In [ ]:

```
# libraries
import matplotlib.pyplot as plt
import shapely.wkb as wkblib
import numpy as np
import keplergl
import pickle
import pandas as pd
import time
import osmium
import os

from collections import OrderedDict
from pyproj import Proj
from h3 import h3

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor
from geopandas import GeoSeries, GeoDataFrame
from rtree import index

from utils.OSRMFramework import OSRMFramework
from utils.RouteAnnotator import RouteAnnotator
from utils.plot_geometry import plot_geometry
from utils.conversor import latlon2linestring

%load_ext autoreload
%autoreload 2
%autosave 0


###### constants
TARGET = 'fare_amount'
DATASET_PATH = 'data/taxi_fare_sample_100000.csv'
H3_RES_ANALYSIS = 7

IS_DOCKER = os.environ.get('IS_DOCKER', False)
if IS_DOCKER:
    OSRM_PATH = 'osrm-router:5000'  # OSRM path for when it's instantiated throu
gh docker-compose (service name)
else:
    OSRM_PATH = 'localhost:5000'    # OSRM path for when it's instantiated local
ly
```

Notebook for AMLD 2020 Workshop: Feature Engineering for Spatial Data Analysis (https://appliedmldays.org/workshops/feature-engineering-for-spatial-data-analysis).

Authors:

- Caio Miyashiro: caiohenrique37@gmail.com
- Eva Jaumann: eva.jaumann@mytaxi.com
- Selim Onat: selim.onat@mytaxi.com

Github Repository: https://github.com/caiomiyashiro/geospatial_data_analysis/tree/master/AMLD-2020 (https://github.com/caiomiyashiro/geospatial_data_analysis/tree/master/AMLD-2020)

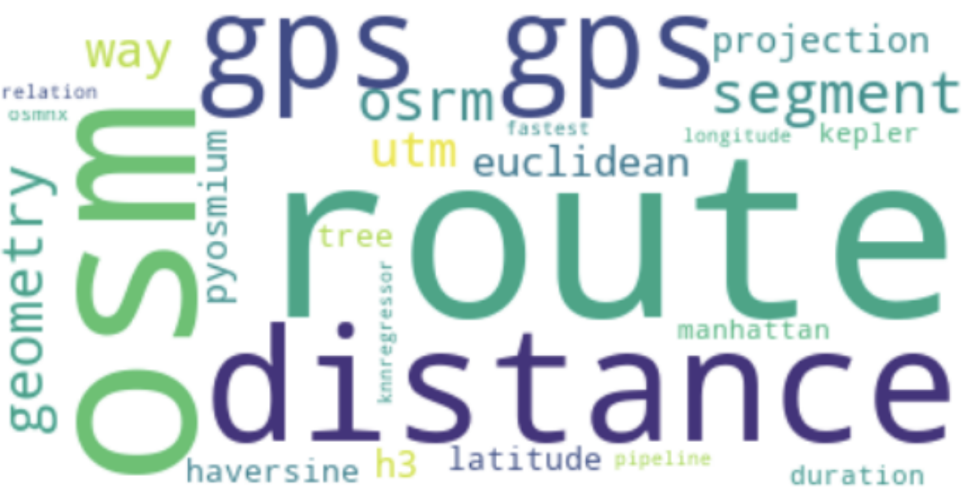
Installation

If you did not already: Please follow the README file for setup instructions.

Make sure that you have a kaggle account.

Introduction

Welcome to the workshop! Today you're going to learn several techniques related to the processing of geospatial data. We're going to use as background motivation the Kaggle dataset on Prediction of Taxis Fares (https://www.kaggle.com/c/new-york-city-taxi-fare-prediction)



The main sections we're going to cover are:

- [ ] Intro + Exploratory Data Analysis
- [ ] Feature Engineering - Distance and Time
- [ ] Spatial Processing Performance
- [ ] Traffic Prototype
- [ ] Map Features

Presentation and full notebook

We prepared a short version to follow during the workshop (this notebook Presentation_AMLD_2020.ipynb). To get more information (also after the workshop) have a look at Complete_AMLD_2020.ipynb. You can always check out this notebook if you get stucked and want to have a look at the solutions for the exercises.

Dataset

Kaggle Taxi Prediction Homepage Picture

The challenge of this dataset is to predict the final fare paid by user by just having a small set of variables to work with:

- **pickup_datetime**: Time that a user was picked-up by a taxi
- **pickup_latitude**: pickup location

In [ ]:

```
df = pd.read_csv(DATASET_PATH)
df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'])
display(df.head())
display(df.dtypes)
```

Latitude/Longitude

Latitude and longitude are imaginary horizontal and vertical lines respectively that run around the earth.

- Latitude run across the globe, and the latitude that runs through the middle of the Earth is given the number zero degrees (0°) and is called the Equator.
- Longitude are vertical lines around Earth. They meet at the poles and are wide apart at Equator. By convention, the line with 0° degrees longitude passes the Royal Observatory in Greenwich, England.

Together, latitude and longitude identify a point on a spherical system, in our case on our planet The Earth.

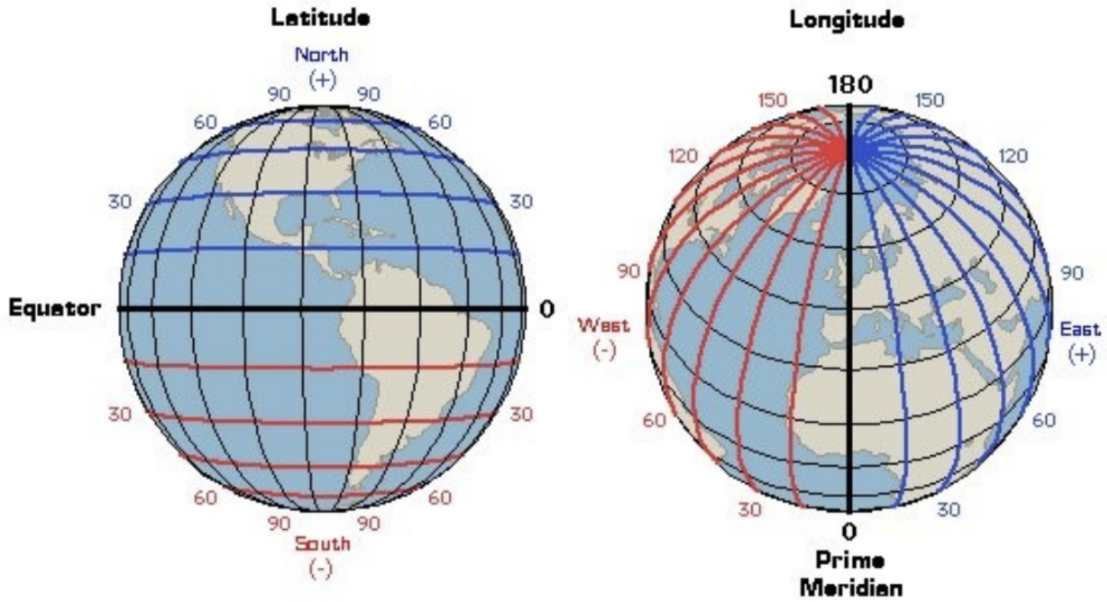


Image source - https://en.wikipedia.org/wiki/Geographic_coordinate_system (https://en.wikipedia.org/wiki/Geographic_coordinate_system)

For example the statue of liberty has a latitude of `40.691332` and a longitude of `-74.0446291`.

In [ ]:

```
df.describe()
```

First Analysis

Before we start understanding our data, we should check its integrity. We apply some sanity checks in the data to remove anything that'd be impossible to happen, such as negative fares.

Check amount of NaN

The number of nan elements are irrelevant. So we just remove them.

In [ ]:

```
# check amount of NaN - just remove them
display(df.isna().sum())

df.dropna(subset=['dropoff_longitude', 'dropoff_latitude'], inplace=True)
```

Check impossible fares

Note that we're removing only **impossible** fares for now. Any filter based on value distribution should be done only using the training set.

In [ ]:

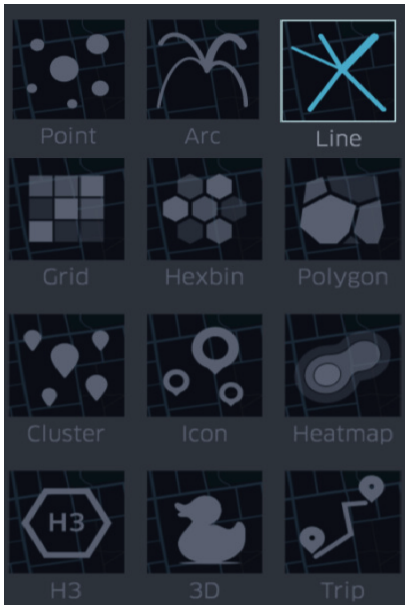
```
display(df['fare_amount'].describe())
print('')

fare_under_0 = np.sum(df['fare_amount']<=0)
print(f'{fare_under_0} rows with fare under or equal 0')

df = df.loc[df['fare_amount'] >= 0]
```

Spatial Data Visual Analysis

From the first `describe` function in the section above, we see that there're a few latitudes/longitudes for both pick up and drop-off that are away from the main mass of data. In order to confirm we can use a visual tool.

Kepler

Kepler can visualize the following polygons:



> In the complete version of this notebook, we go through each of the layers in H3. Check it out.

You can check a few visuals in the examples section (https://github.com/keplergl/kepler.gl/blob/master/docs/user-guides/c-types-of-layers.md) of kepler Github repository.

Check pickup and dropoff points points

For checking pick-up and drop-off locations, we're going to check these `Points` in kepler by sending the latitudes and longitudes from both fields and build a Points layer in Kepler

In [ ]:

```
w1 = keplergl.KeplerGl(height=500)
w1.add_data(data=df[['pickup_latitude', 'pickup_longitude',
                     'dropoff_latitude', 'dropoff_longitude']], name='points')
w1
```

We can see that some points are quite wrong, being not just outside New York, but rather in other countries or even in the middle of the ocean. Lets do a simple approach to remove those points and build a bounding box (https://wiki.openstreetmap.org/wiki/Bounding_Box) (bbox), *i.e.*, a square around New York.

In order to build a squared bbox, we need two pairs of [latitude, longitude] points, the bottom left and top right corners. In order to get them, we can click on specific places on Google Maps (https://www.google.com/maps) and get the returned latitude/longitudes points. Alternatively, boundingbox (https://boundingbox.klokantech.com/) can be used. Finally, everything that's outside the bbox definition, we will filter out from our dataset.

Image source - https://gis.stackexchange.com/questions/255158/get-minimum-and-maximum-latitude-and-longitude-of-new-york (https://gis.stackexchange.com/questions/255158/get-minimum-and-maximum-latitude-and-longitude-of-new-york)

In [ ]:

```
def remove_outside_bbox(df, bbox, lat_col, lon_col):
    df_ = df.copy()
    mask_lat = (df_[lat_col] > bbox[0][0]) & (df_[lat_col] < bbox[1][0])
    mask_lon = (df_[lon_col] > bbox[0][1]) & (df_[lon_col] < bbox[1][1])
    return df_.loc[(mask_lat) & (mask_lon)]

bottom_left_lat_lon = [40.492016, -74.279034]
upper_right_lat_lon = [40.913473, -73.689152]
bbox = [bottom_left_lat_lon, upper_right_lat_lon]

df = remove_outside_bbox(df, bbox, 'pickup_latitude', 'pickup_longitude')
df = remove_outside_bbox(df, bbox, 'dropoff_latitude', 'dropoff_longitude')
```

Lets check it again:

In [ ]:

```
w1 = keplergl.KeplerGl(height=500)
w1.add_data(data=df[['pickup_latitude', 'pickup_longitude',
                     'dropoff_latitude', 'dropoff_longitude']], name='points')
w1
```

Split training and testing

In [ ]:

```
######
###### Helper functions
def split_training_test(df, target=TARGET, test_size=0.2):
    X = df.drop(TARGET, axis=1)
    y = df[TARGET]
    return train_test_split(X, y, test_size=test_size, random_state=42)

def join_Xy(df, target, target_str=TARGET):
    df_ = df.copy()
    df_[target_str] = target
    return df_

######
###### Main function
def get_initial_training_testing_set(df):
    X_train, X_test, y_train, y_test = split_training_test(df)
    print(f'Shape X_train {X_train.shape}')
    print(f'Shape X_test {X_test.shape}')

    # join target for easier exploratory analysis
    df_train = join_Xy(X_train, y_train)
    df_test = join_Xy(X_test, y_test)
    return df_train, df_test

df_train, df_test = get_initial_training_testing_set(df)
```

Data Analysis

In the machine learning context, we want to have the best quality data in our training procedure, as outliers can impact your model training process and therefore, create bad predictions for when your model is in production. With this second step on data analysis, we check the data distribution to check for extreme variations in our dataset. Besides serving for the purpose explained above, this also indicate the type of rules that we should consider when the model is in `production`.

- For example, what should we do if we have a model for predicting Taxi fares made for NY in production and we receive a fare prediction request with a pick-up latitude/longitude referencing Brazil?

Remove fares < minimum fare and outliers > 99th percentile

Lets use a simple percentil rule in order to remove training data whose fare is minimum that the city regulation defines and more than 99% of the other fares presenting in our training set.

In [ ]:

```
# minimum fare = $2.5: https://www1.nyc.gov/site/tlc/passengers/taxi-fare.page

display(df_train['fare_amount'].describe(percentiles=[.01, .05, .1, .90,.95,.99
]))
```

We will remove fares smaller than 2.5 dollars and higher than 52 dollars.

In [ ]:

```
df_train = df_train.loc[(df_train['fare_amount'] >= 2.5) & (df_train['fare_amoun
t'] <= 52)]

plt.hist(df_train['fare_amount'], bins=51)
plt.title('Histogram of Fare')
plt.xlabel('Dollars ($)');
```

Average demand per region over 1 weekday - H3

On the maps we saw the pickup and dropoff locations. How can we define areas with high demand? To do this we need to divide the area of the city in smaller neighborhoods.

Zipcodes?

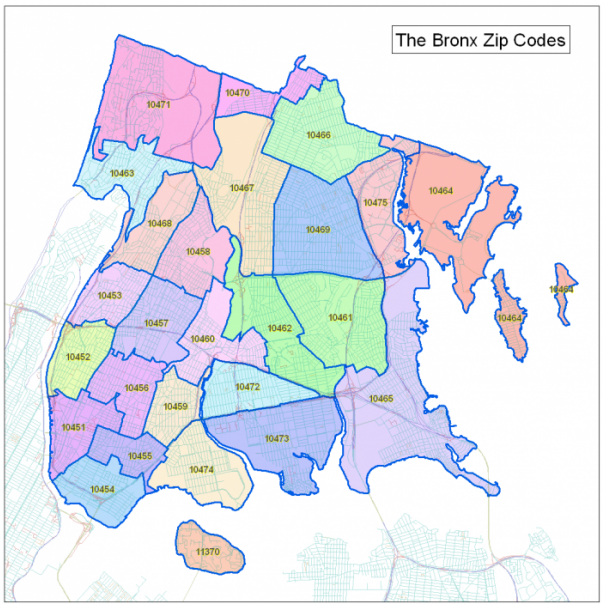
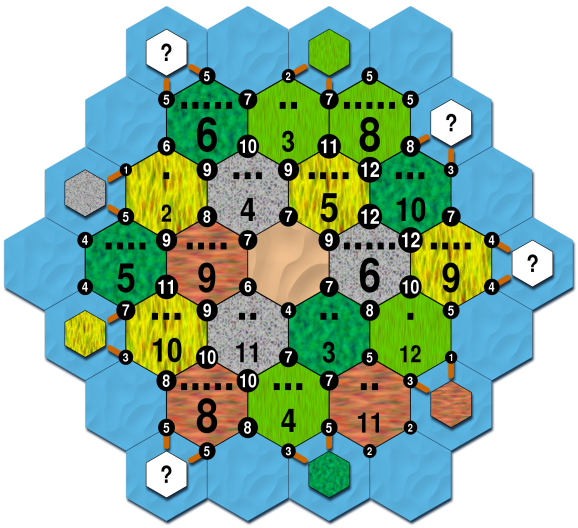


Image source - worldmap (http://map-world.us/bronx-zip-code-map.html)

Grid!



Catan (game) Image source - wikipedia (https://en.wikipedia.org/wiki/Catan#/media/File:Catan_Universe_fixed_setup.svg)

Different resolutions possible.

On a sphere: Fullerene like structure with pentagons (could be placed into water)

Image source - https://eng.uber.com/h3/ (https://eng.uber.com/h3/)

We will use H3 (https://uber.github.io/h3), Uber's open source geospatial indexing system and use it for visualisations. The size of the hexagons is defined by the resolution (https://uber.github.io/h3/#/documentation/core-library/resolution-table), e.g. a resolution of 7 results in a hexagon area of 5 square km.

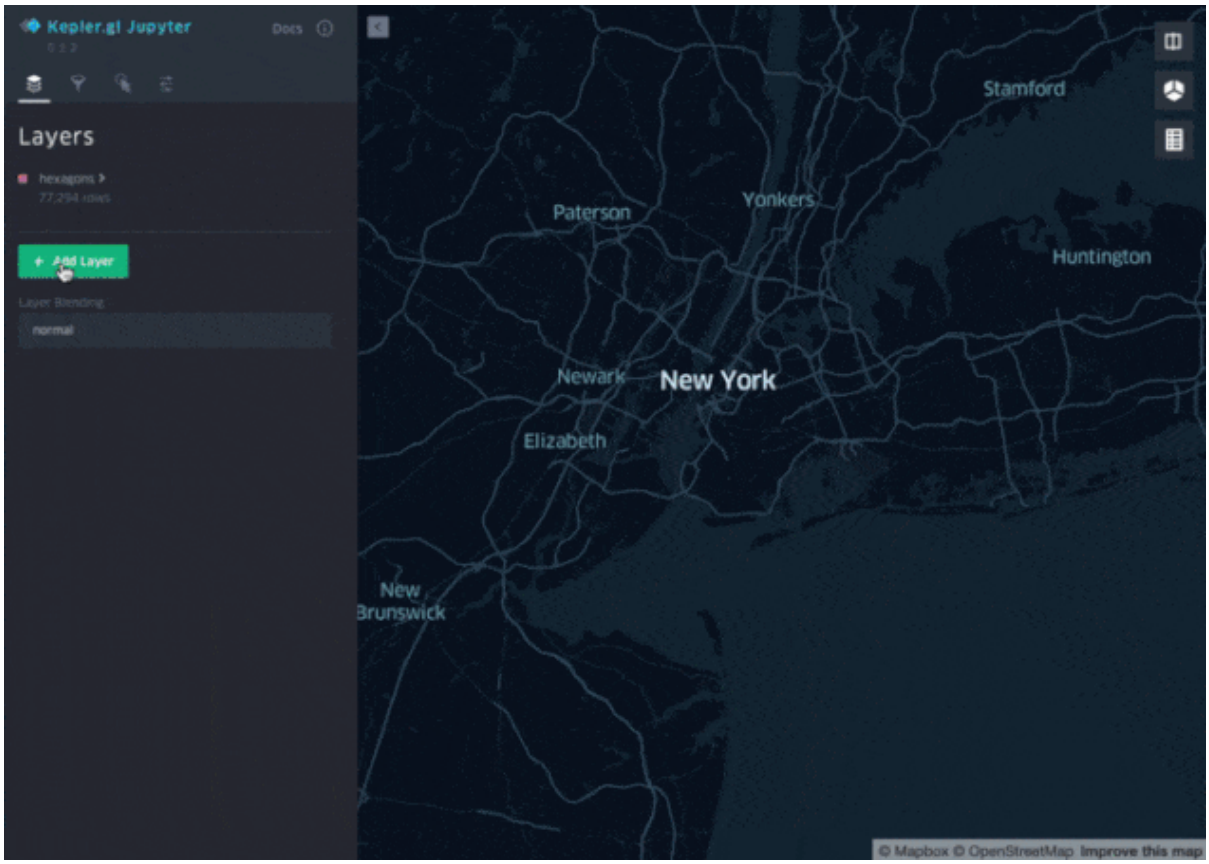
In [ ]:

```
## helper function
from datetime import timedelta


def add_day(day, reference_day='2019-01-07'):
    return pd.to_datetime(reference_day) + timedelta(days=day)

def average_demand_weekday(df,
                           pickup_latitude='pickup_latitude', pickup_longitude=
'pickup_longitude',
                           dropoff_latitude='dropoff_latitude', dropoff_longitud
e='dropoff_longitude',
                           pickup_datetime='pickup_datetime', fare='fare_amount'
, resolution=H3_RES_ANALYSIS):
    df = df.copy()

    # hexagon indices
    df['pickup_h3'] = [h3.geo_to_h3(lat, long, res=resolution) for lat, long in
zip(df[pickup_latitude],

df[pickup_longitude])]
    df['dropoff_h3'] = [h3.geo_to_h3(lat, long, res=resolution) for lat, long in
zip(df[dropoff_latitude],

df[dropoff_longitude])]
    # weekday
    df['weekday'] = df[pickup_datetime].dt.dayofweek

    # aggregate per hexagon and weekday
    df_aggregated = df.groupby(['pickup_h3','weekday']).size().reset_index(name=
'nr_tours')

    df_demand = df.merge(df_aggregated, how = 'inner', on = ['pickup_h3', 'weekd
ay'])

    # introduce a reference day as a reference week to have a playback option in
kepler
    df_demand['ref_week_date'] = df_demand.weekday.apply(add_day)


    return df_demand[['ref_week_date', 'pickup_h3', 'dropoff_h3', 'nr_tours', fa
re]]

df_hex = average_demand_weekday(df_train, resolution=H3_RES_ANALYSIS)

df_hex.head()
```

In [ ]:

```
w1 = keplergl.KeplerGl(height=500)
w1.add_data(data=df_hex, name='hexagons')
w1
```

These are the steps we used to show the hexagon map (open the menu first with the arrow symbol **>**):



Exercise: Origin-Destination map over 1 Week - H3

To see which pickup-dropoff combinations are the most demanded, we connect the pickup hexagon centers with the dropoff hexagon centers.

Use the **arc** layer to show the distances between pickup and dropoff.

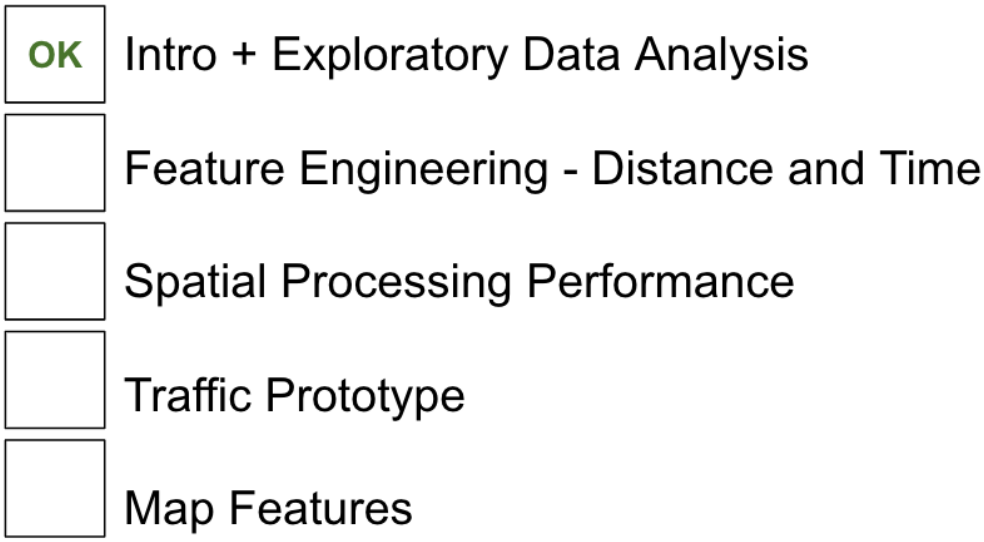
Can you think on the most popular drop-off?

In [ ]:

```
def average_arc_weekday(df,
                        pickup_hex='pickup_h3', dropoff_hex='dropoff_h3',
                        nr_tours='nr_tours', ref_week_date='ref_week_date'):

    center_pickup_hex = 'center_pickup_hex'
    center_dropoff_hex = 'center_dropoff_hex'
    df[center_pickup_hex] = df[pickup_hex].apply(lambda x: h3.h3_to_geo(x))
    df[center_dropoff_hex] = df[dropoff_hex].apply(lambda x: h3.h3_to_geo(x))

    df['lng_pickup'] = df[center_pickup_hex].apply(lambda x: x[1])
    df['lat_pickup'] = df[center_pickup_hex].apply(lambda x: x[0])
    df['lng_dropoff'] = df[center_dropoff_hex].apply(lambda x: x[1])
    df['lat_dropoff'] = df[center_dropoff_hex].apply(lambda x: x[0])
    return df[['lat_pickup', 'lng_pickup', 'lat_dropoff', 'lng_dropoff', nr_tour
s, ref_week_date]].drop_duplicates()

df_arc = average_arc_weekday(df_hex)
df_arc.head()
```

In [ ]:

```
w1 = keplergl.KeplerGl(height=500)
w1.add_data(data=df_arc.drop_duplicates(), name='hexagons')
w1
```



Feature Engineering - Model Iterations

Now that we have a cleaner training set, we can use it to create our predictions! For teaching purposes, we're going to create multiple models, with incremental set of features/complexity. For each iteration, we're going to show a demo with the feature's concept, create a pipeline, train our models and then show and store the errors by their's Root Mean Squared Error - RMSE (https://en.wikipedia.org/wiki/Root-mean-square_deviation).

Start experiment tracking

We're going to store the RMSEs for each iteration in this dictionary and look at it at the end of each iteration.

In [ ]:

```
iteration_results = OrderedDict()
```

Model 0 - Lazy Estimator - Median Fare

In order to have a baseline, we're going to start with a lazy model, one that, for every row in our training (and testing) dataset, will predict only the median fare of the training set. The only purpose of this is to have a upper error limit and we hope to see it decreasing after every iteration on the model's features.

In [ ]:

```
median_f = df_train['fare_amount'].median()
print(f'Median Fare: {median_f}')

y_test_pred_lazy = [median_f] * df_test.shape[0]                        # array wi
th same length as y_test

rmse = np.sqrt(mean_squared_error(df_test[TARGET], y_test_pred_lazy)) # RMSE
iteration_results['exp_0_lazy_rmse'] = rmse                             # store in
experiments results
print(f'RMSE: {rmse}')
```

Model 1 - Euclidean Distance

In [ ]:

```
## Small Example
##
def euclidean_distance(x1, y1, x2, y2):
    return (((x2-x1)**2 + (y2-y1)**2)**(1/2))

df_train_temp = df_train.head(3).copy()
df_train_temp['euclidean_dist'] = euclidean_distance(df_train_temp['pickup_latit
ude'], df_train_temp['pickup_longitude'],
                                                     df_train_temp['dropoff_latitu
de'], df_train_temp['dropoff_longitude'])

df_train_temp
```

In [ ]:

```
def standardize_features(df):
    df_copy = df.copy()
    col_names = df_copy.columns
    ixs = df_copy.index
    return pd.DataFrame(StandardScaler().fit_transform(df_copy), columns=col_nam
es, index=ixs)

def feature_pipeline_1(df, target_col=TARGET,
                               pickup_latitude='pickup_latitude', pickup_longitud
e='pickup_longitude',
                               dropoff_latitude='dropoff_latitude', dropoff_longi
tude='dropoff_longitude'):

    EUCLIDEAN_FEAT = 'euclidean_dist'
    FEATURES = [EUCLIDEAN_FEAT]

    df_copy = df.copy()
    df_copy[EUCLIDEAN_FEAT] = euclidean_distance(x1=df_copy[pickup_latitude], y1
=df_copy[pickup_longitude],
                                                 x2=df_copy[dropoff_latitude], y
2=df_copy[dropoff_longitude])
    # in the EDA, we probably treat the nulls, so for now, just drop them
    df_copy = df_copy.dropna()

    return df_copy[FEATURES + [target_col]]

df_train_1 = feature_pipeline_1(df_train)
df_test_1 = feature_pipeline_1(df_test)

df_train_1.head()
```

In [ ]:

```
model_1 = LinearRegression()
model_1.fit(df_train_1.drop(TARGET, axis=1), df_train_1[TARGET])

y_test_pred_1 = model_1.predict(df_test_1.drop(TARGET, axis=1))
```

In [ ]:

```
def print_evaluation(y_obs, y_pred, max_lim_y=100, return_errors=True):
    rmse_hist = np.sqrt((y_pred - y_obs)**2)
    plt.hist(rmse_hist[y_obs < max_lim_y], bins=100)
    plt.title('RMSE Distribution');

    if(return_errors == True):
        return mae_hist

#####
rmse = np.sqrt(mean_squared_error(df_test_1[TARGET], y_test_pred_1))
iteration_results['exp_1_rmse'] = rmse
print(f'RMSE: {rmse}')

print_evaluation(df_test_1[TARGET], y_test_pred_1, return_errors=False)
```

Why the feature value above is not correct?

Although it works for our model, if you look at pipeline1's output, the euclidean distance values don't make any sense. Why is that?

Projections

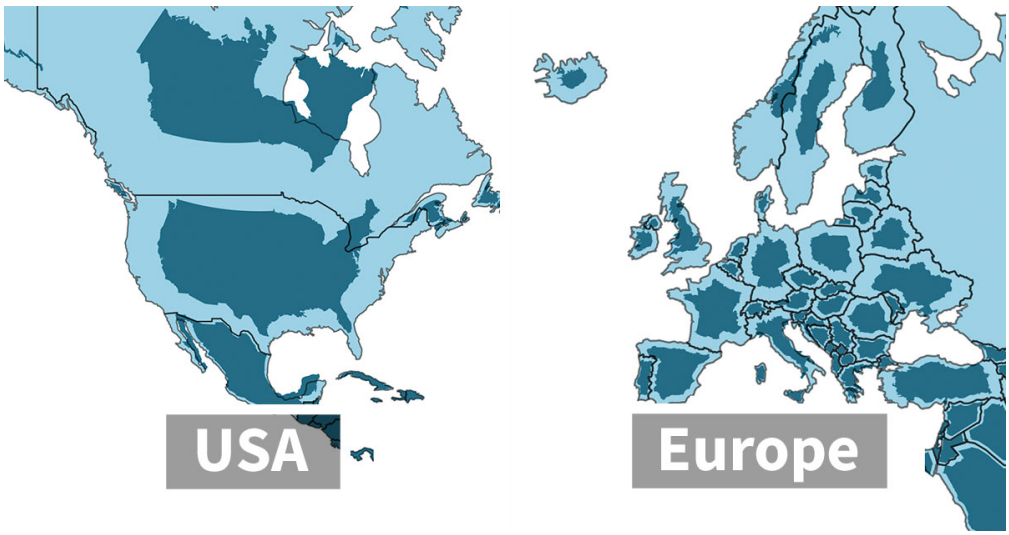
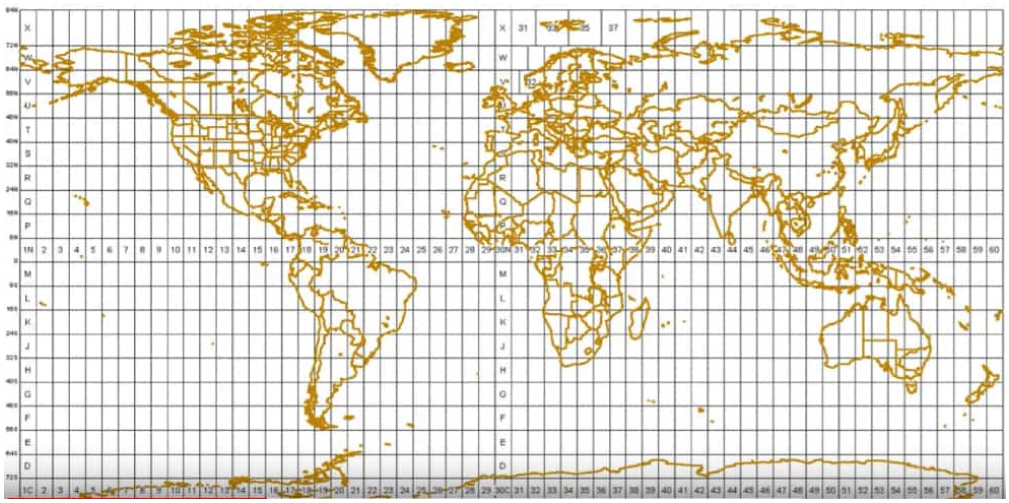


Image source (https://newsini.com/news/this-map-reveals-the-actual-size-and-shape-of-every-country-in-the-world?uid=18225)

There are many types of projections, popular ones include: **Universal transverse Mercator (UTM)** and **Lambert Conformal Conic**. UTM is discussed in detail below, the latter is used mainly in mid-latitude areas.This uses two Standard Parallel (lines of latitudes which are unevenly spaced concentric circles).

Universal Transverse Mercator (UTM)



UTM Projection grid - https://www.youtube.com/watch?v=LcVlx4Gur7I (https://www.youtube.com/watch?v=LcVlx4Gur7I)

UTM projects the latitude and longitude to a three numbers representation (https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system#Locating_a_position_using_

- **Zone** - Zone is a number attributed to every grid as in the picture above. We can use online tools to figure out the UTM zone of a region
- **Easting** - For every grid, their central meridian starts with a value of 500.000 and the *easting* value represents how many `meters` east is the point to this grid's meridian

- **Northing** - For every grid, this value represents how far away the point is from the south pole `in meters`

Adapted Pipeline 1 code for UTM

To project our lat/lon points to UTM, we use the pyproj (https://pypi.org/project/pyproj/#description) package. **Notice** how, in the function, we have to send the coordinates in the correct order - **longitude/latitude**:

In [ ]:

```
## Small Example
##
def latlon2UTM(latitudes, longitudes, utm_proj_zone=18):
    # project points to UTM
    # New York city zone = 18 - https://www.latlong.net/lat-long-utm.html
    lonlat2UTM = Proj(proj='utm', zone=utm_proj_zone, ellps='WGS84')
    point_easting, point_northing = lonlat2UTM(longitudes, latitudes)
    # lonlat2UTM(point_easting,point_northing,inverse=True) # to convert UTM bac
k to lon/lat
    return point_easting, point_northing

df_train_temp = df_train.head(3).copy()

pickup_dropoff_cols = ['pickup', 'dropoff']
for point in pickup_dropoff_cols:
    point_easting, point_northing = latlon2UTM(latitudes=df_train_temp[f'{point}
_latitude'].values,
                                               longitudes=df_train_temp[f'{poin
t}_longitude'].values)
    df_train_temp[f'{point}_easting'] = point_easting
    df_train_temp[f'{point}_northing'] = point_northing

df_train_temp
```

In [ ]:

```
def feature_pipeline_1b(df, target_col=TARGET,
                        pickup_latitude='pickup_latitude', pickup_longitude='pick
up_longitude',
                        dropoff_latitude='dropoff_latitude', dropoff_longitude='d
ropoff_longitude',
                        utm_proj_zone=18):

    EUCLIDEAN_FEAT = 'euclidean_dist_km'
    FEATURES = [EUCLIDEAN_FEAT]

    df_copy = df.copy()

    pickup_dropoff_cols = ['pickup', 'dropoff']
    for point in pickup_dropoff_cols:
        point_easting, point_northing = latlon2UTM(latitudes=df_copy[f'{point}_l
atitude'].values,
                                                   longitudes=df_copy[f'{point}_
longitude'].values)
        df_copy[f'{point}_easting'] = point_easting
        df_copy[f'{point}_northing'] = point_northing

    df_copy[EUCLIDEAN_FEAT] = euclidean_distance(x1=df_copy['pickup_northing'],
y1=df_copy['pickup_easting'],
                                                 x2=df_copy['dropoff_northing'],
y2=df_copy['dropoff_easting'])/1000
    # in the EDA, we probably treat the nulls, so for now, just drop them
    df_copy = df_copy.dropna()

    return df_copy[FEATURES + [target_col]]

df_train_1b = feature_pipeline_1b(df_train)
df_test_1b = feature_pipeline_1b(df_test)

df_train_1b.head()
```

In [ ]:

```
model_1b = LinearRegression()
model_1b.fit(df_train_1b.drop(TARGET, axis=1), df_train_1b[TARGET])

y_test_pred_1b = model_1b.predict(df_test_1b.drop(TARGET, axis=1))
```

In [ ]:

```
rmse = np.sqrt(mean_squared_error(df_test_1b[TARGET], y_test_pred_1b))
iteration_results['exp_1b_rmse'] = rmse
print(f'RMSE: {rmse}')

print_evaluation(df_test_1b[TARGET], y_test_pred_1b, return_errors=False)
```

We can see that the score didn't change much, for small distances there's a big chance that latitude/longitude are highly correlated to UTM projections, except for the unit, which in UTM refers to meters, while lat/lon distances doesn't have any real unit.

In [ ]:

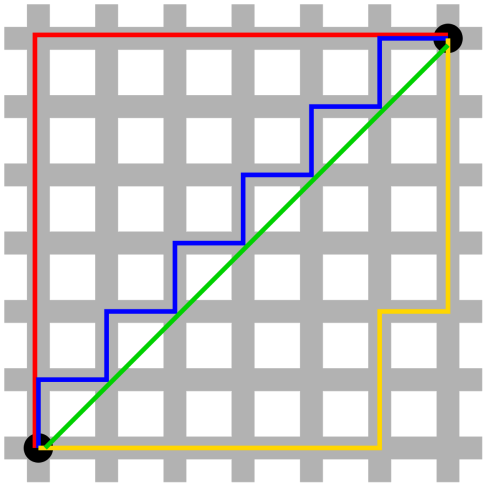
```
df_train_1[['euclidean_dist']].merge(df_train_1b[['euclidean_dist_km']], left_in
dex=True, right_index=True).corr()
```

Manhattan

The manhattan distance, also coincidentally called the taxi cab distance (https://en.wikipedia.org/wiki/Taxicab_geometry), is a different distance definition, in which the distance between two points A and B is the sum of the absolute differences of their Cartesian coordinates, that is:

$$d_1(p, q) = \|p - q\|_1 = \sum_{i=1}^{n} |p_i - q_i|$$

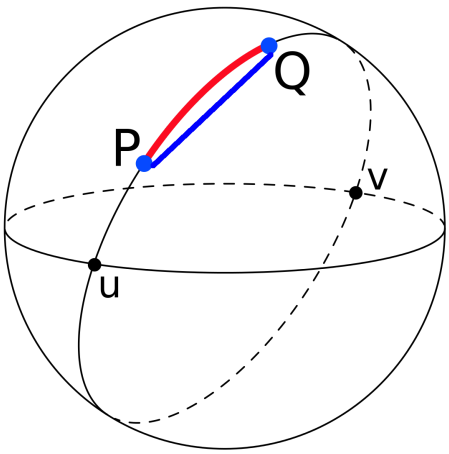
The name refers to the minimum distance a taxi would have to travel in a place with regular square blocks instead of a direct straight line.



Manhattan distance from Wikipedia (https://de.wikipedia.org/wiki/Datei:Manhattan_distance_bgiu.png)

Haversine Distance:

Both euclidean and manhattan are calculated considering that point P and Q are in a plane like a square. However, depending on the distance between the two points, these distances don't take the curvature of the earth into consideration, and their values would measure a distance that would cross through the earth!



Blue line representing euclidean distance between two further points P and Q and Haversine distance in red.
Adapted from Wikipedia (https://en.wikipedia.org/wiki/Great-circle_distance)

As in this workshop we're dealing with small enough distances considering the whole earth as parameter, we don't expect big differences between the euclidean and Haversine. However, if you're dealing with bigger distances, Haversine should be preferred.

In [ ]:

```
## Small Example
##
def manhattan_distance(x1, y1, x2, y2):
    dy = np.abs(y2 - y1)
    dx = np.abs(x2 - x1)
    return dy + dx

def lat_longs2radians(lat1, lon1, lat2, lon2):
    lat1, lon1, lat2, lon2 = map(np.radians, [lat1, lon1, lat2, lon2])
    return lat1, lon1, lat2, lon2

def haversine_distance(lat1, lon1, lat2, lon2):
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees)

    All args must be of equal length.

    """
    lat1, lon1, lat2, lon2 = lat_longs2radians(lat1, lon1, lat2, lon2)
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = np.sin(dlat/2.0)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2.0)**2

    c = 2 * np.arcsin(np.sqrt(a))
    km = 6367 * c
    return km


#####
#####
df_train_temp = df_train.head(3).copy()
df_train_temp['haversine_dist'] = haversine_distance(df_train_temp['pickup_latit
ude'], df_train_temp['pickup_longitude'],
                                                     df_train_temp['dropoff_lati
tude'], df_train_temp['dropoff_longitude'])

pickup_dropoff_cols = ['pickup', 'dropoff']
for point in pickup_dropoff_cols:
    point_easting, point_northing = latlon2UTM(latitudes=df_train_temp[f'{point}
_latitude'].values,
                                               longitudes=df_train_temp[f'{poin
t}_longitude'].values)
    df_train_temp[f'{point}_easting'] = point_easting
    df_train_temp[f'{point}_northing'] = point_northing
df_train_temp['manhattan_dist'] = manhattan_distance(df_train_temp['pickup_north
ing'], df_train_temp['pickup_easting'],
                                                     df_train_temp['dropoff_nort
hing'], df_train_temp['dropoff_easting'])

df_train_temp
```

In [ ]:

```
def feature_pipeline_2(df, target_col=TARGET,
                               pickup_latitude='pickup_latitude', dropoff_latitud
e='dropoff_latitude',
                               pickup_longitude='pickup_longitude', dropoff_longi
tude='dropoff_longitude'):
    EUCLIDEAN_FEAT = 'euclidean_dist_km'
    HAVERSINE_FEAT = 'haversine_dist_km'
    MANHATTAN_FEAT = 'manhattan_dist_km'
    FEATURES = [EUCLIDEAN_FEAT, MANHATTAN_FEAT, HAVERSINE_FEAT]

    df_copy = df.copy()

    # Execute previous features' pipeline
    df_euclidean = feature_pipeline_1b(df_copy)
    df_copy[EUCLIDEAN_FEAT] = df_euclidean[EUCLIDEAN_FEAT]

    pickup_dropoff_cols = ['pickup', 'dropoff']
    for point in pickup_dropoff_cols:
        point_easting, point_northing = latlon2UTM(latitudes=df_copy[f'{point}_l
atitude'].values,
                                                   longitudes=df_copy[f'{point}_
longitude'].values)
        df_copy[f'{point}_easting'] = point_easting
        df_copy[f'{point}_northing'] = point_northing

    df_copy[MANHATTAN_FEAT] = manhattan_distance(x1=df_copy['pickup_northing'],
y1=df_copy['pickup_easting'],
                                                 x2=df_copy['dropoff_northing'],
y2=df_copy['dropoff_easting'])/1000
    df_copy[HAVERSINE_FEAT] = haversine_distance(lat1=df_copy[pickup_latitude],
lon1=df_copy[pickup_longitude],
                                                 lat2=df_copy[dropoff_latitude],
lon2=df_copy[dropoff_longitude])

    # in the EDA, we probably treat the nulls, so for now, just drop them
    df_copy = df_copy.dropna()

    return df_copy[FEATURES + [target_col]]

df_train_2 = feature_pipeline_2(df_train)
df_test_2 = feature_pipeline_2(df_test)

df_train_2.head()
```

In [ ]:

```
model_2 = LinearRegression()
model_2.fit(df_train_2.drop(TARGET, axis=1), df_train_2[TARGET])

y_test_pred_2 = model_2.predict(df_test_2.drop(TARGET, axis=1))
```

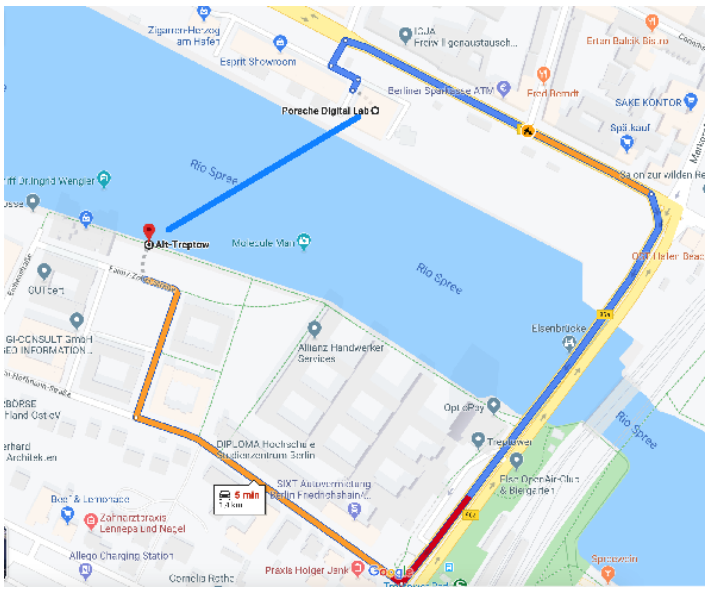
In [ ]:

```
rmse_2 = np.sqrt(mean_squared_error(df_test_2[TARGET], y_test_pred_2))
iteration_results['exp_2_rmse'] = rmse_2
print(f'RMSE Error: {rmse_2}')

print_evaluation(df_test_2[TARGET], y_test_pred_2, return_errors=False)
```

In [ ]:

```
iteration_results
```

Using fastest route distance and duration - OSRM



Blue line representing straight line distance between points A and B while real route distance is displayed by Google Maps. Source: Google Maps (https://shorturl.at/cqz37)

Routers such as google maps have the city street layout with them and so they can calculate routes by different means of transportation. A open source tool works over Open Street Maps (https://www.openstreetmap.org/about) (OSM) and can calculate these routes for us. For testing their API, they make a demo (https://map.project-osrm.org/) and a public server (https://github.com/Project-OSRM/osrm-backend/wiki/Demo-server) available, but in both you're limited to a limited amount of requests, in order not to overload them.

Setup

We can setup OSRM by two ways:

- Use a standard pre-created dockerfile - Source (https://hub.docker.com/r/osrm/osrm-backend/)
- Compile from source - Source (https://github.com/Project-OSRM/osrm-backend#building-from-source)

OSRM works with many configuration files, such as transportation profiles, which configure streets max speeds and allowed paths depending on which transportation mean you're using.

While setting up this workshop project (with `make`) we've already set up a functional OSRM for you and you can go to next cells to test it out :)

However, for the learning process, we can do it locally with the instructions below, which, by the way, are the same commands that the `makefile` is doing in `make setup`.

Instructions:

> * Open a terminal and go to your project folder, create a *router* folder for cleanliness * wget http://download.geofabrik.de/north-america/us/new-york-latest.osm.pbf (file is ~200 Mb) or (**preferably**) the custom smaller version of just New York city - https://amldspatial.s3.eu-central-1.amazonaws.com/new_york_city.osm.pbf * In the terminal, move to the folder where the file above was put and execute the following commands, considering that **the file name must match with the downloaded file**: * docker run -t -v "
> $PWD:/data$" osrm/osrm-backend osrm-extract -p /opt/car.lua /data/new_york_city.osm.pbf
> osrm/osrm-backend osrm-partition /data/new_york_city.osm.pbf * docker run -t -v "

$PWD : /data " osrm/osrm - backendosrm - customize/data/new_yorkcity.osm.pbf * dockerrun -$
osrm/osrm-backend osrm-routed --algorithm mld /data/new_york_city.osm.pbf

Now you have the router running in a dedicated process. In order to test it, open another tab in the terminal and type the following:

Test: `curl "http://localhost:5000/route/v1/driving/-73.996070,40.732605;-73.980675,40.761864?steps=false&geometries=geojson&annotations=true&overview=full"`

This is a command to make a route between the two provided lon/lats. You can check more options at the API documentation (http://project-osrm.org/docs/v5.5.1/api/#route-service)

Calculate street distances and durations

For our pipelines, we're going to use the custom class `utils.OSRMFramework.OSRMFramework` to request and retrieve route data from a OSRM instance. When building the class, you have to send the instance's URL. When it's set up locally with docker, the address is `localhost:5000`. After it, you can use `OSRMFramework.route` sending the pick-up and dropoff's latitudes and longitudes and obtain back 5 types of data:
1 - and 2. - Latitude and longitudes that compose the route
3 - Estimated route distance
4 - Estimated route duration
5 - OSM Node ids. These will be explained in the Traffic Prototype section.

In [ ]:

```
## Small Example
##
lat1, lon1 = 40.732605,-73.996070
lat2, lon2 = 40.761864,-73.980675

osm = OSRMFramework(OSRM_PATH)
lat, lon, distance, duration, node_ids = osm.route(lat1, lon1, lat2, lon2)

t = latlon2linestring(lat, lon)

plot_geometry(t)
```

In [ ]:

```
## Small Example
##
def get_route(osm, lat1, lon1, lat2, lon2):
    lat, lon, distance, duration, node_ids = osm.route(lat1, lon1, lat2, lon2)
    return_col_names = ['route', 'distance_m', 'duration_sec', 'node_ids']
    if(type(lat) == float):
        return pd.Series([np.nan] * len(return_col_names), index=return_col_name
s)
    else:
        return pd.Series([latlon2linestring(lat, lon), distance, duration, node_
ids], index=return_col_names)

routes = df_train.head().apply(lambda row: get_route(osm,
                                                     row['pickup_latitude'],
                                                     row['pickup_longitude'],
                                                     row['dropoff_latitude'],
                                                     row['dropoff_longitude']), axis=1)
routes['route'] = GeoSeries([elem[0] if type(elem) == GeoSeries else np.nan for
elem in routes['route']], index=routes.index)

routes
```

In [ ]:

```
# TEMP: because router take too much time, I've saved the routes from this sampl
e here. If you want to
# use the router anyway, comment lines where pickle is read and uncomment commen
ted lines

def feature_pipeline_3(df, target_col=TARGET,
                               pickup_latitude='pickup_latitude', dropoff_latitud
e='dropoff_latitude',
                               pickup_longitude='pickup_longitude', dropoff_longi
tude='dropoff_longitude',
                               osm_router=OSRM_PATH,
                               test_mode=None):
    EUCLIDEAN_FEAT = 'euclidean_dist_km'
    HAVERSINE_FEAT = 'haversine_dist_km'
    MANHATTAN_FEAT = 'manhattan_dist_km'
    ROUTE_DISTANCE = 'route_distance_km'
    ROUTE_DURATION = 'route_duration_min'

    FEATURES = [EUCLIDEAN_FEAT,
                MANHATTAN_FEAT,
                HAVERSINE_FEAT,
                ROUTE_DISTANCE,
                ROUTE_DURATION]

    df_copy = df.copy()

    # Execute previous features' pipeline
    df_pipeline2 = feature_pipeline_2(df_copy)
    df_copy[EUCLIDEAN_FEAT] = df_pipeline2[EUCLIDEAN_FEAT]
    df_copy[MANHATTAN_FEAT] = df_pipeline2[MANHATTAN_FEAT]
    df_copy[HAVERSINE_FEAT] = df_pipeline2[HAVERSINE_FEAT]

    if(test_mode == 'train'):
# TEST MODE
        with open('data/temp_routes_train.pickle', 'rb') as f:
# TEST MODE
            routes = pickle.load(f)
# TEST MODE
    elif(test_mode == 'test'):
# TEST MODE
        with open('data/temp_routes_test.pickle', 'rb') as f:
# TEST MODE
            routes = pickle.load(f)
# TEST MODE

#     routes = df_copy.apply(lambda row: get_route(osm,
#                                                  row['pickup_latitude'],
#                                                  row['pickup_longitude'],
#                                                  row['dropoff_latitude'],
#                                                  row['dropoff_longitude']), axis=1)
#     routes['route'] = GeoSeries([elem[0] if type(elem) == GeoSeries else np.na
n for elem in routes['route']], index=routes.index)
    df_copy[ROUTE_DISTANCE] = routes['distance_m']/1000
    df_copy[ROUTE_DURATION] = routes['duration_sec']/60

    # in the EDA, we probably treat the nulls, so for now, just drop them
    df_copy = df_copy.dropna()

    return df_copy[FEATURES + [target_col]], routes
```

```
df_train_3, routes_train = feature_pipeline_3(df_train, test_mode='train')
df_test_3, routes_test = feature_pipeline_3(df_test, test_mode='test')

df_train_3.head()
```

In [ ]:

```
model_3 = LinearRegression()
model_3.fit(df_train_3.drop(TARGET, axis=1), df_train_3[TARGET])
y_test_pred_3 = model_3.predict(df_test_3.drop(TARGET, axis=1))
```

In [ ]:

```
#####
rmse_3 = np.sqrt(mean_squared_error(df_test_3[TARGET], y_test_pred_3))
iteration_results['exp_3_osrm'] = rmse_3
print(f'RMSE Error: {rmse_3}')

print_evaluation(df_test_3[TARGET], y_test_pred_3, return_errors=False)
plt.xlim([0, 50]);
```

In [ ]:

```
iteration_results
```

Exercise OSRM

1. Set up router in your local machine and call the route service using the example above
2. Call the `route` service by using a pick-up point from inside New York city and one outside the city, in another state, for example. What happened? What if both pick-up and drop off are outside New York city? If you want to confirm, try to use the "Small Example" code from below to check your assumptions

In [ ]:

```
# One point inside, one outside NY

inside_ny_lat, inside_ny_lon = [40.870450, -73.879747]
outside_ny_lat, outside_ny_lon = [42.297001, -71.113800]

osm = OSRMFramework(OSRM_PATH)
lat, lon, distance, duration, node_ids = osm.route(inside_ny_lat, inside_ny_lon,
outside_ny_lat, outside_ny_lon)

t = latlon2linestring(lat, lon)

print(f'Distance (KM): {distance/1000}')
print(f'Duration (minutes): {duration/60}')
plot_geometry(t)
```

In [ ]:

```
# Both points outside, 1 points gets snapped, the other not

outside1_ny_lat, outside1_ny_lon = [42.329999, -71.072593]
outside2_ny_lat, outside2_ny_lon = [42.297001, -71.113800]

osm = OSRMFramework(OSRM_PATH)
lat, lon, distance, duration, node_ids = osm.route(outside1_ny_lat, outside1_ny_
lon, outside2_ny_lat, outside2_ny_lon)

t = latlon2linestring(lat, lon)

print(f'Distance (KM): {distance/1000}')
print(f'Duration (minutes): {duration/60}')
plot_geometry(t)
```

2nd Exploratory Analysis

Check why we have some route distance == 0

In [ ]:

```
df_train_3_analysis = df_train_3.merge(df_train, how='left', left_index=True, ri
ght_index=True)

df_train_3_analysis['route_distance_km'] = routes_train['distance_m']/1000
display(df_train_3_analysis.loc[df_train_3_analysis['route_distance_km'] == 0].h
ead()) # PROBLEM pickup == dropoff

############################################################
### Filter where pick up and drop off are the same
############################################################
```

Some leftover routes still have route_distance == 0, investigate

In [ ]:

```
df_train_3_analysis = df_train_3_analysis.loc[df_train.index].copy()

df_train_3_analysis['distances'] = haversine_distance(df_train_3_analysis['picku
p_latitude'], df_train_3_analysis['dropoff_latitude'],
                                df_train_3_analysis['pickup_longitude'], df_train
_3_analysis['dropoff_longitude'])

df_train_3_analysis.loc[df_train_3_analysis['route_distance_km'] == 0].head() ##
# ?? - ~65 examples

############################################################
### Filter where route_distance == 0 for next iteration###
############################################################
```

Check price per kilometer

In [ ]:

```
temp = df_train_3_analysis.loc[df_train_3_analysis['route_distance_km'] != 0]

price_km = temp['fare_amount_y']/temp['route_distance_km']
display(price_km.describe(percentiles=[.9, .95, .99]))

##########################################################
### Filter where price per km > 5 - 95th percentile    ###
##########################################################
```

Embed Nearest neighbors fares

What do have trips in common?

Last approach, we're going to find the most similar trips that were performed in the past. Based on the fare prices specification (https://www1.nyc.gov/site/tlc/passengers/taxi-fare.page), we can see that the day of the week, hour and distance make an impact in the final overall price. Therefore, we'll try to cover this features based on:

- Trips that started in the same point in the week (full weekly seasonality)
- Trips that started and finished roughly at the same locations

Given the most similar trips for a given specific taxi trip, we can then average out their fares to produce a new feature for our model, `nn_avg_fare`. The only hyperparameter we have to deal with is the number of nearest neighbors that we are going to average our final fare from.

In [ ]:

```
## Small Example
##
def nn_avg_fare(df, target_col=TARGET,
                pickup_latitude='pickup_latitude', pickup_longitude='pickup_long
itude',
                dropoff_latitude='dropoff_latitude', dropoff_longitude='dropoff_
longitude',
                pickup_datetime_col='pickup_datetime', n_neighbors=7, test_mode=
'test',
                nn_avg_fare_model=None):

    df_copy = df.copy()
    minutes_since_monday_midnight = 'minutes_since_monday_midnight' # monday 00:
00 = 0, tuesday 00:00 = 24, so on..
    df_copy[minutes_since_monday_midnight] = df_copy[pickup_datetime_col].dt.day
ofweek * (24*60) + \
                                             df_copy[pickup_datetime_col].dt.h
our * 60 + \
                                             df_copy[pickup_datetime_col].dt.m
inute


    nn_features = [pickup_latitude, pickup_longitude,
                       dropoff_latitude, dropoff_longitude,
                       minutes_since_monday_midnight]
    nn_data = standardize_features(df_copy[nn_features])             # for k-mea
ns it's important to standardize feat.
    nn_data[target_col] = df_copy[target_col]
    if(test_mode == 'train'):                                        # if it's t
raining, used dataset to create model
        nn_avg_fare_model = KNeighborsRegressor(n_neighbors=n_neighbors)
        nn_avg_fare_model.fit(nn_data.drop(target_col, axis=1), nn_data[target_c
ol])
    predictions = nn_avg_fare_model.predict(nn_data.drop(target_col, axis=1))
    return predictions, nn_avg_fare_model

#####
#####
df_train_temp = df_train.head(50).copy()
df_test_temp = df_train.iloc[50:53].copy()
df_train_temp['nn_avg_fare'], nn_avg_fare_model = nn_avg_fare(df_train_temp, tes
t_mode='train')
df_test_temp['nn_avg_fare'] , nn_avg_fare_model = nn_avg_fare(df_test_temp,
                                                              test_mode='test',
                                                              nn_avg_fare_model=
nn_avg_fare_model)

df_test_temp
```

In [ ]:

```
def feature_pipeline_4(df, target_col=TARGET,
                               pickup_latitude='pickup_latitude', dropoff_latitud
e='dropoff_latitude',
                               pickup_longitude='pickup_longitude', dropoff_longi
tude='dropoff_longitude',
                               pickup_datetime_col='pickup_datetime',
                               osm_router=OSRM_PATH,
                               nn_avg_fare_model=None, n_neighbors=7,
                               test_mode=None):
    EUCLIDEAN_FEAT = 'euclidean_dist_km'
    HAVERSINE_FEAT = 'haversine_dist_km'
    MANHATTAN_FEAT = 'manhattan_dist_km'
    ROUTE_DISTANCE = 'route_distance_km'
    ROUTE_DURATION = 'route_duration_min'
    NN_AVG_FARE = 'nn_avg_fare'

    FEATURES = [EUCLIDEAN_FEAT,
                MANHATTAN_FEAT,
                HAVERSINE_FEAT,
                ROUTE_DISTANCE,
                ROUTE_DURATION,
                NN_AVG_FARE]

    df_copy = df.copy()

    # Execute previous features' pipeline
    df_pipeline3, routes3 = feature_pipeline_3(df_copy, test_mode=test_mode)
    df_copy[EUCLIDEAN_FEAT] = df_pipeline3[EUCLIDEAN_FEAT]
    df_copy[MANHATTAN_FEAT] = df_pipeline3[MANHATTAN_FEAT]
    df_copy[HAVERSINE_FEAT] = df_pipeline3[HAVERSINE_FEAT]
    df_copy[ROUTE_DISTANCE] = df_pipeline3[ROUTE_DISTANCE]
    df_copy[ROUTE_DURATION] = df_pipeline3[ROUTE_DURATION]


    ###### Outliers Filtering
    # in the EDA, we probably treat the nulls, so for now, just drop them
    df_copy = df_copy.dropna()

    df_copy = df_copy.loc[df_copy[ROUTE_DISTANCE] != 0].copy()

    price_per_km = df_copy[target_col]/(df_copy[ROUTE_DISTANCE])
    df_copy = df_copy.loc[price_per_km < 5].copy()
    ######

    ###### NN avg fare model
    if(test_mode == 'train'):
        df_copy[NN_AVG_FARE], nn_avg_fare_model = nn_avg_fare(df_copy, test_mode
='train', n_neighbors=n_neighbors)
    else:
        df_copy[NN_AVG_FARE], nn_avg_fare_model = nn_avg_fare(df_copy, test_mode
='test',
                                                              nn_avg_fare_model=
nn_avg_fare_model)
    ######

    return df_copy[FEATURES + [target_col]], routes3, nn_avg_fare_model

df_train_4, routes_train, nn_avg_fare_model = feature_pipeline_4(df_train, test_
mode='train')
```

```
df_test_4, routes_test, nn_avg_fare_model = feature_pipeline_4(df_test, test_mod
e='test', nn_avg_fare_model=nn_avg_fare_model)

df_train_4.head()
```

In [ ]:

```
model_4 = LinearRegression()
model_4.fit(df_train_4.drop(TARGET, axis=1), df_train_4[TARGET])
y_test_pred_4 = model_4.predict(df_test_4.drop(TARGET, axis=1))
```

In [ ]:

```
#####
rmse_4 = np.sqrt(mean_squared_error(df_test_4[TARGET], y_test_pred_4))
iteration_results['exp_4_knn_rmse'] = rmse_4
print(f'RMSE Error: {rmse_4}')

print_evaluation(df_test_4[TARGET], y_test_pred_4, return_errors=False)
plt.xlim([0, 50]);
```

In [ ]:

```
iteration_results
```

Personal Note: Although we've seen improvements in this iteration, it's in the modelling's objetive to determine the usefulness of this feature. The previous iterations' results didn't improve much because, besides that the features were correlated, they also didn't capture all the factors that compose the final fare price given the dataset that we were given to. The KNNRegression approach that we execute, capture close by trips and "mimics" their final price as feature, without putting much thought on what are the **main driving factors that compose the final fare**. If performance is the objective, this is always a good feature. However, if we were modelling/explaining this process, I wouldn't personally go for this feature

Can you think of more features?

Exercise KNNRegressor

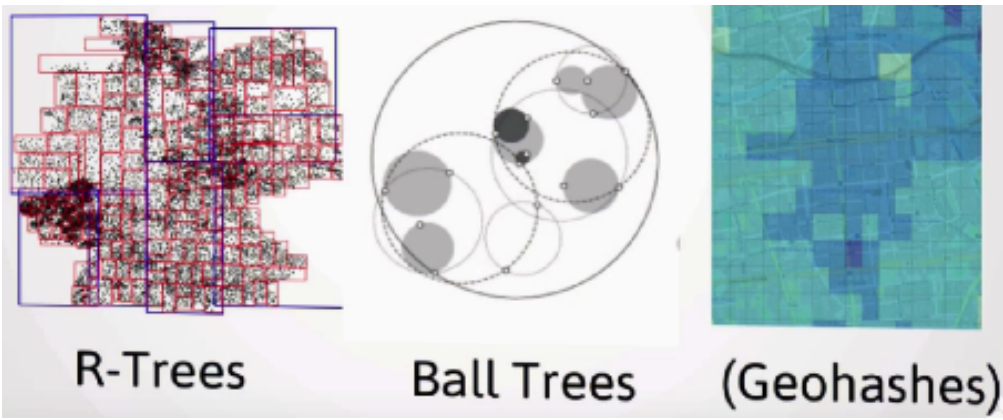
Using the codes above, define what is the best value of `n_neighbors` that minimizes the test RMSE?

In [ ]:

```
##############################################################################
############################
## Enter values to check different values for neighbors
##############################################################################
############################
# replace 1 by the minimum number of neighbors you want to test
START_N = 1

# replace 2 by the maximum number of neighbors you want to test (if you chose a
 too high number and the execution
# takes too long: interrupt the kernel with the square symbol and choose a small
er number)
END_N = 2
##############################################################################
############################

n_knn = range(START_N, END_N+1)
for n in n_knn:
    df_train_ex_knn, _, nn_avg_fare_model = feature_pipeline_4(df_train, test_mo
de='train', n_neighbors=n)
    df_test_ex_knn, _, _ = feature_pipeline_4(df_test, test_mode='test', nn_avg_
fare_model=nn_avg_fare_model)

    model_ex_knn = LinearRegression()
    model_ex_knn.fit(df_train_ex_knn.drop(TARGET, axis=1), df_train_ex_knn[TARGE
T])
    y_test_pred_ex_knn = model_ex_knn.predict(df_test_ex_knn.drop(TARGET, axis=1
))

    rmse_ex_knn = np.sqrt(mean_squared_error(df_test_ex_knn[TARGET], y_test_pred
_ex_knn))
    print(f'RMSE Error - N = {n}: {rmse_ex_knn}')
```

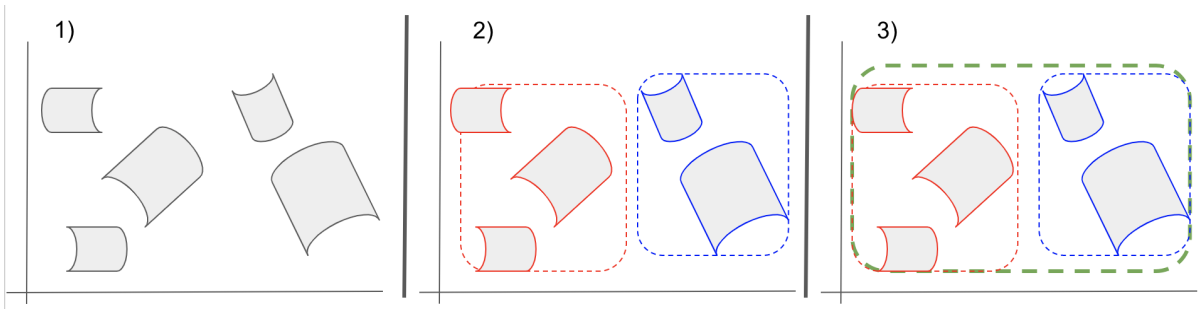
- [x] OK Intro + Exploratory Data Analysis
- [x] OK Feature Engineering - Distance and Time
- [ ] Spatial Processing Performance
- [ ] Traffic Prototype
- [ ] Map Features

How to speed up spatial queries:



Example of spatial indexes. Source: Youtube: Alexander Müller - Spatial Range Queries Using Python In-Memory Indices (https://www.youtube.com/watch?v=_95bSEqMzUA)

Spatial Indexes:



Insert process of a R-Tree

> The R-Tree helps with filtering out the majority of the dataset that lives outside of the point's bounding box. After the big filtering is done, all the other calculations follow in the same way.

Here (https://geoffboeing.com/2016/10/r-tree-spatial-index-python/) is an example that filter streets based on a city polygon using R-Trees.

Lets say we wanted to check which routes intersect with a route of interest that we manually selected. Geopandas (http://geopandas.org/) can perform spatial joins (https://medium.com/@bobhaffner/spatial-joins-in-geopandas-c5e916a763f3) in order to join tables based on the relationship types between two geometry columns (https://shapely.readthedocs.io/en/latest/manual.html#binary-predicates). Here, we use the `intersects` operation in GeoPandas to check whether two linestrings intersect each other. However, the operation is unfortunately still done element per element (https://matthewrocklin.com/blog/work/2017/09/21/accelerating-geopandas-1). With the R-Tree, we still do element by element search, but we can firstly subselect close-by elements and then perform `intersects` with a small subset of the total dataset. With bigger datasets, this difference in approaches can lead to big differences in performance, as we can see in the example below:

In [ ]:

```
def create_r_tree(gdf, geometry_col):
    idx = index.Index()
    #Populate R-tree index with bounds of grid cells
    for ix, cell in gdf.iterrows():
        # in GeoPandas, there's always a geometry col and it's always a shapely
        idx.insert(ix, cell[geometry_col].bounds)
    return idx
```

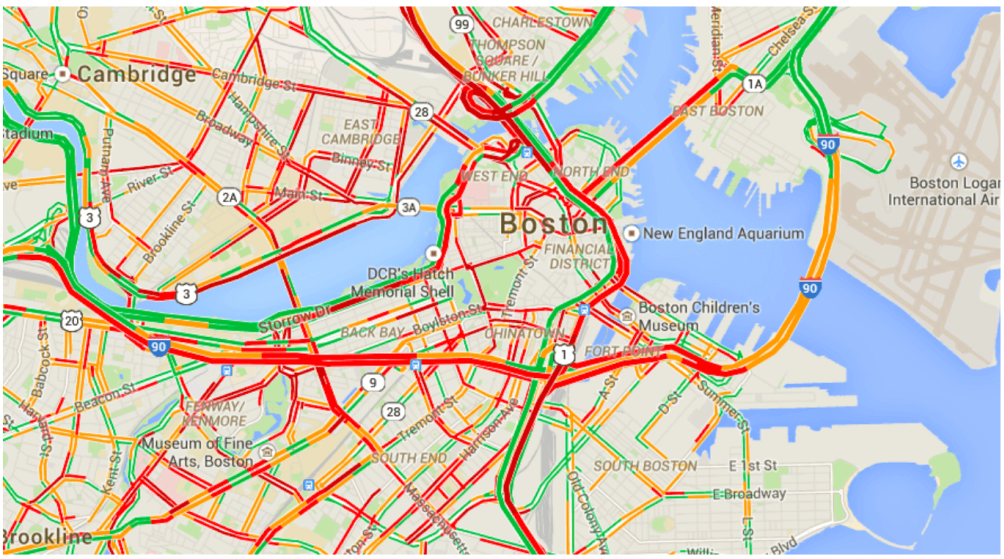
In [ ]:

```
df_geoix = routes_train.loc[(routes_train['distance_m'] > 100) & (routes_train[
'distance_m'] < 500)]
df_geoix = GeoDataFrame(df_geoix, geometry='route') # convert df to GeoDf, allow
methods such as 'intersects'
n_samples = np.arange(7500, 75001, 7500)            # different resample sizes
 (10)
times_no_geoix = []                                 # store processing time for
 searches with no spatial index
times_geoix = []                                    # store processing time for
 searches with R-Tree
intersect_line = df_geoix.loc[1584]['route']        # linstring geometry

for n_sample in n_samples:
    print(f'N Samples: {n_sample}')
    df_geoix_sample = df_geoix.sample(n_sample, replace=True, random_state=42).r
eset_index(drop=True)
    idx = create_r_tree(df_geoix_sample, 'route')

    #### without index
    time_no_geoix = time.time()
    filter_1 = df_geoix_sample.intersects(intersect_line)
    routes_ix = filter_1[filter_1 == True].index
    times_no_geoix.append(time.time() - time_no_geoix)
    print(f'* intersecting routes found without index: {len(routes_ix)}')

    #### with index
    # Filter possible candidates by bouding boxes
    time_with_geoix = time.time()
    idxs = list(idx.intersection(intersect_line.bounds))
    if(len(idxs) > 0):
        # Now do actual intersection
        filter_2 = df_geoix_sample.loc[idxs].intersects(intersect_line)
        routes_ix = df_geoix_sample.loc[filter_2[filter_2 == True].index]
        times_geoix.append(time.time() - time_with_geoix)
        print(f'* intersecting routes found with index: {len(routes_ix)} \n')

plt.plot(n_samples, times_no_geoix)
plt.plot(n_samples, times_geoix)
plt.title('Comparison between search times without index and R-Tree')
plt.xlabel('Number of Geometries')
plt.ylabel('Search Time (seconds)')
plt.legend(['Without Index', 'R-Tree']);
```

You might have notices the first line of code in the code block above, where I filtered for short routes. The reason I've done this is that the spatial indexes are efficient when the object you're trying to join with results in a small number of intersection with other elements. If we had selected a long street that intersected with most routes from our dataset, the index would filter almost no elements and the following search operation would be similar to the one performed by GeoPandas

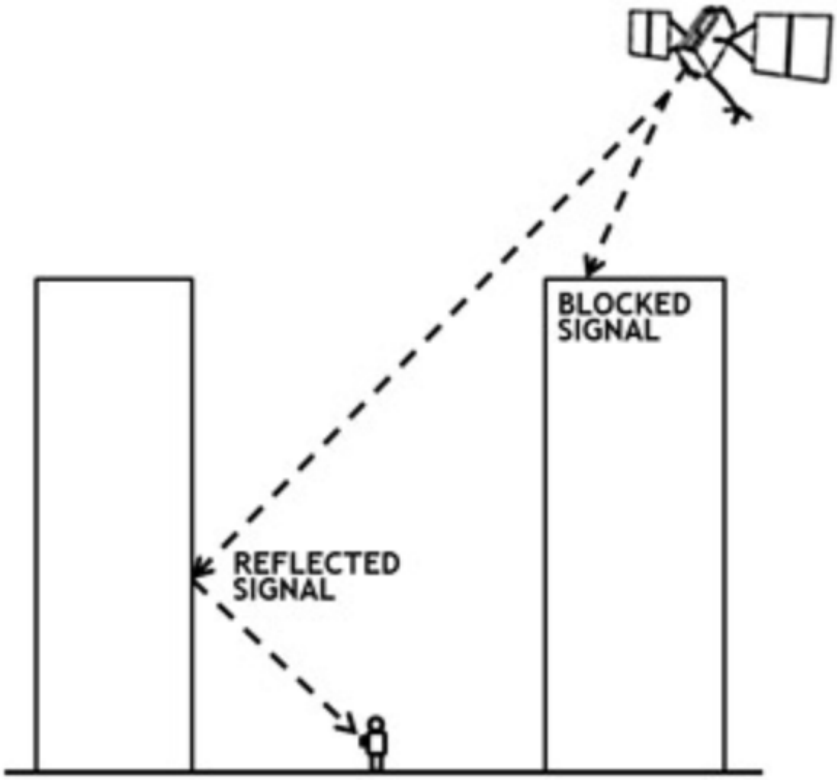
- [x] OK Intro + Exploratory Data Analysis
- [x] OK Feature Engineering - Distance and Time
- [x] OK Spatial Processing Performance
- [ ] Traffic Prototype
- [ ] Map Features

Traffic Prototype



Traffic information from Google Maps

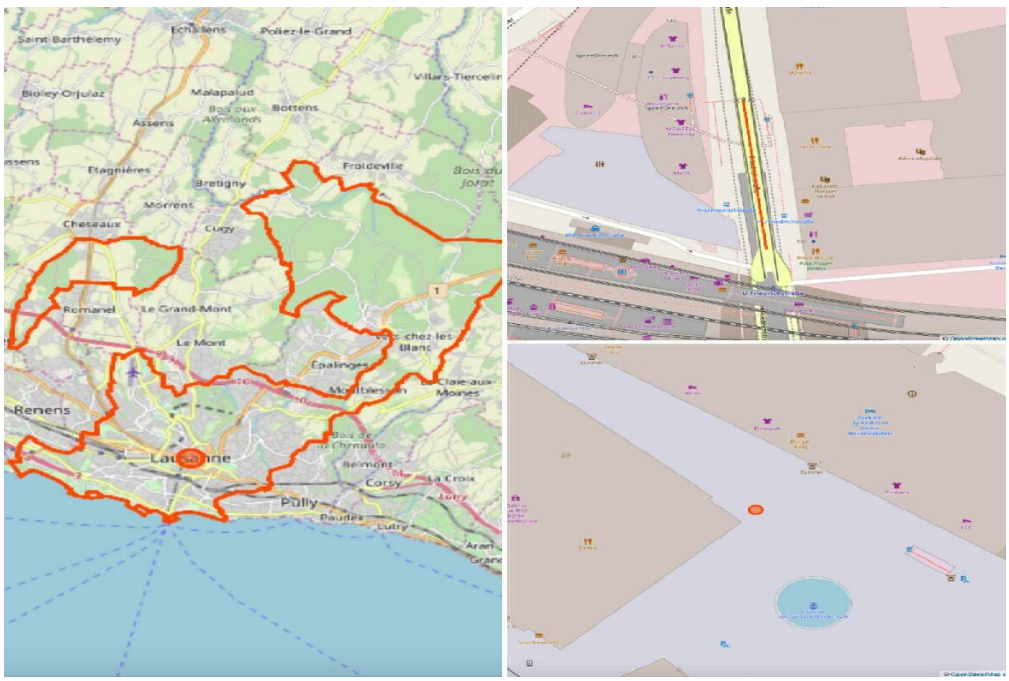
GPS Traces



Bad GPS Signal. Source: https://www.gps.gov/systems/gps/performance/accuracy/ (https://www.gps.gov/systems/gps/performance/accuracy/)

OSM Data Representation

Nodes, Ways and Relations



Data types in OSM

Everything that is inside OSM is represent by one of three possible geometries (Reference (https://labs.mapbox.com/mapping/osm-data-model/)):

- **Node** Example: https://www.openstreetmap.org/node/6343276469 (https://www.openstreetmap.org/node/6343276469)

- **Way** Example: https://www.openstreetmap.org/way/4402228 (https://www.openstreetmap.org/way/4402228)
- **Relations** Example: https://www.openstreetmap.org/relation/1685018 (https://www.openstreetmap.org/relation/1685018)

Tags

Tags are the metadata of each data type defined above. All data types contains a set of possible (but not mandatory) tags related to the tags' semantics. For instance, if you look at the left side of the screen in any of the data types examples above, you'll see that:

- **Nodes** Example: `latitude/longitude`
- Street **Ways** Example: `bicycle path` , if it has a `sidewalk` and mainly, what is it's `maximum driving speed` . **It doesn't always have maximum speed**.
- An important feature about **Relations** is their administrative level (https://wiki.openstreetmap.org/wiki/Key:admin%20level?uselang=en-GB).

Traffic Proof of Concept (POC)

OSRM allows you to define what is the street speed by allowing you to provide an external CSV file that sets it. For that, you need to define the speeds segment by segment, *i.e.*, by each pair of sequential node ids, you can set what is the street speed at that point (Reference (https://github.com/Project-OSRM/osrm-backend/wiki/Traffic)).

In [ ]:

```
# redo route command with previous router

lat1, lon1 = 40.732605,-73.996070
lat2, lon2 = 40.761864,-73.980675

lat, lon, distance, duration, node_ids = osm.route(lat1, lon1, lat2, lon2)

t = latlon2linestring(lat, lon)

plot_geometry(t)
```

For each segment between 2 nodes, set speed to lowest possible = 1 km/h. NOT 0

In [ ]:

```
node_from = []
node_to = []
node_speed = []
i = 1
while i < len(node_ids):
    node_from.append(node_ids[i-1])
    node_to.append(node_ids[i])
    node_speed.append(1)
    i += 1
blocked = pd.DataFrame({'node_from': node_from, 'node_to': node_to, 'node_speed'
: node_speed})

# Save file with segment speeds with no columns headers nor row indexes
# blocked.to_csv('router/test_traffic.csv', header=False, index=False)

blocked.head()
```

Reset router updating street speeds informations

The procedure here is the same as the previous router set up. The `only difference is the new parameter in osrm-customize`

`--> --segment-speed-file /data/traffic_file.csv .`

* **Running in you local machine:** * docker run -t -v " $PWD:/data$ " osrm/osrm-backend osrm-extract -p /opt/car.lua /data/router/new_york_city.osm.pbf * docker run -t -v " $PWD:/data$ " osrm/osrm-backend osrm-partition /data/router/new_york_city.osm.pbf * docker run -t -v " $PWD:/data$ " osrm/osrm-backend osrm-customize /data/router/new_york_city.osm.pbf --segment-speed-file /data/data/test_traffic.csv * docker run -t -i -p 5000:5000 -v " $PWD:/data$ " osrm/osrm-backend osrm-routed --algorithm mld /data/router/new_york_city.osm.pbf

* **Manual execution for AMLD Workshop, in the workshop's root directory:** * docker-compose stop osrm-router * docker-compose rm osrm-router * docker-compose create osrm-router * docker run -t -v "${PWD}:/data" osrm/osrm-backend osrm-customize /data/router/new_york_city.osm.pbf --segment-speed-file /data/data/test_traffic.csv * docker-compose start osrm-router

The commands above are already coded in the file `restart_osrm_traffic.sh` . **If you can't execute the shell script, like Windows users, you can execute the above commands by copying/pasting them in your terminal.** By executing it, we're able to stop the current router in the docker-compose containers, update the router with traffic information and restart it without having the stop the workshop's docker-compose structure, *i.e.*, without stoping this jupyter notebook.

- In your terminal, in the workshop's root folder, execute the shell script.

Call router again and check new route

With the new router set up, we can calculate the route for the same pick-up/drop-off points as before and see how it's changed. We can see that the router avoids as best as possible to stay away from the segments we marked as heavy traffic, *i.e.*, segment speed = 1 km/h.

In [ ]:

```
lat, lon, distance, duration, node_ids = osm.route(lat1, lon1, lat2, lon2)

t = latlon2linestring(lat, lon)

plot_geometry(t)
```

Match noisy GPS to Node IDs

The POC works fine if you have the node id's for all your route points, which unfortunately, isn't the case. Besides:

- GPS sampling might be not stable, *e.g.*, some times we have sampling of 5 secs, other times 10 secs.
- Points might not fall exactly on the street due to GPS inaccuracy

So, lets recreate a possible real world GPS trace by taking a trace returned by the router in section Embed Nearest neighboors fares and disturb it:

In [ ]:

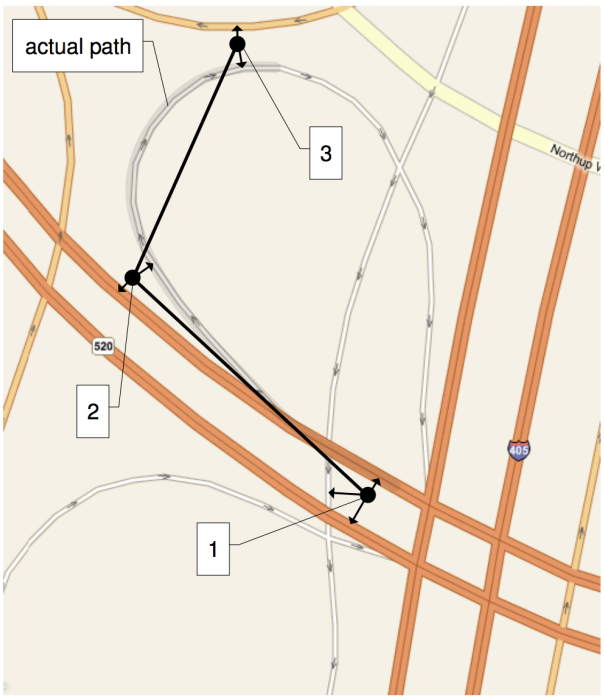
```
lines = df_train.merge(routes_train, how='left', left_index=True, right_index=Tr
ue)
trace = lines['route'].iloc[30]

lon, lat = trace.xy[0], trace.xy[1]

lat, lon = np.delete(lat, -3),  np.delete(lon, -3)       # cut corner at the bot
tom part
np.random.seed(42)                                       # choose 10 random poin
ts to disturb
choices = np.random.choice(range(len(lat)), size=10, replace=False)
lat[choices] = lat[choices] + np.random.normal(0, 0.001) # add noise
lon[choices] = lon[choices] + np.random.normal(0, 0.001) # add noise

t = latlon2linestring(lat, lon)

plot_geometry(t)
```

We can see that some of the points don't even fall on a street segment and, like in the route close to the bottom-right side, a node indicating a turn is missing in a way that, when we connect the nodes with a line, the line goes through the building blocks.

As we know that every node in OSM has a lat/lon associated with it, we could associate each trace point to the closest node. As Newson and Krumm showed (Reference (https://www.ismll.uni-hildesheim.de/lehre/semSpatial-10s/script/6.pdf)):

- This is error prone, as this kind of approach in street condensed networks can match GPS points to unrelated street segments.
- The approach doesn't take the previous and future GPS points in order to match a GPS point to the street, *i.e.*, if previous and next point are over a bridge, it's highly improbable that the current matched GPS point is outside the bridge, even if it's closer to a node outside the bridge.

We can see this kind of examples in the picture below:



GPS matching potential errors - as in https://www.ismll.uni-hildesheim.de/lehre/semSpatial-10s/script/6.pdf (https://www.ismll.uni-hildesheim.de/lehre/semSpatial-10s/script/6.pdf)

Snap noisy GPS points to street using map matching

Newson and Krumm created an approach that takes into account the whole sequence of nodes and then try to match them in a probabilistic way.

OSRM already provide it for us out of the box and we can use the OSRMFramework class to extract the main information using the router we have already set up.

- API - http://project-osrm.org/docs/v5.5.1/api/#match-service (http://project-osrm.org/docs/v5.5.1/api/#match-service)

Main parameters to be understood here are:

- **geometries**
- **timestamps**
- **radiuses**

In the complete version of this notebook, we fully describe the parameter above. Check it out.

Lets now take a look of how the disturbed street is after map matching:

In [ ]:

```
lat, lon, nodes_id = osm.match(lat, lon, timestamps=None, radiuses=None)

t = latlon2linestring(lat, lon)

plot_geometry(t)
```

Traffic Data Processing

Trace accumulation per time window

So until now, we know how we can have multiple GPS points and their associated node ids and timestamps. However, `how do I turn them into traffic information?`

Two sources of data:

- **Real time**
- **Historical aggregates**

You can check that these are the two options used by google here: https://www.google.com/maps/place/Nova+Iorque,+NY,+EUA/@40.6971494,-74.2598655,10z/data=!3m1!4b1!4m5!3m4!1s0x0:0x74.0059728!5m1!1e1 (https://www.google.com/maps/place/Nova+Iorque,+NY,+EUA/@40.6971494,-74.2598655,10z/data=!3m1!4b1!4m5!3m4!1s0x0:0x74.0059728!5m1!1e1)

> The accessibility to public car's GPS traces is still rare. When companies have access to this kind of information, they usually don't make publicly available. In this traffic section we're working with a mock of GPS signals, made by OSRM itself to build routes from A to B. The objective of this part then is not to analyze data or summaries statistics, but to understand possible techniques on how to embed traffic information into your OSRM.

In [ ]:

```
lines_k = lines.loc[lines['pickup_datetime'].dt.year == 2015][['pickup_datetime'
, 'node_ids']].copy()
lines_k['pickup_datetime'] = lines_k['pickup_datetime'].dt.strftime('%H:00:00')

display(lines_k.head(3))
```

The **objective** of the following code blocs is to convert the dataframe above into a a list line segments (2 node ids), the hour bin and how many times this segment appeared in this time bin, like the example below:

datetime	count	segment linestring
13:00	5	LINESTRING (-73.9600437 40.7980478, -73.959547...

For that we need to:

1. Process every node id pair from every route, *i.e.*, the route's segments
2. Count how many times the segment occurred grouped by hour bin
3. Extract node's position (lat/lon) and convert segments to geometric formats

In [ ]:

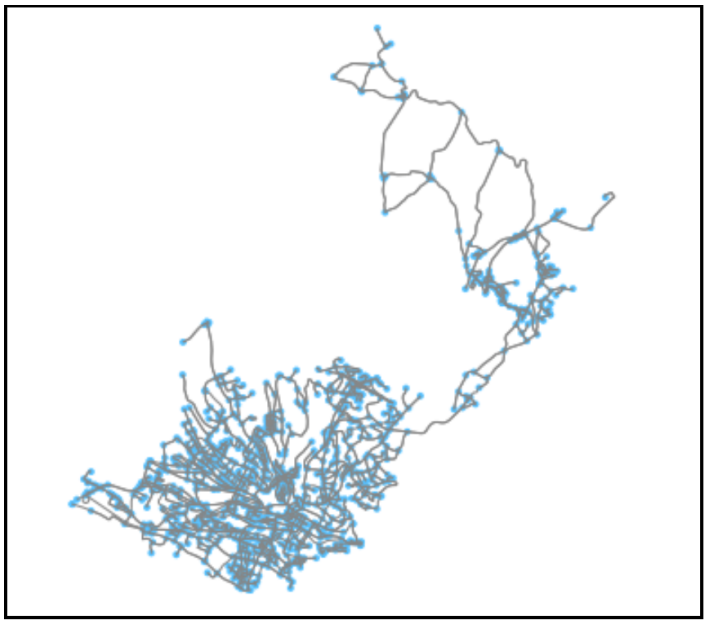
```
# 1. Process every node id pair from every route, i.e., the route's segments

from_node = []
to_node = []
seg_date = []
for ix, row in lines_k.dropna().iterrows():
    row_df = pd.DataFrame()
    lat_lon_pairs = []
    i = 1
    while i < len(row['node_ids']):
        from_node.append(row['node_ids'][i-1])
        to_node.append(row['node_ids'][i])
        seg_date.append(row['pickup_datetime'])
        i += 1

seg_df = pd.DataFrame({'seg_date': seg_date, 'from_node': from_node, 'to_node':
to_node})
seg_df.head()
```

In [ ]:

```
# 2. Count how many times the segment occurred grouped by hour bin

def count_duplicates(group):
    group_res = group.groupby(['from_node', 'to_node']).size().reset_index().ren
ame(columns={0:'records'})
    group_res['seg_date'] = group.name
    return group_res

group_res = seg_df.groupby('seg_date').apply(count_duplicates).reset_index(drop=
True)
group_res.head()
```

Tasks 1 and 2 are straightforward, needing basically some processing. However, for task 3, we need to map our node ids to lat/lons and, unfortunately, **OSRM doesn't provide us with that information**, for optimization reasons. You check some issues in Github here (https://github.com/Project-OSRM/osrm-backend/issues/5310) and here (https://github.com/Project-OSRM/osrm-backend/issues/5490) and some more at their Github repository. Enter...OSMNx (https://geoffboeing.com/2016/11/osmnx-python-street-networks/).

Open Street Maps + NetworkX (OSMnx) and RouteAnnotator



Lausanne's driving network

OSMnx is a framework that works over OSM data and has multiple ways to extract data about street networks, including, nodes, ways, relations and **all the tags associated with them**.

The `RouteAnnotator` class access OSMnx, retrieve desired street network and extract all metadata related to nodes, ways and node segments (node pairs and nodes belonging to same way). For now, it can only extract a network based on osmnx.graph_from_place (https://osmnx.readthedocs.io/en/stable/osmnx.html#osmnx.core.graph_from_place) function but you can read more about other ways to retrieve data in OSMnx's documentation.

RouteAnnotator creates three main functions similar to Mapbox's route-annotator (https://github.com/mapbox/route-annotator). Its main functions are:

- **segment_lookup** - provided a list of node ids with size N, it returns $N - 1$ segments containing data such as the way_id that this segment belongs to
- **way_lookup** - provided a list of ways_id, it returns the metadata associated with all ways, including an ordered list of node_ids that compose that way
- **node_lookup** - provided a list of node_ids, it returns the metadata associated with all nodes, including latitude and longitude.

We're going to use RouteAnnotator in order to retrieve the nodes' lat/lon and keep processing our traffic dataset:

> **Observation**: OSMnx download a uncompressed geoJSON file from OSM and this file tends to be quite big. While executing the code block below inside docker, we had memory problems because of that, so we decided to execute locally, save the lookups and importing them locally for this workshop.

> The lookups are exactly the same as if you executed the commented lines, but mind the memory consumption, as it can break your container.

In [ ]:

```
# download and process new york city street graph
#
# ra = RouteAnnotator('new york, USA', 'drive_service')
# ra.build_lookups()

# AMLD version, load already saved lookups
ra = RouteAnnotator.AMLD_local_lookups('new york, USA', 'drive_service')
```

In [ ]:

```
# 3. Extract node's position (lat/lon) and convert segments to geometric formats

line_seg = []
for ix, row in group_res.iterrows():
    try:
        metadata1 = ra.node_lookup(row['from_node'])  # use RouteAnnotator to re
trieve node's metadata
        m1_lat_lon = [metadata1['y'], metadata1['x']] # such as lat/lon
        metadata2 = ra.node_lookup(row['to_node'])
        m2_lat_lon = [metadata2['y'], metadata2['x']]
        line_segment = latlon2linestring(lat = [m1_lat_lon[0], m2_lat_lon[0]],
                                         lon = [m1_lat_lon[1], m2_lat_lon[1]])[0
]
        line_seg.append(line_segment)
    except Exception:
        line_seg.append(np.nan)
        continue


group_res = GeoDataFrame(group_res, geometry=line_seg)
group_res.head()
```

Finally, we have the street segment and their respective "popularity", *i.e.*, count, aggregated by hour. In a real GPS database, the `records` columns would be replaced by the average speed of gps traces that passed through it. Lets visualize it!

After finalizing the dataset above, we could save them as the first CSV file that we've saved containing our desired metric.

In [ ]:

```
w1 = keplergl.KeplerGl(height=500)
w1.add_data(data=group_res[['seg_date', 'geometry', 'records']].dropna(), name=
'traces')
w1
```

- [x] OK Intro + Exploratory Data Analysis
- [x] OK Feature Engineering - Distance and Time
- [x] OK Spatial Processing Performance
- [x] OK Traffic Prototype
- [ ] Map Features

Retrieving Extra Map Features

Objective: Extract extra information, we're open to new kind of possibilities when doing our analysis/features:

- Do you want to check what are the most illuminated streets in your city? Maybe you can build a router only for safe illuminated streets, or find the streets with most lamps for your happy dog (http://sk53-osm.blogspot.com/2013/04/maps-for-dogs-or-lamp-posts-in-chains.html)
- You can investigate which neighbors have better living features (https://github.com/Z3tt/30DayMapChallenge/blob/master/Day15_Names/Names_BerlinRoads.pdf), such as parks, benches, business.

We can locally search for nearby elements in OSM by clicking with the right click over a map region an choosing `query features` , like in the image below:

Local features query in OSM

pyosmium

As the name implies, pyosmium is a framework to work with the Osmium Library (https://osmcode.org/libosmium/) from OSM. It provides a sequential way to access each of OSM's data types and callbacks for data processing.

To start working with it:

- We first create a class that inherits from `osmium.SimpleHandler`
- Pyosmium will provide standards callback methods for every element present inside the osm.pbf file:
  - `way`
  - `node`
  - `relation`
- And a standard method to read the osm.pbf file, called `apply_file`

Lets say you want to check where and how distributed are New York's schools. The *tag* that defines the node as a school, or other types of objects is usually defined by a tag `amenity`, in this case, `== school`. There's a really good list of amenities at OSM wikipedia page (https://wiki.openstreetmap.org/wiki/Key:amenity). Lets take a look on an example:

In [ ]:

```
class TestPyosmium(osmium.SimpleHandler):          # class must inherit from osm
ium.SimpleHandler
    def __init__(self, pbf_path, amenity_name):
        osmium.SimpleHandler.__init__(self)        # initialize pyosmium
        self.amenity_name = amenity_name           # parameter, search for this
 specific amenity
        self.wkb_fab = osmium.geom.WKBFactory()    # builds geometry over OSM ob
jects
        self.points = []                           # store points geometries
        self.names = []                            # store points names

        self.apply_file(pbf_path, locations=True)  # initialize osm.pbf file pro
cess

        self.points = GeoSeries(self.points)       # AFTER process is done, conv
ert collected geometry into GeoSeries
        self.df = GeoDataFrame({'name': self.names}, geometry=self.points) # con
vert all data to DataFrame


    def node(self, node):
        # TagList can't be converted to dict automatically, see:
        # https://github.com/osmcode/pyosmium/issues/106
        tags_dict = {tag.k: tag.v for tag in node.tags}
        if('amenity' in tags_dict.keys() and tags_dict['amenity'] == self.amenit
y_name):
            wkb = self.wkb_fab.create_point(node)   # extract Point's hex locati
on data
            points = wkblib.loads(wkb, hex=True)    # convert hex data to WKB ge
ometry format
            self.points.append(points)              # store geometry in list
            self.names.append(tags_dict['name'] if 'name' in tags_dict.keys() el
se '') # store name IF name exist
```

In [ ]:

```
init_time = time.time()
PBF_PATH = 'data/new_york_city.osm.pbf'
test_pyosmium = TestPyosmium(PBF_PATH, amenity_name='school')
print(f'Execution time: {(time.time() - init_time)/60} minutes')
```

In [ ]:

```
plot_geometry(test_pyosmium.df['geometry'], marker_cluster=True)
```

Exercise pyosmium

Provide a helper map for tourists in NY showing them where are the concentration of drinkable water or public toilets in the city.

In [ ]:

```
init_time = time.time()
PBF_PATH = 'data/new_york_city.osm.pbf'
test_pyosmium = TestPyosmium(PBF_PATH, amenity_name='toilets')
print(f'Execution time: {(time.time() - init_time)/60} minutes')

plot_geometry(test_pyosmium.df['geometry'], marker_cluster=True)
```

Things I'd like to cover if we had infinite workshop time

- **Spatial Statistics and pysal**
- **GPS Traces database, cleaning and processing**
- **Demand Modelling and Demand Prediction**