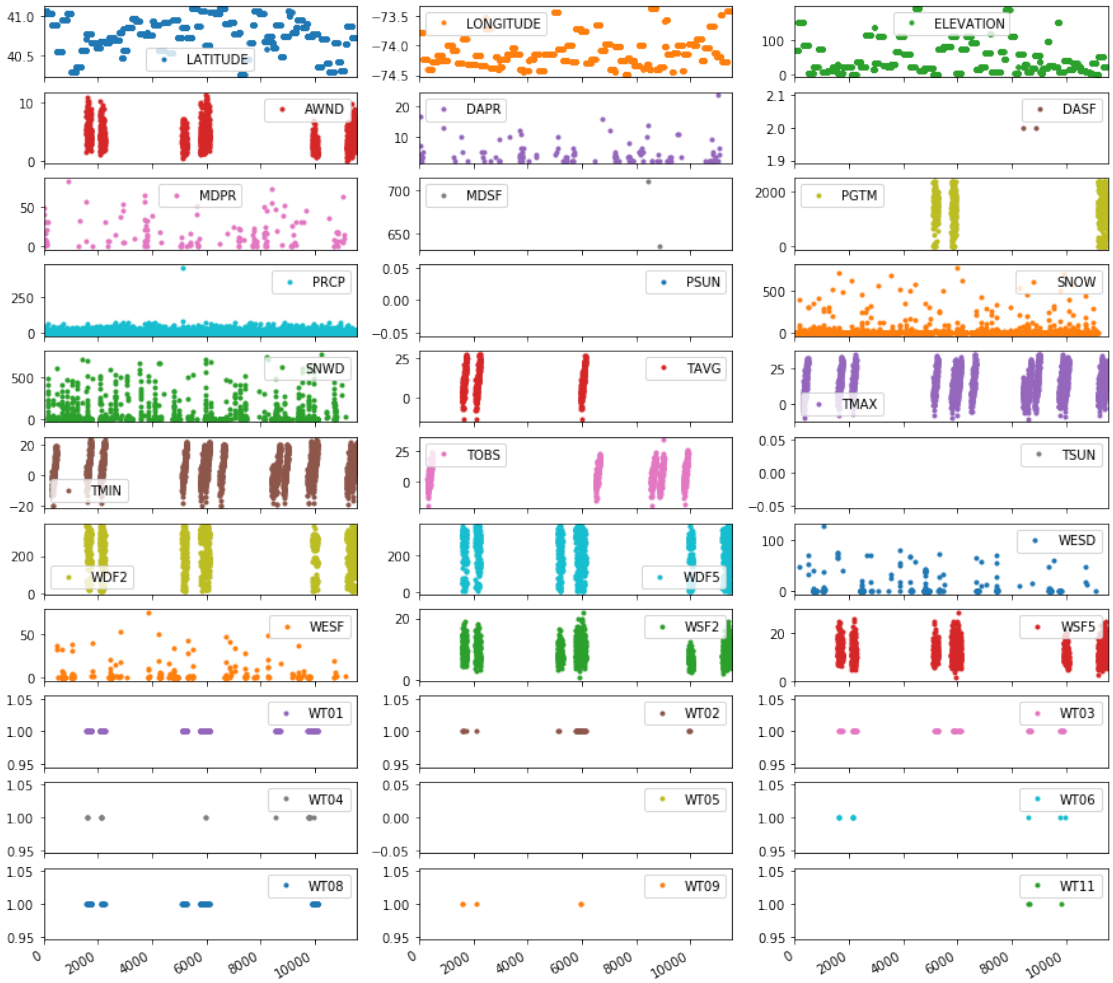
```
Columns that will be dropped:
['AWND_ATTRIBUTES', 'DAPR_ATTRIBUTES', 'DASF_ATTRIBUTES', 'MDPR_ATTRIBUTES',
'MDSF_ATTRIBUTES', 'PGTM_ATTRIBUTES', 'PRCP_ATTRIBUTES', 'PSUN_ATTRIBUTES',
'SNOW_ATTRIBUTES', 'SNWD_ATTRIBUTES', 'TAVG_ATTRIBUTES', 'TMAX_ATTRIBUTES',
'TMIN_ATTRIBUTES', 'TOBS_ATTRIBUTES', 'TSUN_ATTRIBUTES', 'WDF2_ATTRIBUTES',
'WDF5_ATTRIBUTES', 'WESD_ATTRIBUTES', 'WESF_ATTRIBUTES', 'WSF2_ATTRIBUTES',
'WSF5_ATTRIBUTES', 'WT01_ATTRIBUTES', 'WT02_ATTRIBUTES', 'WT03_ATTRIBUTES',
'WT04_ATTRIBUTES', 'WT05_ATTRIBUTES', 'WT06_ATTRIBUTES', 'WT08_ATTRIBUTES',
'WT09_ATTRIBUTES', 'WT11_ATTRIBUTES']
```

3 Dataset cleaning process

Now that the *_ATTRIBUTES* columns are dropped, let's plot the different columns and do some quick analysis.

> Note: The following code to display all features in one global graph has been provided to me as an example by Michael, thanks to him ;-)

[3]:
```
# Plot features
nyc_weather.plot(lw=0, marker='.', subplots=True, figsize=(15,15), layout=(-1,3))
plt.show()
```



3.1 Empty columns

What I can see in the previous overall graph is that some of the colums are empty. Let's confirm this using the combination of *DataFrame.count()*, and *Series.sort_values()* method to get the number of null values present in each column.

Note: The *count()* method returns the number of non-NaN values in each column

[4]:
```
# Get number of non null values in each columns
nyc_weather.count().sort_values()
```

[4]:
```
TSUN    0
WT05    0
PSUN    0
MDSF    2
DASF    2
WT11    3
```

```
WT09             5
                ...
PRCP         11343
ELEVATION    11449
DATE         11544
LONGITUDE    11544
LATITUDE     11544
NAME         11544
STATION      11544
Length: 36, dtype: int64
```

That's it, three columns are empty, we can drop them as they won't give any informations to our models: - PSUN - Daily percent of possible sunshine for the period - TSUN - Total sunshine for the period - WT05 - Hail (may include small hail)

[5]:
```
# Drop empty columns
nyc_weather.drop(['PSUN', 'TSUN','WT05'], axis=1, inplace=True)
```

3.2 Columns with very low data

I've found 2 columns wich the non-null values repesent the majority of the rows. This is very low and might not be usefull to train our model: - DASF - Number of days included in the multiday snow fall total (MDSF) - MDSF - Multiday snowfall total

Looking a bit more on the description of this column, I've found that those columns count the number of consecutive day of snow fall and the cumulated snow that falls during those cumulative days.

There's also two other fields that are similar to DASF and MDSF: - DAPR - Number of days included in the multiday precipitation total (MDPR) - MDPR - Multiday precipitation total (use with DAPR and DWPR, if available)

This columns are related to precipitation total for the multiday precipitaion total.

As I do really not see how this information could be used, I've decided to drop the four of them.

[6]:
```
# Drop cumulative days column informations
nyc_weather.drop(['DASF', 'MDSF','DAPR', 'MDPR'], axis=1, inplace=True)
```

3.3 Rows related the 1st of July 2016

Looking at the dates we have in this dataset, I've found that it covers the 1st of January 2016 to the 1st of July 2016.

The *travel* dataset, covers the same interval but the 1st of July 2016.

As I will never use *weather* information for days that are not in the *travel* dataset, I will simply drop lines where DATE='2016-07-01'

[7]:
```
# Get max day from nyc_weather
```

```
print("Max date in dataset                    :", nyc_weather['DATE'].
 ↪sort_values(ascending=False).iloc[0])

# Count lines related to DATE='2016-07-01'
lines_to_remove=len(nyc_weather[nyc_weather.DATE == '2016-07-01'].index)
print("Number of lines for DATE='2016-07-01' :", lines_to_remove)

# Count total lines
total_lines=len(nyc_weather.index)
print("Total number of lines in dataset       :", total_lines)

# Drop lines related to DATE='2016-07-01'
nyc_weather=nyc_weather[nyc_weather.DATE != '2016-07-01']
print("Number of lines after cleaning         :", len(nyc_weather.index))

if(total_lines == lines_to_remove + len(nyc_weather.index)):
    print("Removal OK")
else:
    print("Number of lines removed does not match total number of lines")
```

```
Max date in dataset                    : 2016-07-01
Number of lines for DATE='2016-07-01' : 57
Total number of lines in dataset       : 11544
Number of lines after cleaning         : 11487
Removal OK
```

4 Categorical and Numerical columns

Using the *describe()* method on the dataset, I've been able to quickly identify *categorical* values, and by exclusion the *numerical* ones. Of course, I do not consider *STATION*, *NAME*, *LATITUDE*, *LONGITUDE* AND *DATE* columns as they are identification keys of the lines.

> Note: I use the *astype()* method to get values as integers. This will be more easy to identify categorical values.

[8]:
```
nyc_weather.describe().astype('int')
```

[8]:

	LATITUDE	LONGITUDE	ELEVATION	AWND	PGTM	PRCP	SNOW	SNWD	TAVG	TMAX	TMIN	TOBS
count	11487	11487	11392	1445	725	11287	6951	3067	546	2410	2420	852
mean	40	-74	46	3	1306	2	4	26	10	14	4	6
std	0	0	44	1	555	8	37	90	8	9	8	8
min	40	-74	2	0	1	0	0	0	-13	-11	-19	-18
25%	40	-74	13	2	1037	0	0	0	4	6		

```
-1      0
50%          40         -74          24      3  1354       0      0      0     10     14
5      7
75%          40         -74          72      5  1627       1      0      0     17     22
10     13
max          41         -73         188     11  2359     451    770    780     27     35
22     33

        WDF2  WDF5  WESD  WESF  WSF2  WSF5  WT01  WT02  WT03  WT04  WT06  WT08
WT09  WT11
count   1448  1444   346   455  1448  1444   436    40    54    17    10   172
5     3
mean     208   206     7     2     8    11     1     1     1     1     1     1
1     1
std       98   102    18     8     3     3     0     0     0     0     0     0
0     0
min       10     5     0     0     0     1     1     1     1     1     1     1
1     1
25%      150   137     0     0     6     8     1     1     1     1     1     1
1     1
50%      220   210     0     0     8    11     1     1     1     1     1     1
1     1
75%      300   300     1     0    10    14     1     1     1     1     1     1
1     1
max      360   360   126    74    21    28     1     1     1     1     1     1
1     1
```

The *WT* columns seems to be categorical ones. Look at the *mean*, *std* and *max* values ;-)

There's also two interseting columns, *WDF2* and *WDF5* which, according to the dataset description, determines the direction of the wind. Those two columns should be defined as categorical one, classifying wind direction in distinct directions like North-East, South-West and so on. We'll see below.

4.1 The WT* columns

Let's explore first what I've identifed as *categorical* columns: The one starting with WT:

[9]:
```
# Get WT column names
cat_columns=nyc_weather.filter(regex='WT').columns
nyc_weather[cat_columns].describe().astype('int')
```

[9]:
```
        WT01  WT02  WT03  WT04  WT06  WT08  WT09  WT11
count    436    40    54    17    10   172     5     3
mean       1     1     1     1     1     1     1     1
std        0     0     0     0     0     0     0     0
min        1     1     1     1     1     1     1     1
25%        1     1     1     1     1     1     1     1
```

```
50%      1     1     1     1     1     1     1     1
75%      1     1     1     1     1     1     1     1
max      1     1     1     1     1     1     1     1
```

As I've seen above, most of the lines into those columns are fill win *NaN* values. We can confirm that looking at the previous *describe()* graph. The min and max values for those column are equal to 1, with a standard deviation of 0. This means that we do have only 1 values in those columns, the rest is set to *NaN*.

[10]:
```
for col in cat_columns:
    print("Unique values in {} column: {}".format(col, nyc_weather[col].
 ↪unique()))
```

```
Unique values in WT01 column: [nan  1.]
Unique values in WT02 column: [nan  1.]
Unique values in WT03 column: [nan  1.]
Unique values in WT04 column: [nan  1.]
Unique values in WT06 column: [nan  1.]
Unique values in WT08 column: [nan  1.]
Unique values in WT09 column: [nan  1.]
Unique values in WT11 column: [nan  1.]
```

So what ? Well, easy, simply replace the *NaN* cells in those columns by a *0* and we are done, the *WT* columns will be categorical ones filles in with 0 and 1 values:

[11]:
```
# fill NaN values in WT column with 0
df=nyc_weather.copy()
df[cat_columns]=df[cat_columns].fillna(0)
nyc_weather=df.copy()
```

Just to confirm, let display the unique values and *describe()* method. Perfect :-)

[12]:
```
for col in cat_columns:
    print("Unique values in {} column: {}".format(col, nyc_weather[col].
 ↪unique()))

nyc_weather[cat_columns].describe().astype('int')
```

```
Unique values in WT01 column: [0. 1.]
Unique values in WT02 column: [0. 1.]
Unique values in WT03 column: [0. 1.]
Unique values in WT04 column: [0. 1.]
Unique values in WT06 column: [0. 1.]
Unique values in WT08 column: [0. 1.]
Unique values in WT09 column: [0. 1.]
Unique values in WT11 column: [0. 1.]
```

[12]:

	WT01	WT02	WT03	WT04	WT06	WT08	WT09	WT11
count	11487	11487	11487	11487	11487	11487	11487	11487
mean	0	0	0	0	0	0	0	0

std	0	0	0	0	0	0	0	0
min	0	0	0	0	0	0	0	0
25%	0	0	0	0	0	0	0	0
50%	0	0	0	0	0	0	0	0
75%	0	0	0	0	0	0	0	0
max	1	1	1	1	1	1	1	1

4.2 The *WDF2* and *WDF5* columns

So, I've mention that those two columns do determine the direction of the wind measured by weather stations. The first one contains the wind direction during the last 2 mn, the second one for the last 5 minutes.

The direction is expressed in degrees, 360 meaning the North, 180 the South.

Let's first look at their characteristics, including informations on the three columns concerning wind speed: - AWSD - WSD2 - WSD5

As an evidence, I should consider values in wind direction columns only when there's wind, which means value in *AWND* is not equal to NaN.

Let's first display the weather station that measure the wind speed.

[13]:
```
nyc_weather[~nyc_weather.AWND.isna()].groupby('STATION')[['AWND']].describe()
```

[13]:

	AWND							
	count	mean	std	min	25%	50%	75%	max
STATION								
USW00014732	182.0	4.903297	1.902249	1.6	3.425	4.6	5.775	10.8
USW00014734	182.0	4.304945	1.824368	1.1	3.000	4.1	5.200	10.2
USW00054743	182.0	2.476923	1.361838	0.1	1.500	2.2	3.075	6.8
USW00054787	180.0	4.236111	1.652111	1.0	3.100	3.9	5.125	9.8
USW00094728	177.0	2.655367	1.155661	0.6	1.900	2.5	3.300	7.0
USW00094741	179.0	3.346369	1.488580	0.5	2.300	3.0	4.300	7.2
USW00094745	181.0	3.616575	1.895360	1.1	2.300	3.1	4.500	9.9
USW00094789	182.0	5.296703	2.142413	1.6	3.700	5.0	6.500	11.2

It seems that only 9 of them do measure the wind speed.

Ok, let's have a look now at the wind direction columns.

[14]:
```
wind_columns=['AWND', 'WSF2', 'WDF2', 'WDF5', 'WSF5']
nyc_weather[wind_columns].describe()
```

[14]:

	AWND	WSF2	WDF2	WDF5	WSF5
count	1445.000000	1448.000000	1448.000000	1444.000000	1444.000000
mean	3.859377	8.771478	208.887431	206.159972	11.938989
std	1.949278	3.050781	98.654138	102.189245	3.984617
min	0.100000	0.900000	10.000000	5.000000	1.300000
25%	2.500000	6.700000	150.000000	137.500000	8.900000
50%	3.500000	8.100000	220.000000	210.000000	11.200000

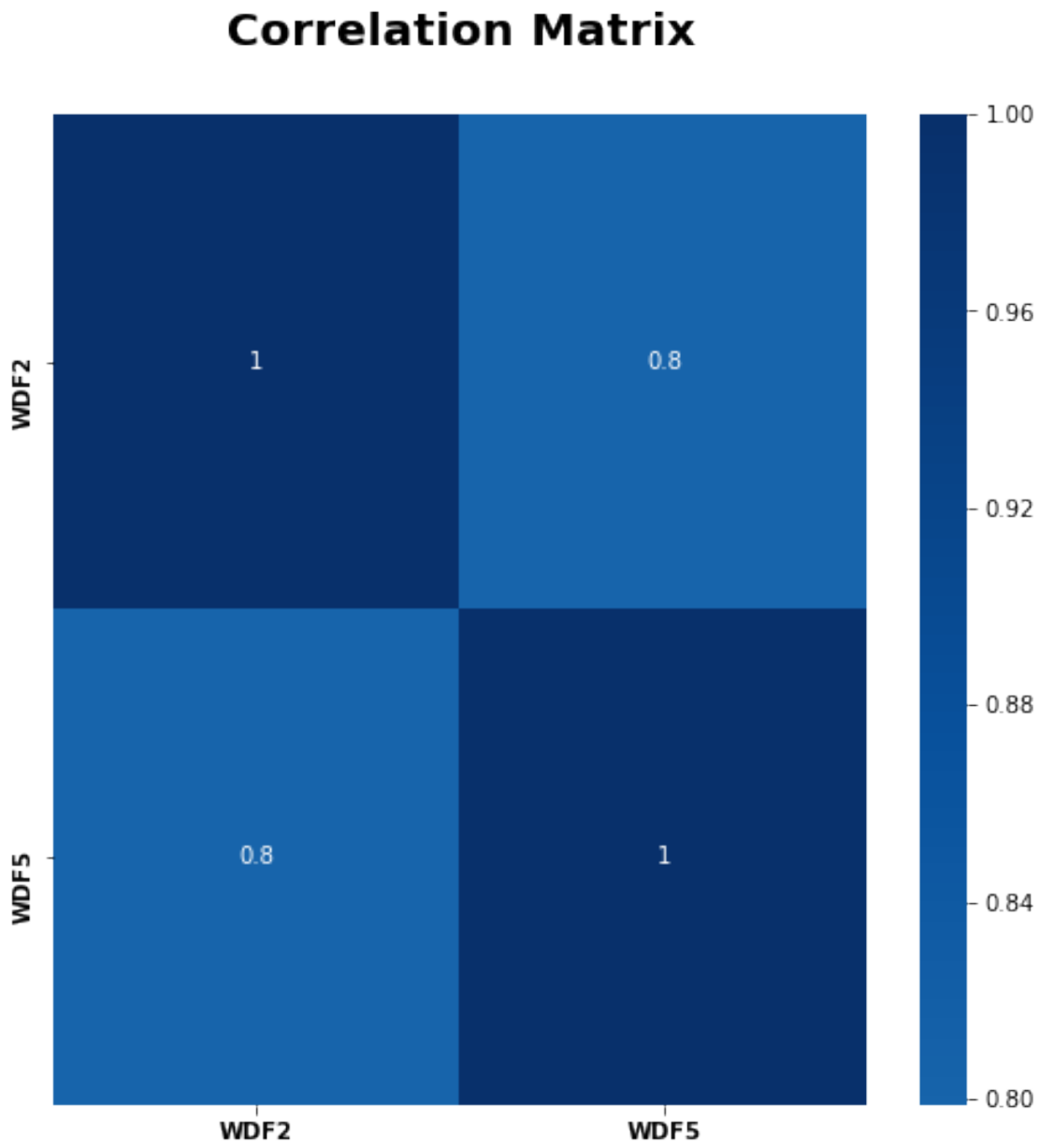
75%	5.000000	10.700000	300.000000	300.000000	14.300000
max	11.200000	21.900000	360.000000	360.000000	28.600000

Around 10% of lines of the Dataset contains information in those columns.

Let's look at their correlation.

> Note: I do remove the lines where *WDF2* is NaN to not negatively influence the correlation matrix

[15]:
```
draw_correlation_matrix(nyc_weather[nyc_weather.WDF2.notna()][['WDF2', 'WDF5']],␣
↪figsize=(8,8))
```



That's clear, those two columns have a correlation factor really high (near 1.0), which means that one of them should be dropped.

I've decided to drop the *WDF5* as it has a bit less of measures (1'452 against 1'456 for the *WDF2*)

Regarding *WDF2*, I will transform it into categorical columns by distributing values into a new feature, *WDIR*, using the following translation table:

> Note: The NaN values will be categorized has No-Wind values with direction code = 'O'

Direction	Direction Code	Low Angle limit	Higher Angle limit
North	N	337.5	22.5
North East	NE	22.5	67.5
East	E	67.5	112.5
South East	SE	112.5	157.5
South	S	157.5	202.5
South West	SW	202.5	247.5
West	W	247.5	292.5
North West	NW	292.5	337.5
No Wind	O	NaN	NaN.

This new classification stored in the new *WDIR* column will be then converted into indicator variables using *pandas.get_dummies()*

> Note: Before doing this classification I will verify that each time I have a wind direction value, I also have a wind speed. Otherwise, the wind direction will be set to *No Wind*

[16]:
```
# Drop *WDF5* column
nyc_weather=nyc_weather.drop(['WDF5'], axis=1)
```

[17]:
```
def get_wind_direction(value) -> str:
    """
    Function that will convert a value in a direction code.

    Note that the value 'No direction' is based on a cell value = -1
    We will have to fill NaN values of dataset with -1 in order for this␣
 ↪function to work.


    Results:
    --------
    str

    """
    if(value == -1):
        return 'O'
```

```
    elif(value <= 22.5):
        return 'N'
    elif(value <= 67.5):
        return 'NE'
    elif(value <= 112.5):
        return 'E'
    elif(value <= 157.5):
        return 'SE'
    elif(value <= 202.5):
        return 'S'
    elif(value <= 247.5):
        return 'SW'
    elif(value <= 292.5):
        return 'W'
    elif(value <= 337.5):
        return 'NW'
    else:
        return 'N'

# Ensure that if AWND column is NaN and WDF2 is not NaN, set WDF2 to -1 (if no␣
 ↪wind, there's no direction)
for id in nyc_weather[np.logical_and(nyc_weather.AWND.isna(), ~nyc_weather.WDF2.
 ↪isna())].index:
    nyc_weather.at[id, 'WDF2']=-1

# Apply conversion function to WDF2 columns (NaN replaced with -1)
nyc_weather['WDIR']=nyc_weather.fillna(-1)['WDF2'].apply(get_wind_direction)


print("Value counts of the new WDIR column:")
nyc_weather['WDIR'].value_counts()
```

```
Value counts of the new WDIR column:
```

[17]:
```
0     10042
NW      298
S       289
W       239
N       140
SW      136
NE      135
SE      116
E        92
Name: WDIR, dtype: int64
```

Now, drop the *WDF2* column as it will be replaced by the categorical one *WDIR*, and run a *pandas.get_dummies()* function on the *WDIR* column.

[18]:
```
# Drop WDF2 column
nyc_weather.drop('WDF2', axis=1, inplace=True)
```

[19]:
```
# OneHotEncoder with pandas.get_dummies() method
nyc_weather = pd.get_dummies(nyc_weather, columns=['WDIR'])
```

The *pandas.get_dummies()* function added 9 columns as I did not set the *drop_first* parameter to *True*. Why ?

Well, I would like to be sure that the column that will be dropped is the *WDIR_O*, the one spcifying that there is no direction as there is no wind.

So the last thing I have to do now is to drop the *WDIR_O* columns as it represents lines where all other values are set to 0.

[20]:
```
# Remove WDIR_O column
nyc_weather.drop('WDIR_O', axis=1, inplace=True)
```

[21]:
```
# Let's display the WDIR_ columns whera AWND is not NaN
nyc_weather[nyc_weather.AWND.notna()][nyc_weather.filter(regex='WDIR_|AWND').
 ↪columns].head(3)
```

[21]:

	AWND	WDIR_E	WDIR_N	WDIR_NE	WDIR_NW	WDIR_S	WDIR_SE	WDIR_SW	WDIR_W
1577	6.0	0	0	0	0	0	0	0	1
1578	5.6	0	0	0	0	0	0	0	1
1579	5.4	0	0	0	0	0	0	0	1

5 Numerical columns

It's time now to have a look at the numerical columns. To get the list of these column names, I will simply remove from the dataset the categorical columns (*WT* and *WDIR*) and the static data columns (*NAME*, *STATION*, *LONGITUDE*, *LATITUDE* and *ELEVATION*)

Note: *ELEVATION* is considered as static data information of the weather stations

[22]:
```
# Build numercial column name list
num_columns=nyc_weather.filter(regex='^(?!WT)').filter(regex='^(?!WDIR)').
 ↪drop(['NAME','STATION','LONGITUDE','LATITUDE','DATE', 'ELEVATION'], axis=1).
 ↪columns

# Describe the columns
nyc_weather[num_columns].describe().astype('int')
```

[22]:

	AWND	PGTM	PRCP	SNOW	SNWD	TAVG	TMAX	TMIN	TOBS	WESD	WESF	WSF2	WSF5
count	1445	725	11287	6951	3067	546	2410	2420	852	346	455	1448	1444
mean	3	1306	2	4	26	10	14	4	6	7	2	8	11

```
std      1   555     8    37    90     8     9     8     8    18     8     3
3
min      0     1     0     0     0   -13   -11   -19   -18     0     0     0
1
25%      2  1037     0     0     0     4     6    -1     0     0     0     6
8
50%      3  1354     0     0     0    10    14     5     7     0     0     8
11
75%      5  1627     1     0     0    17    22    10    13     1     0    10
14
max     11  2359   451   770   780    27    35    22    33   126    74    21
28
```

As I can see, except for the PRCP column, most of them contains values for mostly less then 10% of the lines.

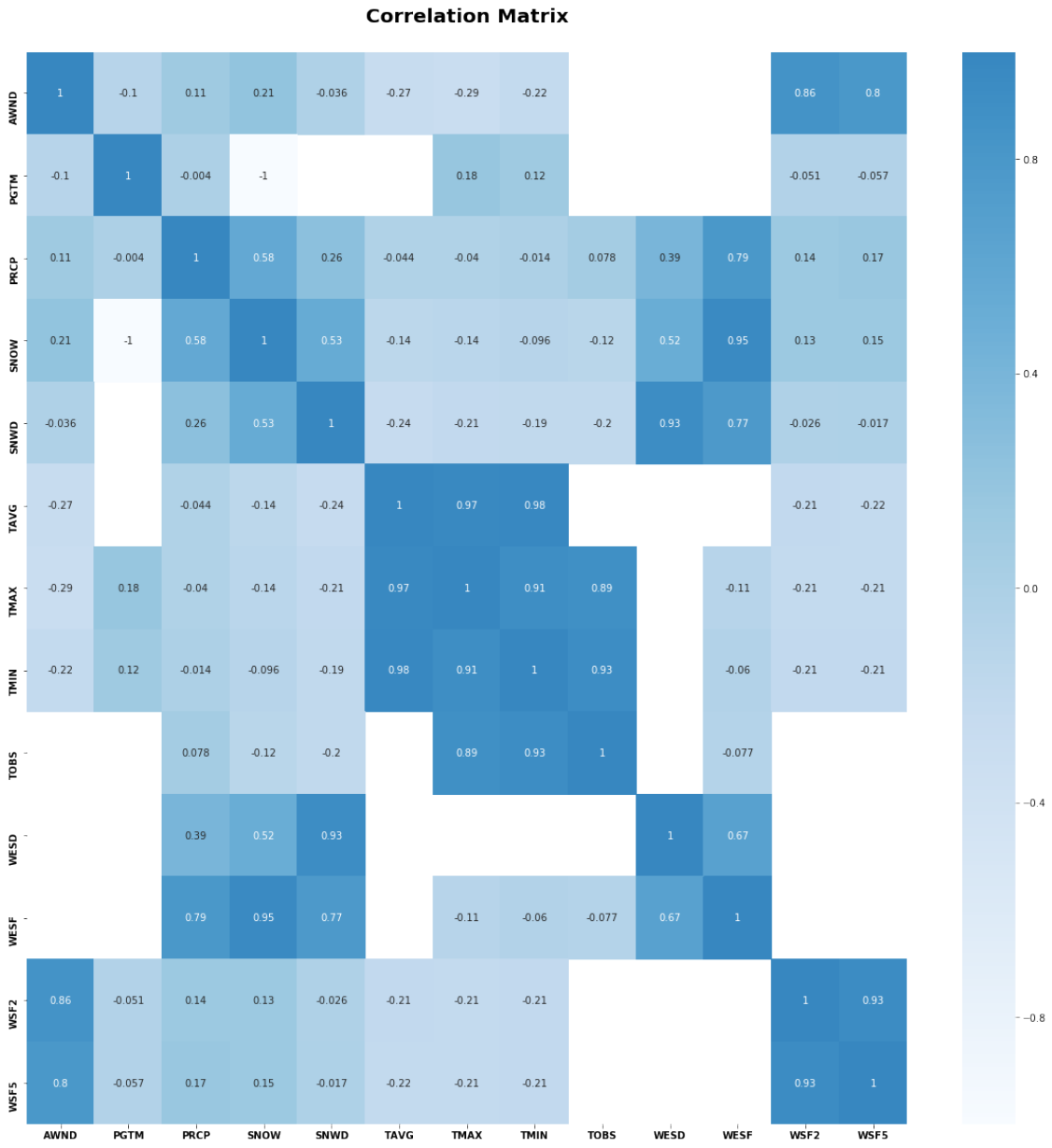
That should be normal. The *SNOW* column gives the quantity of snow fall during the day, it would have been surprising to have snow every day in NYC ;-)

Before doing a deep dive on each column, I will start analyzing any correlation between them.

5.1 Correlation matrix

Let's use my Correlation Matrix Heatmap drawing function on all the features of the dataset.

[23]:
```
# let's graph a correlation matrix with a Seaborn graph
# Function comes from 'my_utils' library
draw_correlation_matrix(dataset=nyc_weather[num_columns], figsize=(20,20))
```



Nice result :-)

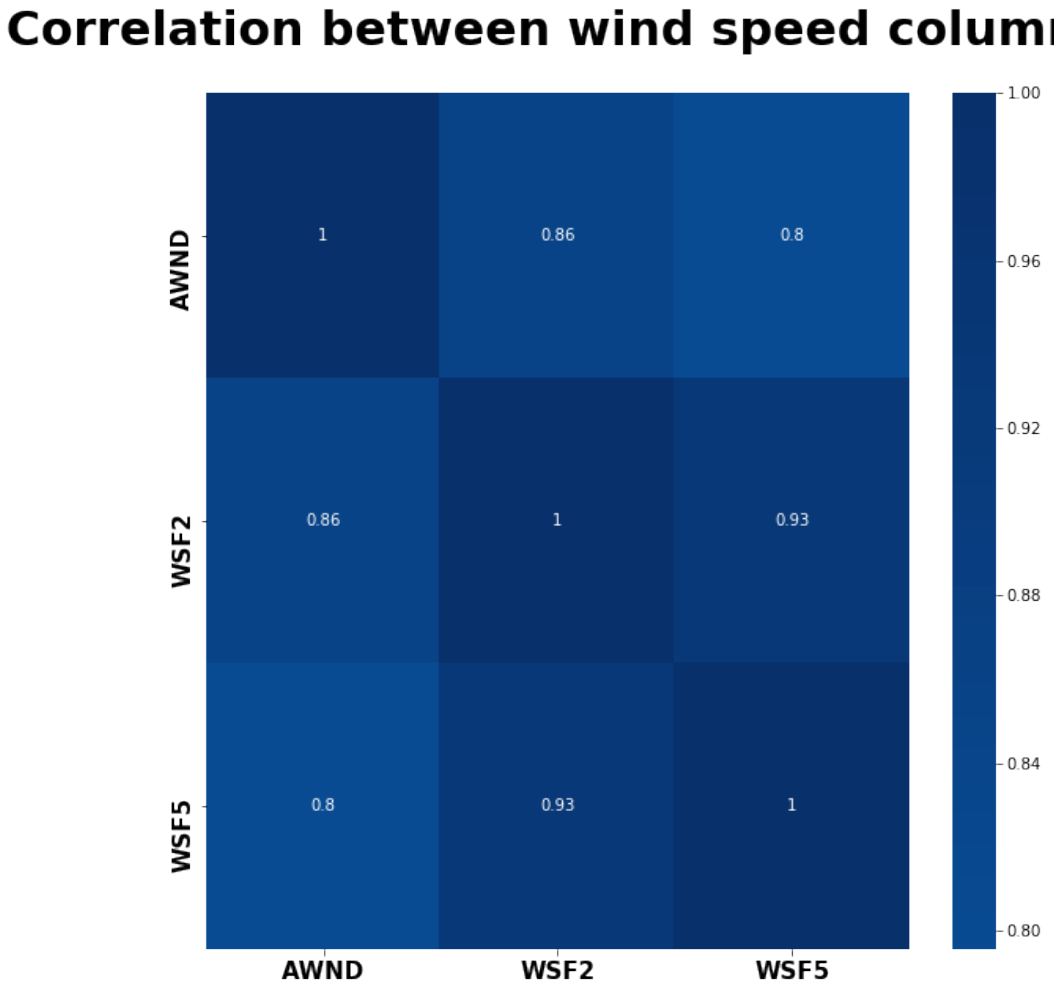
Zoom on some results from the previous graph and identify any features to drop.

5.2 AWND, WSF2 and WSF5 columns

Those columns are highly correlated, they report respectively the Average Wind Speed (AWND), the fastest 2-minutes Wind Speed (WSF2) and the fastest 5-minutes Wind Wpeed (WSF5).

> Note: I've excluded all the values where *AWND* is NaN to not influence negatively the correlation matrix.

[24]:
```
# Correlation between wind speed columns
draw_correlation_matrix(nyc_weather[['AWND', 'WSF2', 'WSF5']],
                        title="Correlation between wind speed columns",
                        fontsize=15, center=0)
```



I decide to drop the *WSF2* and *WSF5* columns, and replace all the NaN values in the *AWND* columns with 0

> Note: Previously, while building the *WDIR* categorical columns, I've checked that each time I had a wind direction (*WDIR_O == 1), the AWND column as a value > 0. This is another good reason to keep that column and drop the others.

[25]:
```
# Drop WSF2 and WSF5 columns
nyc_weather.drop(['WSF2', 'WSF5'], axis=1, inplace=True)
```

[26]:
```
# Replace NaN value in AWND column with 0
nyc_weather['AWND'].fillna(0, inplace=True)
```

```
print("Number of NaN values in AWND column:",nyc_weather['AWND'].isna().sum())
```

```
Number of NaN values in AWND column: 0
```

5.3 PGTM column

There's another column that describes feature about wind: *PGTM*

This feature describe the number of seconds the weather station observed a peak gust.

Instead of keeping this column as is, I will transform it using *pandas.get_dummies()* method to build a categorical column: PEAK = yes or no

[27]:
```
def get_peak_gust_status(value) -> str:
    """
    Function that will convert the peak gust time in Y if > 0, N otherwise.

    Do not forget to fill NaN values with 0 before applying this function

    Results:
    --------
    str

    """
    if(value > 0):
        return 'Y'
    else:
        return 'N'

# Apply conversion function to PGTM columns and store result in PEAK column
nyc_weather['PEAK']=nyc_weather.fillna(0)['PGTM'].apply(get_peak_gust_status)


print("Value counts of the new PEAK column:")
nyc_weather['PEAK'].value_counts()
```

```
Value counts of the new PEAK column:
```

[27]:
```
N    10762
Y      725
Name: PEAK, dtype: int64
```

ok, now convert the *PEAK* columns in categorical one using *pandas.get_dummies()* and drop the useless *PGTM* column.

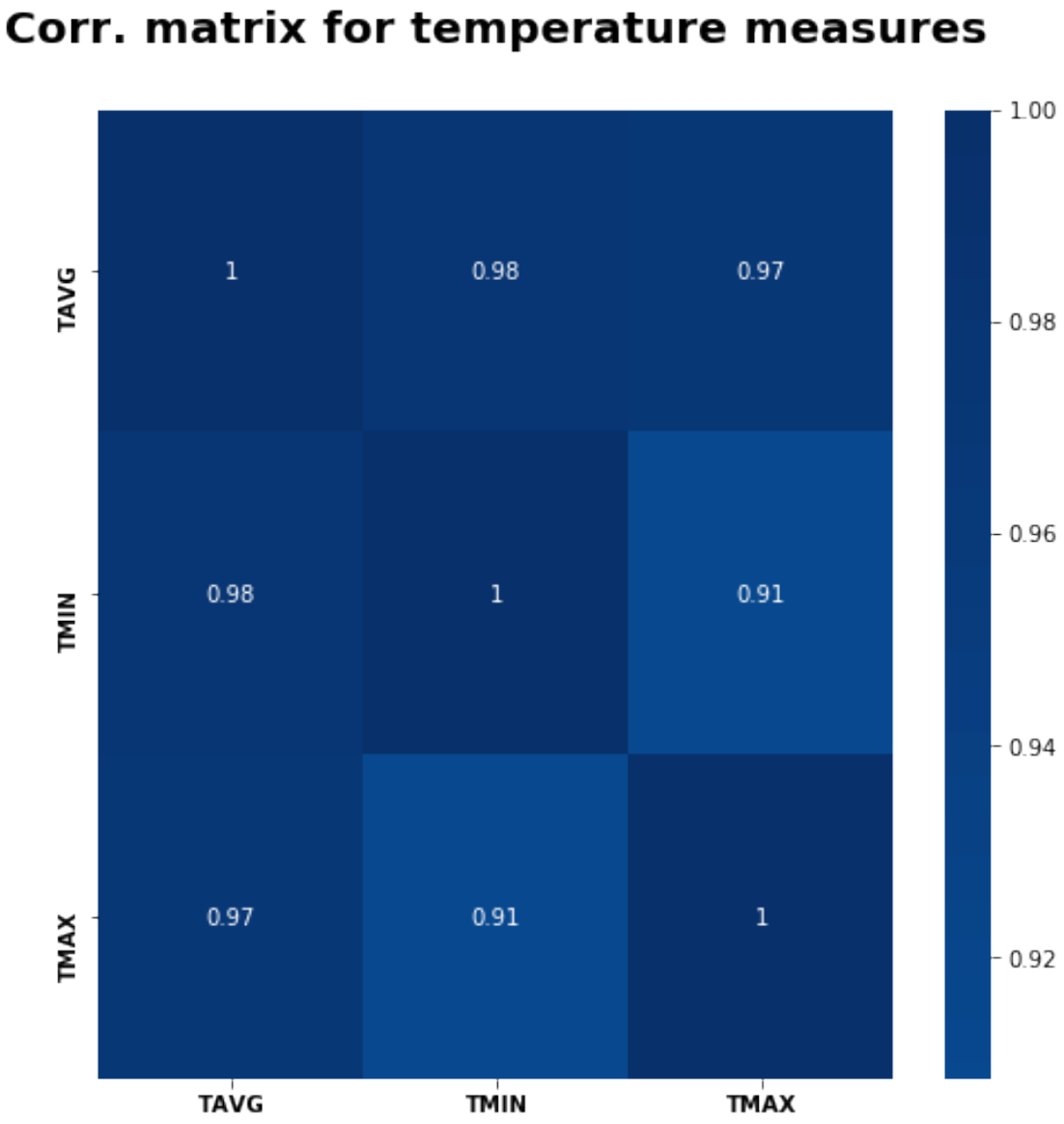
[28]:
```
nyc_weather.drop('PGTM', axis=1, inplace=True)
```

[29]:
```
# OneHotEncoder with pandas.get_dummies() method
nyc_weather = pd.get_dummies(nyc_weather, columns=['PEAK'], drop_first=True)
```

5.4 *TAVG*, *TMAX* and *TMIN* columns

Those columns also are highly correlated, they concern the temperature mesured: - TAVG - Average Temperature. - TMAX - Maximum temperature - TMIN - Minimum temperature

[30]:
```
draw_correlation_matrix(nyc_weather[['TAVG', 'TMIN', 'TMAX']],
                        title="Corr. matrix for temperature measures",
                        figsize=(8,8))
```



Having a closer look at those specific columns, I found that we only have three weather stations that do report values for *TAVG*, while *TMAX* and *TMIN* are reported on more weather station (14 stations).

Furthermore, each weather stations with a *TMIN* value **do have** a *TMAX* value.

[31]:
```
# Filter lines where TMiN.notna() and count values
nyc_weather[nyc_weather.TMIN.notna()][['STATION', 'TMAX', 'TMIN', 'TAVG']].
 ↪groupby(by='STATION').count().sort_values(by='TAVG', ascending=False)
```

[31]:

STATION	TMAX	TMIN	TAVG
USW00014732	182	182	182
USW00014734	182	182	182
USW00094789	182	182	182
USC00066655	172	182	0
USC00280907	180	180	0
USC00281335	182	182	0
USC00282023	60	60	0
USC00283704	182	182	0
USC00301309	182	182	0
USW00054743	182	182	0
USW00054787	180	180	0
USW00094728	182	182	0
USW00094741	180	180	0
USW00094745	182	182	0

For that reason, I've decided to replace the *TAVG* columns with an average based on the TMIN and TMAX values.

I'm not sure this is completly right, but this will solve the correlation between *TMIN* and *TMAX*, as well as increase the number af *TAVG* values.

While doing this transformation, I'll aslo create a new column, *TSTD*, which will contains the absolute value of *TMAX* minus *TAVG*. That way, I will keep the information of the variation amplitude of the temperature for each measures. A sort of standard deviation with the mean of two values ;-)

> Note 1: *TAVG* values where *TMIN* and *TMAX* are equal to NaN will be filled with NaN value, same for the *TSTD* feature.
>
> Note 2: As I've confirmed that each time I have a value for *TMIN*, I have one for *TMAX*, filtering will be done on *TMIN* only

[32]:
```
# get average function
def get_average(tmin, tmax) -> float:
    """
    Returns the average of the the two values passed as parameter.
    This function takes care of parameters received equal to np.nan

    Returns:
    --------
    float
```

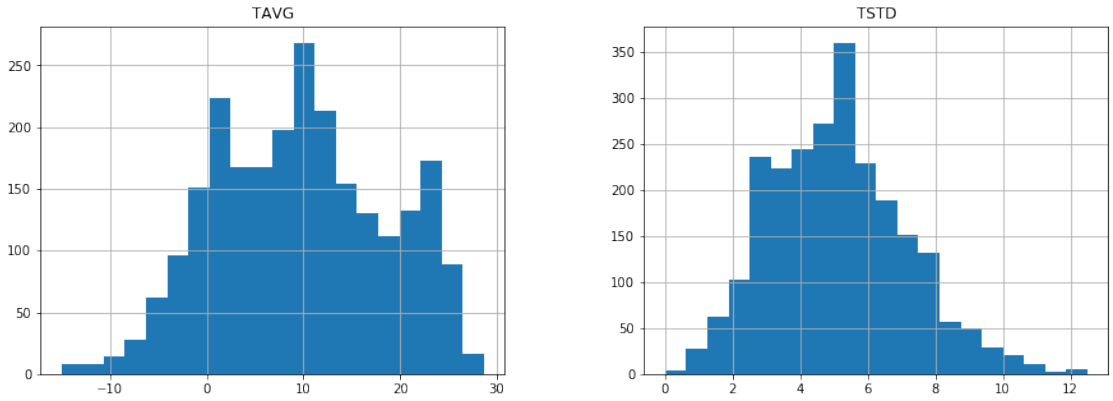
```
    """
    # initialize returned variable
    tavg=np.nan

    # Try to calculate the average
    try:
        tavg=(tmin+tmax)/2
    # if not posible, set return value to np.nan
    except:
        tavg=np.nan

    # Return calculated value
    return tavg

def get_diff(tmax, tavg) -> float:
    """
    Returns the absolute difference the two values passed as parameter.
    This function takes care of parameters received equal to np.nan

    Returns:
    --------
    float

    """

    # initialize returned variable
    tstd=np.nan

    # Try to calculate the difference
    try:
        tstd=tmax-tavg

    # if not posible, set return value to np.nan
    except:
        tstd=np.nan

    # return calculated value
    return tstd

# Set TAVG to NaN where TMIN.isna()
nyc_weather['TAVG']=nyc_weather.apply(lambda x: get_average(x['TMIN'],␣
 ↪x['TMAX']), axis=1)

# Set TAVG to NaN where TMIN.isna()
nyc_weather['TSTD']=nyc_weather.apply(lambda x: get_diff(x['TMAX'], x['TAVG']),␣
 ↪axis=1)
```

```
# Display some results
nyc_weather[nyc_weather.TMAX.notna()][['STATION', 'TMAX', 'TMIN', 'TAVG',␣
 ↪'TSTD']].head(2)
#nyc_weather.loc[0:0]['TMIN']
```

[32]:

	STATION	TMAX	TMIN	TAVG	TSTD
305	USC00280907	8.3	2.8	5.55	2.75
306	USC00280907	3.3	-0.6	1.35	1.95

[33]:
```
# Display histogram of TAVG values
nyc_weather[['TAVG', 'TSTD']].hist(bins=20, figsize=(15,5))
plt.show()
```



[34]:
```
# Display the number of empty values in TAVG (the one equal to -99)
nyc_weather[['TAVG', 'TSTD']].isna().sum()
```

[34]:
```
TAVG    9077
TSTD    9077
dtype: int64
```

The *TAVG* column has been recalculated using *TMIN* and *TMAX* average to fill some missing values, but there is still *TAVG* values that are set to *NaN*.

The reason is some temperature measures in the dataset are missing from weather station for each days reported.

Luckily, as we can see below, we have *at least* one weather station that reported an average temperature for each days of the dataset. What I will do is calculate using extrapolation the *TAVG* values that are missing for some stations, using the value of the nearest one for the same day. I'll do this in the next notebook: NYC Taxi Travel Dataset Feature Engineering

[35]:
```
# Display the days of the dataset with the lowest number of weather station␣
 ↪having an average temperature measure
```

```
# If the first value of the first days is > 0 => we have measures for all the␣
 ↪days
nyc_weather[['DATE','TAVG']].groupby('DATE').count().sort_values('TAVG',␣
 ↪ascending=True).head(3) #['TAVG'].isnull().value_counts()
```

[35]:

DATE	TAVG
2016-06-30	12
2016-05-05	12
2016-05-20	12

About temperature, there's a final *TOBS* column wich contains the temperature at the time of observation. The number of values in this column is very low (857 entries), I'd prefer to drop this column, along with *TMIN* and *TMAX* now that they are useless.

[36]:
```
nyc_weather.drop(['TOBS', 'TMIN', 'TMAX'], axis=1, inplace=True)
```

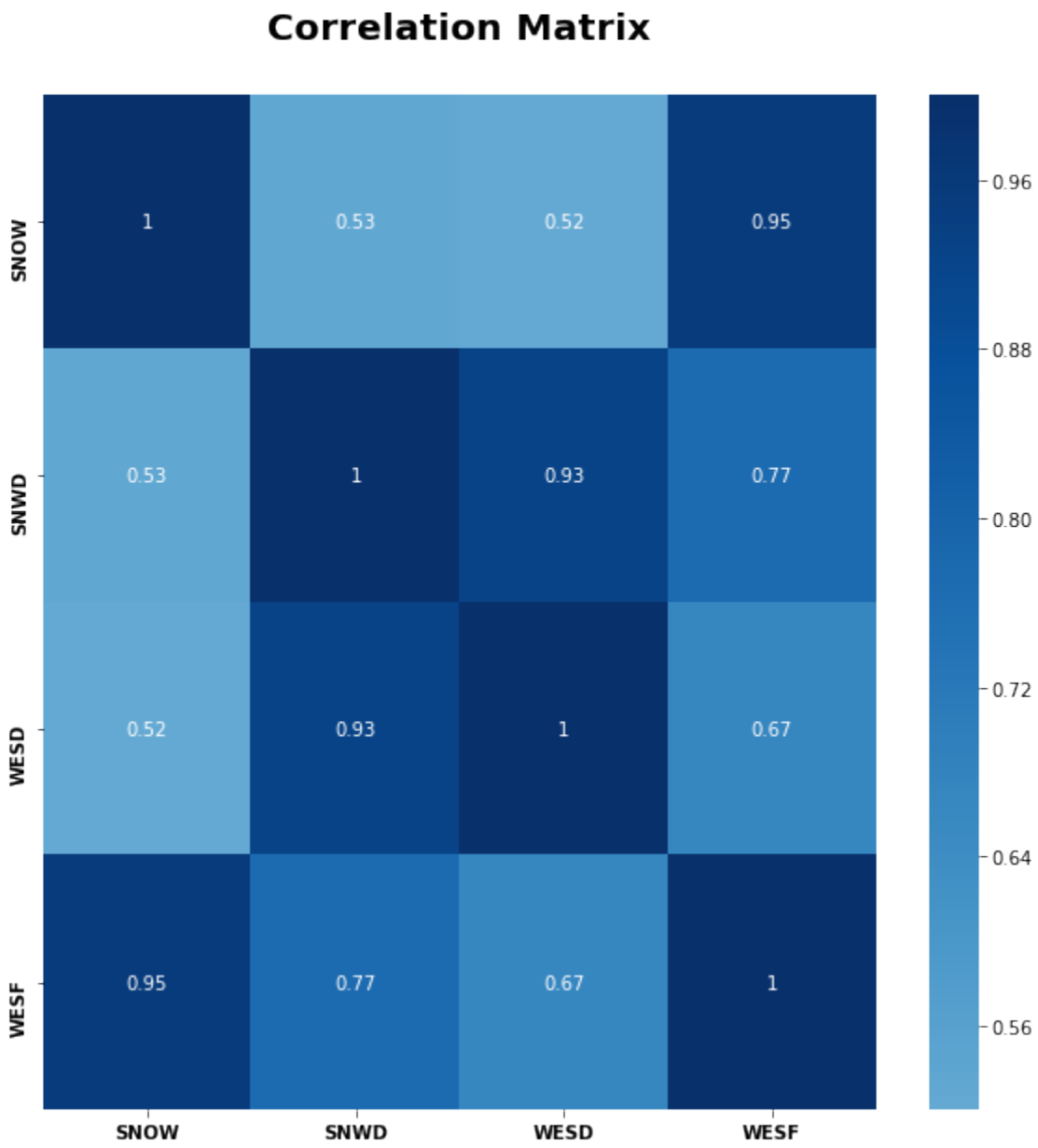
5.5 SNOW, SNWD, WESD and WESF columns

4 columns in the dataset concern the snow:

- SNOW - Snowfall
- SNWD - Snow depth
- WESD - Water equivalent of snow on the ground
- WESF - Water equivalent of snowfall

Let's display again the correlation matrix between those columns:

[37]:
```
# Build numercial column name list
snow_columns=['SNOW', 'SNWD', 'WESD', 'WESF']

# Draw correlation matrix
draw_correlation_matrix(nyc_weather[snow_columns])
```



Not surprisingly, there is a strong correlation between *WESD* (Water Equivalent of Snow on the ground) and *SNWD* (Snow Depth on the road), as well as between *SNOW* (Total snowfall) and *WESF* (Water Equivalent of Snow Fall).

Furthermore, if I check the NaN values in those four columns, I would find that *SNOW* and *SNWD* contains a lot more of information than *WESF* and *WESD*.

[38]:
```
nyc_weather[snow_columns].describe()
```

[38]:

	SNOW	SNWD	WESD	WESF
count	6951.000000	3067.000000	346.000000	455.000000
mean	4.576320	26.544180	7.737283	2.833626
std	37.602428	90.830324	18.801285	8.704819
min	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	1.500000	0.800000
max	770.000000	780.000000	126.700000	74.900000

For that reason, I would prefer to ignore *WESD* and *WESF* columns and drop them from my dataset

[39]:
```
# Remove WESD and WESF columns
nyc_weather.drop(['WESD', 'WESF'], axis=1, inplace=True)
```

Ok, now I will use the *SNOW* and *SNWD* columns to build two new categorical features:

- SNOW_FALL:

Values will be set to 1 if *SNOW* is not NaN or > 0

This will indicate if the day is a snowing day.

- SNOW_ROAD:

Values will be 1 if *SNWD* is not NaN or > 0

This will indicate if there is snow on the road

[40]:
```
def check_na_or_null(value) -> int:
    """
    Check if value passed as parameter is greater than 0

    return 1 if True, 0 otherwise


    Returns:
    --------
    int

    """
    if value > 0:
        return 1
    return 0

# apply check_na_or_null function on SNOW to create new SNOW_FALL categorical␣
 ↪feature
nyc_weather['SNOW_FALL']=nyc_weather['SNOW'].fillna(0).apply(check_na_or_null)
```

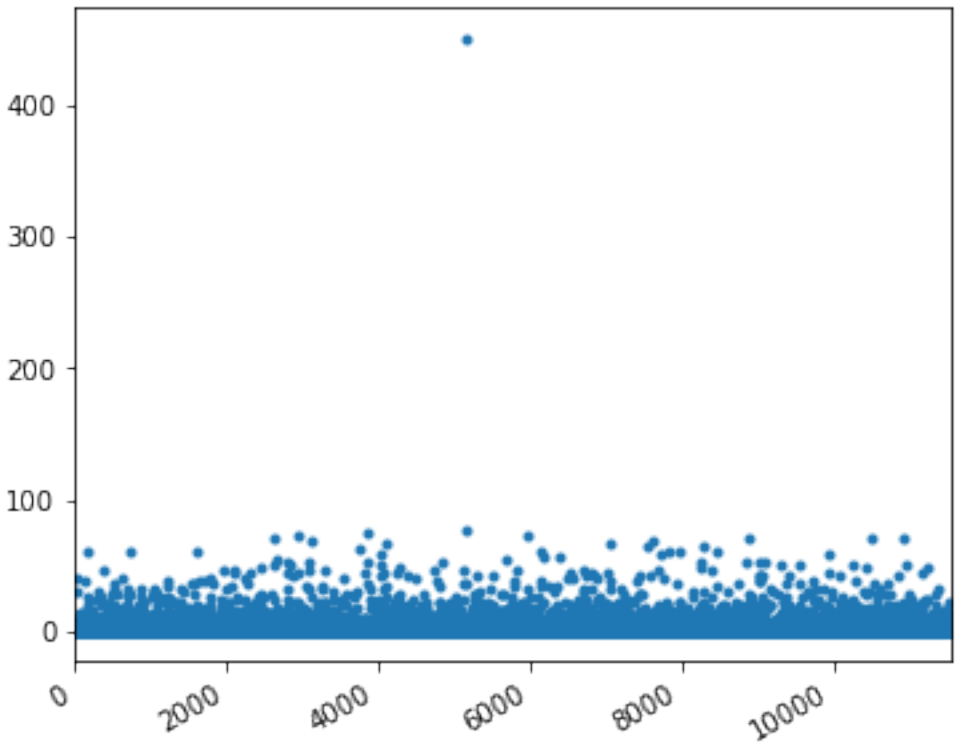
```
# apply check_na_or_null function on SNWD to create new SNOW_ROAD categorical␣
 ↪feature
nyc_weather['SNOW_ROAD']=nyc_weather['SNWD'].fillna(0).apply(check_na_or_null)
```

5.6 PRCP column

And finally, the PRCP feature. This one contains the quantity of precipitation that falls on the weather station that made the measure.

Plotting the values will show up that there is a value that must be considered as an outlier, more than that a mistake in the mesure done (more that 4 meters of precipitation detected in a day by one of the weather station, that sounds wrong.

[41]:
```
# Plot PRCP features
nyc_weather['PRCP'].plot(lw=0, marker='.', subplots=True, figsize=(20,5),␣
 ↪layout=(-1,3))
plt.show()
```



Let's have a look at this upper point.

[42]:
```
# Get line where PRCP > 400
nyc_weather[nyc_weather.PRCP > 400]
```

[42]:
```
          STATION                      NAME  LATITUDE  LONGITUDE  ELEVATION
DATE  AWND  \
5148  USW00094741  TETERBORO AIRPORT, NJ US     40.85  -74.06139        2.7
2016-02-14   5.6

       PRCP  SNOW  SNWD  TAVG  WT01  WT02  WT03  WT04  WT06  WT08  WT09  WT11
WDIR_E  WDIR_N  \
5148  451.1   NaN   NaN -13.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0       0

      WDIR_NE  WDIR_NW  WDIR_S  WDIR_SE  WDIR_SW  WDIR_W  PEAK_Y  TSTD
SNOW_FALL  SNOW_ROAD
5148        0        1       0        0        0       0       1   5.0
0          0
```

Ok, as I can see, we have one weather station concerned by this erroneous value, on *DATE = 2016-02-14*

Looking at the other weather stations for the same day, I found that the *PRCP* value is equal to 0.

Best thing to do here is to replace this 451.1 mm of precipitation on day 2019-02-14 by 0 mm:

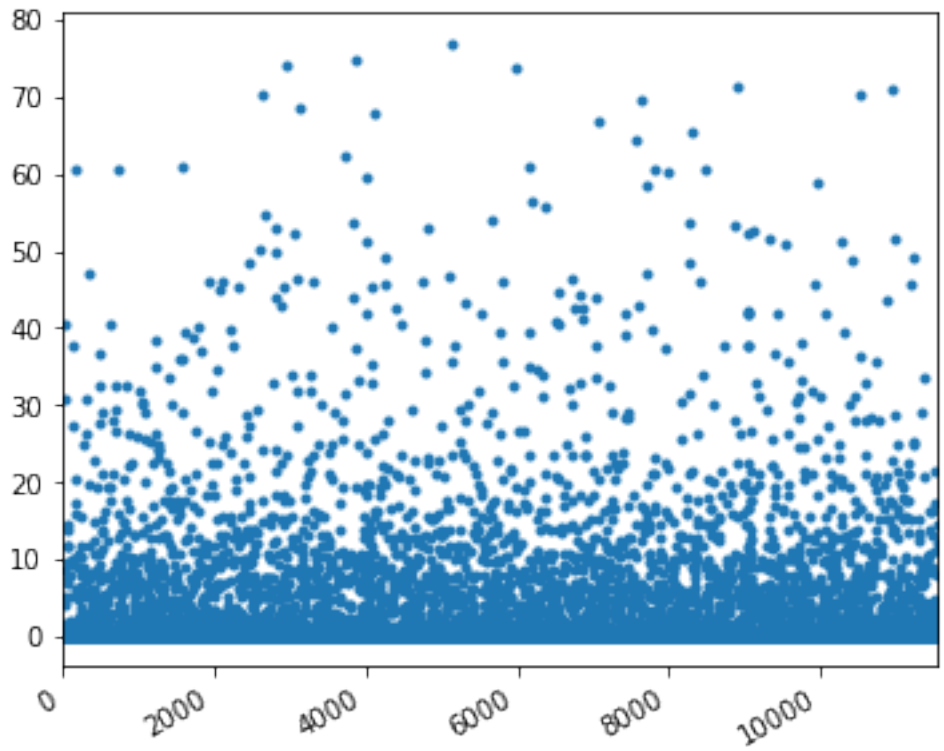
[43]:
```
# Get PRCP measures for the '2016-02-14', groupe by DATE and sum the PRCP values
nyc_weather[nyc_weather.DATE == '2016-02-14'][['DATE', 'PRCP']].groupby('DATE').
 ↪sum()
```

[43]:
```
             PRCP
DATE
2016-02-14  451.1
```

[44]:
```
# sum of PRCP values equal the value of 451.1 => all other weather stations␣
 ↪reported 0
# replace 451.1 value by 0
# row index is avalilable a few cell up ;-)
nyc_weather.at[5148, 'PRCP']=0
```

Plotting again the *PRCP* features shows a more normal situation.

[45]:
```
# Plot PRCP features
nyc_weather['PRCP'].plot(lw=0, marker='.', subplots=True, figsize=(20,5),␣
 ↪layout=(-1,3))
plt.show()
```



6 Save resulting datasets to SQL Database

Ok, all done for this first data cleaning process. As explained in the Data Preparation Introduction notebook, I will build two dataset from the one cleaned previously and store them into our database: *Weather Categorical* and *Weather Numerical*

6.1 Weather Categorical Dataset

This dataset will contains all the *categorical* columns, which are:

- WT_*
- WDIR_*
- PEAK_Y
- SNOW_FALL
- SNOW_ROAD

After creating a filter based on regex and named column, I will save them into the SQL Database.

> Note 1: I will drop the weather station static data (*NAME*, *LATITUDE*, *LONGITUDE*, *ELEVATION*) except the *STATION* id, as I do not want to save those values in the two datasets (they've already been included in *stations* dataset in the The 83 Weather Stations notebook).

Note 2: The *WT* features are encoded as *float64*, which is not right. I will re-encode them as *int64*

[46]:
```
# WT* categorical column encoded as float64, recode them as int
nyc_weather = nyc_weather.astype({
    "WT01": int,
    "WT02": int,
    "WT03": int,
    "WT04": int,
    "WT06": int,
    "WT08": int,
    "WT09": int,
    "WT11": int
})

# Check WT columns except STATION and DATE are int64 dtype
nyc_weather.filter(regex='WT').info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 11487 entries, 0 to 11542
Data columns (total 8 columns):
WT01    11487 non-null int64
WT02    11487 non-null int64
WT03    11487 non-null int64
WT04    11487 non-null int64
WT06    11487 non-null int64
WT08    11487 non-null int64
WT09    11487 non-null int64
WT11    11487 non-null int64
dtypes: int64(8)
memory usage: 807.7 KB
```

[47]:
```
# Create categorical filter
cat_columns=list(nyc_weather.filter(regex='WT.*|WDIR.*|PEAK_Y|SNOW_.*').columns)

# Save to SQL Database
save_sql(nyc_weather[['STATION', 'DATE'] + cat_columns], tablename='weather_cat')
```

```
Saving OK
```

[47]:
```
True
```

[48]:
```
# Check what has been stored into database
load_sql('weather_cat').head(2)
```

```
Query: SELECT * FROM weather_cat
```

[48]:
```
       STATION        DATE  WT01  WT02  WT03  WT04  WT06  WT08  WT09  WT11
 WDIR_E  WDIR_N  \
```

```
0  US1NYWC0003  2016-01-01     0     0     0     0     0     0     0     0
0       0
1  US1NYWC0003  2016-01-02     0     0     0     0     0     0     0     0
0       0

   WDIR_NE  WDIR_NW  WDIR_S  WDIR_SE  WDIR_SW  WDIR_W  PEAK_Y  SNOW_FALL
SNOW_ROAD
0        0        0       0        0        0       0       0          0
0
1        0        0       0        0        0       0       0          0
0
```

Ok, good, here we are with a second dataset of categorical values coming from the NYC Weather Stations Dataset :-)

6.2 Weather Numerical Dataset

This dataset will contains all the *numerical* columns, which are all the column except the weather station static and categorical columns.

To get them, I'll use the *.loc[]* method using an exclusion approach (inspired by code found on this web document: https://www.statology.org/pandas-exclude-column/).

Note: I'll keep the *STATION* id and the *DATE* feature in this dataset.

[49]:
```
# Save numerical features dataset into SQL Database
save_sql(nyc_weather.loc[:, ~nyc_weather.columns.isin(cat_columns)].
 ↪drop(['NAME','LATITUDE','LONGITUDE','ELEVATION'], axis=1),␣
 ↪tablename='weather_num')
```

```
Saving OK
```

[49]:
```
True
```

[50]:
```
# Check what has been stored into database
load_sql('weather_num').head(2)
```

```
Query: SELECT * FROM weather_num
```

[50]:

	STATION	DATE	AWND	PRCP	SNOW	SNWD	TAVG	TSTD
0	US1NYWC0003	2016-01-01	0.0	0.0	0.0	NaN	NaN	NaN
1	US1NYWC0003	2016-01-02	0.0	0.0	0.0	NaN	NaN	NaN

7 Well done !!

Here I am :-)

As expected, the data cleaning process for this NYC Weather Stations dataset is far more complex than the NYC Taxi Travel one.

It's time now to make some *Advanced Feature Engineering* on the 4 new datasets we've created: - travel - stations - weather_cat - weather_num

Let's continue with the next notebook: NYC Taxi Travel Dataset Feature Engineering

14.NYC Taxi Travel Dataset Feature Engineering

February 2, 2022

Ok, there's a lot of feature engineering to be done on this dataset ;-)

Based on the pickup and dropoff location: What's the distance between those two location? What is the nearest weather station for each of these locations ? How far they are from their nearest weather stations ?

Based on pickup and dropoff date and time: What is the day of the trip ? How long does it take ? In which period of the day ? Morning ? Noon ? At night ?

But stop listing all possible features, let's start coding them :-)

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

[2]:
```
# Verify SQL tablename is defined in my_utils library
print("Table name used to save the improved dataset:", TRAVEL_TABLENAME)
```

```
Table name used to save the improved dataset: travel_improved
```

[3]:
```
# Load NYC travel dataset from SQL database
travel_df=load_sql('travel')

# Display the 3 first rows
travel_df.head(3)
```

```
Query: SELECT * FROM travel
```

[3]:

	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latitude
0	2016-03-14 17:24:55	2016-03-14 17:32:30	1	-73.982155	40.767937
1	2016-06-12 00:43:35	2016-06-12 00:54:38	1	-73.980415	40.738564
2	2016-01-19 11:35:24	2016-01-19 12:10:48	1	-73.979027	40.763939

	dropoff_longitude	dropoff_latitude	store_and_fwd_flag	trip_duration
0	-73.964630	40.765602	0	455
1	-73.999481	40.731152	0	663
2	-74.005333	40.710087	0	2124

1 Adapt data types in timestamp columns

First thing to do is redefine the dtypes of columns:

- pickup and dropoff datetime should be converted into *datetime64* pandas type

[4]:
```
# convert pickup_datetime to pandas.datetime[64] type
travel_df['pickup_datetime']=pd.to_datetime(travel_df['pickup_datetime'])

# convert dropoff_datetime to pandas.datetime[64] type
travel_df['dropoff_datetime']=pd.to_datetime(travel_df['dropoff_datetime'])

# Verify the change
travel_df[['pickup_datetime','dropoff_datetime']].dtypes
```

[4]:
```
pickup_datetime     datetime64[ns]
dropoff_datetime    datetime64[ns]
dtype: object
```

2 Date and Time features

2.1 Day of the trip ? Weekday ? Weekend ?

Based on the *pickup_datetime* feature, let's build categorical columns that will define: - If trip started on a weekday (Monday to Friday) - If trip started on week-end (Saturday or Monday)

I will used the *Series.dt.dayofweek* property that returns the day of the week with Monday=0, Sunday=6 to build a new *travel_weekday* categorical feature.

Note: the *is_weekend()* method is coded in the *my_utils* library

[5]:
```
# Apply is_weekend() function to pickup_datetime feature and store result in␣
 ↪'weekend' column
travel_df['weekend']=travel_df['pickup_datetime'].apply(is_weekend)

# Display result
travel_df[['pickup_datetime','weekend']].head(3)
```

[5]:
	pickup_datetime	weekend
0	2016-03-14 17:24:55	0
1	2016-06-12 00:43:35	1
2	2016-01-19 11:35:24	0

2.2 Create bins with the time the trip begins: morning, afternoon, evening, night

I've decided to use the *pickup_datetime* feature to create a new categorical feature wich classify the travels in 4 different categories, depending on the time of start:

Category	Trips starts between
morning	6:00 and 12:00
afternoon	12:00 and 18:00
evening	18:00 and 22:00
night	22:00 and 6:00.

I will use *apply()* and *get_dummies()* method from *pandas* to build new categorical features.

Note: The *get_time_category()* is coded in the *my_utils* library

[6]:
```
# apply get_time_category on pickup_datetime feature
travel_df['day_period']=travel_df['pickup_datetime'].apply(get_time_category)

# use get_dummies() to. build categorical features
travel_df=pd.get_dummies(travel_df, columns=['day_period'])

# Drop one of the column (useless)
travel_df.drop('day_period_night', axis=1, inplace=True)

# Display three first rows
travel_df[[
    'pickup_datetime',
    'day_period_morning',
    'day_period_afternoon',
    'day_period_evening'
]].head(3)
```

[6]:
```
      pickup_datetime  day_period_morning  day_period_afternoon
day_period_evening
0 2016-03-14 17:24:55                   0                     1
0
1 2016-06-12 00:43:35                   0                     0
0
2 2016-01-19 11:35:24                   1                     0
0
```

2.3 Create a column with day of the trip (YYYY-MM-DD)

Even if it looks useless to create a new column made of the date part of the *pickup_datetime*, I will use this column to join lines between the *travel* and the *weather_cat* datasets.

The merge between those two dataset will be done in the Global Dataset - Merging all the datasets into a big one Notebook

[7]:
```
# Apply lambda function on pickup_datetime to build pickup_date columns
travel_df['pickup_date']=travel_df['pickup_datetime'].apply(lambda x: x.
 ↪strftime('%Y-%m-%d'))

# Display result
travel_df[['pickup_datetime', 'pickup_date']].head(5)
```

[7]:

	pickup_datetime	pickup_date
0	2016-03-14 17:24:55	2016-03-14
1	2016-06-12 00:43:35	2016-06-12
2	2016-01-19 11:35:24	2016-01-19
3	2016-04-06 19:32:31	2016-04-06
4	2016-03-26 13:30:55	2016-03-26

3 Distance Features

3.0.1 Calculate the distance bewtween pickup and dropoff in kilometers

In order to build the dependent feature, the average speed of a TAXI travel, I am missing the distance between pickup and dropoff location.

To obtain it, I'll use a function that will calculate, for each travel, the distance between those two locations.

How ?

Using a function, *get_distance_in_km()*, that will calculate the distance between two locations from their latitude/longitude coordinates. Of course, the distance obtained here is the straight line distance between the two points, not the effective distance driven by the TAXI.

Note: The function *get_distance_in_km()* is coded in the my_utils library

[8]:
```
# Build distance_km feature column
travel_df['distance_in_km']=get_distance_in_km(
    travel_df['pickup_latitude'],
    travel_df['pickup_longitude'],
    travel_df['dropoff_latitude'],
    travel_df['dropoff_longitude']
)

# display result
travel_df[[
    'distance_in_km',
    'pickup_latitude',
    'pickup_longitude',
    'dropoff_latitude',
    'dropoff_longitude'
]].head(3)
```

[8]:
```
   distance_in_km  pickup_latitude  pickup_longitude  dropoff_latitude
dropoff_longitude
0        1.498521        40.767937        -73.982155         40.765602
-73.964630
1        1.805507        40.738564        -73.980415         40.731152
-73.999481
2        6.385098        40.763939        -73.979027         40.710087
-74.005333
```

To be sure that my function returns the right value, I've picked-up an example from my dataset and used Google Maps to measure the distance.

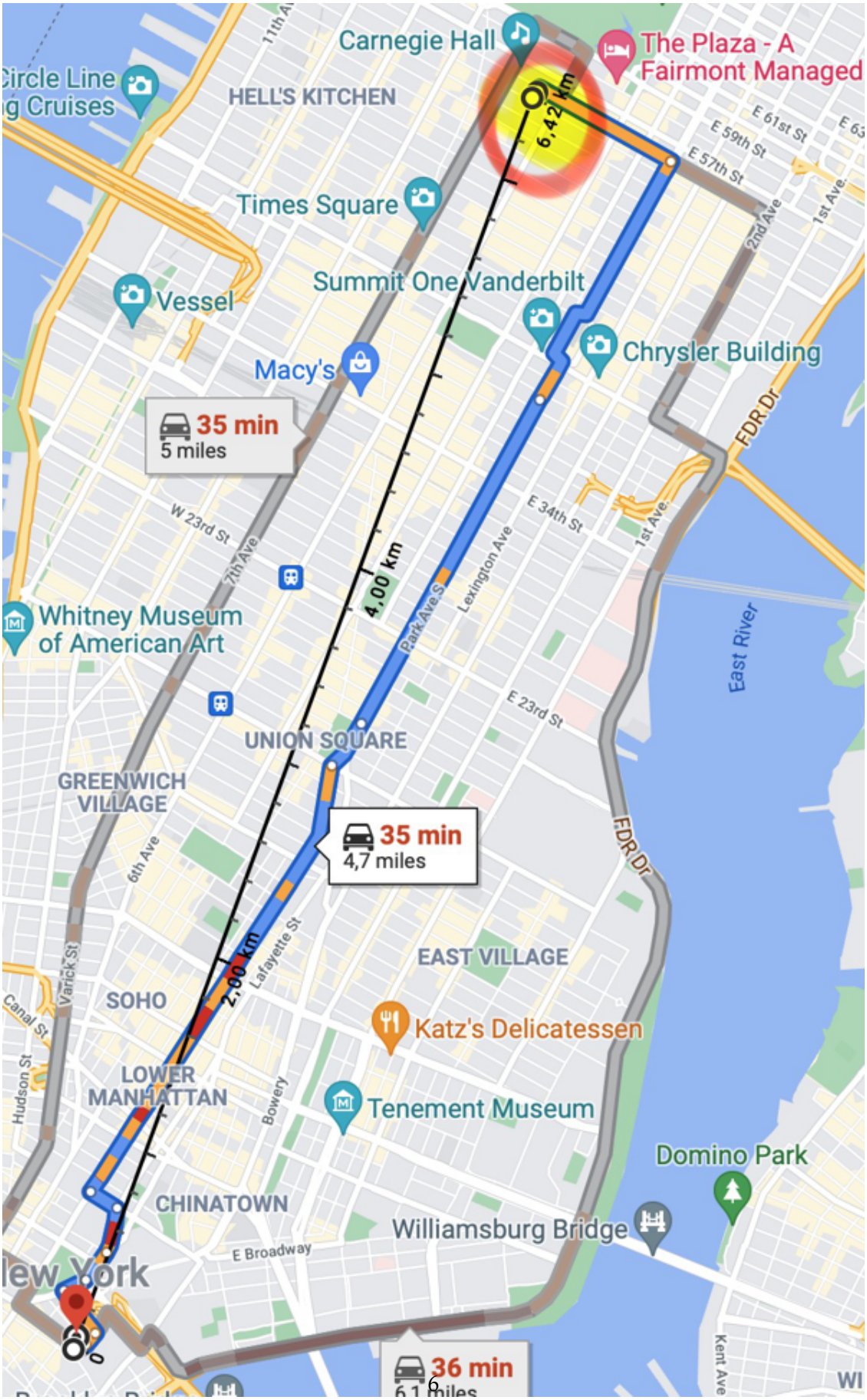
Value from my Dataset used for control: 6.39 km

pickup_latitude	40.763939
pickup_longitude	-73.979027
dropoff_latitude	40.710087
dropoff_longitude	-74.005333
distance_in_km	6.39 km

Value obtained from Google Maps: 6.42 km

[9]:
```
img=Image.open(os.path.join(IMG_PATH, 'distance-between-two-locations.jpg'))
img.thumbnail([600,1000]);img
```

[9]:


Carnegie Hall
The Plaza - A Fairmont Managed
Circle Line Cruises
HELL'S KITCHEN
6,42 km
E 61st St
E 59th St
E 57th St
Times Square
2nd Ave
1st Ave
Summit One Vanderbilt
Vessel
Chrysler Building
Macy's
FDR Dr
35 min
5 miles
W 23rd St
7th Ave
4,00 km
E 34th St
1st Ave
Park Ave S
Lexington Ave
Whitney Museum of American Art
East River
E 23rd St
UNION SQUARE
GREENWICH VILLAGE
35 min
4,7 miles
6th Ave
FDR Dr
2,90 km
Lafayette St
EAST VILLAGE
Varick St
SOHO
Canal St
Katz's Delicatessen
Hudson St
LOWER MANHATTAN
Bowery
Tenement Museum
Domino Park
CHINATOWN
Williamsburg Bridge
E Broadway
New York
36 min
Kent Ave

Google Maps link where the measure has been done: From (40.763939,-73.979027) to (40.710087,-74.005333)

3.1 Calculate the average speed to go from pickup to dropoff location

Now, having the distance between pickup and dropoff, it's possible to build my dependent feature: the TAXI Travel Speed between pickup and dropoff location.

Simply divide the *distance_in_km* by the *trip_duration* and multiply the result by *3'600* to obtain a speed value in kilometers per hour.

> Note: Remember, the *distance_in_km* feature is the *straight line* distance between pickup and dropoff location, speed calculated here is not the real TAXI speed.

[10]:
```
# Build travel speed feature column in km per hour (multiply by 3600 seconds)
travel_df['km_per_hour']=travel_df['distance_in_km']/
 ↪travel_df['trip_duration']*3600
travel_df.head(5)

# display result
travel_df[['km_per_hour', 'distance_in_km', 'trip_duration']].head(3)
```

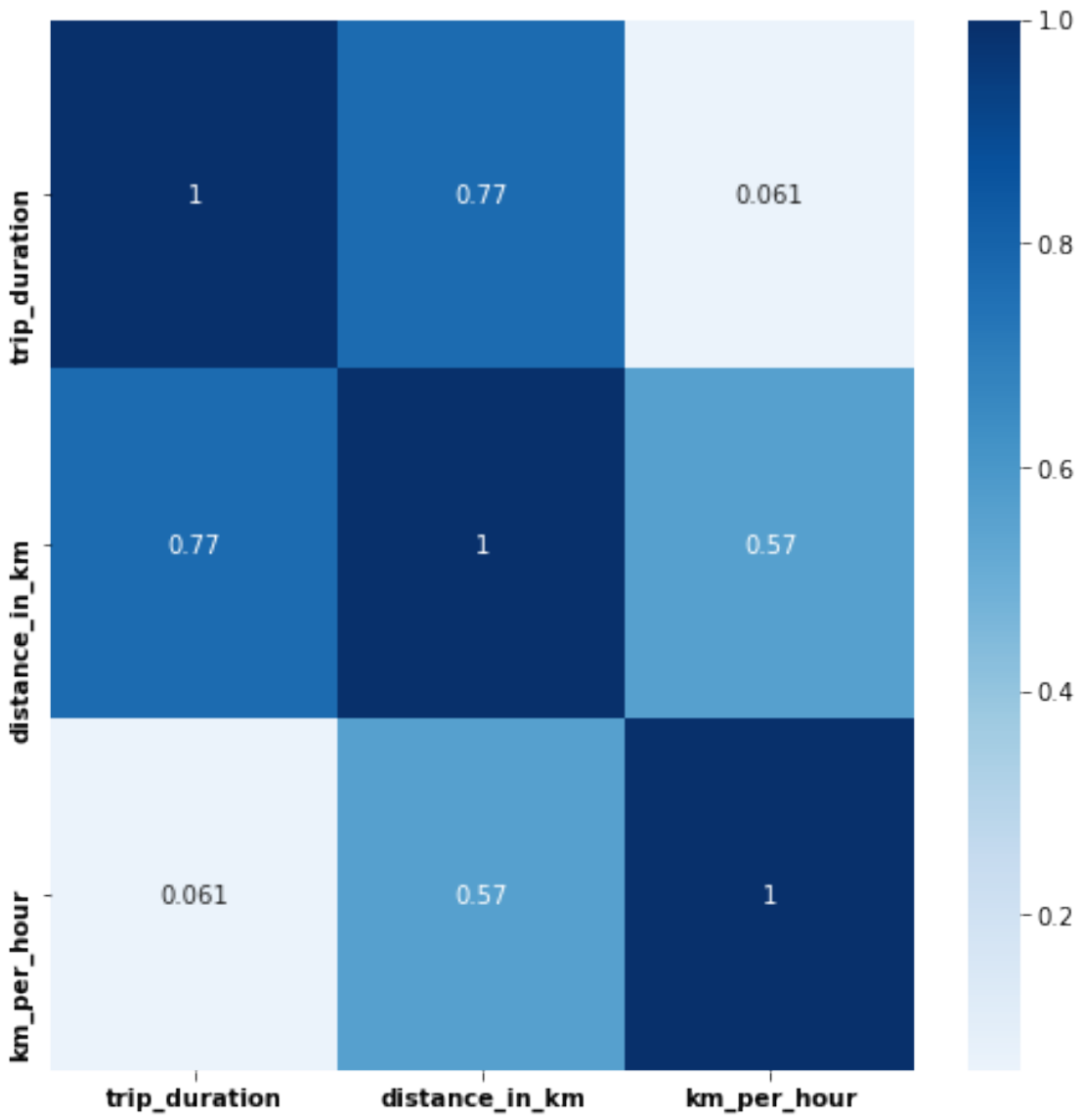
[10]:

	km_per_hour	distance_in_km	trip_duration
0	11.856428	1.498521	455
1	9.803659	1.805507	663
2	10.822201	6.385098	2124

As expected, the result here shows a quite strong correlation between *trip_duration* and *distance_in_km*

Let's drop the *trip_duration* feature.

[11]:
```
# Show correlation matrix between trip_duration, distance_in_km and km_per_hour
draw_correlation_matrix(travel_df[['trip_duration', 'distance_in_km',␣
 ↪'km_per_hour']], figsize=(8,8))
```



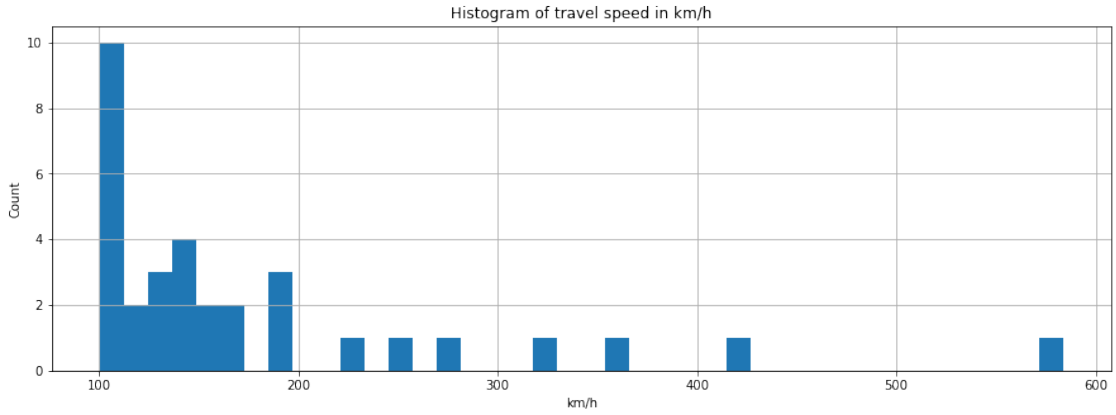
[12]:
```
# Drop trip_duration column
travel_df.drop('trip_duration', axis=1, inplace=True)
```

3.2 Travel speed outliers

Now that my dependent variable has been defined, looking at its values shows some outliers that I was not able to detect before without calculating distance and speed.

Here is an histogram of speed valuoes that are above 100 kilometers, at bit high in my opinion in NYC ;-)

[13]:
```
# Show histogram of speed value higher than 100 km/h
plt.figure(figsize=(15,5))
travel_df[travel_df.km_per_hour > 100]['km_per_hour'].hist(bins=40)
plt.title("Histogram of travel speed in km/h")
plt.xlabel("km/h")
plt.ylabel("Count")
plt.show()
travel_df[travel_df.km_per_hour > 500][['pickup_datetime', 'dropoff_datetime',␣
 ↪'distance_in_km', 'km_per_hour']]
```



[13]:

	pickup_datetime	dropoff_datetime	distance_in_km	km_per_hour
1049962	2016-03-20 11:47:20	2016-03-20 11:49:21	19.619959	583.734313

Wow !!! 583 km/h in New-York :-)

On the opposite side, I've found some trips with 0 km, which implies trip speed of 0 km/h.

I should drop those abnormal values.

Question: What is an *abnormal speed* ?

Well, there's no really good answer, I have to choose some arbitrary values.

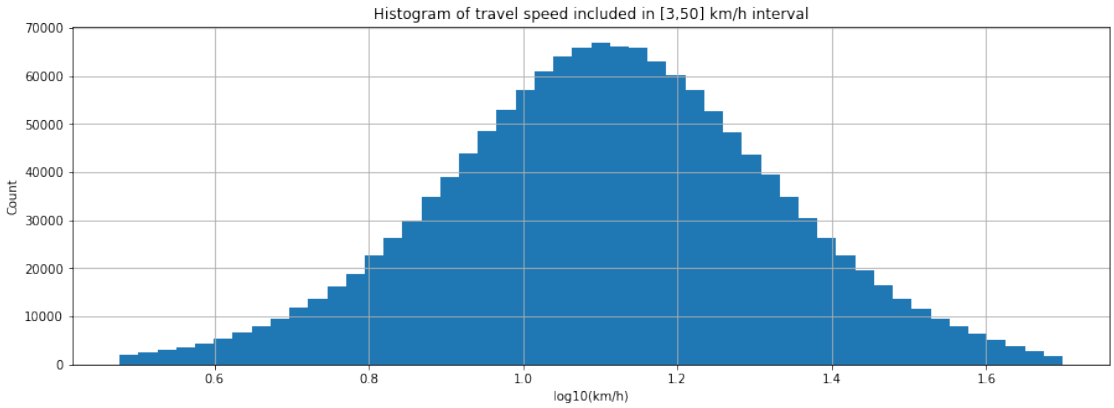
To help me in my choices, I will use an histogram of the *np.log10()* values, removing *abnormal one*, and search the *min* and *max* values that gives an histigram with the most perfect Gauss curve.

Doing this exercise, I've found that good values *could* be travel speed between 3 and 50 km/h

Lines in dataset where travel speed is outside this speed limit will be dropped.

[14]:
```
SPEED_LIMIT_LOW=3
SPEED_LIMIT_HIGH=50

filter_travel_speed=np.logical_and(
    travel_df.km_per_hour >= SPEED_LIMIT_LOW,
    travel_df.km_per_hour <= SPEED_LIMIT_HIGH
```

```
)
plt.figure(figsize=(15,5))
np.log10(travel_df[filter_travel_speed]['km_per_hour']).hist(bins=50)
plt.title("Histogram of travel speed included in [3,50] km/h interval")
plt.xlabel("log10(km/h)")
plt.ylabel("Count")
plt.show()
```



How many lines will be dropped using this Travel speed limit ?

[15]:
```
print("Number of lines removed using the Travel speed interval [{},{}]: {}".
 ↪format(
    SPEED_LIMIT_LOW,
    SPEED_LIMIT_HIGH,
    travel_df[~filter_travel_speed]['km_per_hour'].count()
))
```

```
Number of lines removed using the Travel speed interval [3,50]: 21228
```

Number of lines removed is acceptable, I decide to keep this speed interval and drop trips that are outside of it.

> Note: Later in the Global Dataset - Merging all the datasets into a big one Notebook, I will normalize the *km_per_hour* field using *np.log10()* method.

[16]:
```
# Remove outliers
travel_df=travel_df[filter_travel_speed]
```

4 Passenger numbers, from numerical to categorical

Looking at the passenger numbers distribution in TAXI Travel, I've decided to transform this numerical value into a categorical one:

- Is there more than 1 passenger in the trip ?

This is simply done using *apply()* method on the *passenger_count* feature with a lambda function that returns 1 if passenger numbers is higher than 1:

[17]:
```
# Transform passenger to a categorical one
travel_df['passenger_alone']=travel_df['passenger_count'].apply(lambda x: 0 if␣
 ↪x>1 else 1)

# Drop passenger_count feature
travel_df.drop('passenger_count', axis=1, inplace=True)
```

5 Travel and weather station static data join

The goal here is to map all the pickup and dropoff location of the *travel* dataset with their nearest weather station.

When this is done, I will calculate, again for pickup and dropoff, the distance in km from the nearest weather station found.

And finally, with travel and weather station mapped together, I'll be able to create new features:

- The elevation difference between pickup and dropoff (numerical feature)
- The elevation from pickup to dropoff is ascending or descending (categorical feature)

5.1 Map pickup and dropoff point to their nearest weather station

Ok, something really fun to do: Find the nearest weather station of the pickup and dropoff location.

To do so, I've created a function *get_nearest_station_from_location* in my_utils library that seeks in the 83 weather stations dataset which station is the nearest from a location passed as parameter (see my_utils for more informations on that function implementation)

Then, I've used this function in combination with the *DataFrame.apply()* method to deetermine, for each rows in my *travel* dataset, which are the nearest weather stations for each pickup and dropoff locations.

WARNING !! This process results in heavy computing: 1'500'000 cells have to be processed, each of them requiring 83 calls to a *pythagore()* function which includes heavy calculus. It tooks about 15 minutes on my Apple Mac M1 computrer. So be patient ;-)

After processing each slices, the result will be appended in a *temporary* table of the database. This table will be reloaaded at the end of the process to be saved back and replace values into the *TRAVEL_TABLENAME* table.

> Note 1: As the quantity of data to be manipulated is quite high, in order to avoid any limit problems, I've made the following process working on slices of the dataset, splitting it by slices of 100'000 lines. I've coded a function named *get_slice_list()* that returns a list of tuples of slice indexes. This function *get_slice_list()* is detailed in the my_utils library
>
> Note 2: The name of the features created by joining *travel* and *stations* data, pickup_STATION and dropoff_STATION, uses lowercase and uppercase. Lowercase

indicates features coming from *travel* dataset, uppercase from *weather* one. Now you understand the reason why I kept lowercase and uppercase in the feature names of each dataset, a sort of visual mnemonic to quickly identify from where the feature comes from. This will be more usefull when I'll build the full joined dataset in The global Dataset - Merging all the datasets into a big one

[18]:
```
# Print list of interval tuples for the travel_df dataset using get_slice_list()␣
 ↪from my_utils
print("List of slices calculated for the travel_df dataset:")
print(get_slice_list(travel_df, slice_interval=250000))
```

```
List of slices calculated for the travel_df dataset:
[(0, 249999), (250000, 499999), (500000, 749999), (750000, 999999), (1000000,
1249999), (1250000, 1426415)]
```

[19]:
```
# measure processing time (for fun)
import time

# Drop temporary database
drop_table(tablename='temporary')

# start global timer
global_start=time.time()

print("== Start processing dataset ==")
# Work on slices of dataset to avoid memory limitations
for interval in get_slice_list(dataset=travel_df, slice_interval=250000):

    print("Working on slice", interval)
    # Get dataset slice
    df=travel_df[interval[0]:interval[1]+1].copy()
    print(  "Number of lines to process: ", len(df.index))


    # Search nearset weather stations for pickup and dropoff
    for i in ['pickup', 'dropoff']:
        # start timer
        start = time.time()

        # Start processing
        print(f" Processing {i} locations...")

        # Find nearest stations for interval
        df[f'{i}_STATION']=df[[f'{i}_latitude', f'{i}_longitude']].apply(lambda␣
 ↪x: get_nearest_station_from_location(x[f'{i}_latitude'], x[f'{i}_longitude']),␣
 ↪axis=1)

        # stop timer
```

```
        end = time.time()
        print("   => Processing time for {}: {:.2f} minutes".format(i, (end -␣
 ↪start)/60))

    # Append to SQL database table
    print(" Save to database...")
    save_sql(dataset=df, tablename='temporary', if_exists='append')

    print(f"-- Slice {interval} done --")


# Stop global timer
global_end=time.time()

# Display processing time
print("== Processing terminated. Total time: {:.2f} minutes ==".
 ↪format((global_end - global_start)/60))
```

```
'temporary' table dropped from SQL database
== Start processing dataset ==
Working on slice (0, 249999)
Number of lines to process:  250000
 Processing pickup locations...
   => Processing time for pickup: 4.01 minutes
 Processing dropoff locations...
   => Processing time for dropoff: 3.91 minutes
 Save to database...
Saving OK
-- Slice (0, 249999) done --
Working on slice (250000, 499999)
Number of lines to process:  250000
 Processing pickup locations...
   => Processing time for pickup: 4.01 minutes
 Processing dropoff locations...
   => Processing time for dropoff: 4.04 minutes
 Save to database...
Saving OK
-- Slice (250000, 499999) done --
Working on slice (500000, 749999)
Number of lines to process:  250000
 Processing pickup locations...
   => Processing time for pickup: 3.92 minutes
 Processing dropoff locations...
   => Processing time for dropoff: 3.91 minutes
 Save to database...
Saving OK
-- Slice (500000, 749999) done --
Working on slice (750000, 999999)
```

```
Number of lines to process:  250000
 Processing pickup locations...
   => Processing time for pickup: 3.90 minutes
 Processing dropoff locations...
   => Processing time for dropoff: 3.91 minutes
 Save to database...
Saving OK
-- Slice (750000, 999999) done --
Working on slice (1000000, 1249999)
Number of lines to process:  250000
 Processing pickup locations...
   => Processing time for pickup: 3.90 minutes
 Processing dropoff locations...
   => Processing time for dropoff: 3.92 minutes
 Save to database...
Saving OK
-- Slice (1000000, 1249999) done --
Working on slice (1250000, 1426415)
Number of lines to process:  176415
 Processing pickup locations...
   => Processing time for pickup: 2.77 minutes
 Processing dropoff locations...
   => Processing time for dropoff: 2.75 minutes
 Save to database...
Saving OK
-- Slice (1250000, 1426415) done --
== Processing terminated. Total time: 45.13 minutes ==
```

5.1.1 Save result in database

Ok, now that the complete result is stored in the SQL *temporary* table, reload it, replace the *travel_df* variable, and save it back to the SQL databse.

This new version of the *travel_df* dataset is the one that I'll use in the rest of this Notebook.

[20]:
```
# Reload travel_df from 'temporary' table
travel_df=load_sql('temporary')

# and save it back to SQL database
save_sql(dataset=travel_df, tablename=TRAVEL_TABLENAME)
```

```
Query: SELECT * FROM temporary
Saving OK
```

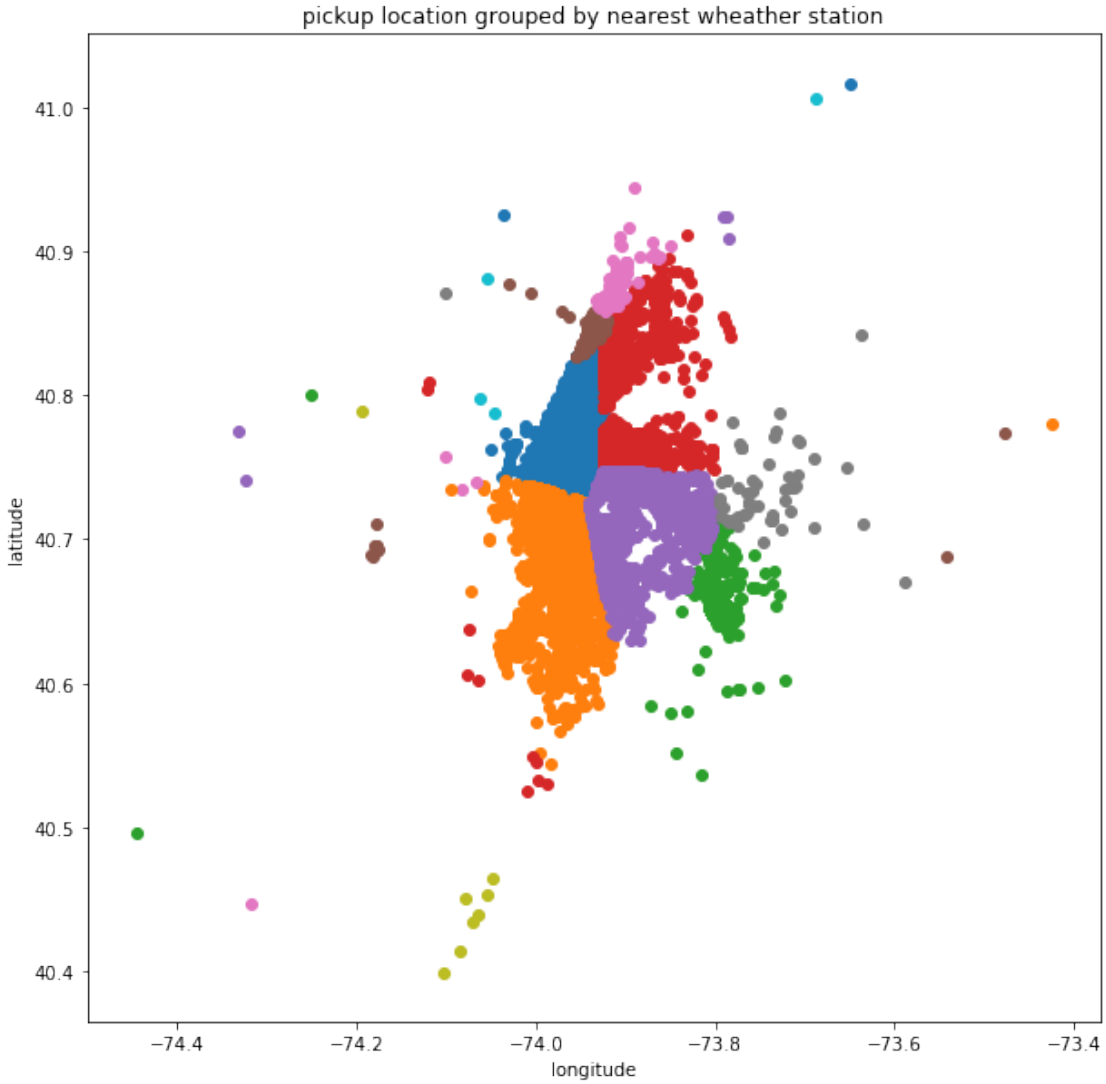
[20]:
```
True
```

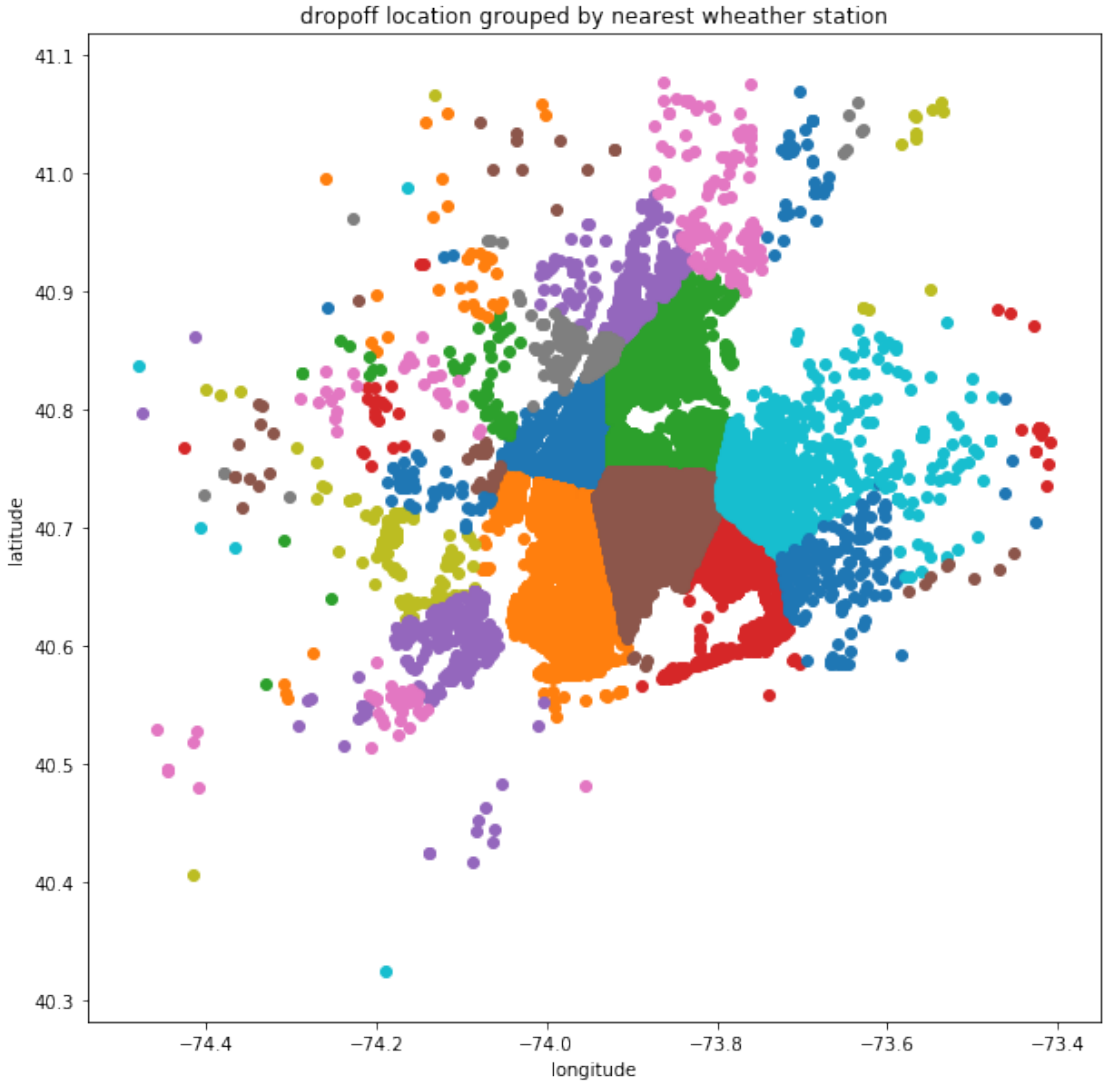
5.1.2 Visual check of the result

Let's do scatter plots of *pickup* and *dropoff* locations using different colors for each weather station locations they are associated with.

> Note: I've removed the legend as the number of stations is a bit too high to be added on the graph

[21]:
```
# Print two graphs, one for pickup, the other for dropoff
for i in ['pickup', 'dropoff']:

    # Set figure size
    plt.figure(figsize=(10,10))

    # Loop for each station found in pickup_STATION feature
    for station in list(travel_df[f'{i}_STATION'].unique()):

        # Get the True/False selection Series filtering on the statio name
        idx = (travel_df[f'{i}_STATION'] == station)

        # Scatter plot of locations concerned by the current weather station
        plt.scatter(travel_df[idx][f'{i}_longitude'],␣
↪travel_df[idx][f'{i}_latitude'], label=station)

    # Add title
    plt.title(f'{i} location grouped by nearest wheather station')

    # Set Y and Y axis labels
    plt.xlabel('longitude')
    plt.ylabel('latitude')

    # Display canvas
    plt.show()
```


pickup location grouped by nearest wheather station
41.0
40.9
40.8
40.7
40.6
40.5
40.4
latitude
−74.4
−74.2
−74.0
−73.8
−73.6
−73.4
longitude



5.1.3 Remark

Looking at the previous graphs, I can find that pickup locations are more *localized* on the center of the map, while *dropoff* one *spread* more far from that center.

5.2 Calculate the distance between pickup and dropoff location with their nearest weather station

One feature that could be interesting would be to calculate the distance between pickup and dropoff location and their nearest weather station.

I will do this using SQL query with *INNER JOIN* approach.

[22]:
```
# Build a select function with two INNER JOIN between travel and stations␣
 ↪datasets, one INNER JOIN per pickup/dropoff location
query='SELECT '
query+='T.pickup_latitude, T.pickup_longitude, pickup_STATION, P.LATITUDE as␣
 ↪pickup_STATION_LATITUDE, P.LONGITUDE AS pickup_STATION_LONGITUDE,'
query+='T.dropoff_latitude, T.dropoff_longitude, dropoff_STATION, D.LATITUDE as␣
 ↪dropoff_STATION_LATITUDE, D.LONGITUDE AS dropoff_STATION_LONGITUDE '
query+=f'FROM {TRAVEL_TABLENAME} as T '
query+='INNER JOIN stations as P ON T.pickup_STATION=P.STATION '
query+='INNER JOIN stations as D ON T.dropoff_STATION=D.STATION '

df=load_sql(query=query)

for i in ['pickup', 'dropoff']:
   ␣
 ↪travel_df[f'{i}_distance_to_STATION']=get_distance_in_km(df[f'{i}_latitude'],␣
 ↪df[f'{i}_longitude'], df[f'{i}_STATION_LATITUDE'],␣
 ↪df[f'{i}_STATION_LONGITUDE'])

travel_df[['pickup_distance_to_STATION', 'dropoff_distance_to_STATION']].head(3)
```

```
Query: SELECT T.pickup_latitude, T.pickup_longitude, pickup_STATION, P.LATITUDE
as pickup_STATION_LATITUDE, P.LONGITUDE AS
pickup_STATION_LONGITUDE,T.dropoff_latitude, T.dropoff_longitude,
dropoff_STATION, D.LATITUDE as dropoff_STATION_LATITUDE, D.LONGITUDE AS
dropoff_STATION_LONGITUDE FROM travel_improved as T INNER JOIN stations as P ON
T.pickup_STATION=P.STATION INNER JOIN stations as D ON
T.dropoff_STATION=D.STATION
```

[22]:

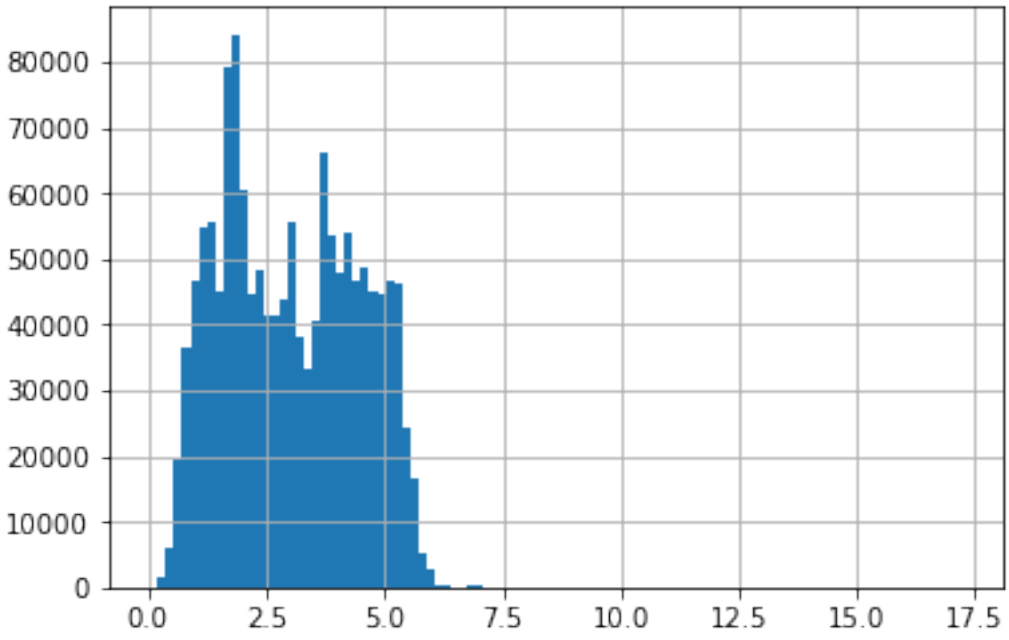
	pickup_distance_to_STATION	dropoff_distance_to_STATION
0	1.639747	1.537584
1	4.591445	5.287223
2	1.864153	3.240232

Displaying the histogram of those two new features shows that we have some outliers here, but anyway, for this feature, I decide to keep them in the dataset.

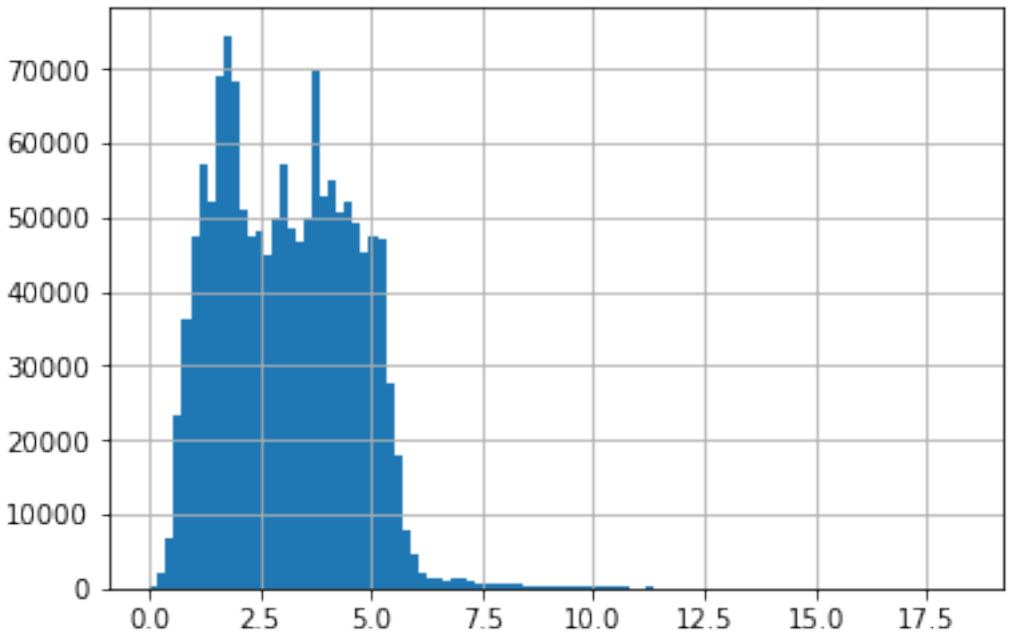
[23]:
```
# Describe feature (using 'int' dtype) and draw histogram (bins=100)
for i in ['pickup','dropoff']:
    print(f'{i} distance to STATION')
    print(travel_df[f'{i}_distance_to_STATION'].describe().astype('int'))
    travel_df[f'{i}_distance_to_STATION'].hist(bins=100)
    plt.show()
    print('------')
```

```
pickup distance to STATION
count    1426415
mean           3
std            1
```

```
min            0
25%            1
50%            2
75%            4
max           17
Name: pickup_distance_to_STATION, dtype: int64
```




```
------
dropoff distance to STATION
count    1426415
mean           3
std            1
min            0
25%            1
50%            3
75%            4
max           18
Name: dropoff_distance_to_STATION, dtype: int64
```



5.2.1 Save dataset

It's time to overwrite the table in our SQL database with this new improved *travel* dataset

[24]:
```
# Save travel_df to SQL database
save_sql(travel_df, tablename=TRAVEL_TABLENAME)
```

```
Saving OK
```

[24]: True

5.3 Elevation feature

Absolute elevation difference between pickup and dropoff, ascending or descending trip Now that pickup and dropoff locations have their nearest weather stations set, I can use the *ELEVATION* feature of the stations to calculate the elevation difference between pickup and dropoff location.

I'll store those informations in a new feature numerical feature:

- diff_ELEVATION

At the same time, I will build two categorical feature that will describe if the trip is going from pickup to dropoff ascending, *diff_ELEVATION* value is positive, or descending, *diff_ELEVATION* is a negative value.

- diff_ASCENDING: Set to 1 if the difference between the dropoff *ELEVATION* and the pickup one is absolutely positive (> 0), 0 otherwise.
- diff_DESCENDING: Set to 1 if the difference between the dropoff *ELEVATION* and the pickup one is absolutely negative (< 0), 0 otherwise.

  Note : if *diff_ASCENDING* and *diff_DESCENDING* are both equal to 0, this means that the travel starts and ends at the same *ELEVATION* value. This *zero ELEVATION* feature is useless (completly correlated with the two others) and not stored in the dataset.

[25]:
```
# Build a select function with two INNER JOIN between travel and stations datasets, one INNER JOIN per pickup/dropoff location to find elevation of the nearest STATION
# and calculate the absolute difference between pickup and dropoff elevation (based on the ELEVATION of the nearest STATION)
query='SELECT '
query+='P.ELEVATION AS pickup_ELEVATION, D.ELEVATION AS dropoff_ELEVATION '
query+=f'FROM {TRAVEL_TABLENAME} as T '
query+='INNER JOIN stations as P ON T.pickup_STATION=P.STATION '
query+='INNER JOIN stations as D ON T.dropoff_STATION=D.STATION '

# Load query as dataset
df=load_sql(query=query)

# Build diff_ELEVATION feature
travel_df['diff_ELEVATION']=np.abs(df['pickup_ELEVATION']-df['dropoff_ELEVATION'])

# build diff_ASCENDING feature
travel_df['diff_ASCENDING']=(df['dropoff_ELEVATION']-df['pickup_ELEVATION']).apply(lambda x: 1 if x > 0 else 0)

# build diff_DESCENDING feature
travel_df['diff_DESCENDING']=(df['dropoff_ELEVATION']-df['pickup_ELEVATION']).apply(lambda x: 1 if x < 0 else 0)

# Display 5 first rows of our dataset
travel_df[['diff_ELEVATION', 'diff_ASCENDING', 'diff_DESCENDING']].head(5)
```

```
Query: SELECT P.ELEVATION AS pickup_ELEVATION, D.ELEVATION AS dropoff_ELEVATION
FROM travel_improved as T INNER JOIN stations as P ON T.pickup_STATION=P.STATION
INNER JOIN stations as D ON T.dropoff_STATION=D.STATION
```

[25]:

	diff_ELEVATION	diff_ASCENDING	diff_DESCENDING
0	0.0	0	0
1	37.2	0	1
2	37.2	0	1
3	0.0	0	0

```
4              0.0                0                  0
```

6 Save result to database and...

[26]:
```
save_sql(travel_df, tablename=TRAVEL_TABLENAME)
```

```
Saving OK
```

[26]:
```
True
```

...follow up with the next Notebook: NYC Weather Categorical Dataset Feature Engineering

15.NYC Weather Categorical Dataset Feature Engineering

February 2, 2022

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

1 Group dataset by *DATE*

What kind of feature engineering could I do on the categorical features build from my weather dataset ?

Well, most of the work has been done while bulding this categorical dataset (refer to NYC Weather Data Preparation notebook for more details), but there's one last thing I've decided to do: Group all those categorical features by day.

Looking at my *weather_cat* dataset, I've found that some weather stations did not reported any metrics for some days and moreover, some stations did not reported always all of their metrics: Were thay out of order ? Stopped for maintenance ? Not equiped to measure some features others do ?

Whatever the reason, it would be nice to have a complete dataset, and for that reason, I've decided to drop the *STATION* feature and group by *DATE* the lines in the *weather_cat* dataset. Doing so, I'll get a dataset with one line per day, and categorical feature set for that day.

To perform this *grouping* operation, I will use an SQL SELECT query to group by *DATE*, and grab the max value of the other columns. As the value are either 0 or 1, taking the max will set the feature to 1 if at least one of the *STATION* for this *DATE* reported 1 for the considered feature.

1.1 What will be the result ?

Grouping by DATE, taking for each categorical feature the maximum value of each of them, and droping the *STATION* feature, I'll obtain a dataset of 182 lines, which is the number of days between the 1st of January 2016 and the 30th of June 2016.

Of course, some of the features might be multivaluated, in the sense that taking, for one particular *DATE*, the max value of each WDIR_* feature, it might result in a day with multiple wind directions.

Is that a problem ? I don't think so. Furthermore, it would be far more complicated and error prone to try to extrapolate features on missing entries.

1.2 Let's do it...

And start by verifying that the name of the SQL tablename is defined

[2]:
```
# Verify SQL tablename is defined in my_utils library
print("Table name used to save the improved dataset:", WEATHER_CAT_TABLENAME)
```

```
Table name used to save the improved dataset: weather_cat_improved
```

Load the *weather_cat* dataset built in NYC Weather Data Preparation notebook.

[3]:
```
# Load weather categorical dataset from SQL Database
df=load_sql('weather_cat')
```

```
Query: SELECT * FROM weather_cat
```

Verify that I do not have any NaN value in it

[4]:
```
# Check there is no NaN values
print("Number of NaN values in dataset: ", df.isna().sum().sum())
```

```
Number of NaN values in dataset:  0
```

Run the following query that will group line by *DATE*, dropping *STATION* column and keeping the MAX() value of the other ones:

```
SELECT
DATE,
MAX(WT01) AS WT01,
MAX(WT02) AS WT02,
MAX(WT03) AS WT03,
MAX(WT04) AS WT04,
MAX(WT06) AS WT06,
MAX(WT08) AS WT08,
MAX(WT09) AS WT09,
MAX(WT11) AS WT11,
MAX(WDIR_E) AS WDIR_E,
MAX(WDIR_N) AS WDIR_N,
MAX(WDIR_NE) AS WDIR_NE,
MAX(WDIR_NW) AS WDIR_NW,
MAX(WDIR_S) AS WDIR_S,
MAX(WDIR_SE) AS WDIR_SE,
MAX(WDIR_SW) AS WDIR_SW,
MAX(WDIR_W) AS WDIR_W,
MAX(PEAK_Y) AS PEAK_Y,
MAX(SNOW_FALL) AS SNOW_FALL,
MAX(SNOW_ROAD) AS SNOW_ROAD
FROM weather_cat
```

```
GROUP BY DATE ORDER BY DATE ASC
```

Note: Column built with the MAX() function are renamed to keep the same feature names.

[5]:
```
# Build query described above
query = 'SELECT DATE'

# Loop for each column name except STATION and DATE ([1:2])
for col in list(df.columns[2:]):
    query+=f', MAX({col}) AS {col}'

# Finalize query
query+=" FROM weather_cat WHERE DATE!='2016-07-01' GROUP BY DATE ORDER BY DATE␣
 ↪ASC"

# Run query
df=load_sql(query=query, verbose=False)

# Display some lines
df.head(3)
```

[5]:
```
         DATE  WT01  WT02  WT03  WT04  WT06  WT08  WT09  WT11  WDIR_E  WDIR_N
WDIR_NE  WDIR_NW  \
0  2016-01-01     0     0     0     0     0     0     0     0       0       0
0        1
1  2016-01-02     0     0     0     0     0     0     0     0       0       0
0        0
2  2016-01-03     0     0     0     0     0     1     0     0       0       0
0        0

   WDIR_S  WDIR_SE  WDIR_SW  WDIR_W  PEAK_Y  SNOW_FALL  SNOW_ROAD
0       0        0        0       1       1          0          0
1       0        0        0       1       1          0          0
2       0        0        1       1       1          0          0
```

Check that the number of lines and columns is correct.

- Number of line must be 182, it's the number of days between the 1st of January 2016 and the 30th of June 2016
- Number of column must be 20, 1 less that the *weather_cat* dataset as we've removed the *STATION* feature

[6]:
```
# Get number of lines:
number_of_lines=len(df.DATE)

# Get number of column:
number_of_columns=len(df.columns)
```

```
if (number_of_lines==182 and number_of_columns==20) : # Number of lines and␣
 ↪columns matches
    print("Number of lines and columns in dataset is correct: {} x {}".
 ↪format(number_of_lines, number_of_columns))
else:
    print("ERROR: Number of lines and columns in dataset is incorrect: {} x {}".
 ↪format(number_of_lines, number_of_columns))
```

```
Number of lines and columns in dataset is correct: 182 x 20
```

[7]:
```
# Save improved dataset to SQL Database
save_sql(df, tablename=WEATHER_CAT_TABLENAME)
```

```
Saving OK
```

[7]:
```
True
```

2 Go to next...

...notebook: NYC Weather Numerical Dataset Feature Engineering

16.NYC Weather Numerical Dataset Feature Engineering

February 2, 2022

As already said in the previous notebook, many of the weather stations did not reported informations for every days. Otherwise, the number of lines in the *weather* dataset would have been 15'106 (<number of stations> X <number of days>).

Furtermore, I've identified some case where, even if the weather station reported values for a particular day, some of the me4trics where empty.

What wouold be nice is to have a complete dataset, replacing missing line and metrics with *coherent* remplacement values.

What's the strategy I choose to fill missing metrics ? Extension and extrapolation !!

- **Extension**: will be the process of creating an extended dataset with the 3'619 mising lines counted above, setting feature values to NaN
- **Extrapolation**: will be the process of *extrapolating* the NaN values using non NaN values for the same day.

More details on how to do that trick in the following cells of this Notebook.

Let's go :-)

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

[2]:
```
# Load current weather_num table in Dataframe
df=load_sql('weather_num', verbose=False)
```

[3]:
```
# Get number of lines from dataset
number_of_lines=len(df.index)

# Calculate number of lines if all the STATION had reported values
expected_number_of_lines=len(get_stations().index) * len(get_days()) #␣
 ↪get_days() comes from my_utils library

# Print the difference
```

```
print("Number of missing lines in the weather dataset: ",␣
 ↪expected_number_of_lines - number_of_lines)
```

```
Number of missing lines in the weather dataset:  3619
```

[4]:
```
# Display the NaN values per feature
print("Number of NaN values per feature in dataset:")
df.isna().sum().sort_values(ascending=False)
```

```
Number of NaN values per feature in dataset:
```

[4]:
```
TSTD       9077
TAVG       9077
SNWD       8420
SNOW       4536
PRCP        200
AWND          0
DATE          0
STATION       0
dtype: int64
```

[5]:
```
# Verify SQL tablename is defined in my_utils library
print("Table name used to save the improved dataset:", WEATHER_NUM_TABLENAME)
```

```
Table name used to save the improved dataset: weather_num_improved
```

1 Extension: Build an extended dataset

The goal here is to extend the *weather_num* dataset with the 3'619 missing line, filling new line features with np.nan values. The result will be a dataset of 83 stations x 182 days = 15106 lines.

To do so, I will first create an empty DataFrame with the same columns as *weather_num*.

Then, I will use the *get_days()* function from my_utils library to create a loop on every days, and for each of them i will :

1. Select all the lines for that day from *weather_num* table of my database
2. Append the result to my empty DataFrame
3. Select *STATION* from *stations* table which **are not** in the lines selected in point (1), appended with the current day
4. Append this second query result to my empty DataFrame.

At the end of the loop, I should have a 15'106 lines dataset, one line for each *DATE* and *STATION* :-)

Here are the SQL described above for day = '2016-01-01':

```
(1) SELECT * FROM weather_num WHERE DATE='2016-01-01'

(2) SELECT STATION, '2016-01-01' AS DATE FROM stations
```

```
WHERE STATION NOT IN ('US1NYWC0003','US1NJBG0023', [...],'USW00054787')
```

[6]:
```
# Create an empty global dataset
stations_and_day_df=pd.DataFrame(columns=df.columns)

# Loop for each days from '2016-01-01' to '2016-01-01'
for date in get_days():

    # Select lines for current date loop
    query=f"SELECT * FROM weather_num WHERE DATE='{date}'"

    # Store result in temp_df
    temp_df=load_sql(query=query, verbose=False)

    # Append temp_df to global dataset
    stations_and_day_df=stations_and_day_df.append(temp_df, ignore_index=True,␣
 ↪sort=False)

    # Select STATION that are not in the temp_df dataset, appending 'date' to␣
 ↪result
    query="SELECT STATION, '{}' AS DATE FROM stations WHERE STATION NOT IN␣
 ↪('{}')".format(date,"','".join(temp_df['STATION']))

    # Run query and append result to global dataset
    stations_and_day_df=stations_and_day_df.append(load_sql(query=query,␣
 ↪verbose=False), ignore_index=True, sort=False)

# Copy result in a new dataset that will be used in the cells below:␣
 ↪weather_df_extended
weather_df_extended=stations_and_day_df.sort_values(by=['DATE', 'STATION'])
```

Let's check the number of lines in our dataset, and the number of NaN value per feature.

[7]:
```
# Display the number of lines, should be 15'106
print("Number of lines in extended dataset:", len(weather_df_extended.index))

# Display the NaN values per feature
print("Number of NaN values per feature in dataset:")
print(weather_df_extended.isna().sum().sort_values(ascending=False))
```

```
Number of lines in extended dataset: 15106
Number of NaN values per feature in dataset:
TSTD       12696
TAVG       12696
SNWD       12039
SNOW        8155
PRCP        3819
AWND        3619
```

```
DATE           0
STATION        0
dtype: int64
```

Well done, I have now a dataset with a line for each *DATE* and *STATION*.

Let's go on with the next part, extrapolation.

2 Extrapolation: Fill missing values with extrapolated one

What is *extrapolation* ?

As we are talking here about numerical values that are *continuous*, I've decided to fill in the NaN cells with the average values of the maximum three nearest weather stations of the concerned cells.

To do so, I've written a function, *get_nan_replacement_value()*, which is detailed in the next cell.

This function simply returns the replacement value for a cell using three parameters: - station: The weather station we are looking for replacement value. - date: the date of the replacement value. - feature: The name of the feature we would like to replace.

[8]:
```
def get_nan_replacement_value(station, date, feature) -> float:
    """
    Returns a replacement value for the 'feature' received as parameter using␣
 ↪the average
    of the maximum three nearest stations, for the 'station' and 'date' received␣
 ↪as parameter

    Here is the algorythm choosen:

    - Performs a SQL SELECT in weather_num where feature is not null.
    - In the previous SELECT, used the pythagore() method two identify the␣
 ↪nearest three stations
    - Form the previous result, performs a SELECT AVG(feature). This will␣
 ↪calculate the average of
      the maximum three values from the nearest weather station.
    - Use Dataframe.at() method to retrieve the average value we are looking␣
 ↪for, and return it.

    Parameters are:
    - station: The weather station we are looking for replacement value.
    - date: the date of the replacement value.
    - feature: The name of the feature we would like to replace.

    Returns:
    --------
    float
```

```
    """

    # First get location of 'station' passed as parameter
    station_df=get_stations(station_list=[station])[['LATITUDE', 'LONGITUDE']]
    station_latitude=float(station_df.LATITUDE.values[0])
    station_longitude=float(station_df.LONGITUDE.values[0])

    # Build first SQL queries (the one who gets the 3 nearest weather station␣
↪with their 'feature' values)
    query='SELECT '
    query+=f"W.STATION, S.LATITUDE, S.LONGITUDE, W.DATE, W.{feature},␣
↪'{station_latitude}', '{station_longitude}' "
    query+=f"FROM weather_num AS W "
    query+=f"INNER JOIN stations AS S ON S.STATION=W.STATION "
    query+=f"WHERE DATE='{date}' AND {feature} IS NOT NULL "
    query+=f"ORDER BY pythagore({station_longitude}, {station_latitude}, S.
↪LATITUDE, S.LONGITUDE) ASC LIMIT 3"

    # Create second SQL query, based on the previous query, to get average of␣
↪the feature
    query=f"SELECT avg({feature}) AS {feature} FROM ({query})"

    # Get the value we are looking for
    df=load_sql(query=query, verbose=False)


    return df[feature].values[0]
```

I'll then loop on each numerical features and apply the *get_nan_replacement_value()* to all cells of the feature that are empty.

[9]:
```
# Crete a copy of our extended dataset
weather_num_extrapolated=weather_df_extended.copy()

# Some logging informations
print("== Extrapolation process starting ==")

# Loop on each numerical features
for feature in weather_df_extended.columns[2:]:

    # Disaply information on current feature processed
    print("Extrapolating value for feature", feature)

    # Display number of NaN values before extrapolation
    print("  Number of NaN values in dataset:",weather_num_extrapolated[feature].
↪isna().sum())
```

```
    # Build a filter on current feature NaN values
    filter_nan=weather_num_extrapolated[feature].isna()

    # Apply get_nan_replacement_value() on filtered feature
    weather_num_extrapolated[feature]=weather_num_extrapolated.apply(lambda x:␣
 ↪get_nan_replacement_value(x['STATION'], x['DATE'], feature) if(pd.
 ↪notnull(x[feature]) == False) else x[feature], axis=1)

    # Display logging informations
    print("  Processing done")

    # Display number of NaN values after extrapolation
    print("  Number of NaN values after extrapolation:
 ↪",weather_num_extrapolated[feature].isna().sum())

# Display logging informations
print("== Extrapolation process terminated ==")
```

```
== Extrapolation process starting ==
Extrapolating value for feature AWND
  Number of NaN values in dataset: 3619
  Processing done
  Number of NaN values after extrapolation: 0
Extrapolating value for feature PRCP
  Number of NaN values in dataset: 3819
  Processing done
  Number of NaN values after extrapolation: 0
Extrapolating value for feature SNOW
  Number of NaN values in dataset: 8155
  Processing done
  Number of NaN values after extrapolation: 0
Extrapolating value for feature SNWD
  Number of NaN values in dataset: 12039
  Processing done
  Number of NaN values after extrapolation: 0
Extrapolating value for feature TAVG
  Number of NaN values in dataset: 12696
  Processing done
  Number of NaN values after extrapolation: 0
Extrapolating value for feature TSTD
  Number of NaN values in dataset: 12696
  Processing done
  Number of NaN values after extrapolation: 0
== Extrapolation process terminated ==
```

Ok, let's display the first 3 lines of the *weather_num* dataset **before** and **after** the extrapolation, as well as the number of lines and the number of NaN values in the extrapolated dataset.

[10]:
```
print("Weather_num dataset BEFORE extrapolation:")
weather_df_extended.head(3)
```

```
Weather_num dataset BEFORE extrapolation:
```

[10]:

	STATION	DATE	AWND	PRCP	SNOW	SNWD	TAVG	TSTD
77	US1CTFR0022	2016-01-01	NaN	NaN	NaN	NaN	NaN	NaN
79	US1CTFR0039	2016-01-01	NaN	NaN	NaN	NaN	NaN	NaN
38	US1NJBG0002	2016-01-01	0.0	0.3	NaN	NaN	NaN	NaN

[11]:
```
print("Weather_num dataset AFTER extrapolation:")
weather_num_extrapolated.head(3)
```

```
Weather_num dataset AFTER extrapolation:
```

[11]:

	STATION	DATE	AWND	PRCP	SNOW	SNWD	TAVG	TSTD
77	US1CTFR0022	2016-01-01	1.3	0.0	0.0	0.0	4.833333	2.233333
79	US1CTFR0039	2016-01-01	1.3	0.0	0.0	0.0	4.833333	2.233333
38	US1NJBG0002	2016-01-01	0.0	0.3	0.0	0.0	4.833333	2.233333

[12]:
```
# Display the number of lines, should be 15'106
print("Number of lines in extended dataset (must be equal to 15'106):",␣
 ↪len(weather_num_extrapolated.index))

# Display the NaN values per feature
print("Number of NaN values per feature in dataset (must be all equal to 0):")
print(weather_num_extrapolated.isna().sum().sort_values(ascending=False))
```

```
Number of lines in extended dataset (must be equal to 15'106): 15106
Number of NaN values per feature in dataset (must be all equal to 0):
TSTD       0
TAVG       0
SNWD       0
SNOW       0
PRCP       0
AWND       0
DATE       0
STATION    0
dtype: int64
```

Nice, NaN values have been replaced by extrapolated ones.

And trust me, the *get_nan_replacement_value()* function works the way I wanted to ;-)

3 Save extrapolated dataset to SQL database

It's now time to save the result as a new dataset in our SQL database.

[13]:
```
# Save extrapolated dataset to SQL database
save_sql(dataset=weather_num_extrapolated, tablename=WEATHER_NUM_TABLENAME)
```

```
Saving OK
```

[13]:
```
True
```

Time to continue with the next notebook: The global Dataset - Merging all the datasets into a big one

17.The global Dataset - Merging all the datasets into a big one

February 2, 2022

Here we are, now cleaning and feature engineering is done on our two original datasets, it's time to build the *full* global Dataset that will be used to train models.

How ?

Using SQL *INNER JOIN* queries to merge the three *improved* dataset built in the previous notebooks and stored in the SQL database.

> Note: The table names of the three *improved* datasets are available as constant in my_utils library: - TRAVEL_TABLENAME - WEATHER_CAT_TABLENAME - WEATHER_NUM_TABLENAME

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

1 Merge all dataset

Before merging those datasets, display two of their fist rows to remember their content:

[2]:
```
print(f"{TRAVEL_TABLENAME} dataset:")
load_sql(TRAVEL_TABLENAME, limit=2)
```

```
travel_improved dataset:
Query: SELECT * FROM travel_improved LIMIT 2 OFFSET 0
```

[2]:
```
       pickup_datetime     dropoff_datetime  pickup_longitude  pickup_latitude
dropoff_longitude  \
0  2016-03-14 17:24:55  2016-03-14 17:32:30        -73.982155        40.767937
-73.964630
1  2016-06-12 00:43:35  2016-06-12 00:54:38        -73.980415        40.738564
-73.999481

   dropoff_latitude  store_and_fwd_flag  weekend  day_period_afternoon
day_period_evening  \
0         40.765602                   0        0                     1
```

```
0
1         40.731152                   0        1                     0
0

   day_period_morning pickup_date  distance_in_km  km_per_hour  passenger_alone
pickup_STATION  \
0                   0  2016-03-14        1.498521    11.856428                1
USW00094728
1                   0  2016-06-12        1.805507     9.803659                1
USW00094728

  dropoff_STATION  pickup_distance_to_STATION  dropoff_distance_to_STATION
diff_ELEVATION  \
0     USW00094728                    1.639747                     1.537584
0.0
1     US1NYKN0025                    4.591445                     5.287223
37.2

   diff_ASCENDING  diff_DESCENDING
0               0                0
1               0                1
```

[3]:
```
print(f"{WEATHER_CAT_TABLENAME} dataset:")
load_sql(WEATHER_CAT_TABLENAME, limit=2)
```

```
weather_cat_improved dataset:
Query: SELECT * FROM weather_cat_improved LIMIT 2 OFFSET 0
```

[3]:
```
         DATE  WT01  WT02  WT03  WT04  WT06  WT08  WT09  WT11  WDIR_E  WDIR_N
WDIR_NE  WDIR_NW  \
0  2016-01-01     0     0     0     0     0     0     0     0       0       0
0        1
1  2016-01-02     0     0     0     0     0     0     0     0       0       0
0        0

   WDIR_S  WDIR_SE  WDIR_SW  WDIR_W  PEAK_Y  SNOW_FALL  SNOW_ROAD
0       0        0        0       1       1          0          0
1       0        0        0       1       1          0          0
```

[4]:
```
print(f"{WEATHER_NUM_TABLENAME} dataset:")
load_sql(WEATHER_NUM_TABLENAME, limit=2)
```

```
weather_num_improved dataset:
Query: SELECT * FROM weather_num_improved LIMIT 2 OFFSET 0
```

[4]:
```
        STATION        DATE  AWND  PRCP  SNOW  SNWD      TAVG      TSTD
0  US1CTFR0022  2016-01-01   1.3   0.0   0.0   0.0  4.833333  2.233333
1  US1CTFR0039  2016-01-01   1.3   0.0   0.0   0.0  4.833333  2.233333
```

1.1 Build the JOIN SQL query

The *JOIN* query is based on three *INNER JOIN* with the main *TRAVEL_TABLENAME* dataset.

Why three joins as we have three datasets ? That should be two ? Well, the *WEATHER_NUM* dataset will be joined two times, first time to join with *pickup* informations, second time with *dropoff*.

1.1.1 *TRAVEL_TABLENAME* joined to *WEATHER_CAT_TABLENAME* using *pickup_DATE* columns

WEATHER_CAT_TABLENAME contains informations per each day, the joing process is base on its *DATE* column and the *pickup_date* column from *TRAVEL_TABLENAME*:

```
SELECT [columns selection] FROM {TRAVEL_TABLENAME} AS T
INNER JOIN {WEATHER_CAT_TABLENAME} AS WC
ON WC.DATE=T.pickup_date
```

1.1.2 *TRAVEL_TABLENAME* and *WEATHER_NUM_TABLENAME* using *pickup_STATION* and *pickup_DATE*

WEATHER_NUM_TABLENAME contains a unique row for a particular day and a particular weather station (remember, I've extrapolated missing data in this notebook). The joining process here will map *STATION* and *DATE* to *pickup_STATION* and *pickup_date* from *TRAVEL_TABLENAME*

```
SELECT [columns selection] FROM {TRAVEL_TABLENAME} AS T
INNER JOIN {WEATHER_NUM_TABLENAME} AS WNP
ON (WNP.STATION=T.pickup_STATION AND WNP.DATE=T.pickup_date)
```

1.1.3 *TRAVEL_TABLENAME* and *WEATHER_NUM_TABLENAME* using *dropoff_STATION* and *pickup_DATE*

Same logic as above, instead that we do not have values for *dropoff_date* in the *TRAVEL_TABLENAME* (choice has been made to consider only the pickup date as the date of the travel)

```
SELECT [columns selection] FROM {TRAVEL_TABLENAME} AS T
INNER JOIN {WEATHER_NUM_TABLENAME} AS WND
ON (WND.STATION=T.dropoff_STATION AND WND.DATE=T.pickup_date)
```

1.1.4 column names to be used in the *JOIN* query

Here is the schema I've used to name the columns of the resulting dataset:

- columns from *TRAVEL_TABLENAME* are taken as is (example: *pickup_datetime*)
- columns from *WEATHER_CAT_TABLENAME* are prefixed with *WC_* (example: *WC_SNOW_FALL*)
- columns from *WEATHER_NUM_TABLENAME* for *pickup JOIN* are prefixed with *WNP_* (example: *WNP_TAVG*)

- columns from *WEATHER_NUM_TABLENAME* for *dropoff JOIN* are prefixed with *WND_* (example: *WND_PRCP*)

[5]:
```
# Build travel columns
columns_to_select=[f'T.{column} AS {column}' for column in␣
 ↪load_sql(TRAVEL_TABLENAME, limit=1, verbose=False).columns]

# Append weather_cat columns
columns_to_select+=[f'WC.{column} AS WC_{column}' for column in␣
 ↪load_sql(WEATHER_CAT_TABLENAME, limit=1, verbose=False).columns]

# Append weather_num for pickup location columns
columns_to_select+=[f'WNP.{column} AS WNP_{column}' for column in␣
 ↪load_sql(WEATHER_NUM_TABLENAME, limit=1, verbose=False).columns]

# Append weather_num for dropoff columns
columns_to_select+=[f'WND.{column} AS WND_{column}' for column in␣
 ↪load_sql(WEATHER_NUM_TABLENAME, limit=1, verbose=False).columns]
```

Build the query using the column list and *INNER JOIN* directive described above:

[6]:
```
query="SELECT {} ".format(','.join(columns_to_select))
query+=f"FROM {TRAVEL_TABLENAME} AS T "
query+=f"INNER JOIN {WEATHER_CAT_TABLENAME} AS WC ON WC.DATE=T.pickup_date "
query+=f"INNER JOIN {WEATHER_NUM_TABLENAME} AS WNP ON (WNP.STATION=T.
 ↪pickup_STATION AND WNP.DATE=T.pickup_date) "
query+=f"INNER JOIN {WEATHER_NUM_TABLENAME} AS WND ON (WND.STATION=T.
 ↪dropoff_STATION AND WND.DATE=T.pickup_date) "
```

1.2 Run the query

Nothing to say here, simply pass the above query to the *load_sql()* function to obtain a *pandas.DataFrame*

[7]:
```
# Get dataset using load_sql
full_df=load_sql(query=query, verbose=False)
```

As expected, my three datasets are merged the way I want, and that's good news :-)

[8]:
```
# Display information on the resulting dataset
full_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1426415 entries, 0 to 1426414
Data columns (total 58 columns):
pickup_datetime                1426415 non-null object
dropoff_datetime               1426415 non-null object
pickup_longitude               1426415 non-null float64
pickup_latitude                1426415 non-null float64
```

```
dropoff_longitude              1426415 non-null float64
dropoff_latitude               1426415 non-null float64
store_and_fwd_flag             1426415 non-null int64
weekend                        1426415 non-null int64
day_period_afternoon           1426415 non-null int64
day_period_evening             1426415 non-null int64
day_period_morning             1426415 non-null int64
pickup_date                    1426415 non-null object
distance_in_km                 1426415 non-null float64
km_per_hour                    1426415 non-null float64
passenger_alone                1426415 non-null int64
pickup_STATION                 1426415 non-null object
dropoff_STATION                1426415 non-null object
pickup_distance_to_STATION     1426415 non-null float64
dropoff_distance_to_STATION    1426415 non-null float64
diff_ELEVATION                 1426415 non-null float64
diff_ASCENDING                 1426415 non-null int64
diff_DESCENDING                1426415 non-null int64
WC_DATE                        1426415 non-null object
WC_WT01                        1426415 non-null int64
WC_WT02                        1426415 non-null int64
WC_WT03                        1426415 non-null int64
WC_WT04                        1426415 non-null int64
WC_WT06                        1426415 non-null int64
WC_WT08                        1426415 non-null int64
WC_WT09                        1426415 non-null int64
WC_WT11                        1426415 non-null int64
WC_WDIR_E                      1426415 non-null int64
WC_WDIR_N                      1426415 non-null int64
WC_WDIR_NE                     1426415 non-null int64
WC_WDIR_NW                     1426415 non-null int64
WC_WDIR_S                      1426415 non-null int64
WC_WDIR_SE                     1426415 non-null int64
WC_WDIR_SW                     1426415 non-null int64
WC_WDIR_W                      1426415 non-null int64
WC_PEAK_Y                      1426415 non-null int64
WC_SNOW_FALL                   1426415 non-null int64
WC_SNOW_ROAD                   1426415 non-null int64
WNP_STATION                    1426415 non-null object
WNP_DATE                       1426415 non-null object
WNP_AWND                       1426415 non-null float64
WNP_PRCP                       1426415 non-null float64
WNP_SNOW                       1426415 non-null float64
WNP_SNWD                       1426415 non-null float64
WNP_TAVG                       1426415 non-null float64
WNP_TSTD                       1426415 non-null float64
WND_STATION                    1426415 non-null object
WND_DATE                       1426415 non-null object
```

```
WND_AWND                        1426415 non-null float64
WND_PRCP                        1426415 non-null float64
WND_SNOW                        1426415 non-null float64
WND_SNWD                        1426415 non-null float64
WND_TAVG                        1426415 non-null float64
WND_TSTD                        1426415 non-null float64
dtypes: float64(21), int64(27), object(10)
memory usage: 631.2+ MB
```

1.3 Drop useless columns for ML training

As I am building a dataset to be used to train ML models, I will drop columns that are useless for that process.

Quite easy to identify the columns to drop: It's the one that are non numeric: station name and dates.

[9]:
```
# Get 'object' dtype column names
columns_to_drop=full_df.head(1).select_dtypes('object').columns

# Print result
print("Non numeric columns to drop:")
for i in columns_to_drop:
    print(" -", i)
```

```
Non numeric columns to drop:
 - pickup_datetime
 - dropoff_datetime
 - pickup_date
 - pickup_STATION
 - dropoff_STATION
 - WC_DATE
 - WNP_STATION
 - WNP_DATE
 - WND_STATION
 - WND_DATE
```

Use *drop()* method with the column name list build previously to drop non numeric columns

[10]:
```
# Drop non numeric column from dataset
full_df.drop(columns_to_drop, axis=1, inplace=True)
```

Display size and some lines from the resulting dataset:

[11]:
```
# print size of the dataset
nb_lines=len(full_df.index)
nb_columns=len(full_df.columns)
print("Size of the dataset: {} lines and {} columns".format(nb_lines,␣
 ↪nb_columns))
```

```
# print first few lines
full_df.head()
```

```
Size of the dataset: 1426415 lines and 48 columns
```

[11]:

```
   pickup_longitude  pickup_latitude  dropoff_longitude  dropoff_latitude
store_and_fwd_flag  \
0        -73.982155        40.767937         -73.964630         40.765602
0
1        -73.980415        40.738564         -73.999481         40.731152
0
2        -73.979027        40.763939         -74.005333         40.710087
0
3        -74.010040        40.719971         -74.012268         40.706718
0
4        -73.973053        40.793209         -73.972923         40.782520
0

   weekend  day_period_afternoon  day_period_evening  day_period_morning
distance_in_km  \
0        0                     1                   0                   0
1.498521
1        1                     0                   0                   0
1.805507
2        0                     0                   0                   1
6.385098
3        0                     0                   1                   0
1.485498
4        1                     1                   0                   0
1.188588

   km_per_hour  passenger_alone  pickup_distance_to_STATION
dropoff_distance_to_STATION  \
0    11.856428                1                    1.639747
1.537584
1     9.803659                1                    4.591445
5.287223
2    10.822201                1                    1.864153
3.240232
3    12.465721                1                    4.397729
3.270370
4     9.836594                1                    1.614272
0.500627

   diff_ELEVATION  diff_ASCENDING  diff_DESCENDING  WC_WT01  WC_WT02  WC_WT03
WC_WT04  WC_WT06  \
0             0.0               0                0        1        0        0
```

```
0        0
1            37.2               0                1        0        0        0
0        0
2            37.2               0                1        0        0        0
0        0
3             0.0               0                0        0        0        0
0        0
4             0.0               0                0        1        0        0
0        0

   WC_WT08  WC_WT09  WC_WT11  WC_WDIR_E  WC_WDIR_N  WC_WDIR_NE  WC_WDIR_NW
WC_WDIR_S  WC_WDIR_SE  \
0        1        0        0          1          0           1           0
0           0
1        1        0        0          0          0           0           1
0           0
2        0        0        0          0          0           0           1
0           0
3        1        0        0          0          0           0           0
1           1
4        0        0        0          0          1           1           0
1           1

   WC_WDIR_SW  WC_WDIR_W  WC_PEAK_Y  WC_SNOW_FALL  WC_SNOW_ROAD  WNP_AWND
WNP_PRCP  WNP_SNOW  \
0           0          0          1             0             0       6.3
7.4       0.0
1           0          1          1             0             0       3.6
0.0       0.0
2           0          1          1             1             1       4.9
0.0       0.0
3           0          0          1             0             0       0.0
0.0       0.0
4           0          0          1             0             0       2.9
0.0       0.0

   WNP_SNWD  WNP_TAVG  WNP_TSTD  WND_AWND  WND_PRCP  WND_SNOW   WND_SNWD
WND_TAVG  WND_TSTD
0       0.0      7.50  3.100000  6.300000       7.4       0.0   0.000000
7.500000  3.100000
1       0.0     22.50  5.800000  2.233333       0.0       0.0   0.000000
22.766667  6.666667
2       0.0     -5.45  3.350000  0.000000       0.0       0.0  16.666667
-4.950000  3.150000
3       0.0      1.70  5.333333  0.000000       0.0       0.0   0.000000
1.700000  5.333333
4       0.0      8.05  4.750000  2.900000       0.0       0.0   0.000000
```

```
8.050000  4.750000
```

Here we are, a full 1'426'415 lines dataset of 48 features, cleaned and complete :-)

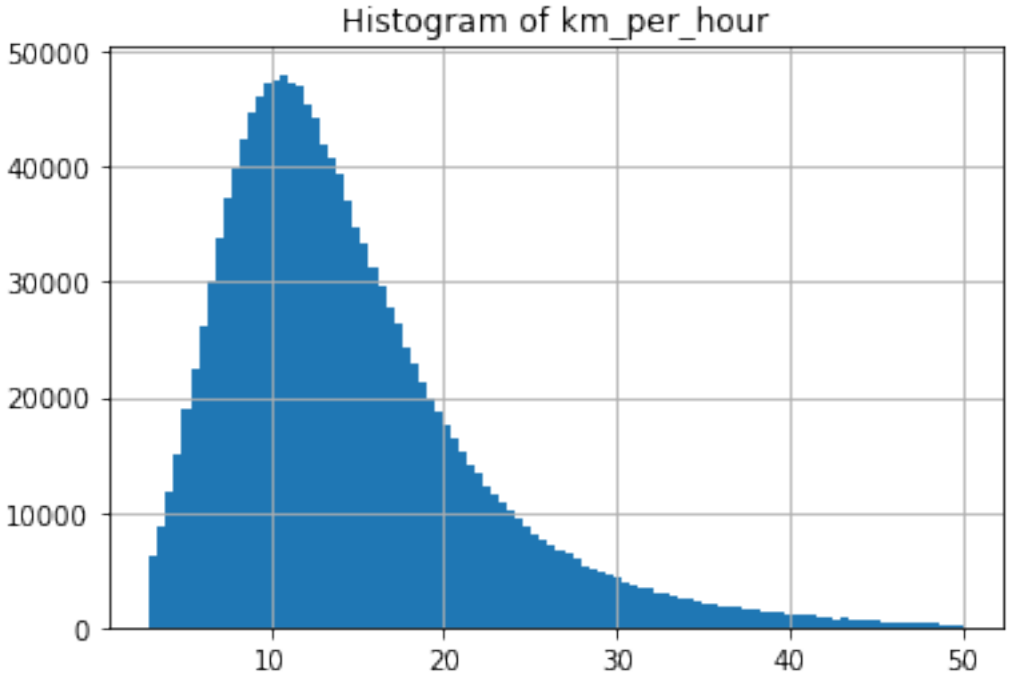
2 Normalize *km_per_hour* feature

Oh, one last thing . Remember, I've used a graphical representation of the *log10()* travel speed in this notebook to eliminate outliers. Doing so gave me a quite perfect *gauss* curve, which is far more better for ML model fitting process.

Before saving this *full* dataset, I'll transform this feature with *np.log10()* method.

Here is the actual distribution of the travel speed feature: *km_per_hour*:

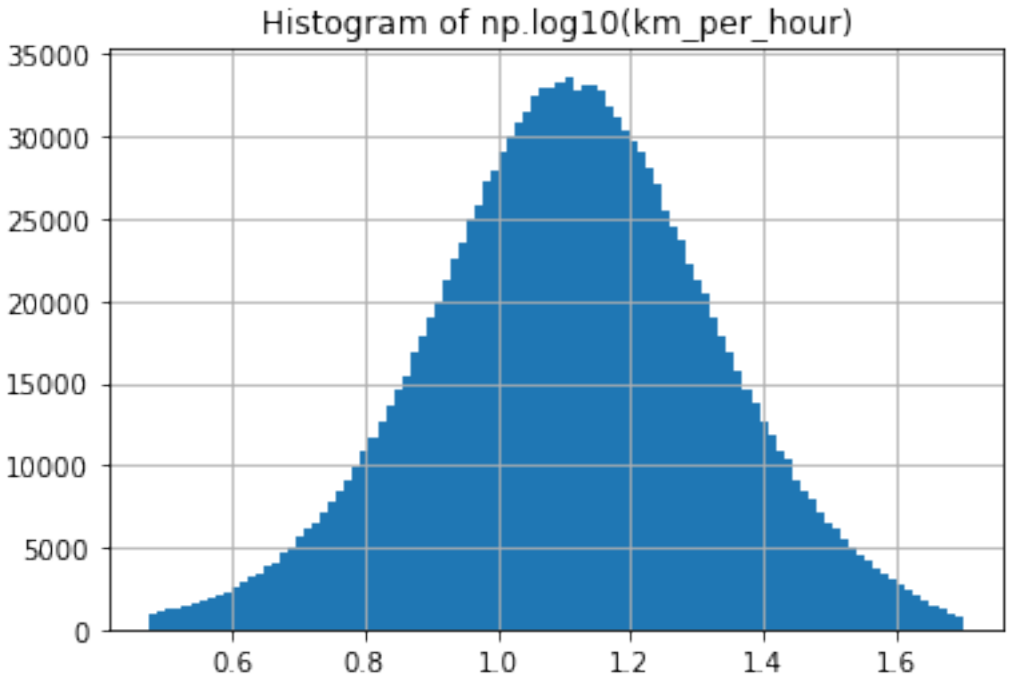
[12]:
```
# Draw histogram
full_df['km_per_hour'].hist(bins=100)
plt.title('Histogram of km_per_hour')
plt.show()
```



Convert the feature using *np.log10()* and show the new distribution:

[13]:
```
# Convert 'km_per_hour' with np.log10() method
full_df['km_per_hour']=np.log10(full_df['km_per_hour'])

# Draw histogram of the new values
full_df['km_per_hour'].hist(bins=100)
```

```
plt.title('Histogram of np.log10(km_per_hour)')
plt.show()
```



Good, it's now time to save our dataset.

Well done :-)

3 Save global dataset to an NPZ file

For the rest of the project, I've decided to store datasets and various informations like *categorical* and *numerical* column names in one NPZ file.

> Note: Code taken from my Course #4 project

3.1 Build *categorical* and *numerical* feature names

The full dataset is made of 48 features. To identify the one wich are *numerical* and which one are *categorical* is as easy as selecting them based on their dtype: - float => numerical - int => categorical

> Note: The dependent variable *km_per_hour*, my Y values, is coded as *float* in the full dataset and have to be removed to build the list of *numerical* features.

[14]:
```
# Get 'float' dtype column names
numerical_features=full_df.head(1).drop('km_per_hour', axis=1).
 ↪select_dtypes('float').columns
```

```
# Display column name and type
full_df[numerical_features].info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1426415 entries, 0 to 1426414
Data columns (total 20 columns):
pickup_longitude               1426415 non-null float64
pickup_latitude                1426415 non-null float64
dropoff_longitude              1426415 non-null float64
dropoff_latitude               1426415 non-null float64
distance_in_km                 1426415 non-null float64
pickup_distance_to_STATION     1426415 non-null float64
dropoff_distance_to_STATION    1426415 non-null float64
diff_ELEVATION                 1426415 non-null float64
WNP_AWND                       1426415 non-null float64
WNP_PRCP                       1426415 non-null float64
WNP_SNOW                       1426415 non-null float64
WNP_SNWD                       1426415 non-null float64
WNP_TAVG                       1426415 non-null float64
WNP_TSTD                       1426415 non-null float64
WND_AWND                       1426415 non-null float64
WND_PRCP                       1426415 non-null float64
WND_SNOW                       1426415 non-null float64
WND_SNWD                       1426415 non-null float64
WND_TAVG                       1426415 non-null float64
WND_TSTD                       1426415 non-null float64
dtypes: float64(20)
memory usage: 217.7 MB
```

[15]:
```
# Get 'int' dtype column names
categorical_features=full_df.head(1).select_dtypes('int').columns

# Display column name and type
full_df[categorical_features].info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1426415 entries, 0 to 1426414
Data columns (total 27 columns):
store_and_fwd_flag      1426415 non-null int64
weekend                 1426415 non-null int64
day_period_afternoon    1426415 non-null int64
day_period_evening      1426415 non-null int64
day_period_morning      1426415 non-null int64
passenger_alone         1426415 non-null int64
diff_ASCENDING          1426415 non-null int64
diff_DESCENDING         1426415 non-null int64
WC_WT01                 1426415 non-null int64
WC_WT02                 1426415 non-null int64
```

```
WC_WT03                 1426415 non-null int64
WC_WT04                 1426415 non-null int64
WC_WT06                 1426415 non-null int64
WC_WT08                 1426415 non-null int64
WC_WT09                 1426415 non-null int64
WC_WT11                 1426415 non-null int64
WC_WDIR_E               1426415 non-null int64
WC_WDIR_N               1426415 non-null int64
WC_WDIR_NE              1426415 non-null int64
WC_WDIR_NW              1426415 non-null int64
WC_WDIR_S               1426415 non-null int64
WC_WDIR_SE              1426415 non-null int64
WC_WDIR_SW              1426415 non-null int64
WC_WDIR_W               1426415 non-null int64
WC_PEAK_Y               1426415 non-null int64
WC_SNOW_FALL            1426415 non-null int64
WC_SNOW_ROAD            1426415 non-null int64
dtypes: int64(27)
memory usage: 293.8 MB
```

3.2 Save to NPZ

In case I will need later in this project one of the original datasets engineered to build this *full* dataset, I will save in my NPZ file: - The full dataset - The categorical and numerical column names of the dataset as list of strings - The dependent variable name as a one element list of string - The *stations* dataset - The *travel improved* dataset - The *weather categorical improved* dataset - The *weather numerical improved* dataset

Except for the *full* dataset, others will be retrieved from the SQLite database.

As I'll do later some more data exploration that might lead me to modify the *full* dataset, I've coded a function in my_utils, *save_npz()*, that will take the *full* dataset as parameter and save it along with other datasets.

> Note: There is two defined constant in my_utils to designate NPZ file names: *NPZ_FILENAME* and *NPZ_NORMALIZED_FILENAME*. The first constant is the name of the file that contains not normalized data, the other constant is for the file that will contain *normalized* data. Normalization will be done in the next chapter, for the moment I save the current dataset in the *NPZ_FILENAME* file.

[16]:
```
# Save datasets to an NPZ file using save_npz() function from my_utils library
save_npz(dataset=full_df, y_column='km_per_hour', y_dtype='float',␣
 ↪npz_filename=NPZ_DATAFILE)
```

```
Numerical features:  pickup_longitude,pickup_latitude,dropoff_longitude,dropoff_
latitude,distance_in_km,pickup_distance_to_STATION,dropoff_distance_to_STATION,d
iff_ELEVATION,WNP_AWND,WNP_PRCP,WNP_SNOW,WNP_SNWD,WNP_TAVG,WNP_TSTD,WND_AWND,WND
_PRCP,WND_SNOW,WND_SNWD,WND_TAVG,WND_TSTD
Categorical features:  store_and_fwd_flag,weekend,day_period_afternoon,day_perio
d_evening,day_period_morning,passenger_alone,diff_ASCENDING,diff_DESCENDING,WC_W
```

```
T01,WC_WT02,WC_WT03,WC_WT04,WC_WT06,WC_WT08,WC_WT09,WC_WT11,WC_WDIR_E,WC_WDIR_N,
WC_WDIR_NE,WC_WDIR_NW,WC_WDIR_S,WC_WDIR_SE,WC_WDIR_SW,WC_WDIR_W,WC_PEAK_Y,WC_SNO
W_FALL,WC_SNOW_ROAD
Build dict to pass to savez_compressed...
Query: SELECT * FROM stations
Query: SELECT * FROM stations LIMIT 1 OFFSET 0
Query: SELECT * FROM travel_improved
Query: SELECT * FROM travel_improved LIMIT 1 OFFSET 0
Query: SELECT * FROM weather_cat_improved
Query: SELECT * FROM weather_cat_improved LIMIT 1 OFFSET 0
Query: SELECT * FROM weather_num_improved
Query: SELECT * FROM weather_num_improved LIMIT 1 OFFSET 0
Save dict to NPZ file ./data/capstone-data.npz
Process terminated
```

4 Here we are

Our global dataset, the merge of the three engineered datasets (*travel*, *weather categorical* and *weather numerical*), is now ready to be used in EDA and Machine Learning models training.

Let's continue with the next notebook: 20.Exploratory Data Analysis

20.Exploratory Data Analysis

February 2, 2022

The merged database is made of fairly lot of features, I will explore them and see if some could be optimized, using two technics:

- Graphical Scatter Plots between numerical features and the dependent variable.
- Using PCA and Scree plot.

Before going further, I've taken time to write a new function into my_utils library to facilitate the loading of datasets from the NPZ backup file created in the previous notebook

- load_npz_as_dict()
- load_dataset()
- load_Xy()

Code of those functions is inspired by the code written in my my_lib.py library while completing my course #4 project, and the function *duildDataMatrix()* written in House Price model training chapter while completing course #3 project.

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

1 load_npz_as_dict()

Remember, in the previous notebook, I've stored all the datasets into an NPZ file, to easyly reload them from disk.

The *load_npz_as_dict()* function aims to return a *dict* Python object filled in with the datasets loaded from an NPZ file. The function expects two main parameters: - filename, the path to the NPZ file on disk, default set to *NPZ_NORMALIZED_DATAFILE* constant. - dataset, the name of the dataset I'd like to load, default being hte 'full' one. Others are *stations*, *travel*, *weather_num* and *weather_cat*

Two other optional parameters are available: - frac, a value between 0 and 1 used to get a sample of the dataset requested, 1 being 100% and 0 none. This parameter will be usefull when coding and training model to work on small subset of the data, 1.5 millions of line and 48 features might be too heavy to be processed on my desktop computer, even if it runs Apple M1 Silicon ;-) - verbose,

a boolean parameter, simply ask the function to display some informations when its value equal *True*

When used, this function opens the NPZ file, loads the requested dataset, convert them to a *pandas.DataFrame* object using column names loaded from the NPZ file, add it to a *dict()* Python object along with the *frac* parameter value and, if the requested dataset is the *full* one, add to the *dict()* object the feature name details of this *full* dataset (numerical, categorical, y and all).

This utility function will be used each time I need to get one of the *engineered* dataset, using the *frac* parameter to work on subset of it.

Note: When verbose=True, this function display the *shape* of the dataset returned.

1.1 Header of *load_npz_as_dict()*

implementation can be read in my_utils libary.

```
def load_npz_to_dict(filename=NPZ_DATAFILE, dataset='full', frac=1, verbose=True) -> dict:
    """
    This function returns one of the dataset stored in the NPZ file passed as parameter,
    and if the dataset claimed is the full one, then its feature names are added to the
    dict returned by the function.

    The dict structure returned looks like this:
        - feature_names: (if requested dataset is the full one)
            - numerical
            - categorrical
            - all
            - y
        - dataset
        - frac

    The NPZ file passed should contain a Python dict built in Notebook No 17

    The dataset parameter is used to determine which dataset the function should return.
    Dafault value is 'full'

    The frac parameter

    Returns:
    --------
    dict

    """
```

1.2 Demonstration of the *load_npz_to_dict()* function

I'd like to load 10% of the *stations* dataset and display the first three lines.

As the *stations* dataset contains 83 lines and 5 columns, I should obtain a dataset with shape=(8, 5), 10% of 83 lines with 5 columns.

[2]:
```
npz_dict=load_npz_as_dict(dataset='stations', frac=0.1)
df=npz_dict['dataset']
df.head(3)
```

```
Loading dataset 'stations' from NPZ file ./data/capstone-data-normalized.npz
Building sample from dataset (frac=0.1)
 Dataset shape: (8, 5)


Dataset loaded, returning dict
```

[2]:

	STATION	NAME	LATITUDE	LONGITUDE	ELEVATION
58	USC00282023	CRANFORD, NJ US	40.6666	-74.3235	24.4
25	US1NJHD0002	KEARNY 1.7 NW, NJ US	40.7729	-74.1409	29
14	US1NJUN0014	WESTFIELD 0.6 NE, NJ US	40.6588	-74.3358	36.3

2 load_dataset()

Along with the *load_npz_to_dict()* function, I've written a wrapper function around it: *load_dataset()*

This wrapper function returns a tuple instead of a dict for code simplification. For example, the following instruction loads the full dataset, store it in *df* variable, and initialize a *feature* variable that contains a list of the rest of the values of the returned tuple.

```
df,*features=load_dataset()
```

Be aware that this function removed from the retunred features column name list the name of the result vector column. This is a big difference with *load_npz_as_dict()* function which returns **all** dataframe column names in a single list (*load_npz_as_dict()['features']['all']*)

2.1 Header of *load_dataset()*

implementation can be read in my_utils libary.

```
def load_dataset(frac=1, random_state=5, verbose=True, y_dtype='float', npz_filename=NPZ_DATAFIL
    """
    Convenient wrapper around the load_npz_as_dict() function that returns the full dataset, its
    as a tuple.

    This function exists to simplify the code when loading full dataset. For example, the follwi
    loads the full dataset store it in df variable, feature variable will contain a list of the

        df,*features=load_dataset()

    Parameters are passed as is to the load_npz_as_dict() function.

    Returned tuple is:
        - dataset
        - all feature names
        - y result vector name
```

```
        - numerical feature names
        - categorical feature names

    Returns:
    --------
    tuple

    """
```

2.2 Demonstration of the *load_dataset()* function

I'd like to load 10% of the *full* dataset and display the feature names and the first three lines of the dataset returned:

[3]:
```
df,x_column,*_=load_dataset(verbose=False)

print("Feature column names:")
print(','.join(x_column))

print("\nFirst three line of dataset:")
df.head(3)
```

```
Feature column names:
pickup_longitude,pickup_latitude,dropoff_longitude,dropoff_latitude,store_and_fwd_flag,weekend,day_period_afternoon,day_period_evening,day_period_morning,passenger_alone,diff_ELEVATION,diff_ASCENDING,diff_DESCENDING,WC_WT01,WC_WT02,WC_WT03,WC_WT04,WC_WT06,WC_WT08,WC_WT09,WC_WT11,WC_WDIR_E,WC_WDIR_N,WC_WDIR_NE,WC_WDIR_NW,WC_WDIR_S,WC_WDIR_SE,WC_WDIR_SW,WC_WDIR_W,WC_PEAK_Y,WC_SNOW_FALL,WC_SNOW_ROAD,distance_in_km_square_log10,dropoff_distance_to_STATION_log1p,WNP_AWND_log1p,WNP_SNWD_log1p,WND_AWND_log1p,WND_SNWD_log1p

First three line of dataset:
```

[3]:

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag	weekend	day_period_afternoon	day_period_evening	day_period_morning	km_per_hour
0	-73.982155	40.767937	-73.964630	40.765602	0	0	1	0	0	1.073954
1	-73.980415	40.738564	-73.999481	40.731152	0	1	0	0	0	0.991388
2	-73.979027	40.763939	-74.005333	40.710087	0					

```
2        0                    0                   0                   1
1.034316

   passenger_alone  diff_ELEVATION  diff_ASCENDING  diff_DESCENDING  WC_WT01
WC_WT02  WC_WT03  \
0                1             0.0               0                0        1
0        0
1                1            37.2               0                1        0
0        0
2                1            37.2               0                1        0
0        0

   WC_WT04  WC_WT06  WC_WT08  WC_WT09  WC_WT11  WC_WDIR_E  WC_WDIR_N  WC_WDIR_NE
WC_WDIR_NW  \
0        0        0        1        0        0          1          0           1
0
1        0        0        1        0        0          0          0           0
1
2        0        0        0        0        0          0          0           0
1

   WC_WDIR_S  WC_WDIR_SE  WC_WDIR_SW  WC_WDIR_W  WC_PEAK_Y  WC_SNOW_FALL
WC_SNOW_ROAD  \
0          0           0           0          0          1             0
0
1          0           0           0          1          1             0
0
2          0           0           0          1          1             1
1

   distance_in_km_square_log10  dropoff_distance_to_STATION_log1p
WNP_AWND_log1p  WNP_SNWD_log1p  \
0                     0.351326                           0.931212
1.987874            0.0
1                     0.513198                           1.838520
1.526056            0.0
2                     1.610335                           1.444618
1.774952            0.0

   WND_AWND_log1p  WND_SNWD_log1p
0        1.987874         0.00000
1        1.173514         0.00000
2        0.000000         2.87168
```

Cool, isn't it ?

3 load_Xy_as_dict()

Training model is based on *np.array* of features (X) and vector result (y) build from datasets, splitted in train and validation subsets.

I've coded the *load_Xy_as_dict()* function to do the job in one call:

- Load the full dataset using *load_dataset()*
- Split dataframe in two subset, *train* and *valid*, using *sklearn.model_selection.train_test_split()* method
- Print some information on the shape of the subset created (if verbose=True)
- Return a dict of the different *np.arrays* created

3.1 Header of *load_Xy_as_dict()*

implementation can be read in my_utils libary.

```
def load_Xy(train_size=TRAIN_SIZE_DEFAULT, frac=1, random_state=5, verbose=True, npz_filename=NP
    """
    Used to get features and vector result of the 'full' dataset as X and y np.array, splitted i
    and valid one.

    The 'train_size' parameter may be used to fix the train size (defaults 0.8). This parameter
    'sklearn.model_selection.train_test_split()' method.

    The value returned is a dict object:

        - train:
            - X:    Train set of X features
            - y:    Train set of y vector result

        - valid:
            - X:    Validation set of X features
            - y:    Validation set of y vector result

        - all:
            - X:    Complete set of X features
            - y:    Complete set of y vector result

        - features: List of feature names
        - result:   Name of the y result vector

    The 'full' dataset is retrived using the 'load_dataset()' function.

    Returns:
    --------
    dict

    """
```

3.2 Demonstration of the *load_Xy_as_dict()* function

I'd like to load 1% of the X train feature values from the *full* dataset

[4]:
```
X_tr=load_Xy_as_dict(frac=0.01, verbose=False)['train']['X']

print("Shape of the X train dataset using 1% of the full dataset:", X_tr.shape)
```

```
Shape of the X train dataset using 1% of the full dataset: (11411, 38)
```

4 load_Xy()

A wrapper function around *load_Xy_as_dict()* that returns train and valid X/y values as a tuple, to simplify code in next Notebooks.

More informations below in the header of the function.

4.1 Header of *load_Xy()*

implementation can be read in my_utils libary.

```
def load_Xy(train_size=TRAIN_SIZE_DEFAULT, frac=1, random_state=5, verbose=True, npz_filename=NP
    """
    A wrapper function around load_Xy_as_dict() that returns X_tr, y_tr, X_va and y_va as a tupl

    This function aims to simplify the code in Notebooks

    Returns:
    --------
    (X_tr, y_tr, X_va, y_va)

    """
```

4.2 Demonstration of the *load_Xy()* function

I'd like to load 1% of the train and validation feature values of the *full* dataset in one instruction, with train/valid split = 60%

[5]:
```
X_tr, y_tr, X_va, y_va = load_Xy(frac=0.01, train_size=0.6)
print("X_tr shape", X_tr.shape)
print("y_tr shape", y_tr.shape)
print("X_va shape", X_va.shape)
print("y_va shape", y_va.shape)
```

```
X_tr shape (8558, 38)
y_tr shape (8558,)
X_va shape (5706, 38)
y_va shape (5706,)
```

5 Let's continue

to the next notebook, Scatter plots for EDA

21.Scatter plots for EDA

February 2, 2022

Before going further in Machine Learning model training, I've decided to determine if some of the feature in the *full* dataset might be useless for fitting models, or need some more improvements.

The first approach I'll use is a visual one, plotting Scatter Plots of the numerical values against the values of my dependent variable: *km_per_hour*

The goal here is to eliminate features that looks like to be **not** correlated with the dependent variable (first derivative of the linear function is null), identify any features which should be normalized using an *np.logxx()* approach, and apply *Plynomial increase approach* on features that looks like being correlated with the vector result using a *polynomial* function applied to their values.

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

1 Load *full* dataset

Load the *full* dataset, store it in a global variable, and initialize two variables: x_columns and y_column.

> Note : Data is loaded from the *NPZ_DATAFILE* file which contains non normalized data. At the end of this Notebook, when data will be normalized, the result will be saved in the *NPZ_NORMALIZE_DATAFILE* file.

[2]:
```
# Fraction of the dataset to load
DATASET_FRAC=0.01

# Load dataset dict
df, _, y_column, x_columns, _=load_dataset(frac=DATASET_FRAC,␣
 ↪npz_filename=NPZ_DATAFILE, verbose=True)
```

```
Loading dataset 'full' from NPZ file ./data/capstone-data.npz
Apply correct dtype to dataset column
{'km_per_hour': 'float', 'pickup_longitude': 'float', 'pickup_latitude':
'float', 'dropoff_longitude': 'float', 'dropoff_latitude': 'float',
'distance_in_km': 'float', 'pickup_distance_to_STATION': 'float',
```

```
'dropoff_distance_to_STATION': 'float', 'diff_ELEVATION': 'float', 'WNP_AWND':
'float', 'WNP_PRCP': 'float', 'WNP_SNOW': 'float', 'WNP_SNWD': 'float',
'WNP_TAVG': 'float', 'WNP_TSTD': 'float', 'WND_AWND': 'float', 'WND_PRCP':
'float', 'WND_SNOW': 'float', 'WND_SNWD': 'float', 'WND_TAVG': 'float',
'WND_TSTD': 'float', 'store_and_fwd_flag': 'int', 'weekend': 'int',
'day_period_afternoon': 'int', 'day_period_evening': 'int',
'day_period_morning': 'int', 'passenger_alone': 'int', 'diff_ASCENDING': 'int',
'diff_DESCENDING': 'int', 'WC_WT01': 'int', 'WC_WT02': 'int', 'WC_WT03': 'int',
'WC_WT04': 'int', 'WC_WT06': 'int', 'WC_WT08': 'int', 'WC_WT09': 'int',
'WC_WT11': 'int', 'WC_WDIR_E': 'int', 'WC_WDIR_N': 'int', 'WC_WDIR_NE': 'int',
'WC_WDIR_NW': 'int', 'WC_WDIR_S': 'int', 'WC_WDIR_SE': 'int', 'WC_WDIR_SW':
'int', 'WC_WDIR_W': 'int', 'WC_PEAK_Y': 'int', 'WC_SNOW_FALL': 'int',
'WC_SNOW_ROAD': 'int'}
Building sample from dataset (frac=0.01)
 Dataset shape: (14264, 48)


Dataset loaded, returning dict
```

2 Draw Scatter plots

To simplify the work, I will define a function that will draw a scatter plot from a continuous column, first removing lines with value == 0, and secondly remove outliers using z-score approach. This will help to decide what to do with those columns.

At the same time, this function will draw, on top of the scatter plots, *polynomial regression curves* that will be helpfull to identify any polynomial correlation between features and result vector.

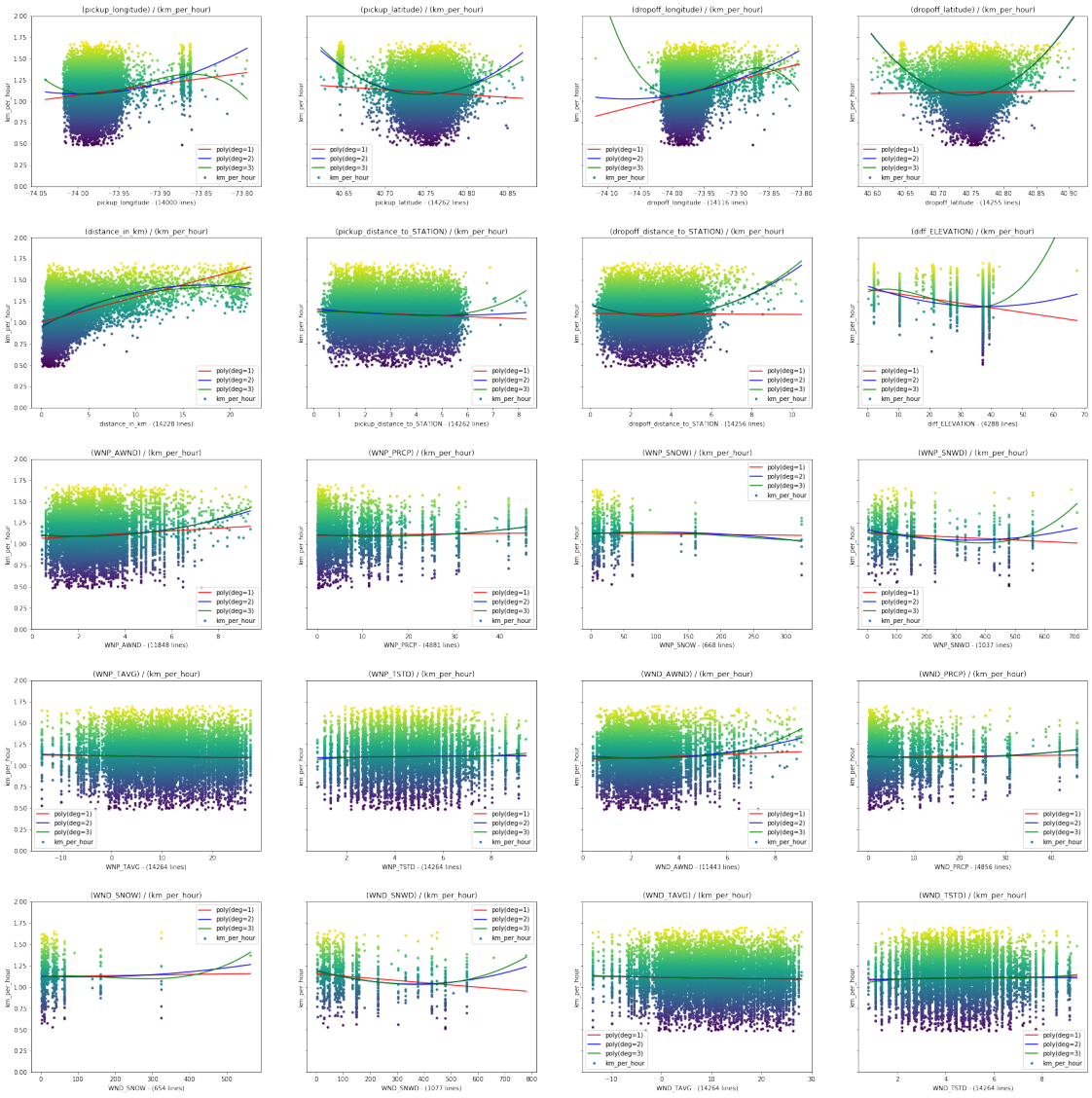
The code of this function is inspired from the *drawScatterPlot()* I've implemented in my course #3 project

> Note: Remember, the vector result *y*, *km_per_hour*, has been converted to its *no.log10()* value. *ylim* value passed to *drawScatterPlot()* should be set to [0,2] ;-)

[3]:
```
# Draw scatter plots of numerical features
draw_scatter_plot(dataset=df, x_columns=x_columns, y_column=y_column,␣
 ↪polyfit_deg=3, figsize=(30,30), z_factor=5)
```

```
Drawing graphs 4 x 5 (number of x_columns = 20)
Plotting feature: pickup_longitude
Plotting feature: pickup_latitude
Plotting feature: dropoff_longitude
Plotting feature: dropoff_latitude
Plotting feature: distance_in_km
Plotting feature: pickup_distance_to_STATION
Plotting feature: dropoff_distance_to_STATION
Plotting feature: diff_ELEVATION
Plotting feature: WNP_AWND
Plotting feature: WNP_PRCP
```

```
Plotting feature: WNP_SNOW
Plotting feature: WNP_SNWD
Plotting feature: WNP_TAVG
Plotting feature: WNP_TSTD
Plotting feature: WND_AWND
Plotting feature: WND_PRCP
Plotting feature: WND_SNOW
Plotting feature: WND_SNWD
Plotting feature: WND_TAVG
Plotting feature: WND_TSTD
Processing done, display result (may take some time)
```



3 What decisions those Scatter plots implies ?

3.1 Polynomial feature increase approach

Looking at the polynomial curves on the scatter plots, I've identified one features that could be a good candidate for polynomial increase: *distance_in_km*

Let's add a new feature named *distance_in_km_square* which contains the square values of *distance_in_km* and plot the result.

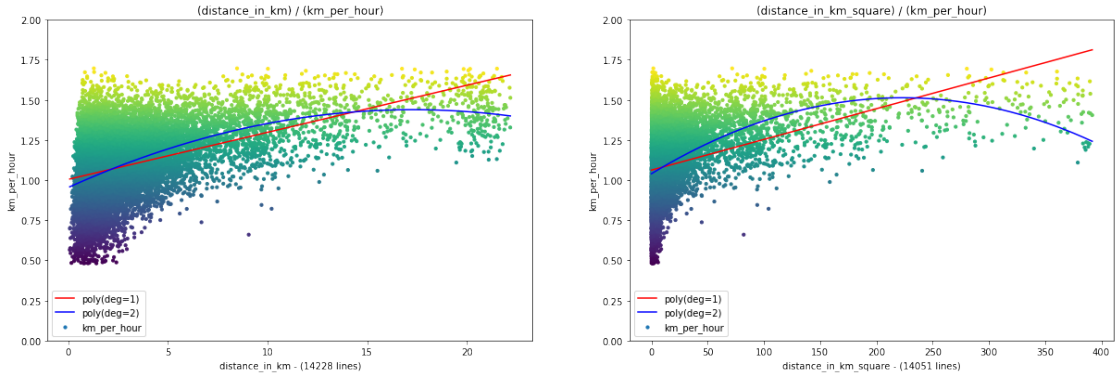
[4]:
```
# Store the square value of distance_in_k into distance_in_km_square
df['distance_in_km_square']=df['distance_in_km']**2

# Show result
df[['distance_in_km_square','distance_in_km']].head(2)
```

[4]:

	distance_in_km_square	distance_in_km
789677	428.080199	20.690099
1176476	7.999932	2.828415

[5]:
```
# scatter plot one feature and its square increased feature
draw_scatter_plot(dataset=df, x_columns=['distance_in_km',␣
↪'distance_in_km_square'], y_column=y_column, graph_per_line=2, polyfit_deg=2,␣
↪figsize=(20,6), z_factor=5)
```

```
Drawing graphs 2 x 1 (number of x_columns = 2)
Plotting feature: distance_in_km
Plotting feature: distance_in_km_square
Processing done, display result (may take some time)
```

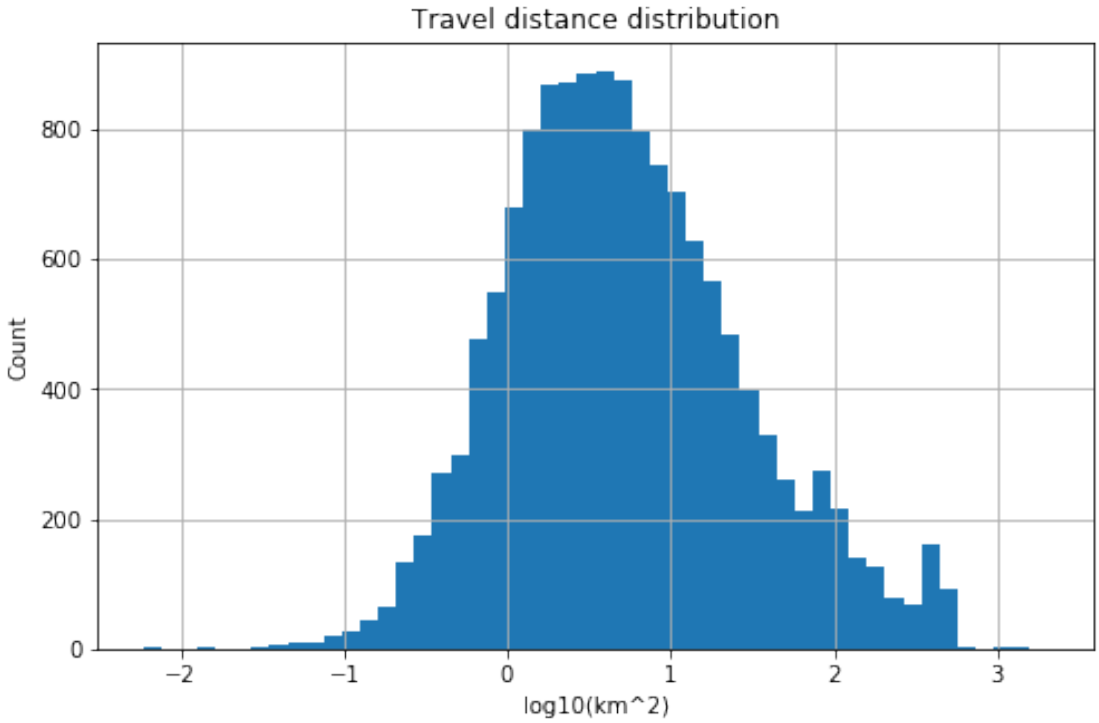


The result looks good, we can now drop the *distance_in_km* feature:

[6]:
```
df.drop(['distance_in_km'], axis=1, inplace=True)
```

And finally normalize *distance_in_km_square* using *np.log()* transformation.

[7]:
```
# Transform features values with np.log10()
df['distance_in_km_square_log10'] = np.log10(df['distance_in_km_square'])

# Display histogram
plt.figure(figsize=(8,5))
plt.title('Travel distance distribution')
df['distance_in_km_square_log10'].hist(bins=50)
plt.xlabel('log10(km^2)')
plt.ylabel('Count')
plt.show()
```



Ok, *distance_in_km* feature engineering looks good. Drop the *distance_in_km_square* and I am done with this first feature.

[8]:
```
df.drop(['distance_in_km_square'], axis=1, inplace=True)
```

3.2 Drop non correlated features

Non correlated numerical features in this dataset can be identified looking at the polynomial regression curve with degree=1. For that non-correlated features, the regression curve is a straight line with first derivative value equal to 0, in other words, whatever the value of the feature, the travel speed predicted by this polynomial regression is a fixed value.

For example, if we look at *WNP_TSTD* (Temperature standard deviation at pickup location), the polynomial curves are close to be horizontal lines.

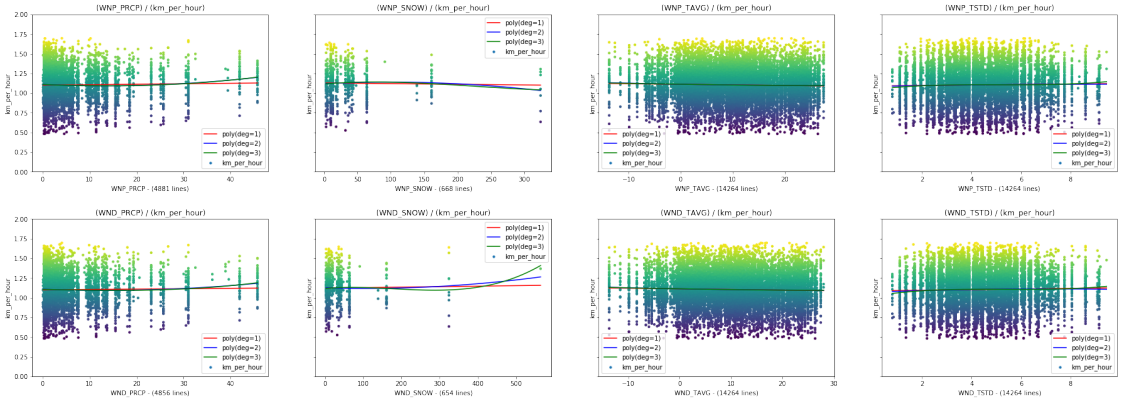
3.2.1 List of non-correlated features

TAVG, TSTD, PRCP and SNOW looks like non-correlated features.

SNWD, even if it looks apparently not correlated, I can see small correlation when its value increase. I've choosen to keep it (I will log transformed it later in this Notebook)

[9]:
```
import re

# Build filtered column list
r = re.compile('.*TAVG.*|.*TSTD.*|.*SNOW.*|.*PRCP.*')
cols_to_drop=list(filter(r.match, x_columns))

# Draw scatter plots for TAVG, TSTD, PRCP and SNOW
draw_scatter_plot(dataset=df, x_columns=cols_to_drop, y_column=y_column,␣
 ↪graph_per_line=4, polyfit_deg=3, figsize=(30,10), z_factor=5)
```

```
Drawing graphs 4 x 2 (number of x_columns = 8)
Plotting feature: WNP_PRCP
Plotting feature: WNP_SNOW
Plotting feature: WNP_TAVG
Plotting feature: WNP_TSTD
Plotting feature: WND_PRCP
Plotting feature: WND_SNOW
Plotting feature: WND_TAVG
Plotting feature: WND_TSTD
Processing done, display result (may take some time)
```



3.2.2 Drop non-correlated columns

Decision made to drop those 8 non correlated columns

[10]:
```
# Drop column and save dataset to alternate NPZ file
print("Columns to drop:", ','.join(cols_to_drop))
df.drop(cols_to_drop, axis=1, inplace=True)
```

```
Columns to drop:
WNP_PRCP,WNP_SNOW,WNP_TAVG,WNP_TSTD,WND_PRCP,WND_SNOW,WND_TAVG,WND_TSTD
```

3.3 A particular feature: *diff_ELEVATION*

Looking at the scatter plot of this feature, the distribution of the data points on the graph looks like to be quantified. This non linearity could be used to transform this numerical feature into a categorical one.

Finally, I've decided to keep this numerical feature as it is.

3.4 Pickup and dropoff latitude

Those two features are globaly equaly distributed, the polynomial regression with degree=3 confirm this.

I will keep those two features as is, scaling pre-processing that will be applied during Machine Learning models training will normalize their values.

3.5 Pickup and dropoff longitude

I'm a bit sceptic about those two features. I've tried some log transformations but the result did not convinced me.

I've choosen to keep them as is and let the Machine Learning model algorythms handle them.

3.6 *np.log10()* to normalize other features.

The six remaining features, looking at their respective feature plots, shouold be normalized using an *np.log()* approach.

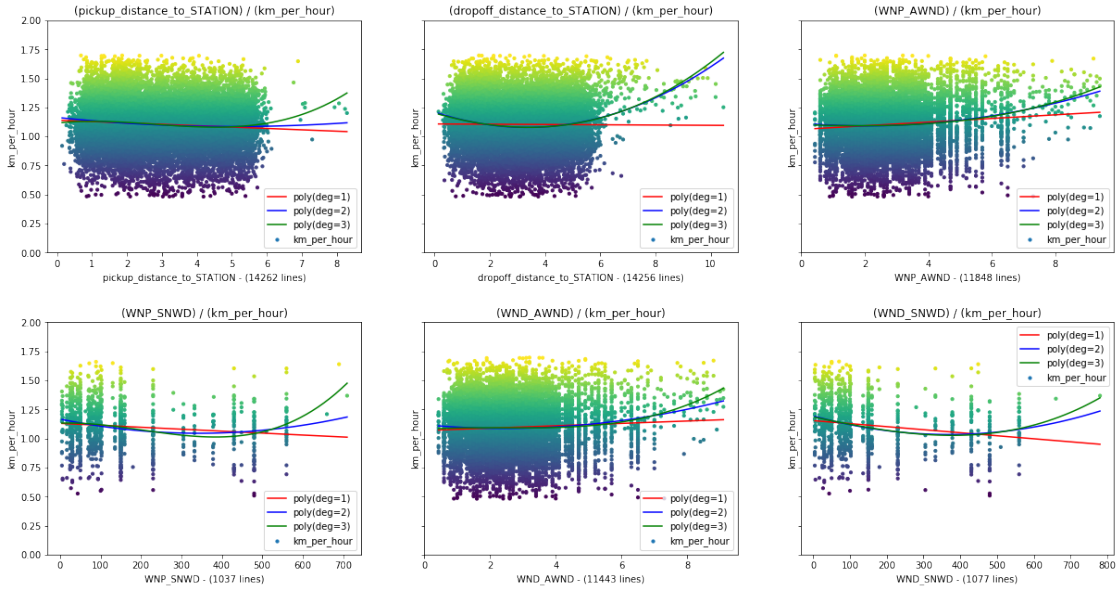
As I have null values for them, I'll use the *np.log1p()* method that takes care of null values (adding +1 to avoid infinite result)

Let's display their current scatter plots

[11]:
```
# Build filtered column list
r = re.compile('.*STATION.*|.*AWND.*|.*SNWD.*')
cols_to_log_transform=list(filter(r.match, x_columns))

# Draw scatter plots
draw_scatter_plot(dataset=df, x_columns=cols_to_log_transform,␣
 ↪y_column=y_column, graph_per_line=3, polyfit_deg=3, figsize=(20,10),␣
 ↪z_factor=5)
```

```
Drawing graphs 3 x 2 (number of x_columns = 6)
Plotting feature: pickup_distance_to_STATION
Plotting feature: dropoff_distance_to_STATION
Plotting feature: WNP_AWND
Plotting feature: WNP_SNWD
Plotting feature: WND_AWND
```

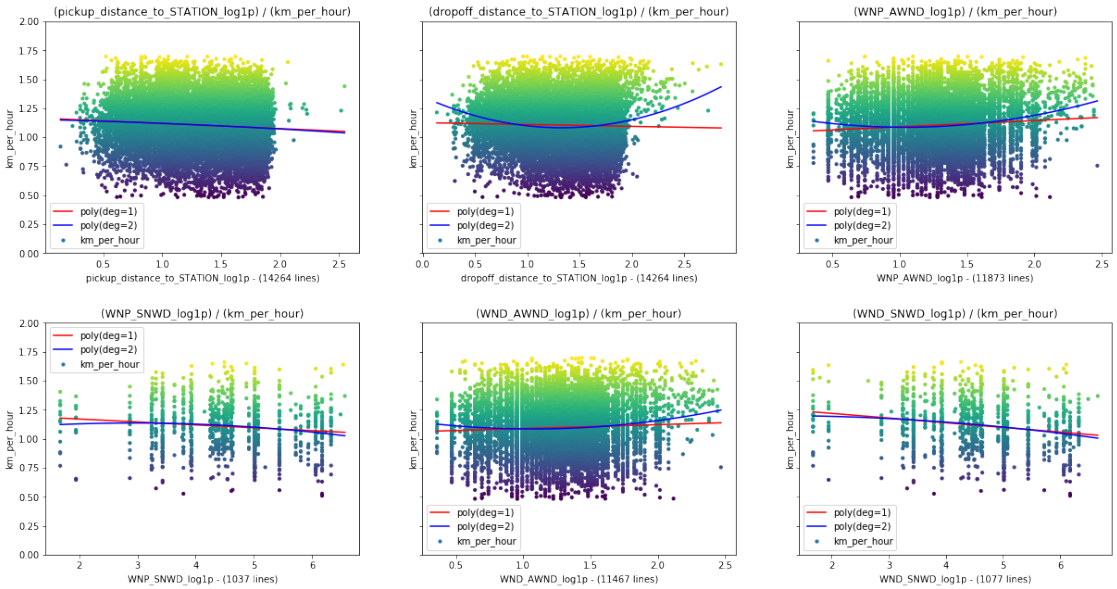
```
Plotting feature: WND_SNWD
Processing done, display result (may take some time)
```



Transform their value and display the modified scatter plots

[12]:
```
# Build log transformed features
new_cols=[]
for col in cols_to_log_transform:
    new_col=f'{col}_log1p'
    new_cols.append(new_col)
    df[new_col]=np.log1p(df[col])

# Display resulting scatter plots
draw_scatter_plot(dataset=df, x_columns=new_cols, y_column=y_column,␣
 ↪graph_per_line=3, polyfit_deg=2, figsize=(20,10), z_factor=5)
```

```
Drawing graphs 3 x 2 (number of x_columns = 6)
Plotting feature: pickup_distance_to_STATION_log1p
Plotting feature: dropoff_distance_to_STATION_log1p
Plotting feature: WNP_AWND_log1p
Plotting feature: WNP_SNWD_log1p
Plotting feature: WND_AWND_log1p
Plotting feature: WND_SNWD_log1p
Processing done, display result (may take some time)
```



3.6.1 A special case: pickup and dropoff distance to STATION

Looking at the polynomial regression degree=2 of the *pickup_distance_to_STATION* log transformed feature, it appears to be non correlated to the travel speed. On the opposite side, the *dropoff_distance_to_STATION* seems to have an influence on the result.

Even if that sound strange, this is what I can conclude looking at the previous Scatter plots.

So, what should I do ?

Drop the pickup one and keep the dropoff as is :-)

3.6.2 What about *SNWD* features ?

Well, they're near to be non correlated as well, but the linear regression shows a little inverted correlation. Furthermore, I must avoid to drop too many features ;-)

So I'll decide to keep them log transformed.

3.7 Data optimization done

Let reload the full dataset, re-apply the above transformations and drops, and save result into the *NPZ_NORMALIZED_DATAFILE*

4 Save cleaned version of the full dataset

Here is the process to follow:

- Reload full dataset from NPZ file
- Do polynomial increase as described above

- Do log transformed as described above
- Drop unused columns
- Save it back to the *NPZ_FILENAME* NPZ file

4.1 Load full dataset and numerical feature names

[13]:
```
# Load full dataset and feature column names
df, _, y, x_columns, _=load_dataset(verbose=False, npz_filename=NPZ_DATAFILE)

print("Shape of the full dataset:", df.shape)
```

```
Shape of the full dataset: (1426415, 48)
```

4.2 Add polynomial increase and log transformation to *distance_in_km* feature

[14]:
```
# Increase and log transform
df['distance_in_km_square_log10'] = np.log10(df['distance_in_km']**2)
df.drop('distance_in_km', axis=1, inplace=True)
```

4.3 Drop non-correlated features

[15]:
```
# Drop column and save dataset to alternate NPZ file
print("Columns to drop:", ','.join(cols_to_drop))
df.drop(cols_to_drop, axis=1, inplace=True)
```

```
Columns to drop:
WNP_PRCP,WNP_SNOW,WNP_TAVG,WNP_TSTD,WND_PRCP,WND_SNOW,WND_TAVG,WND_TSTD
```

4.4 Log transform the needed features

[16]:
```
# Build log transformed features
print("Columns to log transform:", cols_to_log_transform)

for col in cols_to_log_transform:
    df[f'{col}_log1p']=np.log1p(df[col])

df.drop(cols_to_log_transform, axis=1, inplace=True)
```

```
Columns to log transform: ['pickup_distance_to_STATION',
'dropoff_distance_to_STATION', 'WNP_AWND', 'WNP_SNWD', 'WND_AWND', 'WND_SNWD']
```

4.5 Drop pickup_distance_to_STATION

Note: This feature was in the list of log transformed columns. To drop it, we should drop the newly created feature: *pickup_distance_to_STATION_log1p*

[17]:
```
df.drop('pickup_distance_to_STATION_log1p', axis=1, inplace=True)
```

4.6 Save this new normalized full dataset

and save it to the *NPZ_NORMALIZED_DATAFILE* to keep the original one safe, in case of ;-)

[18]:
```
# Save to NPZ
save_npz(dataset=df, npz_filename=NPZ_NORMALIZED_DATAFILE)
```

```
Numerical features:  pickup_longitude,pickup_latitude,dropoff_longitude,dropoff_
latitude,diff_ELEVATION,distance_in_km_square_log10,dropoff_distance_to_STATION_
log1p,WNP_AWND_log1p,WNP_SNWD_log1p,WND_AWND_log1p,WND_SNWD_log1p
Categorical features:  store_and_fwd_flag,weekend,day_period_afternoon,day_perio
d_evening,day_period_morning,passenger_alone,diff_ASCENDING,diff_DESCENDING,WC_W
T01,WC_WT02,WC_WT03,WC_WT04,WC_WT06,WC_WT08,WC_WT09,WC_WT11,WC_WDIR_E,WC_WDIR_N,
WC_WDIR_NE,WC_WDIR_NW,WC_WDIR_S,WC_WDIR_SE,WC_WDIR_SW,WC_WDIR_W,WC_PEAK_Y,WC_SNO
W_FALL,WC_SNOW_ROAD
Build dict to pass to savez_compressed...
Query: SELECT * FROM stations
Query: SELECT * FROM stations LIMIT 1 OFFSET 0
Query: SELECT * FROM travel_improved
Query: SELECT * FROM travel_improved LIMIT 1 OFFSET 0
Query: SELECT * FROM weather_cat_improved
Query: SELECT * FROM weather_cat_improved LIMIT 1 OFFSET 0
Query: SELECT * FROM weather_num_improved
Query: SELECT * FROM weather_num_improved LIMIT 1 OFFSET 0
Save dict to NPZ file ./data/capstone-data-normalized.npz
Process terminated
```

5 Here we are

[19]:
```
npz_dict=load_npz_as_dict(dataset='full', verbose=False)
print("Shape of this new dataset (stored in NPZ_NORMALIZED_DATAFILE):",␣
 ↪npz_dict['dataset'].shape)
```

```
Shape of this new dataset (stored in NPZ_NORMALIZED_DATAFILE): (1426415, 39)
```

That's all folks for the Scatter Plots EDA journey ;-)

Let's continue with PCA on the next notebook

22.Principal Component Analysis

February 2, 2022

In this Notebook, I'll explore my dataset with *Principal Component Analysis (PCA)* from Scikit-Learn.

The goal is to see if PCA might help in reducing the dataset size doing a linear combination of my features and finding the one that could explain most of the variance.

This Notebook is inspired from the information found on PCA in this article and the code I've written during my project Course #4: Data Exploration

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

1 Load features

Note: For this PCA analysis, I will use the full dataset by setting the *frac* parameter to value 1 in the *load_Xy_as_dict()* function call.

[2]:
```
# load train X/y data matrix
data=load_Xy_as_dict(frac=1)

# Store features in X variable
X=data['all']['X']
```

```
Loading dataset...
Splitting dataset...
Load and split process terminated
  Shape of X train variable: (1141132, 38)
  Shape of y train variable: (1141132,)
  Shape of X valid variable: (285283, 38)
  Shape of y valid variable: (285283,)
  Shape of X variable   : (1426415, 38)
  Shape of y variable   : (1426415,)
```

2 Scale data

PCA algorithm is very sensitive to the relative ranges of the features, for that reason we should scale it.

[3]:
```
from sklearn.preprocessing import StandardScaler

transformer = StandardScaler().fit(X)
X_scaled=transformer.transform(X)
```

3 Fit and Transform

Now that we have a *scaled* feature matrix, fit it to PCA and transform it.

As I'd like to identify the number of *Principal Components* I should keep to explain most of the variance, I'll then set *n_components* parameter to *None*, which is equivalent to setting it to the number of features in the dataset.

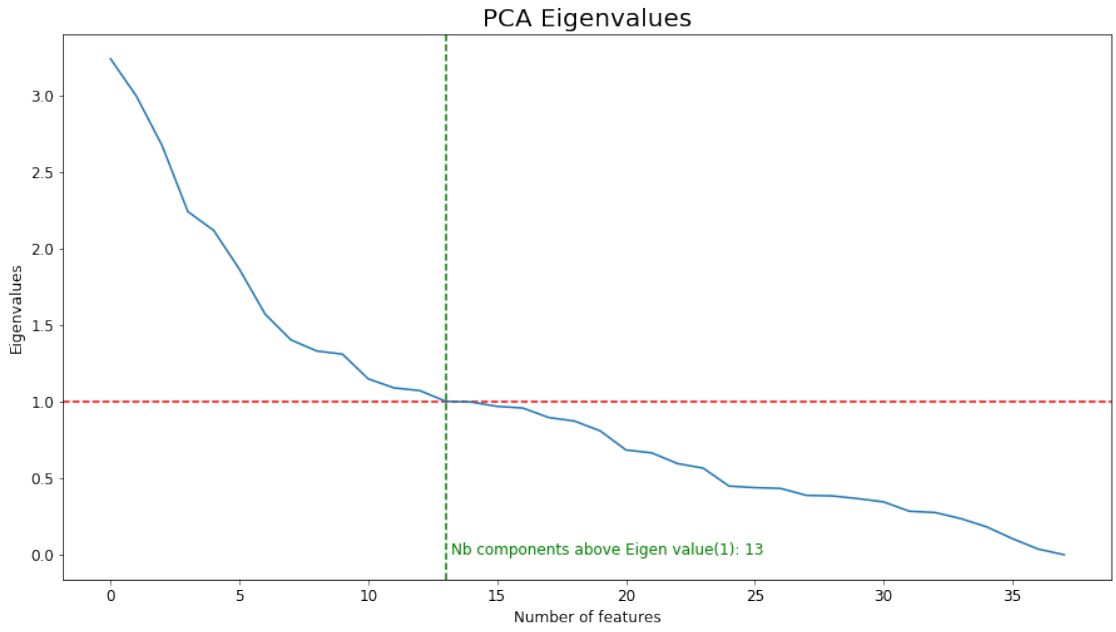
[4]:
```
from sklearn.decomposition import PCA

pca=PCA(n_components=None)

pca.fit(X_scaled)

X_pca=pca.transform(X_scaled)
```

4 Plot the PCA Eigenvalues

After fitting PCA with all features, I'll use the *PCA.explained_variance_* property to get the *Eigenvalues* of each components built by the PCA fitting process.

[5]:
```
# Get the first component id that is below 1
eigen_limit=0
for i, value in zip(list(range(0,len(pca.explained_variance_))), pca.
 ↪explained_variance_):
    if value < 1:
        eigen_limit=i-1
        break

# Set some plot parameters
plt.figure(figsize=(15,8))
plt.rcParams.update({'font.size': 12})
plt.style.context('seaborn-whitegrid')

# Set axis labels ant title
plt.ylabel('Eigenvalues')
plt.xlabel('Number of features')
```

```
plt.title('PCA Eigenvalues', fontsize=20)

# Define Y axis limit
plt.ylim=(0,max(pca.explained_variance_))

# Draw an horizontal line where Eigenvalue = 1
plt.axhline(y=1, color='r', linestyle='--')

# Plot a vertical line to delimite compnents below or above 1
plt.axvline(x=eigen_limit, color='g', linestyle='--')
plt.text(eigen_limit + 0.2, 0, f'Nb components above Eigen value(1):␣
 ↪{eigen_limit}', color='g')

# Plot Eigenvalues
plt.plot(pca.explained_variance_)

# Show canvas
plt.show()
```



PCA theory says that using the primary components that have an Eigenvalue above 1.0 is enough to explain most of the variance, wich in my case is 13.

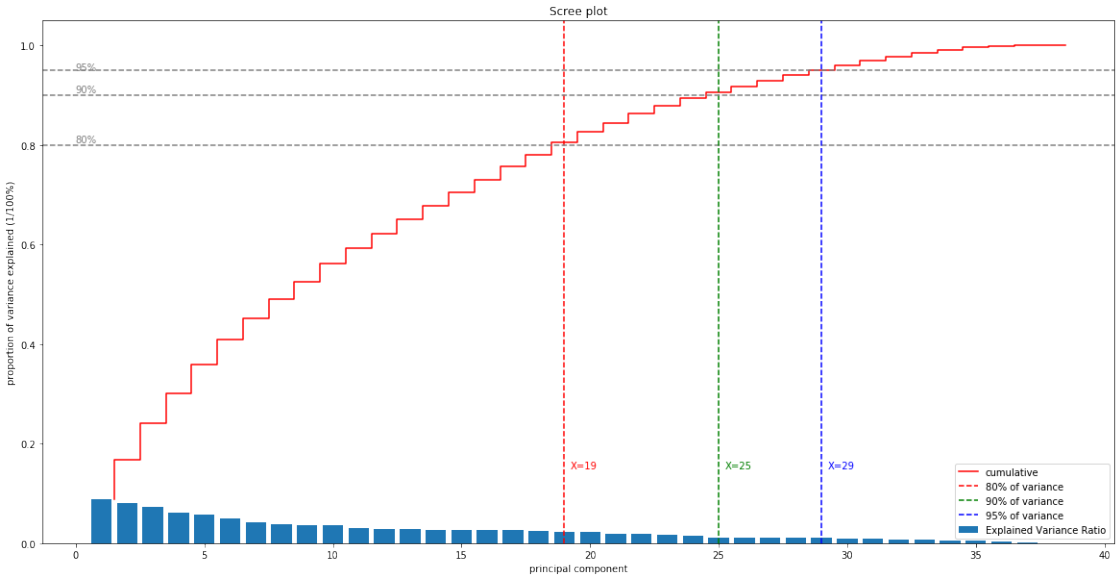
Let's draw another type of plotting function that displays how many principal components could explain any percentage of the variance: The ScreePlot.

5 Draw a Screeplot

A Scree plot draws the ratio cumulative sum of the explained variance and helps identify how many components we must keep to get a specific percentage of variance.

[6]:
```
# Get explained variance ratio
pve=pca.explained_variance_ratio_

# Build its cumulative sum
pve_cumsum = np.cumsum(pve)

# Set some plot parameters
plt.figure(figsize=(20,10))
plt.rcParams.update({'font.size': 10})

# Create bar plot with explained variance ration
xcor = np.arange(1, len(pve) + 1) # 1,2,..,n_components
plt.bar(xcor, pve, label='Explained Variance Ratio')

# Create a step graph using the cummulated value of explained usariance ratio
plt.step(
    xcor+0.5, # 1.5,2.5,..,n_components+0.5
    pve_cumsum, # Cumulative sum
    label='cumulative',
    color='r'
)

# Draw horizontal lines every y = y + 0.1
for i in np.arange(0.1, 1.1, 0.1).tolist():
    #plt.plot([0,50], [i, i], label='{}% of variance'.format(int(i*100)))
    pass

for (variance, color) in zip([80, 90, 95], ['r', 'g', 'b']):
    i=1
    for cumsum in pve_cumsum:
        if cumsum > variance/100:
            # Draw vertical line were variance explained = 95%
            plt.axvline(x=i, color=color, linestyle='--', label=f'{variance}% of␣
 ↪variance')
            plt.text(i+0.25,0.15,f'X={i}', color=color)

            # Draw horizontal line
            plt.axhline(y=variance/100, linestyle='--', color='grey')
            plt.text(0, cumsum, f'{variance}%', color='grey')

            break
        i=i+1
```

```
# Set axis labels and title
plt.xlabel('principal component')
plt.ylabel('proportion of variance explained (1/100%)')
plt.title('Scree plot')

# Show legend
plt.legend()

# Show canvas
plt.show()
```



Smart graph, isn't it ?

I can conclude that PCA should be tried in Machine Learning training pipelines as 90% of the variance is explained using half of the features.

In the next Notebooks, dedicated to model training, I will include in grid search process different PCA reduction as grid parameter.

6 This is the end of EDA

Data preparation and EDA is now terminated.

It's time now to go to the Machine Learning chapter and train our models.

30.Machine Learning Models

February 2, 2022

1 Time to train models :-)

I've made the choice of training four different types of Machine Learning models and compare results between them:

- Gradient Descent based: Ridge Regressor
- Distance based: KNeighborsRegressor
- Category based: RandomForestRegressor
- Neural network: MLPRegressor

Due to the quite big dataset I have, I will use a RandomizedSearchCV step with large parameter scopes to identify the best ML model parameter intervals, and refine the search using reduced scopes with a GridSearchCV. To do preprocessing using scaler like RobustScaler and/or component reduction with PCA, I will combine grid search with Pipeline.

The combination of GridSearchCV and Pipeline is an approach that I've already used in my course #3 project, inspired by this article: SKlearn: Pipeline & GridSearchCV.

Before going to train. models, let me expose some tricks and functions that will be used when training models.

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

2 Work on a fraction of the datasets

The *full* dataset I've built to train models is made of about 1.5 millions of lines and 50 features.

In order to speed-up model training, I will work on a fraction of this dataset. To have the same value accross all the Notebooks of this fraction parameter, I've coded it as a *CONSTANT* in my_utils library.

The current value is:

[2]:
```
print("Fraction of the dataset used to train models: {:.2f}%".
 ↪format(FRAC_VALUE_FOR_ML*100))
```

```
Fraction of the dataset used to train models: 10.00%
```

3 *Mean Absolute Error* and *Mean Absolute Percent Error* to evaluate model performance.

I've choosen the mean_absolute_error() approach from *sklearn.metrics* to evaluate my models performance. It helps to determine the *average* error made by models on prediction.

In order to simplify the use of this mean_absolute_error() in the next Notebooks, I've coded two functions: *mae()* and *mape()*

3.1 mae(): Mean Absolute Error

This function returns the *Mean Absolute Error* of a prediction in km/h. It takes care of the fact that the result vector of the *Full* dataset, *km_per_hour*, has been log transformed.

```
def mae(y_pred, y) -> np.array:
    """
    Returns 10^mean_absolute_error() between the two result
    vector passed as parameter.

    Returns:
    --------
    10^(mean absolute error)

    """

    return 10**mean_absolute_error(y_pred, y)
```

3.2 mape(): Mean Absolute Percent Error

This function returns the *Mean Absolute Error* expressed in percentage, 100% representing a perfect prediction (without any errors).

Expressing the score of prediction in percentage is an advantage as it takes care of the context. Saying that the model has a MAPE of 88% gives more representative information on the model performance than saying that MAE is 1.5 km/h.

```
def mape(y_pred, y) -> float:
    """
    Define a performance metric in percentage

    Returns:
    --------
    mean absolute percentage score
```

```
    """

    # Return percentage value
    return 100 - np.mean(100 * (mean_absolute_error(y_pred, y) / y))
```

4 Define a baseline

In order to evaluate our models, one quite easy and direct method would be to compare them to a simple baseline built with *sklearn.dummy.DummyRegressor*

As I've decided to evaluate my models against the *mean absolute error*, let's use this *DummyRegressor* with parameter *strategy='mean'*

[3]:
```
from sklearn.dummy import DummyRegressor

# Load X and y dataset
X_tr, y_tr, X_va, y_va=load_Xy(frac=FRAC_VALUE_FOR_ML)

dummy = DummyRegressor(strategy='mean')
dummy.fit(X_tr, y_tr)

y_pred=dummy.predict(X_va)

print("Dummy classifier accuracy in km/h       : {:.2f} km/h".format(mae(y_pred,␣
 ↪y_va)))
print("Dummy classifier accuracy in percentage : {:.2f} %".format(mape(y_pred,␣
 ↪y_va)))
```

```
Dummy classifier accuracy in km/h       : 1.48 km/h
Dummy classifier accuracy in percentage : 84.02 %
```

[4]:
```
# Save model for later use
save_model(model=dummy, name='dummy')
```

```
Saving model dummy to ./data/model-dummy.sav using 'pickle' library
```

5 GridSearchCV scoring function

GridSearchCV use internal scoring function to determine prediction performance for each parameters tested, and performs a classification of the parameters combination to determine the best one.

I've decided to not use this internal scoring function (which by defaut is an *R2* function for regression) and define my own scoring function using sklearn.metrics.make_scorer and the *mape()* function defined above.

The two following lines will be added to my_utils library to make it available in ML training Notebooks.

Using this scoring function will let me draw more understable train graphs as the results on the Y-axis will be expressed in *percentage of performance*.

```
from sklearn.metrics import make_scorer
custom_scorer = make_scorer(mape, greater_is_better=True)
```

> Note: The *greater_is_best* parameter to *True* will instruct the GridSearchCV objects that best result is the one with the higher *MAPE* value

6 GridSearchCV results plotting function

As I've decided to use *RandomizedSearchCV* and *GridSearchCV* to tune some hyperparameters, I've reimplemented a function initially coded during my cource #4 project used to draw the results from the search (using *xSearchCV()cv_results_* property).

This will help to visualy seek for the best hyperparameters.

Header of the function is copied below, implementation is available in my_utils library.

```
def plot_grid_search_results(results_df,
                             x_param,
                             y_param=['mean_test_score', 'mean_train_score'],
                             semilogx=True,
                             xlabel='',
                             ylabel='Score (%)',
                             title='GridSearch results',
                             figsize=(15,10),
                             std_params={'mean_test_score': 'std_test_score'},
                             std_factor=1,
                             show_best_result=['mean_test_score'],
                             greater_is_best=False
                            ) -> None:
    """
    Function to graph data points from GridSearchCV results., used to graph
    the mean test score of a GridSearchCV fitted object.

    Mandatory parameters are:
        results_df: A dataframe built from GridSearchCV.cv_results_ property
        x_param: The column name of the results_df dataframe to be used as X axis
        y_param: An array of column to be plotted on the Y axis.

    Optionnal parameters:
        semilogx: If True, the X data points are plotted using a log10 scale
        xlabel: Label of the X axis
        ylabel: Label of the Y axis
        title: Title of the graph
        figsize: Size ot the graph
        std_param: A dict with key=y_param element and value the corresponding
                   standard deviation column name.
                   This parameters is used to draw the std deviation of the
```

```
                y_params as a filled area around the data plot
    std_factor: This parameter is used to amplify the standard deviation
                when building the std dev filled area. Default value is 1 and
                changing increasing it allows displaying standard 'small'
                deviation behaviours.
                Be warn that when changing this parameter to a value other
                that 1, the filled area does not represent absolute values
                but a trend of it.

    The function will also determine, for each of the y_param to be plotted,
    which is the plot with the highest y_param value, and use the coordinates
    to draw a red cross on the plotted line, along with horizontal and vertical
    lines to the X and Y axis.

    For that purpose, the function first sort the results_df dataframe using
    the x_param column in ascending order.

    Returns:
    --------
    None

    """
```

7 Functions to save and load fitted models

Same as I've done in my course #4 project, I will save the train models on disk using *pickle.dump()* method.

To do so, I've coded into my_utils two functions, *get_model_filename()*, *save_model()* and *load_model()*

7.1 Function headers

7.1.1 get_model_filename()

```
def get_model_filename(model_name) -> str:
    """
    Basic function that will return the filename used to store on disk
    the model passed as parameter

    Returns:
    --------
    str

    """
```

7.1.2 save_model()

```
def save_model(model, name) -> None:
```

```
    """
    Function that saves on disk the fitted model passed as first
    parameter using pickle or keras library, depending on model type
    It uses the function getModelFilename() with the 'name'
    parameter to get the filename where to save the model.

    Returns:
    --------
    None

    """
```

7.1.3 load_model()

```
def load_model(name):
    """
    Function that loads from disk the model of which name is passed
    as first parameter. It uses the function getModelFilename() with
    the 'name' parameter to get the filename from where to load the model.

    Returns:
    --------
    Fitted model

    """
```

7.2 How to use those functions ?

Saving a trained model would ba as easy as:

`save_model('model_name', <fitted_model>)`

Loading it will be:

`model=load_model('model_name')`

Models are save on disk, in the *data* directory with the following pattern:

`model-<model_name>.sav`

8 Time to go...

To the first model: Gradient Descent Based - Ridge Regressor

31.Gradient Descent Based - Ridge Regressor

February 2, 2022

Let's start with our first Machine Learning Model: sklearn.linear_model.Ridge

This model is a Gradient Descent one, which implies that I should use a scaler to normalize my dataset before fitting the model.

I've choosen to tune two of its parameters:

- The regularization strenght alpha, a float number
- The solver, to be choosen from a list provided by the Ridge model

Let's start by loading my_utils and other usefull libraries.

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

[2]:
```
# Scalers
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler

# PCA
from sklearn.decomposition import PCA

# Grid Search
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

# Ridge Model
from sklearn.linear_model import Ridge
```

1 Load train/valid data

Nothing special here, use the *loadXy()* function to load train/valid X and y datasets

[3]:
```
# Load X and y
X_tr, y_tr, X_va, y_va=load_Xy(frac=FRAC_VALUE_FOR_ML)
```

2 Pipeline and Grid Search

As explained in the previous notebook, I will train my model using a Grid Search approach in combination with the Pipeline object.

Pipeline will include the following step:

- scaler, to scale the dataset if required by the model
- pca, to apply feature dimension reduction using PCA()
- model, the model to train

I'll do first a randomized search using as much parameters values as possible and, based on the result obtained, run a GridSearchCV with the best parameter intervals found with the randomized process.

2.1 Define pipeline and grid parameters

Scaler and PCA will be configured in the Pipeline and manage as grid parameters:

2.1.1 Scaler

- StandardScaler()
- MinMaxScaler()
- RobustScaler()

  Note: Scaler selection is inspired from that article: All about Feature Scaling

2.1.2 PCA

- PCA(0.8)
- PCA(0.9)
- PCA(0.95)

For both PCA and Scaler, the pipeline will be configured to let the search process disable them (setting *None* as grid parameter), even if it's recommended to scale data when using *Gradient Descent* model types.

[4]:
```
# Define list of scaler used in grid search
scalers=[StandardScaler(), MinMaxScaler(), RobustScaler(), None]

# Define list of PCA reduction used in grid search
pcas=[PCA(0.8), PCA(0.9), PCA(0.95), None]
```

[5]:
```
# Initialize Ridge ML model
model = Ridge()

# Define pipeline with scaler, pca and model choosen
pipe = Pipeline(steps=[
    ("scaler", StandardScaler()),
    ("pca", PCA()),
    ("model", model),
])

# Define base grid search parameters
param_grid = {
    "scaler": scalers,
    "pca": pcas
}
```

2.2 RandomizedSearchCV with large parameters scope

I will start with a *RandomizedSearchCV* on a very large scope of parameters, setting the iterations number to 100. My objective is to find out which hyperparameters and their interval works better to refine search later on using *GridSearchCV*.

> Note: Instead of using the default scoring function of the *GridSearchCV* class, I will use the one I've defined in my_utils library based on *Mean Absolute Percentage Error*.

[6]:
```
# Set specific model parameters to param_grid
param_grid["model__alpha"]=np.logspace(-4, 15, 200)
param_grid["model__solver"]=['svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag',␣
 ↪'saga']

# Number of random search iterations (the more the slower ;-) )
ITERATIONS=100

# Initialize RandomizedSearchCV object. Note that return_train_score=True
random_search = RandomizedSearchCV(pipe, param_grid, scoring=custom_scorer,␣
 ↪random_state=RANDOM_STATE,
                                   n_iter=ITERATIONS, n_jobs=-1, cv=5,␣
 ↪verbose=1, return_train_score=True)

# Do the random hyperparameter tuning search
random_search.fit(X_tr, y_tr)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   14.9s
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed:  1.4min
[Parallel(n_jobs=-1)]: Done 434 tasks      | elapsed:  3.4min
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed:  3.8min finished
```

[6]:
```
RandomizedSearchCV(cv=5, error_score='raise-deprecating',
          estimator=Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True,
with_std=True)), ('pca', PCA(copy=True, iterated_power='auto',
n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)), ('model', Ridge(alpha=1.0,
copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001))]),
          fit_params=None, iid='warn', n_iter=100, n_jobs=-1,
          param_distributions={'scaler': [StandardScaler(copy=True,
with_mean=True, with_std=True), MinMaxScaler(copy=True, feature_range=(0, 1)),
RobustScaler(copy=True, quantile_range=(25.0, 75.0), with_centering=True,
       with_scaling=True), None], 'pca': [PCA(copy=True, iterated_power='auto',
n_compone...02643e+14, 1.00000e+15]), 'model__solver': ['svd', 'cholesky',
'lsqr', 'sparse_cg', 'sag', 'saga']},
          pre_dispatch='2*n_jobs', random_state=47, refit=True,
          return_train_score=True, scoring=make_scorer(mape), verbose=1)
```

2.2.1 Results

Ok, randomized search will now help me to determine wich parameters I should use for best results.

The following code will display as a *pd.DataFrame* the result, sorted by *mean_test_score*, the score obtained with the test set (test set has been made by the *RandomSearchCV* class using a *KFold* approach).

[7]:
```
# Build a dataframe from search.cv_results_
random_df=pd.DataFrame(random_search.cv_results_)

# Restrict to interesting columns
cols = [f'param_{key}' for key in param_grid.keys()]
cols+=['mean_test_score', 'std_test_score', 'mean_train_score',␣
 ↪'std_train_score']
random_df=random_df[cols]

# Print result
print("Best parameters found:", random_search.best_params_)
print("10 best results:")
random_df.sort_values('mean_test_score', ascending=False).head(10)
```

```
Best parameters found: {'scaler': StandardScaler(copy=True, with_mean=True,
with_std=True), 'pca': None, 'model__solver': 'saga', 'model__alpha':
0.0003739937302478798}
10 best results:
```

[7]:
```
                                        param_scaler param_pca
param_model__alpha  \
```

```
51  StandardScaler(copy=True, with_mean=True, with...      None
0.000373994
65                                                 None      None
0.0471375
21  RobustScaler(copy=True, quantile_range=(25.0, ...      None
5.94113
86      MinMaxScaler(copy=True, feature_range=(0, 1))      None
0.000465953
44  StandardScaler(copy=True, with_mean=True, with...      None
5.94113
7       MinMaxScaler(copy=True, feature_range=(0, 1))      None
0.0471375
40      MinMaxScaler(copy=True, feature_range=(0, 1))      None
17.8343
22                                                 None      None
1.58857
96  StandardScaler(copy=True, with_mean=True, with...      None
5415.87
89  RobustScaler(copy=True, quantile_range=(25.0, ...      None
2800.5

   param_model__solver  mean_test_score  std_test_score  mean_train_score
std_train_score
51                saga        87.802608        0.049827         87.807794
0.011967
65            cholesky        87.802574        0.049867         87.807779
0.011961
21                lsqr        87.802567        0.048638         87.807863
0.011794
86           sparse_cg        87.802476        0.050082         87.807621
0.011915
44                lsqr        87.801730        0.049668         87.806981
0.011957
7                 lsqr        87.795993        0.051412         87.801157
0.013136
40                lsqr        87.794566        0.053687         87.799967
0.011835
22                saga        87.792082        0.056615         87.796927
0.012579
96                saga        87.786525        0.047088         87.791230
0.011854
89                 sag        87.744943        0.046490         87.749055
0.011978
```

2.2.2 What can I do according to the above results ?

Well, decide which parameters I should keep and wich value I should give to them for the next Grid Search (without randomized approach)

- PCA

Top most results have been obtained without any PCA reduction, I will remove this feature reduction from the next grid search.

- Scaler

Regarding the scaler, *RobustScaler()*, *MinMaxScaler()* and *StandardScaler()* seems to perform equally, they are in the top most performer results.

There's one noticeable thing in this grid search result. One of the best performer is obtained without any scaler, certainly because the work done previously to normalize the dataset helps models in their training process. But looking at the *alpha* value used to perform without scaler, 4.10^{-2}, lead me to ignore that result and prefer another result.

Same remark with *StandardScaler()* and *MinMaxScaler()*, their best results implies very low *alpha* values.

For that reason, I would prefer to choose the *RobustScaler()* which has it's best performance combination using an alpha value of 5.95

- Alpha

According to the choices made on the scaler, optimal value would be around *alpha=6*. I'll set the *alpha* search parameter to a range from 10^{-1} to 10^{3} to obtain a nice curve.

- What about solver ?

Difficult to say which solver performs best. Let's use the *lsqr*, the one that performs best with my scaler *RoubustScaler()* choice.

2.3 GridSearchCV on more precise parameter intervals

Build and run a *GridSearchCV* with reduced parameter scope, fixing scaler to *RobustScaler()*, PCA to *None* and solver to *lsqr*

[8]:
```
# Set specific model parameters to param_grid
param_grid["model__alpha"]=np.logspace(-1, 4, 100)
param_grid["model__solver"]=['lsqr']
param_grid['scaler']=[RobustScaler()]
param_grid['pca']=[None]


# Initialize GridSearchCV object. Note that return_train_score=True
grid_search = GridSearchCV(pipe, param_grid, scoring=custom_scorer, n_jobs=-1,␣
 ↪cv=5, verbose=1, return_train_score=True)

# Do the grid search
```

```
grid_search.fit(X_tr, y_tr)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   11.0s
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed:   55.3s
[Parallel(n_jobs=-1)]: Done 434 tasks      | elapsed:  2.2min
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed:  2.5min finished
```

[8]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True,
with_std=True)), ('pca', PCA(copy=True, iterated_power='auto',
n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)), ('model', Ridge(alpha=1.0,
copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001))]),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'scaler': [RobustScaler(copy=True, quantile_range=(25.0,
75.0), with_centering=True,
       with_scaling=True)], 'pca': [None], 'model__alpha': array([1.00000e-01,
1.12332e-01, ..., 8.90215e+03, 1.00000e+04]), 'model__solver': ['lsqr']},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=make_scorer(mape), verbose=1)
```

2.3.1 Results

[9]:
```
# Build a dataframe from search.cv_results_
grid_df=pd.DataFrame(grid_search.cv_results_)

# Restrict to interesting columns
cols = [f'param_{key}' for key in param_grid.keys()]
cols+=['mean_test_score', 'std_test_score', 'mean_train_score',␣
 ↪'std_train_score']

# Restrict to interesting columns
grid_df=grid_df[cols]

print("Best parameters found:", grid_search.best_params_)
print("10 best results:")
grid_df.sort_values('mean_test_score', ascending=False).head(10)
```

```
Best parameters found: {'model__alpha': 23.644894126454073, 'model__solver':
'lsqr', 'pca': None, 'scaler': RobustScaler(copy=True, quantile_range=(25.0,
75.0), with_centering=True,
       with_scaling=True)}
10 best results:
```

[9]:

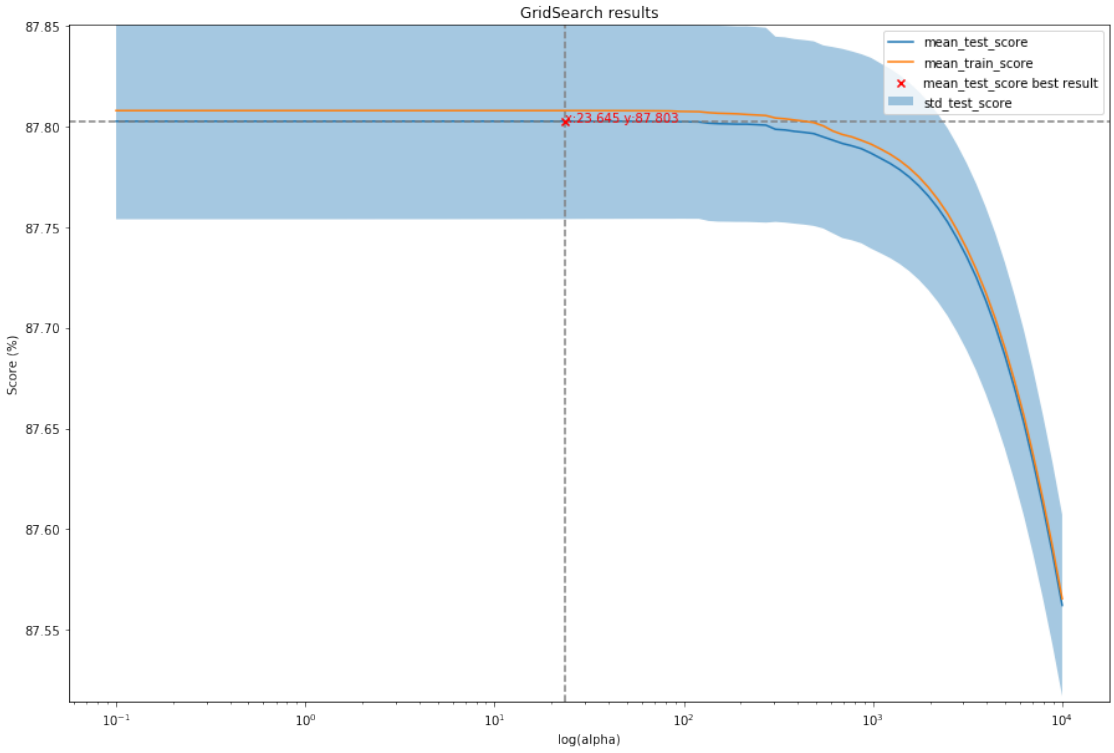
	param_scaler	param_pca	param_model__alpha
47	RobustScaler(copy=True, quantile_range=(25.0, ...	None	23.6449
48	RobustScaler(copy=True, quantile_range=(25.0, ...	None	26.5609
46	RobustScaler(copy=True, quantile_range=(25.0, ...	None	21.049
49	RobustScaler(copy=True, quantile_range=(25.0, ...	None	29.8365
45	RobustScaler(copy=True, quantile_range=(25.0, ...	None	18.7382
44	RobustScaler(copy=True, quantile_range=(25.0, ...	None	16.681
50	RobustScaler(copy=True, quantile_range=(25.0, ...	None	33.516
43	RobustScaler(copy=True, quantile_range=(25.0, ...	None	14.8497
42	RobustScaler(copy=True, quantile_range=(25.0, ...	None	13.2194
41	RobustScaler(copy=True, quantile_range=(25.0, ...	None	11.7681

	param_model__solver	mean_test_score	std_test_score	mean_train_score	std_train_score
47	lsqr	87.802579	0.048563	87.807859	0.011804
48	lsqr	87.802579	0.048552	87.807856	0.011805
46	lsqr	87.802579	0.048574	87.807861	0.011802
49	lsqr	87.802578	0.048538	87.807853	0.011807
45	lsqr	87.802578	0.048584	87.807862	0.011801
44	lsqr	87.802577	0.048592	87.807863	0.011799
50	lsqr	87.802576	0.048524	87.807847	0.011809
43	lsqr	87.802576	0.048599	87.807864	0.011798
42	lsqr	87.802575	0.048606	87.807864	0.011797
41	lsqr	87.802574	0.048613	87.807864	0.011797

2.3.2 Graphical representation of results

Here is a nice curve showing the role of hyperparameters tuning :-)

[10]:
```
# Parameter name to plot
MODEL_PARAM='alpha'

# Plot train and validation curve
results_df=grid_df.sort_values(f'param_model__{MODEL_PARAM}')

plot_grid_search_results(results_df, x_param=f'param_model__{MODEL_PARAM}',␣
 ↪semilogx=True, xlabel=MODEL_PARAM)
```



3 Prediction

Now that I've found the best parameters and get the best estimator via *GridSearchCV()*, let's calculate prediction score of this model.

[11]:
```
# Get best estimator from grid search and predict using X_va
y_pred=grid_search.predict(X_va)

# Get the MAE and MAPE from y_pred
```

```
print("Ridge model mean absolute error         : {:.3f} km/h".format(mae(y_pred,␣
 ↪y_va)))
print("Ridge model mean absolute percent error : {:.2f} %".format(mape(y_pred,␣
 ↪y_va)))
```

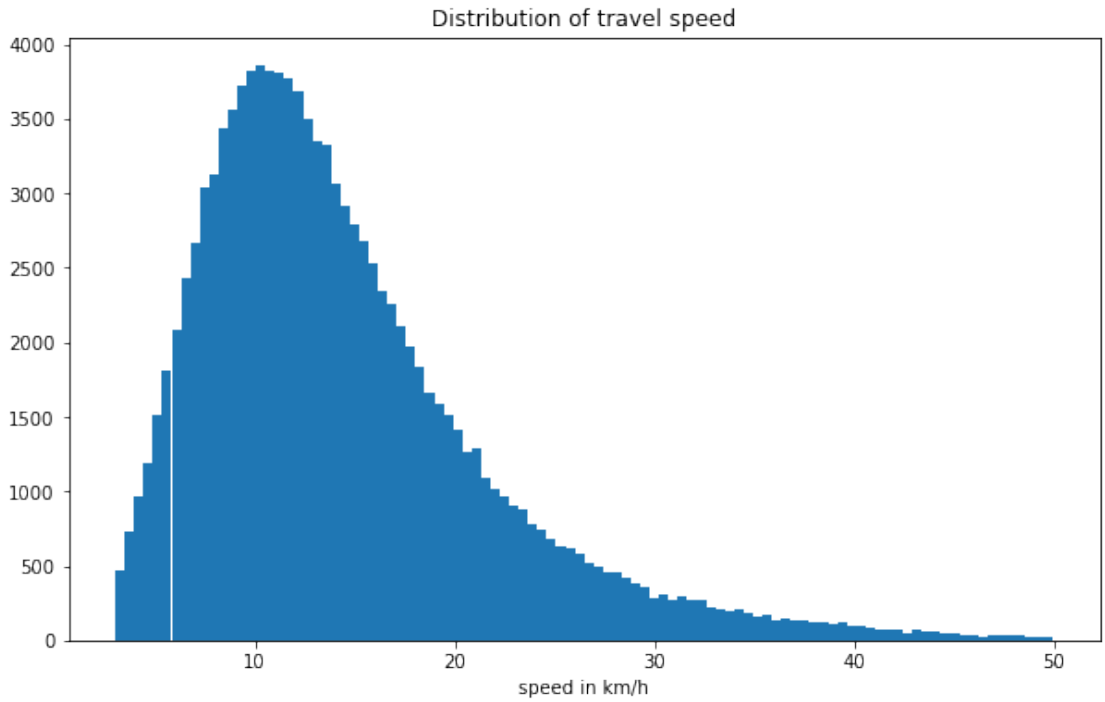
```
Ridge model mean absolute error         : 1.357 km/h
Ridge model mean absolute percent error : 87.55 %
```

3.1 Remark

I suspect since the beginning of my work on data analysis that my dataset might have a bias.

Looking at the y vector result, I can see that most of the travel are centered around 11 km/h.

[12]:
```
plt.figure(figsize=(10,6))
plt.hist(10**y_tr, bins=100)
plt.title('Distribution of travel speed')
plt.xlabel('speed in km/h')
plt.show()
```



Consequence of this dataset characteristics, the bias I suspect is that if you systematically predict a value around the average of travel speed, the same way *DummyRegressor* work, your prediction score would be not so bad.

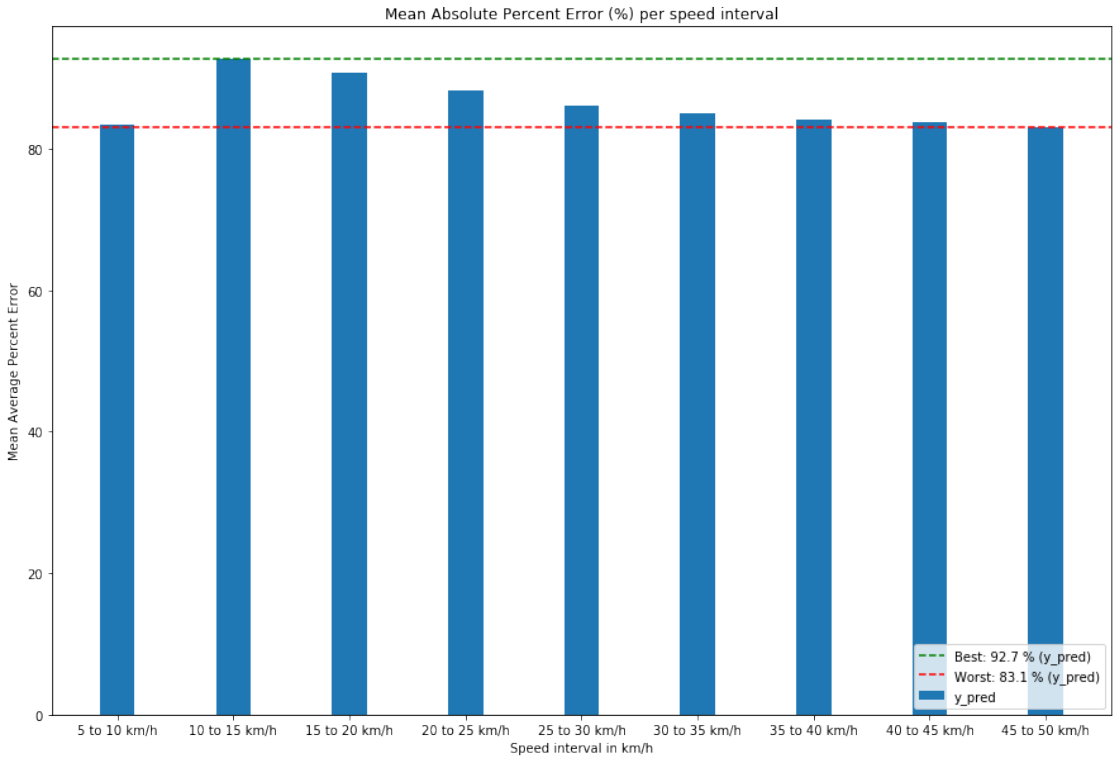
This can be confirmed by the score obtained by the *DummyRegressor()* on this dataset which reach 84%.

3.1.1 Can we observe this bias ?

Let's graph the prediction score of the *Ridge* model I've trained, splitted by groups of 5 km/h real value (the one taken from *y_va* vector): 5 to 10 km/h, 10 to 15 km/h, 15 to 20 km/h and so on...

What I can observe is that score value is the highest in the 10 to 15 km/h interval, the one where most of the travels are, and score decrease when we get far from this optimal interval.

[13]:
```
# Draw graph using function from my_utils library
df=pd.DataFrame({'y': y_va, 'y_pred': y_pred})
draw_mape_per_speed_interval(df,columns=['y_pred'])
```



That's it, I was not so wrong with the travel speed bias.

The previous graphs demonstrate that when speed is predicted between 10 to 15 km/h, the *Mean Average Percent Error* score is the best (more than 90%), and as long we get far from this value, the model prediction is going worst (a bit more than 80%)

4 Save model

Note: The model saved here is the *GridSearchCV* object as when I will reload it, I want to be sure that any pre-processing step like scaling will be applied when computing predictions.

[14]:
```
save_model(grid_search, 'ridge')
```

```
Saving model ridge to ./data/model-ridge.sav using 'pickle' library
```

5 What about PCA ?

We'seen in the Principal Component Analysis notebook that most of the variance (90%) could be explained using nearly half of the features.

Let's run the same grid search as above twice, first run adding a *PCA(0.9)* step to the *Pipeline* and second run adding a *PCA(0.8)* step, and compare the prediction of the best estimators for those two runs with the result obtain without *PCA()*.

5.1 GridSearch with PCA(0.9)

Set *PCA(0.9)* to the *pcas* grid search parameter and rerun the search process.

[15]:
```
# Set pca grid param to PCA(0.9)
param_grid['pca']=[PCA(0.9)]

# Initialize GridSearchCV object. Note that return_train_score=True
pca_90_grid_search = GridSearchCV(pipe, param_grid, n_jobs=-1, cv=5, verbose=1,␣
 ↪return_train_score=True)

# Do the grid search
pca_90_grid_search.fit(X_tr, y_tr)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   22.4s
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done 434 tasks      | elapsed:  3.8min
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed:  4.5min finished
```

[15]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True,
with_std=True)), ('pca', PCA(copy=True, iterated_power='auto',
n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)), ('model', Ridge(alpha=1.0,
copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001))]),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'scaler': [RobustScaler(copy=True, quantile_range=(25.0,
75.0), with_centering=True,
       with_scaling=True)], 'pca': [PCA(copy=True, iterated_power='auto',
n_components=0.9, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)], 'model__alpha':
```

```
array([1.00000e-01, 1.12332e-01, ..., 8.90215e+03, 1.00000e+04]),
'model__solver': ['lsqr']},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=None, verbose=1)
```

[16]:
```
# Get best estimator from grid search and predict using X_va
pca_90_y_pred=pca_90_grid_search.predict(X_va)

# Get the MAE and MAPE from y_pred
print("Ridge model with PCA(0.9) mean absolute error         : {:.3f} km/h".
 ↪format(mae(pca_90_y_pred, y_va)))
print("Ridge model with PCA(0.9) mean absolute percent error : {:.2f} %".
 ↪format(mape(pca_90_y_pred, y_va)))
```

```
Ridge model with PCA(0.9) mean absolute error         : 1.373 km/h
Ridge model with PCA(0.9) mean absolute percent error : 87.05 %
```

5.2 GridSearch with PCA(0.8)

Let's redo the same grid search using *PCA(0.8)*.

[17]:
```
# Set pca grid param to PCA(0.8)
param_grid['pca']=[PCA(0.8)]

# Initialize GridSearchCV object. Note that return_train_score=True
pca_80_grid_search = GridSearchCV(pipe, param_grid, n_jobs=-1, cv=5, verbose=1,␣
 ↪return_train_score=True)

# Do the grid search
pca_80_grid_search.fit(X_tr, y_tr)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   14.9s
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done 434 tasks      | elapsed:  3.7min
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed:  4.3min finished
```

[17]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True,
with_std=True)), ('pca', PCA(copy=True, iterated_power='auto',
n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)), ('model', Ridge(alpha=1.0,
copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001))]),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'scaler': [RobustScaler(copy=True, quantile_range=(25.0,
```

```
75.0), with_centering=True,
       with_scaling=True)], 'pca': [PCA(copy=True, iterated_power='auto',
n_components=0.8, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)], 'model__alpha':
array([1.00000e-01, 1.12332e-01, ..., 8.90215e+03, 1.00000e+04]),
'model__solver': ['lsqr']},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=None, verbose=1)
```

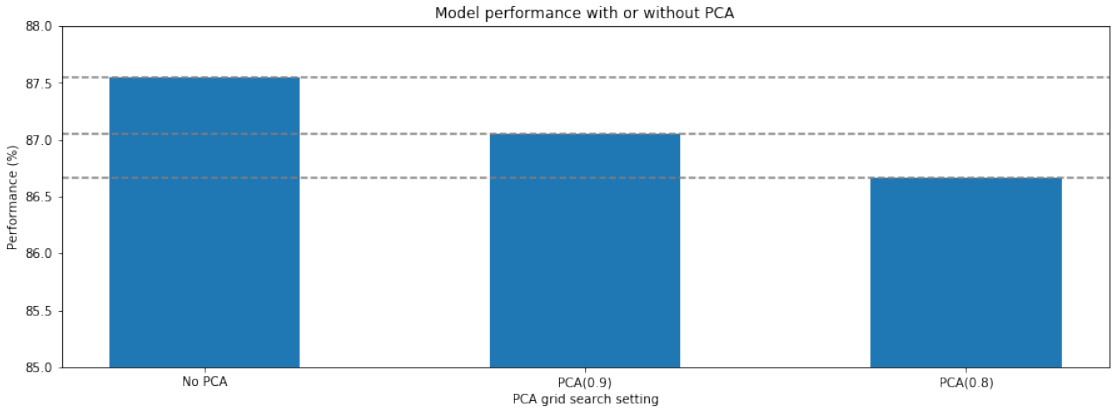
[18]:
```
# Get best estimator from grid search and predict using X_va
pca_80_y_pred=pca_80_grid_search.predict(X_va)

# Get the MAE and MAPE from y_pred
print("Ridge model with PCA(0.8) mean absolute error         : {:.3f} km/h".
 ↪format(mae(pca_80_y_pred, y_va)))
print("Ridge model with PCA(0.8) mean absolute percent error : {:.2f} %".
 ↪format(mape(pca_80_y_pred, y_va)))
```

```
Ridge model with PCA(0.8) mean absolute error         : 1.386 km/h
Ridge model with PCA(0.8) mean absolute percent error : 86.67 %
```

5.3 Compare results

The bar graph below represents the model performance obtained with different *PCA()* configuration.

[19]:
```
df=pd.DataFrame({
    'pca': [0,1,2],
    'prediction': [mape(y_pred, y_va), mape(pca_90_y_pred, y_va),␣
 ↪mape(pca_80_y_pred, y_va)]
})
plt.figure(figsize=(15,5))
plt.bar(df['pca'], df['prediction'], width=0.5)

for i in df['prediction']:
    plt.axhline(y=i, color='grey', linestyle='--')

plt.xticks(ticks=[0,1,2], labels=['No PCA', 'PCA(0.9)', 'PCA(0.8)'])
plt.xlabel("PCA grid search setting")

plt.ylim(85, 88)
plt.ylabel("Performance (%)")

plt.title("Model performance with or without PCA")
plt.show()
```



Compared to the result we've obtained without PCA, conclusion is that using PCA reduction does not perform so bad, but a bit lower than without PCA.

5.4 Save model with PCA(0.9) and PCA(0.8)

[20]:
```
save_model(pca_90_grid_search, 'ridge-pca90')
```

```
Saving model ridge-pca90 to ./data/model-ridge-pca90.sav using 'pickle' library
```

[21]:
```
save_model(pca_80_grid_search, 'ridge-pca80')
```

```
Saving model ridge-pca80 to ./data/model-ridge-pca80.sav using 'pickle' library
```

5.5 Time to go to the next model

A distance based one, the KNeighborsRegressor

32.Distance Based - KNeighborsRegressor

February 2, 2022

Let's start playing with a Distance based ML model, KNeighborsRegressor

This kind of model tries to predict a value by averaging the nearest kneighbors of the value we are looking for. That's why we call them *Distance Based*.

Same remark as the one done with Gradient Descent based model, I should use a scaler to normalize my dataset before fitting the model.

I've choosen to tune two of its parameters:

- The n_neighbors parameter, an int setting the number of kneighbors to search to predict value
- The p parameter to define the *distance* calculation algorythm that could be either p=1 for *manhattan* algorythm, p=2 for *euclidian* one.

Let's start by loading my_utils and other usefull libraries.

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

1 Import libraries

[2]:
```
# Scalers
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler

# PCA
from sklearn.decomposition import PCA

# Grid Search
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
```

```
# KNeighbors regressor Model
from sklearn.neighbors import KNeighborsRegressor
```

2 Load train/valid data

Nothing special here, use the *loadXy()* function to load train/valid X and y datasets.

[3]:
```
# Load X and y
X_tr, y_tr, X_va, y_va=load_Xy(frac=FRAC_VALUE_FOR_ML)
```

3 GridSearch

3.1 Define pipeline and grid parameters

Scaler and PCA will be configured in the Pipeline and managed as grid parameters:

3.1.1 Scaler

- StandardScaler()
- MinMaxScaler()

3.1.2 PCA

- PCA(0.8)
- PCA(0.9)
- PCA(0.95)

For both PCA ans Scaler, the grid search will be configured to let the search process to disable them (setting *None* as grid parameter)

[4]:
```
# Define list of scaler used in grid search
scalers=[StandardScaler(), MinMaxScaler(), None]

# Define list of PCA reduction used in grid search
pcas=[PCA(0.8), PCA(0.9), PCA(0.95), None]
```

[5]:
```
# Initialize KNeighborsRegressor ML model
model = KNeighborsRegressor()

# Define pipeline with scaler, pca and model choosen
pipe = Pipeline(steps=[
    ("scaler", StandardScaler()),
    ("pca", PCA()),
    ("model", model),
])
```

```
# Define base grid search parameters
param_grid = {
    "scaler": scalers,
    "pca": pcas
}
```

3.2 RandomizedSearchCV with large parameters scope

Start first with a *RandomizedSearchCV* on a very large scope of parameters, setting the number random iterations to 100.

[6]:
```
# Set specific model parameters to param_grid
param_grid["model__n_neighbors"]=[*range(1,100,1)]
param_grid["model__p"]=[1,2]

# Number of random search iterations (the more the slower ;-) )
ITERATIONS=100

# Initialize RandomizedSearchCV object. Note that return_train_score=True
random_search = RandomizedSearchCV(pipe,
                                   param_grid,
                                   n_iter=ITERATIONS,
                                   scoring=custom_scorer,
                                   random_state=RANDOM_STATE,
                                   n_jobs=-1,
                                   cv=5,
                                   verbose=1,
                                   return_train_score=True
                                  )

# Do the random hyperparameter tuning search
random_search.fit(X_tr, y_tr)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed: 20.4min
[Parallel(n_jobs=-1)]: Done 176 tasks      | elapsed: 91.3min
[Parallel(n_jobs=-1)]: Done 426 tasks      | elapsed: 390.9min
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed: 460.8min finished
```

[6]:
```
RandomizedSearchCV(cv=5, error_score='raise-deprecating',
          estimator=Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True,
with_std=True)), ('pca', PCA(copy=True, iterated_power='auto',
n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)), ('model',
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
```

```
          metric_params=None, n_jobs=None, n_neighbors=5, p=2,
          weights='uniform'))]),
          fit_params=None, iid='warn', n_iter=100, n_jobs=-1,
          param_distributions={'scaler': [StandardScaler(copy=True,
with_mean=True, with_std=True), MinMaxScaler(copy=True, feature_range=(0, 1)),
None], 'pca': [PCA(copy=True, iterated_power='auto', n_components=0.8,
random_state=None,
  svd_solver='auto', tol=0.0, whiten=False), PCA(copy=True, iterated_powe...80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99],
'model__p': [1, 2]},
          pre_dispatch='2*n_jobs', random_state=47, refit=True,
          return_train_score=True, scoring=make_scorer(mape), verbose=1)
```

3.2.1 Results

Ok, randomized search now help me to determine wich parameter intervals I should focus on.

[7]:
```
# Build a dataframe from search.cv_results_
random_df=pd.DataFrame(random_search.cv_results_)

# Restrict to interesting columns
cols = [f'param_{key}' for key in param_grid.keys()]
cols+=['mean_test_score', 'std_test_score', 'mean_train_score',␣
 ↪'std_train_score']
random_df=random_df[cols]

# Print result
print("Best parameters found:", random_search.best_params_)
print("10 best results:")
random_df.sort_values('mean_test_score', ascending=False).head(10)
```

```
Best parameters found: {'scaler': StandardScaler(copy=True, with_mean=True,
with_std=True), 'pca': PCA(copy=True, iterated_power='auto', n_components=0.95,
random_state=None,
  svd_solver='auto', tol=0.0, whiten=False), 'model__p': 1,
'model__n_neighbors': 21}
10 best results:
```

[7]:

	param_scaler	\
6	StandardScaler(copy=True, with_mean=True, with...	
75	StandardScaler(copy=True, with_mean=True, with...	
58	StandardScaler(copy=True, with_mean=True, with...	
46	StandardScaler(copy=True, with_mean=True, with...	
80	StandardScaler(copy=True, with_mean=True, with...	
97	StandardScaler(copy=True, with_mean=True, with...	
7	StandardScaler(copy=True, with_mean=True, with...	
63	StandardScaler(copy=True, with_mean=True, with...	
3	StandardScaler(copy=True, with_mean=True, with...	

```
23  StandardScaler(copy=True, with_mean=True, with...

                                          param_pca param_model__n_neighbors
param_model__p  \
6   PCA(copy=True, iterated_power='auto', n_compon...                      21
1
75  PCA(copy=True, iterated_power='auto', n_compon...                      10
1
58                                               None                      19
1
46                                               None                      13
2
80                                               None                      15
2
97                                               None                      18
2
7                                                None                      23
1
63  PCA(copy=True, iterated_power='auto', n_compon...                      44
1
3   PCA(copy=True, iterated_power='auto', n_compon...                      13
2
23  PCA(copy=True, iterated_power='auto', n_compon...                      20
2

    mean_test_score  std_test_score  mean_train_score  std_train_score
6         87.671666        0.046494         88.262933         0.017493
75        87.648695        0.045734         88.858427         0.015628
58        87.634064        0.054257         88.283751         0.015487
46        87.610582        0.056624         88.557870         0.015858
80        87.606463        0.051543         88.433153         0.016057
97        87.603257        0.038722         88.289314         0.013853
7         87.599111        0.047268         88.140216         0.012331
63        87.507065        0.032378         87.788879         0.018375
3         87.498386        0.048432         88.454352         0.022355
23        87.489737        0.046934         88.115284         0.025058
```

3.2.2 What can I do according to the above results ?

- PCA

Top most results have been obtained with a PCA component reduction of 95%, I will use this in the next grid search.

- Scaler

StandardScaler is the only one present in the Top10 results. I'll include it in my next grid search as well.

- N-Neighbors

Best number of neighbors seems to be somewhere between 10 and 50. I'll use the (10,50) interval in the grid search.

- P parameters

Difficult to say, it seems that this parameter has no influence. Let keep the two values in the grid search.

3.3 GridSearchCV on more precise parameter intervals

Build and run a *GridSearchCV* with reduced parameter scope, setting scaler to *StandardScaler()* and pca to *PCA(0.95)*

[17]:
```
# Set specific model parameters to param_grid
param_grid["model__n_neighbors"]=[*range(10,50,2)]
param_grid["model__p"]=[1,2]

param_grid['scaler']=[StandardScaler()]
param_grid['pca']=[PCA(0.95)]


# Initialize GridSearchCV object. Note that return_train_score=True
grid_search = GridSearchCV(pipe,
                           param_grid,
                           scoring=custom_scorer,
                           n_jobs=-1,
                           cv=5,
                           verbose=1,
                           return_train_score=True
                          )

# Do the grid search
grid_search.fit(X_tr, y_tr)
```

```
Fitting 5 folds for each of 40 candidates, totalling 200 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed: 27.4min
[Parallel(n_jobs=-1)]: Done 176 tasks      | elapsed: 216.9min
[Parallel(n_jobs=-1)]: Done 200 out of 200 | elapsed: 250.1min finished
```

[17]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True,
with_std=True)), ('pca', PCA(copy=True, iterated_power='auto',
n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)), ('model',
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
```

```
           metric_params=None, n_jobs=None, n_neighbors=5, p=2,
           weights='uniform'))]),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'scaler': [StandardScaler(copy=True, with_mean=True,
with_std=True)], 'pca': [PCA(copy=True, iterated_power='auto',
n_components=0.95, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)], 'model__n_neighbors': [10, 12, 14,
16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48], 'model__p':
[1, 2]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=make_scorer(mape), verbose=1)
```

Results

[18]:
```
# Build a dataframe from search.cv_results_
grid_df=pd.DataFrame(grid_search.cv_results_)

# Restrict to interesting columns
cols = [f'param_{key}' for key in param_grid.keys()]
cols+=['mean_test_score', 'std_test_score', 'mean_train_score',␣
 ↪'std_train_score']

# Restrict to interesting columns
grid_df=grid_df[cols]

print("Best parameters found:", grid_search.best_params_)
print("10 best results:")
grid_df.sort_values('mean_test_score', ascending=False).head(10)
```

```
Best parameters found: {'model__n_neighbors': 14, 'model__p': 1, 'pca':
PCA(copy=True, iterated_power='auto', n_components=0.95, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False), 'scaler': StandardScaler(copy=True,
with_mean=True, with_std=True)}
10 best results:
```

[18]:

	param_scaler	\
4	StandardScaler(copy=True, with_mean=True, with...	
6	StandardScaler(copy=True, with_mean=True, with...	
8	StandardScaler(copy=True, with_mean=True, with...	
2	StandardScaler(copy=True, with_mean=True, with...	
10	StandardScaler(copy=True, with_mean=True, with...	
12	StandardScaler(copy=True, with_mean=True, with...	
14	StandardScaler(copy=True, with_mean=True, with...	
0	StandardScaler(copy=True, with_mean=True, with...	
16	StandardScaler(copy=True, with_mean=True, with...	
7	StandardScaler(copy=True, with_mean=True, with...	

	param_pca	param_model__n_neighbors

```
param_model__p  \
4   PCA(copy=True, iterated_power='auto', n_compon...                    14
1
6   PCA(copy=True, iterated_power='auto', n_compon...                    16
1
8   PCA(copy=True, iterated_power='auto', n_compon...                    18
1
2   PCA(copy=True, iterated_power='auto', n_compon...                    12
1
10  PCA(copy=True, iterated_power='auto', n_compon...                    20
1
12  PCA(copy=True, iterated_power='auto', n_compon...                    22
1
14  PCA(copy=True, iterated_power='auto', n_compon...                    24
1
0   PCA(copy=True, iterated_power='auto', n_compon...                    10
1
16  PCA(copy=True, iterated_power='auto', n_compon...                    26
1
7   PCA(copy=True, iterated_power='auto', n_compon...                    16
2

    mean_test_score  std_test_score  mean_train_score  std_train_score
4         87.697129        0.052080         88.567213         0.016650
6         87.696306        0.052513         88.461889         0.017180
8         87.688607        0.050257         88.373981         0.015038
2         87.683434        0.052239         88.696442         0.015346
10        87.679413        0.046802         88.296879         0.016829
12        87.667389        0.043576         88.230195         0.017581
14        87.655984        0.038663         88.170379         0.016243
0         87.648695        0.045734         88.858427         0.015628
16        87.640310        0.037013         88.117353         0.015237
7         87.633300        0.038349         88.403426         0.014975
```

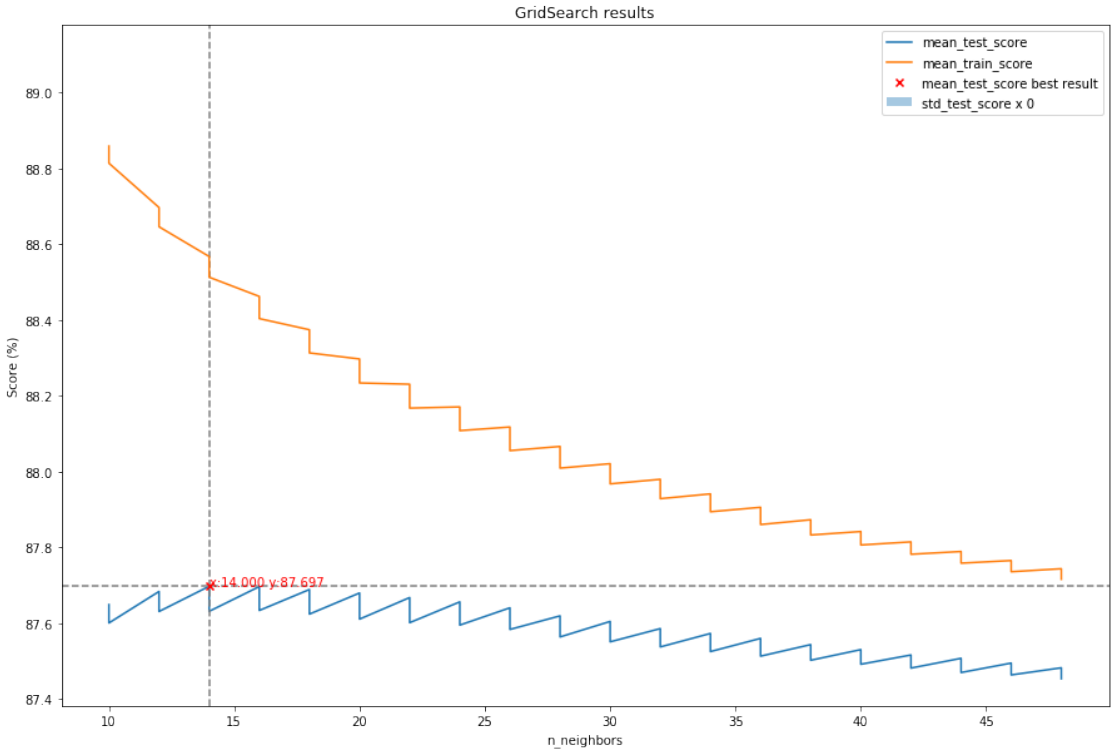
3.3.1 Graphical representation of results

In the following graph, we can see the effect of the grid search with two varying parameters: *n_neighbors* and *p*.

As the graph represent the performance of train/validation set for each combination of *n_neighbors*/*p* valuesused by the grid search, the *spikes* we can observe represent the alternance between 1 and 2 of the *p* parameter.

[19]:
```
# Parameter name to plot
MODEL_PARAM='n_neighbors'

# Plot train and validation curve
results_df=grid_df.sort_values(f'param_model__{MODEL_PARAM}')
```

```
plot_grid_search_results(results_df, x_param=f'param_model__{MODEL_PARAM}',␣
 ↪semilogx=False, xlabel=MODEL_PARAM, std_factor=0)
```

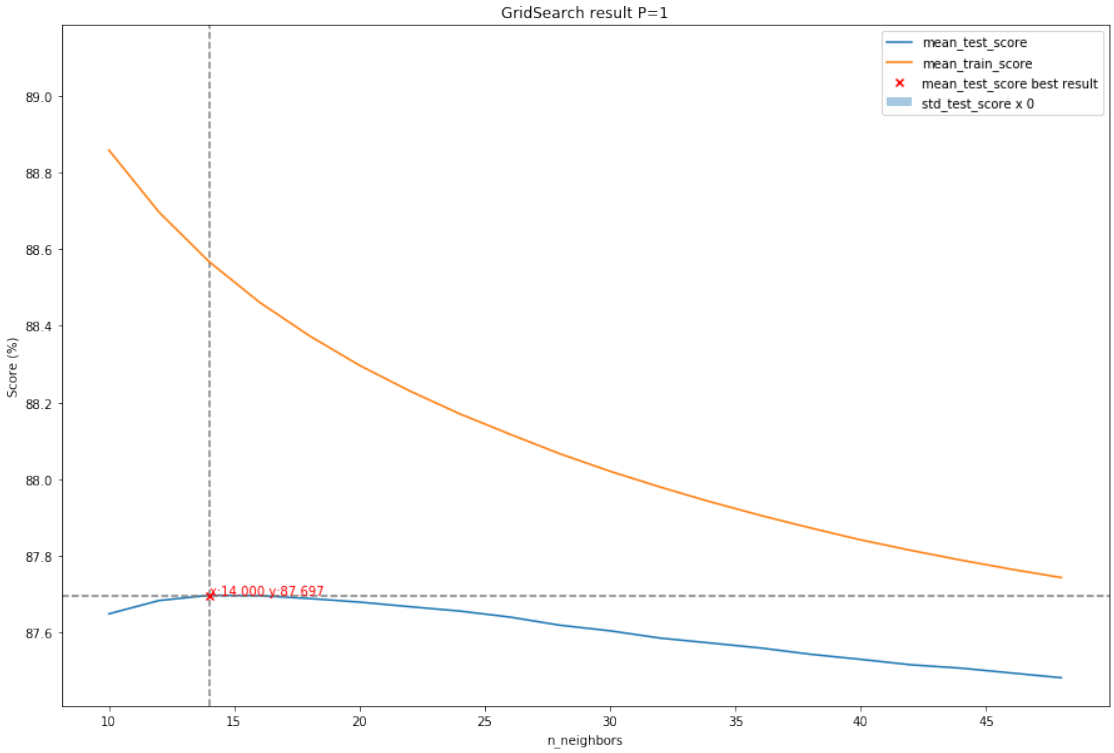


3.3.2 Graphical representation with p=1 only

Results of the previous grid search indicates that p=1 is the best choice.

Here is the same graph as above filtered on values where *p* parameter equals 1.

[20]:
```
# Parameter name to plot
MODEL_PARAM='n_neighbors'

# Filter data point on p=P_VALUE
P_VALUE=1

# Plot train and validation curve
results_df=grid_df[grid_df['param_model__p'] == P_VALUE].
 ↪sort_values(f'param_model__{MODEL_PARAM}')

plot_grid_search_results(results_df, x_param=f'param_model__{MODEL_PARAM}',␣
 ↪semilogx=False, xlabel=MODEL_PARAM, std_factor=0,
                          title='GridSearch result P=1')
```



4 Prediction

Ok, now that we've found the best parameters and get the best estimator via *GridSearchCV()*, let's calculate the *mean_absolute_error* in km/h and %

[21]:
```
# Get best estimator from grid search and predict using X_va
y_pred=grid_search.predict(X_va)

# Get the MAE from y_pred
print("KNeighborsRegressor model mean absolute error         : {:.3f} km/h".
 ↪format(mae(y_pred, y_va)))
print("KNeighborsRegressor model mean absolute percent error : {:.2f} %".
 ↪format(mape(y_pred, y_va)))
```

```
KNeighborsRegressor model mean absolute error         : 1.354 km/h
KNeighborsRegressor model mean absolute percent error : 87.62 %
```

5 Save model

[22]:
```
save_model(grid_search, 'kneighbors')
```

```
Saving model kneighbors to ./data/model-kneighbors.sav using 'pickle' library
```

6 What results could we expect if we reduce the size of training dataset ?

The previous grid search has been done with 10% of the original dataset and took a quite huge of computation time. The question now would be:

- What results do I get if I reduce dataset size to speed up computing time ?

Size of the training dataset have a direct effect on model prediction scores. To illustrate, I will redo a grid search using reduced size of the training dataset and compare results.

> Note: I will fix the *p* paramter to 1 and *n_neighbours* to (10,50) interval to limit computing time

6.1 Train models with different training dataset size

Training dataset fraction that will be selected to train model are:

- 0.05%
- 0.1%
- 0.5%
- 1%
- 2%
- 5%

[53]:
```
# Set specific model parameters to param_grid
param_grid["model__n_neighbors"]=[*range(10,50,2)]
param_grid["model__p"]=[1]
param_grid['scaler']=[StandardScaler()]
param_grid['pca']=[PCA(0.95)]

# Size of training dataset to test
size_to_train=[0.05, 0.1, 0.5, 1, 2, 5]

# All mape calulated will be store in this list
mape_y_pred=[]

for size in size_to_train:

    print("Training dataset fraction used: {}%".format(size))

    # Load X and y
    X_tr, y_tr, X_va, y_va=load_Xy(frac=size/100)
    print("Number of lines in training dataset:", X_tr.shape[0])

    # Initialize GridSearchCV object. Note that return_train_score=True
    reduced_grid_search = GridSearchCV(pipe,
                                       param_grid,
                                       scoring=custom_scorer,
```

```
                                    n_jobs=-1,
                                    cv=5,
                                    verbose=1,
                                    return_train_score=True,
                                    iid=True
                                   )

    # Do the grid search
    reduced_grid_search.fit(X_tr, y_tr)

    # Get best estimator from grid search and predict using X_va
    reduced_y_pred=reduced_grid_search.predict(X_va)

    # Evaluate preformance
    reduced_mape=mape(reduced_y_pred, y_va)

    # Add result to global variable
    mape_y_pred.append(reduced_mape)

    # Get the MAPE from reduced_y_pred
    print("KNeighborsRegressor MAPE : {:.2f} %\n".format(reduced_mape))
```

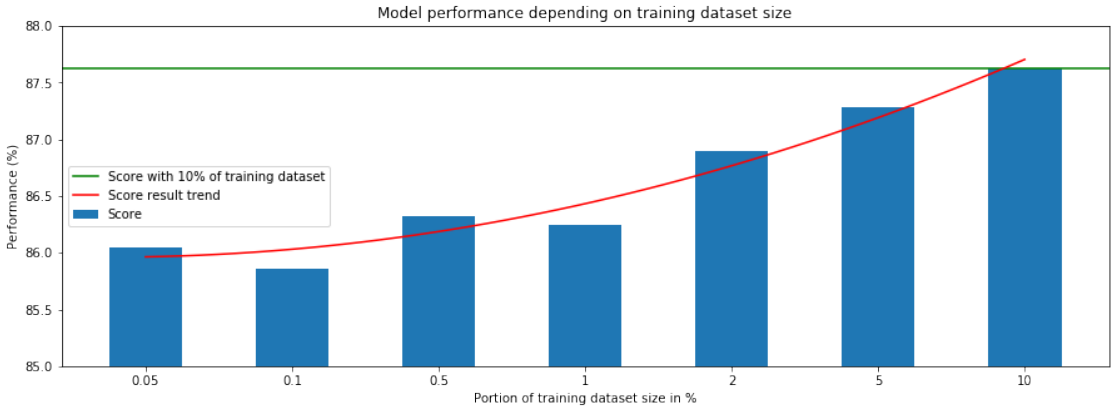
```
Training dataset fraction used: 0.05%
Number of lines in training dataset: 570
Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    4.3s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    4.7s finished

KNeighborsRegressor MAPE : 86.05 %


Training dataset fraction used: 0.1%
Number of lines in training dataset: 1140
Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    1.2s finished

KNeighborsRegressor MAPE : 85.86 %


Training dataset fraction used: 0.5%
Number of lines in training dataset: 5705
Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    7.7s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:   20.8s finished

KNeighborsRegressor MAPE : 86.32 %
```

```
Training dataset fraction used: 1%
Number of lines in training dataset: 11411
Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   33.6s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:  1.5min finished

KNeighborsRegressor MAPE : 86.24 %


Training dataset fraction used: 2%
Number of lines in training dataset: 22822
Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:  3.3min
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:  9.0min finished

KNeighborsRegressor MAPE : 86.89 %


Training dataset fraction used: 5%
Number of lines in training dataset: 57056
Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed: 20.5min
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed: 61.9min finished

KNeighborsRegressor MAPE : 87.28 %
```

6.2 Display performance results per training dataset size

[75]:
```
# Append the performance obtained previously with 10% of training dataset
y_results=mape_y_pred.copy()
y_results.append(87.62)

# Define xticks for bar plt
x_ticks=list(range(len(y_results)))

# Define x_labels
x_labels=size_to_train.copy()
x_labels.append(10)

# Define dataset to be plotted
df=pd.DataFrame({
    'size': x_ticks,
    'prediction': y_results
})
```

```
# Set figure size
plt.figure(figsize=(15,5))

# Plot bar graph using dataset values
plt.bar(df['size'], df['prediction'], width=0.5, label='Score')

# Draw line of the performance obtained previously with 10% of training dataset
plt.axhline(y=87.62, color='green', label='Score with 10% of training dataset')

# Draw a degree=2 poly reg. to show trend
poly = np.poly1d(np.polyfit(x_ticks, df['prediction'], 2))
plt.plot(np.linspace(x_ticks[0], x_ticks[-1]), poly(new_x), color='red',␣
 ↪label='Score result trend')

# Set X & Y axis properties
plt.xticks(ticks=x_ticks, labels=x_labels)
plt.xlabel("Portion of training dataset size in %")
plt.ylim(85, 88)
plt.ylabel("Performance (%)")

# Add legend
plt.legend()

# Set title
plt.title("Model performance depending on training dataset size")

# Display graph
plt.show()
```



Without doubt, the more data in the training set, the better the prediction score is.

On the other size, when the training dataset is very small, the results obtained vary a lot, small

datasets might perform well compared to a bigger one (see 0.05% and 0.1% scores). This can be explained by the fact that reduced training dataset parts are built using random functions. With small portion of it, the probability to have representative train/test values decrease and lead to random results.

Anyway, small dataset even if they demonstrate some noises in prediction, are unable to beat biggest one. All of the previous score are beyond the one obtained with 10% of the *full* training dataset.

7 Time to go to the next model

A distance based one, the RandomForestRegressor

33.Category Based - RandomForestRegressor

February 2, 2022

In this Notebook I will explore the RandomForestRegressor as ML model.

1 How *Random Forest Regressor* works ?

It tries to build sort of binary decision trees, based on feature values, that leads to the prediction.

The main parameter of this ML model is the number of estimators which determines the number of decision trees to use. This is the only parameter I will vary to find the best model.

One cool thing with this *RandomForestRegressor* is that it expose a method to draw the decision tree trained, and a property to identify the top most relevant features for prediction.

> Note 1: To do this graphical exploration about top most relevant features from my dataset, I won't use PCA. Otherwise, I'll loose the feature names. This is an arbitrary choice :-)
>
> Note 2: I won't use any scaling preprocessing as with category based model like decision trees, it doesn't make a lot of sense.

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

2 Import usefull libraries

[2]:
```
# Scalers
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler

# PCA
from sklearn.decomposition import PCA

# Grid Search
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
```

```
from sklearn.pipeline import Pipeline

# RandomForest regressor Model
from sklearn.ensemble import RandomForestRegressor

# Import tools needed for tree visualization
from sklearn.tree import export_graphviz
from graphviz import Source
from PIL import Image
# Disable DecompressionBombWarning warnings (raised when handling large picture␣
 ↪files)
Image.warnings.simplefilter('ignore', Image.DecompressionBombWarning)
```

3 Load train/valid data

Nothing special here, use the *loadXy()* function to load train/valid X and y datasets.

[3]:
```
# Load X and y
X_tr, y_tr, X_va, y_va=load_Xy(frac=FRAC_VALUE_FOR_ML)
```

4 GridSearch

4.1 Define pipeline and grid parameters

Scaler and PCA will be removed from pipeline as I do not want to use them (see introduction notes above for more explanation on that choice).

Regarding the model parameters, I've decided to make grid search varying only the *n_estimators* parameter.

> Note: Using a *Pipeline* object become useless here as I have only one step to process(the ML model). Anyway, I'll keep the code as is in case I'd like to add any step later in this project.

[4]:
```
# Initialize RandomForestRegressor ML model
model = RandomForestRegressor()

# Define pipeline with scaler, pca and model choosen
pipe = Pipeline(steps=[
    ("model", model),
])

# Define an empty base grid search parameters
param_grid = {}
```

4.2 RandomizedSearchCV with large parameters scope: Wont'do

As I will perform a grid search using only one model parameter, *n_estimators*, and because this parameter is an integer with default value is 100, I've decided to don't use a randomized approach and directly perform a grid search with *n_estimators* set to a range going from 1 to 300 with an increment step of 5.

4.3 GridSearchCV and training curves

Grid search will be set with one parameter only: *n_estimators=range(1, 300, 10)*

[5]:
```
# Set specific model parameters to param_grid
param_grid["model__n_estimators"]=[*range(1,300,10)]

# Initialize GridSearchCV object. Note that return_train_score=True
grid_search = GridSearchCV(pipe,
                           param_grid,
                           scoring=custom_scorer,
                           n_jobs=-1,
                           cv=5,
                           verbose=1,
                           return_train_score=True)

# Do the grid search
grid_search.fit(X_tr, y_tr)
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:  2.4min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed: 62.9min finished
```

[5]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('model', RandomForestRegressor(bootstrap=True, criterion='mse',
max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators='warn', n_jobs=None,
           oob_score=False, random_state=None, verbose=0, warm_start=False))]),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'model__n_estimators': [1, 11, 21, 31, 41, 51, 61, 71, 81,
91, 101, 111, 121, 131, 141, 151, 161, 171, 181, 191, 201, 211, 221, 231, 241,
251, 261, 271, 281, 291]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=make_scorer(mape), verbose=1)
```

4.3.1 Results

[6]:
```
# Build a dataframe from search.cv_results_
grid_df=pd.DataFrame(grid_search.cv_results_)

# Restrict to interesting columns
cols = [f'param_{key}' for key in param_grid.keys()]
cols+=['mean_test_score', 'std_test_score', 'mean_train_score',␣
 ↪'std_train_score']

# Restrict to interesting columns
grid_df=grid_df[cols]

print("Best parameters found:", grid_search.best_params_)
print("10 best results:")
grid_df.sort_values('mean_test_score', ascending=False).head(10)
```

```
Best parameters found: {'model__n_estimators': 291}
10 best results:
```

[6]:

	param_model__n_estimators	mean_test_score	std_test_score	mean_train_score	std_train_score
29	291	89.529453	0.053548	96.102936	0.007947
28	281	89.528274	0.055131	96.102925	0.008224
26	261	89.527594	0.052574	96.099724	0.008555
21	211	89.527442	0.060557	96.094201	0.007926
23	231	89.526604	0.054122	96.097457	0.007530
22	221	89.525312	0.055115	96.093459	0.008674
27	271	89.524911	0.050797	96.099680	0.008117
24	241	89.523926	0.057866	96.096483	0.007875
20	201	89.523828	0.054930	96.094615	0.006564
19	191	89.523009	0.057539	96.090387	0.006698

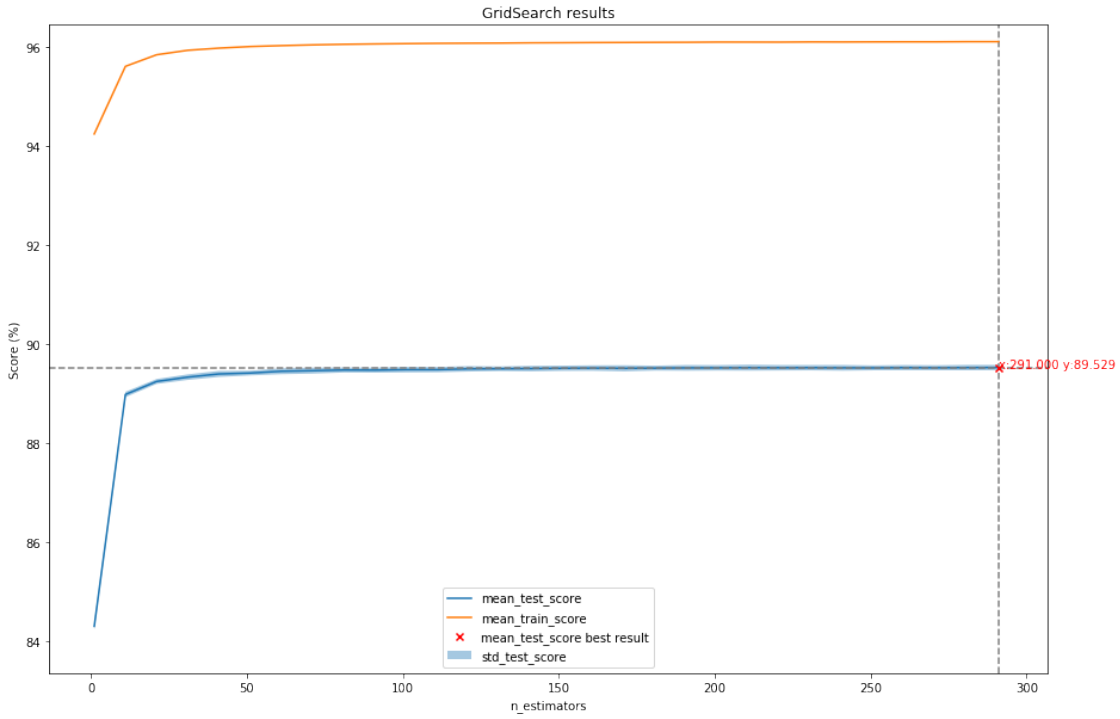
4.3.2 Graphical representation of results

Looking at the curve below, I can say that the *RandomForest* model starts to reach it's best performance when *n_estimators* reach a value of 50~60.

Saying that the model default value for *n_estimators* is 100, I could have use it as is, without per-

forming any grid search.

[7]:
```
# Parameter name to plot
MODEL_PARAM='n_estimators'

# Plot train and validation curve
results_df=grid_df.sort_values(f'param_model__{MODEL_PARAM}')

plot_grid_search_results(results_df, x_param=f'param_model__{MODEL_PARAM}',␣
 ↪semilogx=False, xlabel=MODEL_PARAM)
```



4.4 Prediction

Now that we've found the best parameters and get the best estimator via *GridSearchCV()*, let's calculate the prediction score of this model.

[8]:
```
# Get best estimator from grid search and predict using X_va
y_pred=grid_search.predict(X_va)

# Get the MAE and MAPE from y_pred
print("RandomForestRegressor model mean absolute error         : {:.3f} km/h".
 ↪format(mae(y_pred, y_va)))
print("RandomForestRegressor model mean absolute percent error : {:.2f} %".
 ↪format(mape(y_pred, y_va)))
```

```
RandomForestRegressor model mean absolute error         : 1.296 km/h
RandomForestRegressor model mean absolute percent error : 89.40 %
```

4.5 Save model

[9]:
```
save_model(grid_search, 'random-forest')
```

```
Saving model random-forest to ./data/model-random-forest.sav using 'pickle'
library
```

5 Explore and graph the *RandomForestRegressor* model trained

Models trained during grid search process are stored in *Pipeline* objects stored in *GridSearchCV* one.

The best trained *RandomForestRegressor* object can be grabbed from the *Pipeline* object retrieved from *GridSearchCV* using its *best_estimator_* property.

As the *Pipeline* returned contains only one step (the model to train), I can get the best *RandomForestRegressor* trained model using the following syntax:

[10]:
```
# Get best estimator Pipeline from GridSearchCV
pipeline=grid_search.best_estimator_

# Get RandomForestRegressor trained model
rf=pipeline.named_steps['model']

print("Number of estimators in the best model trained:", len(rf.estimators_))
```

```
Number of estimators in the best model trained: 291
```

As expected, the number of estimators matches the choice made by the grid search.

5.1 Visualizing the Decision Tree

5.1.1 Display Decision Tree as a picture

Using the *export_graphviz()* method, I will obtain a graphical representation of the Decision Tree for one of the estimators of the *RandomForestRegressor* model trained before.

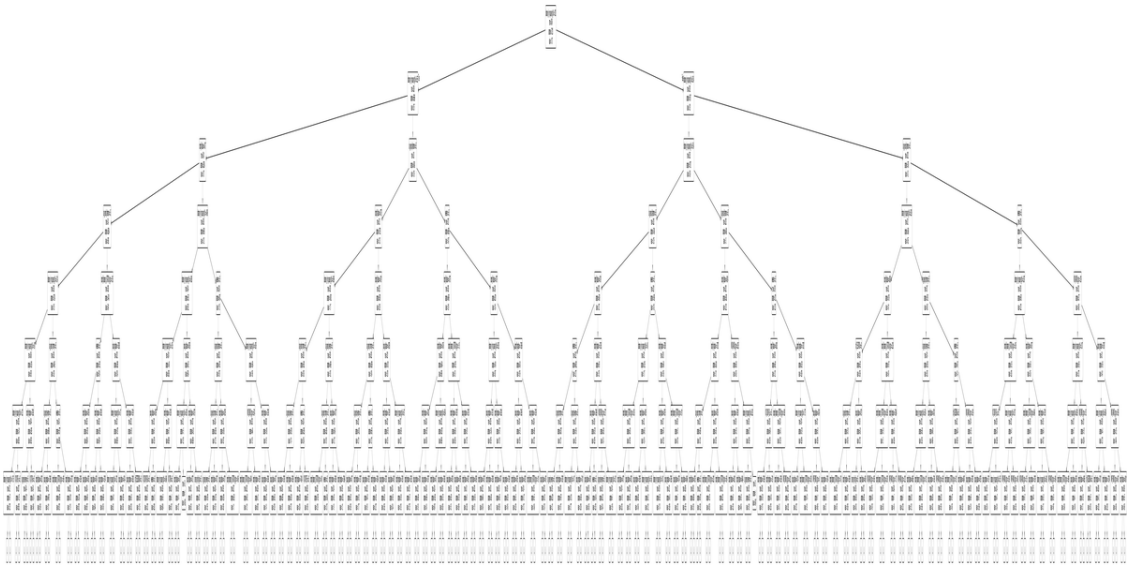
> Note 1: By default, I use the first estimator of the model (*model.estimators_[0]*)
>
> Note 2: I will limit the tree display to a max depth of 7, due to computation limit.

[11]:
```
def draw_estimator(estimator, max_depth=None, resize=[1200,600]) -> Image.Image :
    """
    Function used to draw a graphical representation of a RandomForestRegresor␣
 ↪estimator.

    Returns:
    --------
```

```
    An PIL.Image object representing the graph

    """

    # Define the name of the temporary PNG file created
    png_filename='random-forest-tree'

    # Load feature list
    _, feature_list, *_ = load_dataset(verbose=False)

    # USe Source class to create the PNG file
    graph=Source( export_graphviz(estimator, out_file=None, max_depth=max_depth,␣
↪feature_names=feature_list, rounded=True), format='png', directory=IMG_PATH,␣
↪filename=png_filename)
    graph.render(png_filename,view=False)

    # Return PIL.Image from the PNG file created before
    return Image.open(os.path.join(IMG_PATH, png_filename+'.png')).
↪resize(resize, Image.ANTIALIAS)
```

[12]:
```
# Draw estimator with depth 7
draw_estimator(rf.estimators_[0], max_depth=7)
```

[12]:

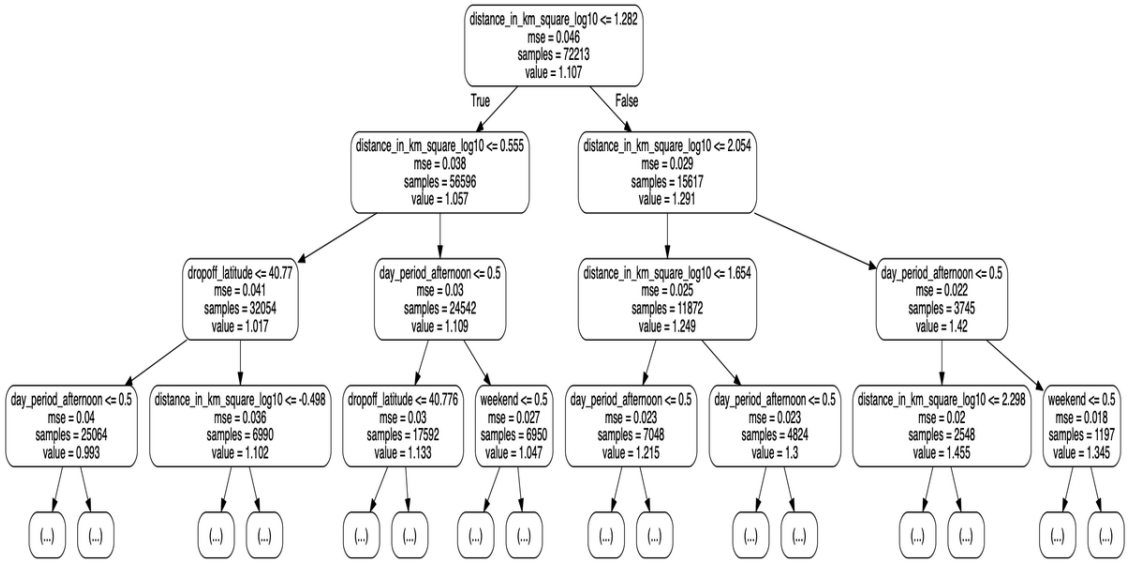


Of course, the previous graph is not really usable, as it is a quite huge graph that I was forced to resize.

5.1.2 Limit to the first three levels

Display again this Decision Tree but using a *max_depth* value of 3 to see how the model *classify* datasets entries to preduct results.

[13]:
```
# Draw estimator with max_depth=3
draw_estimator(rf.estimators_[0], max_depth=3)
```

[13]:


That's more clear, now we can see on this graph how the features and their values are used to predict results.

But wait, is there some features that are more important in terms of prediction ?

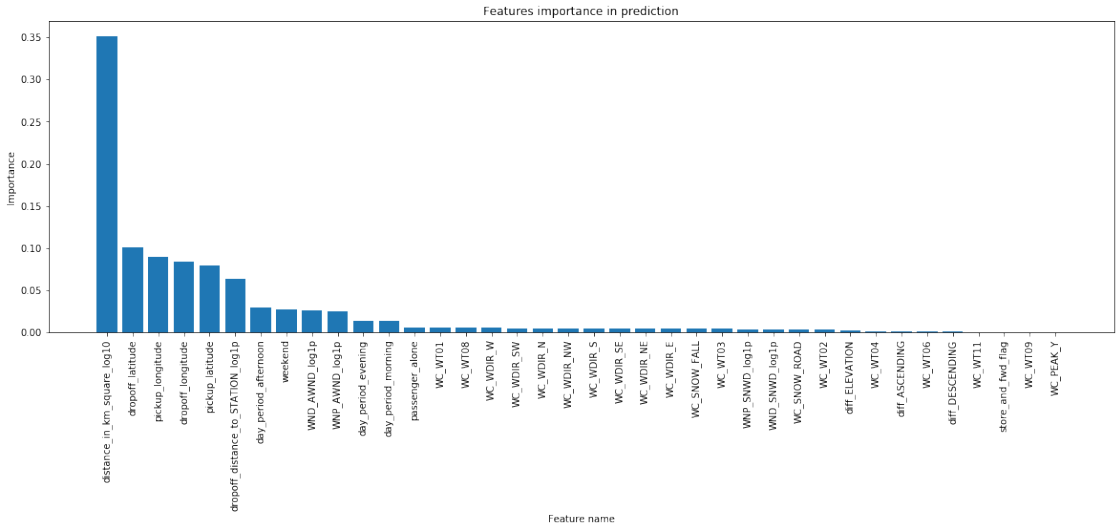
5.2 Top most relevant features

The *RandomForestRegressor* model expose a property named *feature_importances_* which give, for each feature, their relative importance in the Decision Tree. Looking at this property, we can identify which features are the most important one to predict result.

To show this importance par feature in a smart manner, I'll draw a bar graph with features on X-axis and importance on Y-axis, the most important feature being on the left side of the X-axis.

[14]:
```
# Load feature name list
_, feature_list, *_ = load_dataset(verbose=False)

# Get feature importances
importances = rf.feature_importances_

# Map feature list name with feature importance in a dict
#feature_importances = [(feature, importance) for feature, importance in␣
 ↪zip(feature_list, importances)]
```

```
feature_importances = { feature: importance for feature, importance in␣
 ↪zip(feature_list, importances)}
feature_importances={k: v for k, v in sorted(feature_importances.items(),␣
 ↪key=lambda item: item[1], reverse=True)}

plt.figure(figsize=(20,6))

# Calculate x_values range (trick to use when X-axis is made of strings)
x_values = list(range(len(importances)))

# Make a bar chart
plt.bar(x_values, list(feature_importances.values()))

# Tick labels for x axis, replacing range values
plt.xticks(x_values, feature_importances.keys(), rotation='vertical')

# Axis labels and title
plt.ylabel('Importance')
plt.xlabel('Feature name')
plt.title('Features importance in prediction')
plt.show()
```



Interesting results, the top most important feature is the distance in km of the Taxi trip, followed by the pickup and dropoff location, and the dropoff distance to the nearest weather station.

A good demonstration of how it has been usefull to engineer new features from the original one :-)

6 Time to go to the next model

A Neural Network one, the MLPRegressor

34.Neural Network - MLPRegressor

February 2, 2022

In this Notebook, I will play with the MLPRegressor, a Neural Networks based regressor used for prediction of continuous results.

This Machine Learning model is based on at least three layers neural network:

- input and output layers, based on shapes of the input feature and output results. We can not change any parameters for this two layers.
- One or more hidden layers that can be defined using the *hidden_layer_sizes* parameter.

Which hidden layer sizes should I define ?

This is the role of the grid search here, try to find the best combination of layers to obtain the best performance.

Along with the *hidden_layer_sizes* parameters, there is some more parameters to play with to optimize the model: activation and solver algorythm, alpha, the L2 penalty, and many more.

What I'll do in this Notebook is first a randomized search with a fix *alpha* value set to default (10^{-4}) to identify wich values I should use for the other parameters, especially the *hidden_layer_size* one. When this is done, I will vary the *alpha* parameter in a grid search to find the best estimator.

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

1 Import usefull libraries

[2]:
```
# Scalers
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler

# PCA
from sklearn.decomposition import PCA
```

```
# Grid Search
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

# MLPRegressor Model
from sklearn.neural_network import MLPRegressor
```

2 Load train/valid data

Nothing special here, use the *loadXy()* function to load train/valid X and y datasets

[3]:
```
# Load X and y
X_tr, y_tr, X_va, y_va=load_Xy(frac=FRAC_VALUE_FOR_ML)
```

3 Grid Search

3.1 Define pipeline and grid parameters

Scaler and PCA will be configured in the Pipeline and manage as grid parameters:

3.1.1 Scaler

Scaler will not be disabled during the grid search, otherwise the model training process might fail due to inifinite value error. So the *None* choice will be excluded from the search process.

- StandardScaler()
- MinMaxScaler()
- RobustScaler()

3.1.2 PCA

The pipeline will be configured to let the search process to disable the PCA (setting *None* in grid parameter) and use the following definitions:

- PCA(0.8)
- PCA(0.9)
- PCA(0.95)

[4]:
```
# Define list of scaler used in grid search
scalers=[StandardScaler(), MinMaxScaler(), RobustScaler()]

# Define list of PCA reduction used in grid search
pcas=[PCA(0.8), PCA(0.9), PCA(0.95), None]
```

[5]:
```
# Initialize Ridge ML model
model = MLPRegressor()

# Define pipeline with scaler, pca and model choosen
pipe = Pipeline(steps=[
    ("scaler", StandardScaler()),
    ("pca", PCA()),
    ("model", model),
])

# Define base grid search parameters
param_grid = {
    "scaler": scalers,
    "pca": pcas
}
```

3.2 RandomizedSearchCV with large parameters scope

For this first randomized search, I will try to vary a lot of different parameters, except the *alpha* one that I will let to its default value 10^{-4}:

- *hidden_layer_size*
- *solver* function
- *activation* function

3.2.1 What value for *hidden_layer_size* parameter ?

I did found a good comment on Stack Exchange wich gives a formula on how much neurons we should have max.

$N_n = \frac{N_s}{\alpha(N_i+N_o)}$

with:

- Nn = Maximum number of neurons.
- Ni = number of input neurons.
- No = number of output neurons.
- Ns = number of samples in training data set.
- α = an arbitrary scaling factor usually 2-10. My choice will go to $\alpha = 5$

[6]:
```
# Fix alpha to scaling factor
alpha=5

# Set number of input neurons to number of features
n_i=X_tr.shape[1]
```

```
# Set number of output to 1 (only one result vector)
n_o=1

# Define number of samples in train dataset
n_s=X_tr.shape[0]

# Calculate the maximum neurons to use (convert to int type to get integer value)
max_neurons=int(n_s/(alpha*(n_i + n_o)))


print("Max neurons to use:", max_neurons)
```

```
Max neurons to use: 585
```

Ok, this function gives the maximum numbers of neuron I should define in my model, but what about the number of hidden layers and their respective size ?

This is where a randomized search could help. I will define all kind of possibilities using the *max_neurons* values by combining them into 1, 2 and 3 hidden layers.

Let's build a list of all that possibilites.

Note: I will set the minimum number of neurons of 2 for each hidden layer defined

[7]:
```
# Build a list based on range of max_neurons
neuron_list=list(range(2, max_neurons+1, 5))

# Build one layer structure
hidden_layers=neuron_list.copy()

# Build two layer structure
for i in neuron_list:
    for j in neuron_list:
        if i*j < max_neurons:
            hidden_layers.append((i,j))

# Build three layer structure
for i in neuron_list:
    for j in neuron_list:
        for k in neuron_list:
            if i*j*k < max_neurons+1:
                hidden_layers.append((i,j,k))
```

I can now define my grid search parameters and start the search process.

[8]:
```
# Set specific model parameters to param_grid
param_grid["model__hidden_layer_sizes"]=hidden_layers
param_grid["model__activation"]=["identity", "logistic", "tanh", "relu"]
param_grid["model__solver"]=["lbfgs", "sgd", "adam"]
param_grid["model__alpha"]=[0.0001]
```

```
# Set the max_iter to a higher value than default to avoid missing convergence␣
 ↪errors
param_grid["model__max_iter"]=[1000]

# Number of random search iterations (the more the slower ;-) )
ITERATIONS=100

# Initialize RandomizedSearchCV object. Note that return_train_score=True
random_search = RandomizedSearchCV(pipe, param_grid, scoring=custom_scorer,␣
 ↪n_iter=ITERATIONS, random_state=RANDOM_STATE,
                                    n_jobs=-1, cv=5, verbose=1,␣
 ↪return_train_score=True)

# Do the random hyperparameter tuning search
random_search.fit(X_tr, y_tr)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:  2.3min
[Parallel(n_jobs=-1)]: Done 176 tasks      | elapsed: 14.8min
[Parallel(n_jobs=-1)]: Done 426 tasks      | elapsed: 104.3min
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed: 164.4min finished
```

```
[8]: RandomizedSearchCV(cv=5, error_score='raise-deprecating',
          estimator=Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True,
with_std=True)), ('pca', PCA(copy=True, iterated_power='auto',
n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)), ('model',
MLPRegressor(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
      ...=True, solver='adam', tol=0.0001,
       validation_fraction=0.1, verbose=False, warm_start=False))]),
          fit_params=None, iid='warn', n_iter=100, n_jobs=-1,
          param_distributions={'scaler': [StandardScaler(copy=True,
with_mean=True, with_std=True), MinMaxScaler(copy=True, feature_range=(0, 1)),
RobustScaler(copy=True, quantile_range=(25.0, 75.0), with_centering=True,
       with_scaling=True)], 'pca': [PCA(copy=True, iterated_power='auto',
n_components=0...u'], 'model__solver': ['lbfgs', 'sgd', 'adam'],
'model__alpha': [0.0001], 'model__max_iter': [1000]},
          pre_dispatch='2*n_jobs', random_state=47, refit=True,
          return_train_score=True, scoring=make_scorer(mape), verbose=1)
```

3.2.2 Results

Display results as a *pd.DataFrame,* sorted by *mean_test_score*.

[9]:
```
# Build a dataframe from search.cv_results_
random_df=pd.DataFrame(random_search.cv_results_)

# Restrict to interesting columns
cols = [f'param_{key}' for key in param_grid.keys()]
cols+=['mean_test_score', 'std_test_score', 'mean_train_score',␣
 ↪'std_train_score']
random_df=random_df[cols]

# Print result
print("Best parameters found:", random_search.best_params_)
print("10 best results:")
random_df.sort_values('mean_test_score', ascending=False).head(10)
```

```
Best parameters found: {'scaler': RobustScaler(copy=True, quantile_range=(25.0,
75.0), with_centering=True,
       with_scaling=True), 'pca': PCA(copy=True, iterated_power='auto',
n_components=0.95, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False), 'model__solver': 'lbfgs',
'model__max_iter': 1000, 'model__hidden_layer_sizes': (82, 7), 'model__alpha':
0.0001, 'model__activation': 'tanh'}
10 best results:
```

[9]:

	param_scaler	param_pca	param_model__hidden_layer_sizes
21	RobustScaler(copy=True, quantile_range=(25.0, ...	PCA(copy=True, iterated_power='auto', n_compon...	(82, 7)
59	RobustScaler(copy=True, quantile_range=(25.0, ...	None	247
53	StandardScaler(copy=True, with_mean=True, with...	None	(127, 2)
54	StandardScaler(copy=True, with_mean=True, with...	PCA(copy=True, iterated_power='auto', n_compon...	(12, 12)
46	RobustScaler(copy=True, quantile_range=(25.0, ...	PCA(copy=True, iterated_power='auto', n_compon...	382
96	StandardScaler(copy=True, with_mean=True, with...		
8	RobustScaler(copy=True, quantile_range=(25.0, ...		
18	RobustScaler(copy=True, quantile_range=(25.0, ...		
41	StandardScaler(copy=True, with_mean=True, with...		
45	RobustScaler(copy=True, quantile_range=(25.0, ...		

```
96  PCA(copy=True, iterated_power='auto', n_compon...
427
8                                                   None
(97, 2)
18  PCA(copy=True, iterated_power='auto', n_compon...
(82, 2)
41                                                  None
(12, 7)
45                                                  None
342

   param_model__activation param_model__solver param_model__alpha
param_model__max_iter  \
21                    tanh               lbfgs             0.0001
1000
59                    relu               lbfgs             0.0001
1000
53                    tanh               lbfgs             0.0001
1000
54                    tanh               lbfgs             0.0001
1000
46                    relu                adam             0.0001
1000
96                    tanh               lbfgs             0.0001
1000
8                     tanh                adam             0.0001
1000
18                    relu                adam             0.0001
1000
41                    tanh               lbfgs             0.0001
1000
45                logistic                adam             0.0001
1000

    mean_test_score  std_test_score  mean_train_score  std_train_score
21        89.628323        0.070906         89.804815         0.090836
59        89.511726        0.053237         90.411369         0.020481
53        89.478863        0.051583         90.060253         0.075515
54        89.291827        0.112548         89.364894         0.060457
46        89.257904        0.144561         89.592318         0.110451
96        89.193455        0.052047         89.759281         0.036657
8         89.176912        0.065193         89.288227         0.053119
18        89.114149        0.160816         89.256688         0.133280
41        89.074996        0.371477         89.169041         0.323249
45        88.969243        0.225623         89.026365         0.180687
```

3.2.3 What can I do according to the above results ?

The results from this randomized search promotes *PCA(0.95)* transformation and *RobustScaler()* as the best pre-processing combination for that model.

Solver function choice looks clear as well: *lbfgs*.

For the activation function and the layer size, it's a bit unclear what's the best combination. It could be one layer and *relu* activation function, or two layers with *tanh* function.

My preference will go with two hidden layers *(82,7)* with *tanh* activation function as it is the combination that performs the best with *RobustScaler()* and *PCA(0.95)*

In conclusion, here are the parameters I will fix to refine my grid search on *alpha* parameter:

- *RobustScaler()*
- *PCA(0.95)*
- activation=*tanh*
- solver=*lbfgs*
- hidden_layers=*(82,7)*

3.2.4 GridSearchCV on more precise parameter intervals

Build and run a *GridSearchCV* with reduced parameter scope, fixing scaler and PCA to *None*.

[10]:
```
# Set specific model parameters to param_grid
param_grid["model__hidden_layer_sizes"]=[(82,7)]
param_grid["model__activation"]=["tanh"]
param_grid["model__solver"]=["lbfgs"]
param_grid["model__alpha"]=np.logspace(-5, 2, 50)


param_grid['scaler']=[RobustScaler()]
param_grid['pca']=[PCA(0.95)]


# Initialize GridSearchCV object. Note that return_train_score=True
grid_search = GridSearchCV(pipe, param_grid, scoring=custom_scorer, n_jobs=-1,␣
 ↪cv=5, verbose=1, return_train_score=True)

# Do the grid search
grid_search.fit(X_tr, y_tr)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed: 29.8min
[Parallel(n_jobs=-1)]: Done 176 tasks      | elapsed: 156.9min
[Parallel(n_jobs=-1)]: Done 250 out of 250 | elapsed: 199.9min finished
```

[10]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('scaler', StandardScaler(copy=True, with_mean=True,
with_std=True)), ('pca', PCA(copy=True, iterated_power='auto',
n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)), ('model',
MLPRegressor(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
      ...=True, solver='adam', tol=0.0001,
       validation_fraction=0.1, verbose=False, warm_start=False))]),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'scaler': [RobustScaler(copy=True, quantile_range=(25.0,
75.0), with_centering=True,
       with_scaling=True)], 'pca': [PCA(copy=True, iterated_power='auto',
n_components=0.95, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)], 'model__hidden_layer_sizes': [(82,
7)], 'mode...       2.68270e+01, 3.72759e+01, 5.17947e+01, 7.19686e+01,
1.00000e+02]), 'model__max_iter': [1000]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=make_scorer(mape), verbose=1)
```

3.2.5 Results

[11]:
```
# Build a dataframe from search.cv_results_
grid_df=pd.DataFrame(grid_search.cv_results_)

# Restrict to interesting columns
cols = [f'param_{key}' for key in param_grid.keys()]
cols+=['mean_test_score', 'std_test_score', 'mean_train_score',␣
 ↪'std_train_score']

# Restrict to interesting columns
grid_df=grid_df[cols]

print("Best parameters found:", grid_search.best_params_)
print("10 best results:")
grid_df.sort_values('mean_test_score', ascending=False).head(10)
```

```
Best parameters found: {'model__activation': 'tanh', 'model__alpha':
0.005179474679231208, 'model__hidden_layer_sizes': (82, 7), 'model__max_iter':
1000, 'model__solver': 'lbfgs', 'pca': PCA(copy=True, iterated_power='auto',
n_components=0.95, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False), 'scaler': RobustScaler(copy=True,
quantile_range=(25.0, 75.0), with_centering=True,
       with_scaling=True)}
10 best results:
```

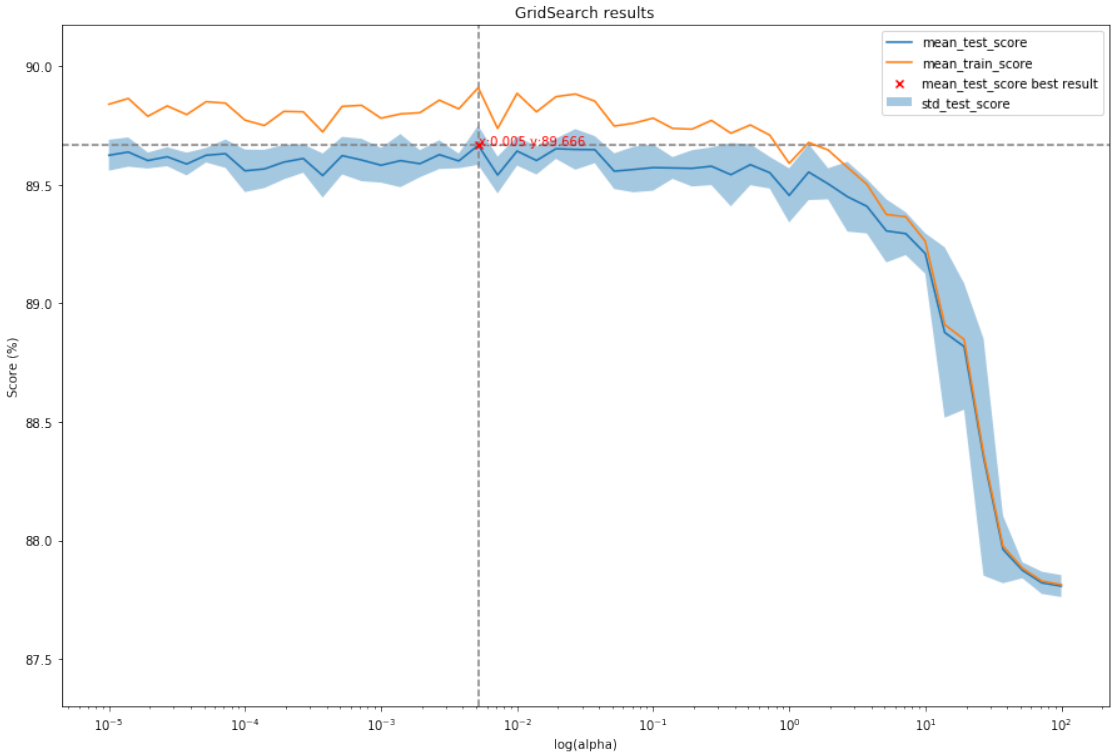
[11]:
```
                                        param_scaler  \
19  RobustScaler(copy=True, quantile_range=(25.0, ...
23  RobustScaler(copy=True, quantile_range=(25.0, ...
24  RobustScaler(copy=True, quantile_range=(25.0, ...
25  RobustScaler(copy=True, quantile_range=(25.0, ...
21  RobustScaler(copy=True, quantile_range=(25.0, ...
1   RobustScaler(copy=True, quantile_range=(25.0, ...
6   RobustScaler(copy=True, quantile_range=(25.0, ...
17  RobustScaler(copy=True, quantile_range=(25.0, ...
5   RobustScaler(copy=True, quantile_range=(25.0, ...
0   RobustScaler(copy=True, quantile_range=(25.0, ...

                                           param_pca
param_model__hidden_layer_sizes  \
19  PCA(copy=True, iterated_power='auto', n_compon...
(82, 7)
23  PCA(copy=True, iterated_power='auto', n_compon...
(82, 7)
24  PCA(copy=True, iterated_power='auto', n_compon...
(82, 7)
25  PCA(copy=True, iterated_power='auto', n_compon...
(82, 7)
21  PCA(copy=True, iterated_power='auto', n_compon...
(82, 7)
1   PCA(copy=True, iterated_power='auto', n_compon...
(82, 7)
6   PCA(copy=True, iterated_power='auto', n_compon...
(82, 7)
17  PCA(copy=True, iterated_power='auto', n_compon...
(82, 7)
5   PCA(copy=True, iterated_power='auto', n_compon...
(82, 7)
0   PCA(copy=True, iterated_power='auto', n_compon...
(82, 7)

   param_model__activation param_model__solver param_model__alpha
param_model__max_iter  \
19                    tanh               lbfgs         0.00517947
1000
23                    tanh               lbfgs           0.019307
1000
24                    tanh               lbfgs           0.026827
1000
25                    tanh               lbfgs          0.0372759
1000
21                    tanh               lbfgs               0.01
1000
```

```
1                     tanh               lbfgs          1.3895e-05
1000
6                     tanh               lbfgs         7.19686e-05
1000
17                    tanh               lbfgs           0.0026827
1000
5                     tanh               lbfgs         5.17947e-05
1000
0                     tanh               lbfgs               1e-05
1000
```

	mean_test_score	std_test_score	mean_train_score	std_train_score
19	89.666126	0.082475	89.909088	0.043575
23	89.651655	0.042575	89.871219	0.066817
24	89.648758	0.085519	89.882346	0.054916
25	89.647843	0.057062	89.852417	0.059214
21	89.642011	0.061247	89.885367	0.041156
1	89.637700	0.061635	89.863654	0.043442
6	89.630690	0.059697	89.843909	0.092532
17	89.626846	0.060687	89.856719	0.037975
5	89.624244	0.030528	89.850026	0.068684
0	89.624235	0.065106	89.839387	0.081099

3.2.6 Graphical representation of results

[12]:
```
# Parameter name to plot
MODEL_PARAM='alpha'

# Plot train and validation curve
results_df=grid_df.sort_values(f'param_model__{MODEL_PARAM}')

plot_grid_search_results(results_df, x_param=f'param_model__{MODEL_PARAM}',␣
 ↪semilogx=True, xlabel=MODEL_PARAM, std_factor=1)
```



4 Prediction

Ok, now that we've found the best parameters and get the best estimator via *GridSearchCV()*, let's calculate the *mean_absolute_error* in km/h and %

[13]:
```
# Get best estimator from grid search and predict using X_va
y_pred=grid_search.predict(X_va)

# Get the MAE from y_pred
print("MLPRegressor model mean absolute error         : {:.3f} km/h".
 ↪format(mae(y_pred, y_va)))
print("MLPRegressor model mean absolute percent error : {:.2f} %".
 ↪format(mape(y_pred, y_va)))
```

```
MLPRegressor model mean absolute error         : 1.295 km/h
MLPRegressor model mean absolute percent error : 89.45 %
```

5 Save model

[14]:
```
save_model(grid_search, 'mlp')
```

```
Saving model mlp to ./data/model-mlp.sav using 'pickle' library
```

5.1 Here we are...

All models choosen for this project have been trained, it's time to display some results in the next Notebook: Results and Communication.

40.Results and Communication

February 2, 2022

After exploring data, engineer it, training and tuning models, time has come to analyze results and compare model performance.

After displaying a bar graph of the model scores to find the best performer, I will explore the way they react when prediction is far from the value predicted by the *DummyRegressor*. This will help conclude about the bias I've suspected, is it real or not. Maybe I'll get a good surprise...

[1]:
```
# Load my_utils.ipynb in Notebook
from ipynb.fs.full.my_utils import *
```

```
Opening connection to database
Add pythagore() function to SQLite engine
Fraction of the dataset used to train models: 10.00%
my_utils library loaded :-)
```

1 Load full *X*/*y* validation set only

For prediction score, I will use the complete validation dataset by setting the *frac* parameter to 1

> Note: We won't use training dataset in this Notebook, I do keep only X and y validation set

[2]:
```
# Load X and y validation vector
_, _, X, y=load_Xy(frac=1)
print("Shape of the validation features :", X.shape)
print("Shape of the validation vector   : ", y.shape)
```

```
Shape of the validation features : (285283, 38)
Shape of the validation vector   :  (285283,)
```

Nice, 285'283 travel for speed prediction evaluation. Let's go !

2 Load models and build a dataframe with their characteristics

I will use the trained models saved on disk to predict values using the *validation* dataset. Those prediction will be used to calculate the *Mean Absolute Error* (in km/h) and the *Mean Absolute Percent Error* (in %) for each model.

I'll then store this *MAE* and *MAPE* in a dataframe, along with the time spent in the grid search process to find their best hyperparameters.

2.1 Compute prediction for each model

After loading each previously fitted models, I will evaluate their prediction results using the X validation features loaded previously.

The result will be stored in a *pandas.DataFrame*, the first column being the y validation values loaded from dataset, and next columns being the results of the column name model.

[3]:
```
model_list=[ 'dummy', 'ridge', 'ridge-pca90', 'ridge-pca80', 'kneighbors',␣
 ↪'random-forest', 'mlp']

# Initialize dict with y validation values
predictions={
    'y': y
}

for i in model_list:
    print("Model", i)
    model=load_model(i)
    print("  Predict values...")
    predictions[i]=list(model.predict(X))
    print("  Process terminated.")

predict=pd.DataFrame(predictions)
```

```
Model dummy
Loading model from  ./data/model-dummy.sav
Model loaded using pickle()
  Predict values...
  Process terminated.
Model ridge
Loading model from  ./data/model-ridge.sav
Model loaded using pickle()
  Predict values...
  Process terminated.
Model ridge-pca90
Loading model from  ./data/model-ridge-pca90.sav
Model loaded using pickle()
  Predict values...
  Process terminated.
Model ridge-pca80
Loading model from  ./data/model-ridge-pca80.sav
Model loaded using pickle()
  Predict values...
  Process terminated.
Model kneighbors
Loading model from  ./data/model-kneighbors.sav
Model loaded using pickle()
  Predict values...
```

```
  Process terminated.
Model random-forest
Loading model from  ./data/model-random-forest.sav
Model loaded using pickle()
  Predict values...
  Process terminated.
Model mlp
Loading model from  ./data/model-mlp.sav
Model loaded using pickle()
  Predict values...
  Process terminated.
```

[4]:
```
# Let's display first lines of this prediction dataset
predict.head(3)
```

[4]:
```
          y     dummy     ridge  ridge-pca90  ridge-pca80  kneighbors  random-
forest       mlp
0  1.547166  1.108545  1.473139     1.534104     1.453226    1.453658
1.517538  1.437138
1  0.932664  1.108545  1.050087     1.050057     1.054383    1.056632
0.991835  1.041009
2  0.972171  1.108545  1.187904     1.110241     1.127601    1.083668
1.005874  1.183418
```

2.1.1 Prediction in km per hour

Remember, the value we obtain here are *np.log10(km/h)* values.

Let's do a copy of this *predict* dataset converting values to km/h

[5]:
```
# Do a copy of the dataset
predict_kmh=predict.copy()

# Compute values in km/h
predict_kmh=10**predict_kmh

# Describe the result
predict_kmh.describe().astype('int')
```

[5]:
```
            y   dummy   ridge  ridge-pca90  ridge-pca80  kneighbors  random-
forest     mlp
count  285283  285283  285283       285283       285283      285283
285283  285283
mean       14      12      13           13           13          13
13      13
std         7       0       4            4            4           4
5       5
min         3      12       3            3            4           4
```

```
3       2
25%         9       12       10            10            10            10
9       9
50%        12       12       12            12            12            12
12      12
75%        17       12       15            14            14            15
16      16
max        49       12       58            79            83            40
46      65
```

Looking at the previous result, I can confirm the suspicion I have concerning the dataset Bias.

The *50%* value for all the models are nearly equal to the *50%* value of the *y* validation set. And if we look at the *75%* and *max*, we can confirm that there is quite important difference.

Except maybe for the *random-forest* models that seems to reserve good surprises.

Let's continue with some graphs to explore the performance of each models.

2.2 Calculate MAE and MAPE, append training time

As explained before, I will build a *pd.DataFrame* object that will contains, for each models, its MAE, MAPE and grid search time in seconds.

This dataset will be used later to draw some graphs.

[6]:
```
# Calculate MAE and MAPE for each model and store it in arrays
mae_result=[]
mape_result=[]
for col in model_list:
    mae_result.append(mae(predict[col], predict['y']))
    mape_result.append(mape(predict[col], predict['y']))

# Grid search time in seconds has been retrieved manually from previous Notebooks
# Order: [ 'dummy', 'ridge', 'ridge-pca90', 'ridge-pca80', 'kneighbors',␣
 ↪'random-forest', 'mlp']
grid_time=[4, 2.5*60, 4.5*60, 4.3*60, 250.1*60, 62.9*60, 199.9*60]

# Build dataframe
model_results=pd.DataFrame(
    {
        'model': model_list,
        'mae': mae_result,
        'mape': mape_result,
        'grid_time': grid_time
    }
)

# display dataframe
model_results
```

[6]:

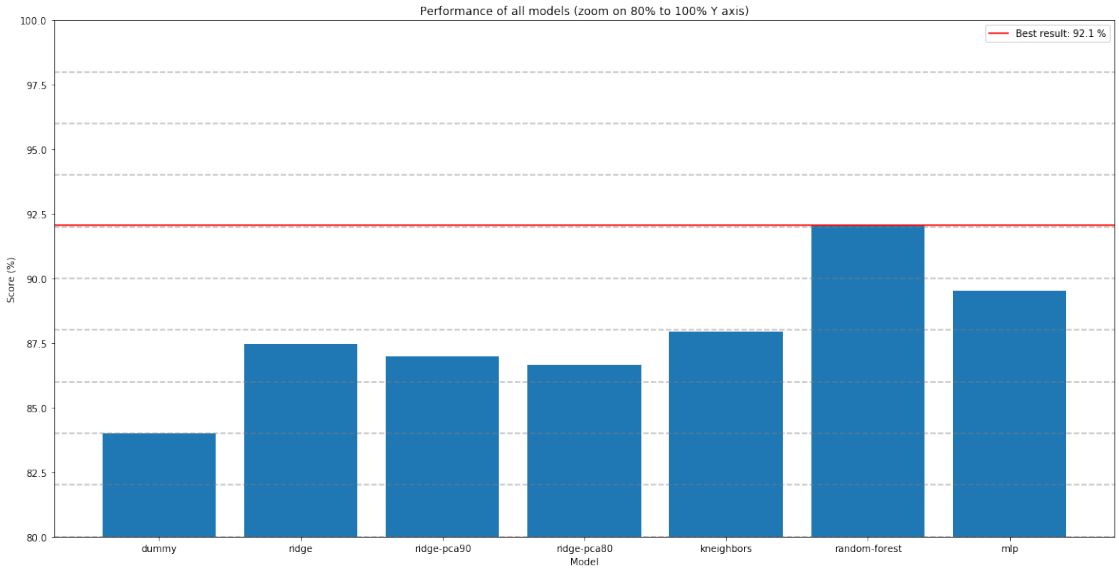
	model	mae	mape	grid_time
0	dummy	1.479496	84.006025	4.0
1	ridge	1.359001	87.474777	150.0
2	ridge-pca90	1.375086	86.994327	270.0
3	ridge-pca80	1.386903	86.644931	258.0
4	kneighbors	1.343402	87.946178	15006.0
5	random-forest	1.214454	92.066561	3774.0
6	mlp	1.292135	89.534899	11994.0

3 A bar graph to compare model performance

The most simple way to determine the best model trained: Use a bar graph to display their *Mean Absolute Percent Error*

[7]:
```
def draw_bar_graph(df=model_results,
                   column='mape',
                   ylabel='Score (%)',
                   xlabel='Model',
                   title='Performance of all models (zoom on 80% to 100% Y␣
 ↪axis)',
                   ylim=[80,100],
                   best_label='Best result',
                   dash_line=[80,100,2]
                  ) -> None:
    # get MAPE from model results dataset
    mean_percent=df[column]

    # Calculate x_values range (trick to use when X-axis is made of strings)
    x_values = list(range(len(mean_percent)))

    # Set figure size
    plt.figure(figsize=(20,10))

    # Make a bar chart
    plt.bar(x_values, mean_percent)

    # Tick labels for x axis, replacing range values
    plt.xticks(x_values, model_list)#, rotation='vertical')

    # Draw horizontal line for best result
    plt.axhline(y=max(mean_percent), color='r', label=best_label)

    # Draw dashed if defined
    if dash_line != None:
        for i in list(range(*dash_line)):
            plt.axhline(y=i, color='grey', linestyle='--', alpha=0.5)
```

```
    # Axis labels, legend and title
    if ylim!= None:
        plt.ylim(*ylim)
    plt.legend()
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    plt.title(title)
    plt.show()


draw_bar_graph(best_label='Best result: {:.1f} %'.
 ↪format(max(model_results['mape'])))
```



Wow… What a good surprise. *RandomForestRegressor* seems to beat other models. That's good news.

3.1 What can we say from the previous graph ?

First, *dummy* regressor performs not so bad, that's a first signal about the bias I suspect. The way I've configured the *DummyRegressor* model, it always predict the mean of the training set, and the training set has a huge concentration of travels between 10 and 15 km/h. Predicting a travel trip in NYC at 12 km/h (this is the prediction done by the *DummyRegressor*) has good chance to be right.

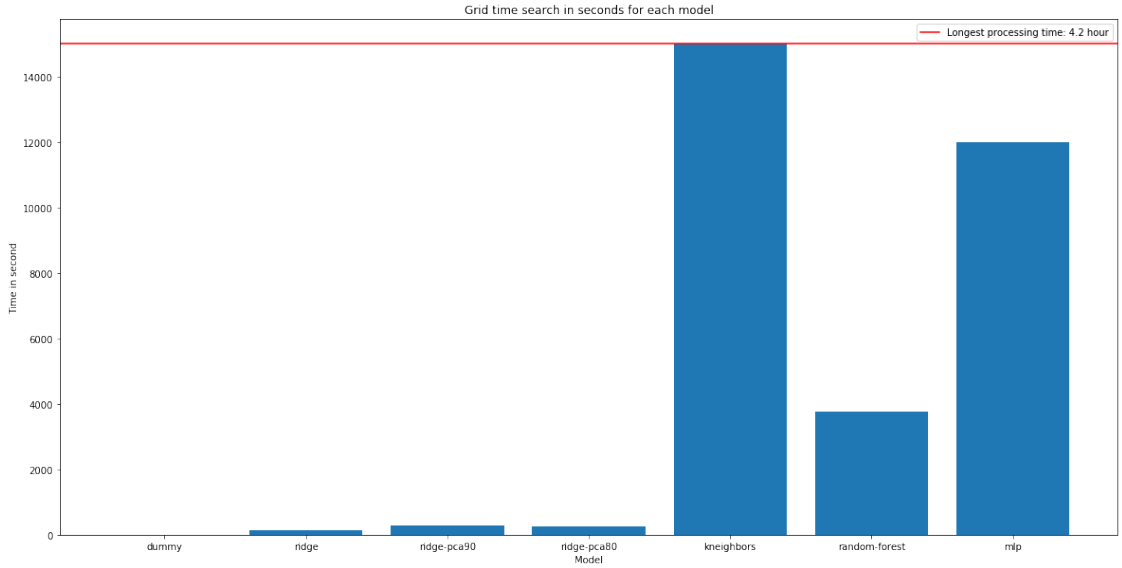
Second, all models trained performs better than the *dummy* one. Better, but not so much.

Third, *Ridge* regressor combined with *PCA()* shows a slightly lower performance when percentage of primary component decrease. This is not a surprise as *PCA* is made to reduce computing time against prediction performance.

Fourth, we should focus on *random-forest* as it has a significative better performance than other models (near of 95%). Does that mean that the bias in dataset has been handled by this model ? I'll see that later...

Fifth, the grid search time is clearly not correlated to the model performance. This can be observed on the next graph.

[8]:
```
# Draw grid search time bar graph
best_label='Longest processing time: {:.1f} hour'.
 ↪format(max(model_results['grid_time'])/3600)
draw_bar_graph(column='grid_time',
               ylabel='Time in second',
               title='Grid time search in seconds for each model',
               best_label=best_label,
               ylim=None,
               dash_line=None
              )
```



kneighbors took nearly 4 times more time to be trained than *random-forest*, *mlp* took 3 times more time.

Another reason to see the *RandomForestRegressor* as the best model to choose to predict Taxi trip in NYC.

> Note: Grid search might have not been done with the same iterations, values presented here could be optimized by reducing grid parameters scope without changing results. Anyway, it's the time I've spent in grid search and it shows the efficiency at seeking the best parameters of the *RandomForestRegressor*.

4 Some more graphs

4.1 Scatter plot with polyfit

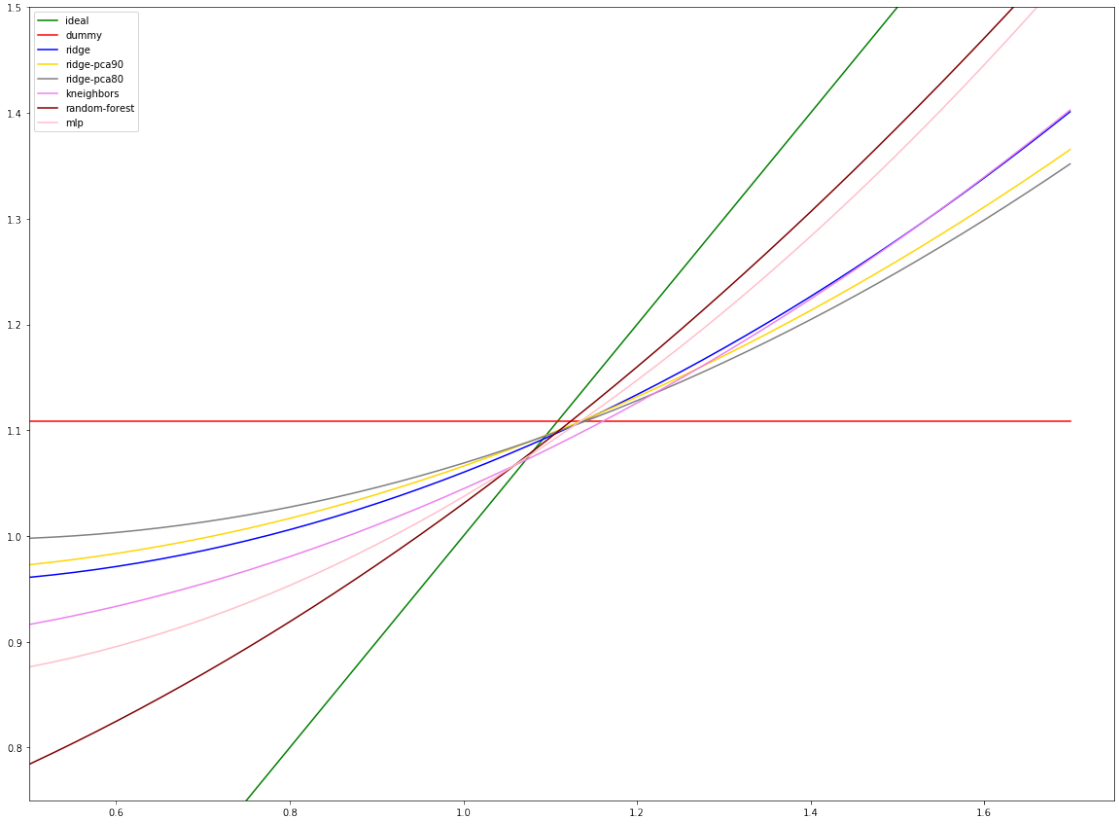
The following graph represent polynomial regression of degree=2 of scatter plots made between the real values (y) and their predictions, one regression per models.

On the X-axis is the real y value, on the Y-axis is the predicted one by models.

I've added a green line which represent the perfect prediction y=*prediction* as a baseline. The nearer model regression line is, the better is the performance score.

> Note 1: To make the graph easy to read, I do not add scatter plots, only regression curves.
>
> Note 2: Remember, value here are *np.log10(km/h).

[9]:
```
def draw_poly_regression(df, degree=2, figsize=(8,8), xlim=(0.5,1.75), ylim=(0.
↪75,1.5), model_list=model_list) -> None :

    plt.figure(figsize=figsize)

    plt.plot([0,2], [0, 2], label='ideal', color='green')

    y=df['y']

    for col, color in zip(model_list, ['red', 'blue', 'gold','grey', 'violet',␣
↪'maroon', 'pink']):

        y_pred=df[col]

        coef=np.polyfit(y,y_pred,degree)

        x_values=np.linspace(y.min(), y.max())
        y_values=np.polyval(coef, x_values)

        plt.plot(x_values, y_values, label=col, color=color)

    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.legend()
    plt.show()

draw_poly_regression(predict, figsize=(20,15))
```



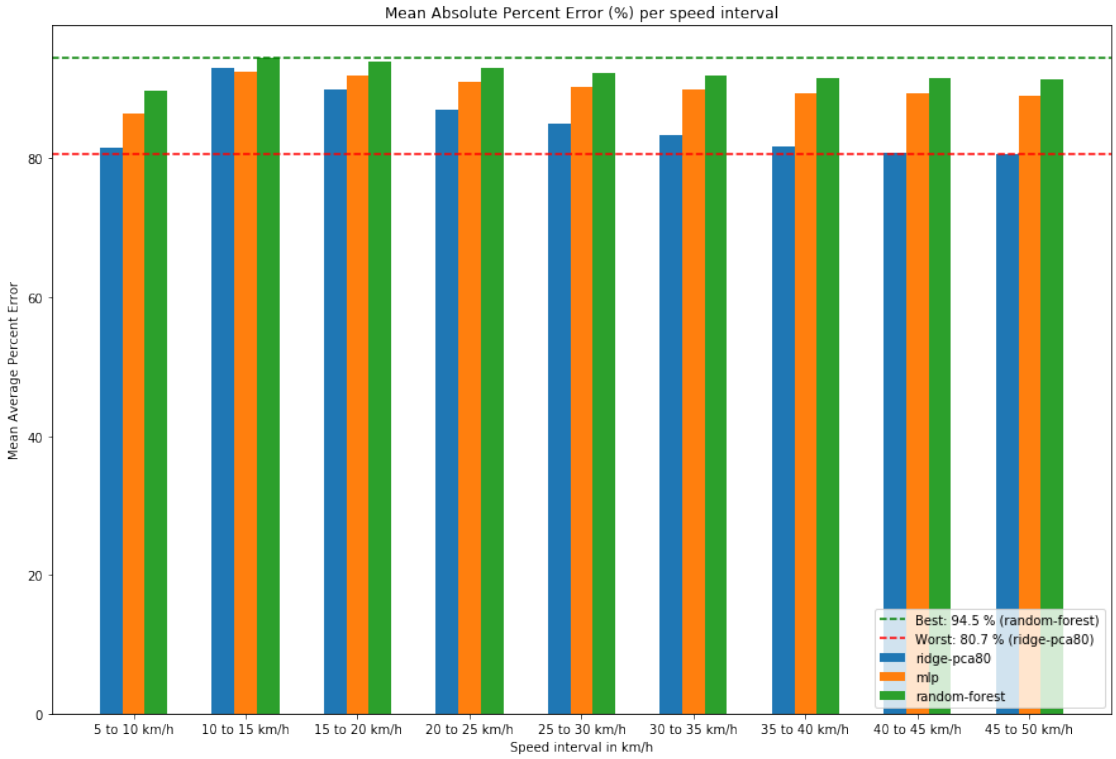
Again, this graph confirms that *RandomForestRegressor()* is the best choice. It's polynomial regression curve is the closest one to the perfect curve (the green one).

4.2 Confirm bias with *MAPE* per speed interval

To definitely conclude on the reality of the bias and the fact that *RandomForestRegressor* model beats this bias, I'll plot, for each model, a bar graph of the performance reached per speed interval (the same graph used in the Ridge Regressor Notebook.

> Note: I will draw bargraph for *ridge-pca80*, *mlp* and *random-forest* predictions only for better graph clarity

[15]:
```
draw_mape_per_speed_interval(predict, columns=['ridge-pca80',␣
↪'mlp','random-forest'], barwidth=1, figsize=(15,10))
```



Looking at this bargraph, it's now clear that *RandomForest* performs more equally on each speed interval, where *RidgePCA(0.8)* doesn't.

Another proof that *RandomForest* model was able to handle the dataset bias better than other models.

5 Conclusion

5.1 The *RandomForestRegressor* is the right choice.

Remember the problem I'd like to solve in this project:

What are the Taxi travel speed prevision for New-York City (NYC) depending on the travel characteristics and weather forecast ?

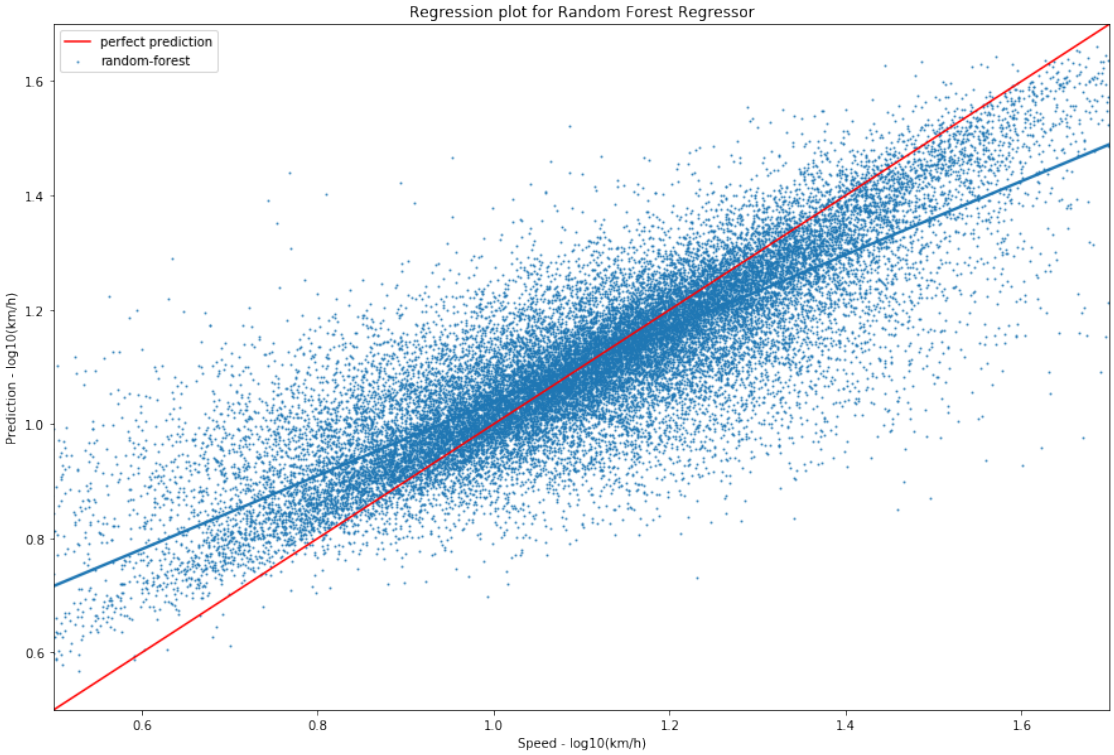
The trained *RandomForestRegressor* provides an average right prediction score of 95%, and has been able to manage the dataset bias which consist of having much more Taxi travels in the speed interval of 10 to 15 km/h compared to other intervals. This lead the model to be efficient whatever the prediction speed is.

5.2 Confirm with seaborn *regplot*

One good representation of the performance of the *RandomForest* model is using a Seaborn.regplot.

This graph shows how the y values and *prediction* do fit closely the perfect line.

[11]:
```
# Set figure size
plt.figure(figsize=(15,10))

# Get sample for better graph look
df=predict.sample(frac=0.1)

# draw seaborn.regplot
sns.regplot(df['y'], df['random-forest'], fit_reg=True, scatter_kws={"s": 1},␣
 ↪label='random-forest')

# draw perfect prediction line
plt.plot([0,2], [0, 2], label='perfect prediction', color='red')

# title and axis labels
plt.title("Regression plot for Random Forest Regressor")
plt.xlabel('Speed - log10(km/h)')
plt.xlim(0.5, 1.7)
plt.ylabel('Prediction - log10(km/h)')
plt.ylim(0.5,1.7)

# display legend and graph
plt.legend()
plt.show()
```



5.3 Why is that model the best one ?

Looking back to the *RandomForest* model exploration made in previous Notebook, I've found that the most important feature found by the model to predict values is the *travel distance*.

Altough, I could suppose that if the distance is high, then the travel course starts or ends far from the town center, which then might implies a higher travel speed (cars goes faster outside of town than in town center).

I should re-explore the dataset to confirm this, and maybe engineer a new feature like *How far pickup and dropoff locations are from town center* to help other models beat the bias.

But that would be another story for another project... ;-)

6 Thanks...

...to all members of the EPFL Learn Extension School team, for this wonderful Machine Learning journey I've spent with all of you, it was really a pleasure.

Special dedicace to my Capstone project Course Instructor Michael Notter

Hope you'll find the same pleasure reading my project :-)

See you soon...

[12]:
```
img=Image.open(os.path.join(IMG_PATH, 'thank-you.png'))
img.thumbnail([600,1000]);img
```

[12]:


THANK
YOU!