Part 1
Part 2

# Statistical Learning - Homework 1

Code

*Davide Aureli, Andrea Marcocchia and Dario Stagnitto*

*15/04/2018*

# Part 1

## Load packages

Hide

```
library(MeanShift)
library(dplyr)
library(plotly)
library(MASS)
library(plot3D)
library(shiny)
library(TDA)
library(ggmap)
library(doParallel)
library(doSNOW)
library(foreach)
library(meanShiftR)
library(ks)
library(sparr)
library(LPCM)
library(pracma)
library(knitr)
library(dbscan)
```

## Loading data

The dataset *trackme.RData* contains info of about 60 running sessions. The dataset is composed by:

- Latitude
- Longitude
- Altitude
- ID
- Time

Hide

```
#load("/Users/andreamarcocchia/Desktop/Statistical learning/HW1/trackme.RData")
#load("C:/Users/user/Desktop/DATA SCIENCE/ANNO I SEMESTRE II 2017_2018/Statistical Learn
ing/Homework/HW_1/trackme.RData")
load("/Users/Dario/Desktop/Statistical Learning/HW1/trackme.RData")
```

Just take a look:

Hide

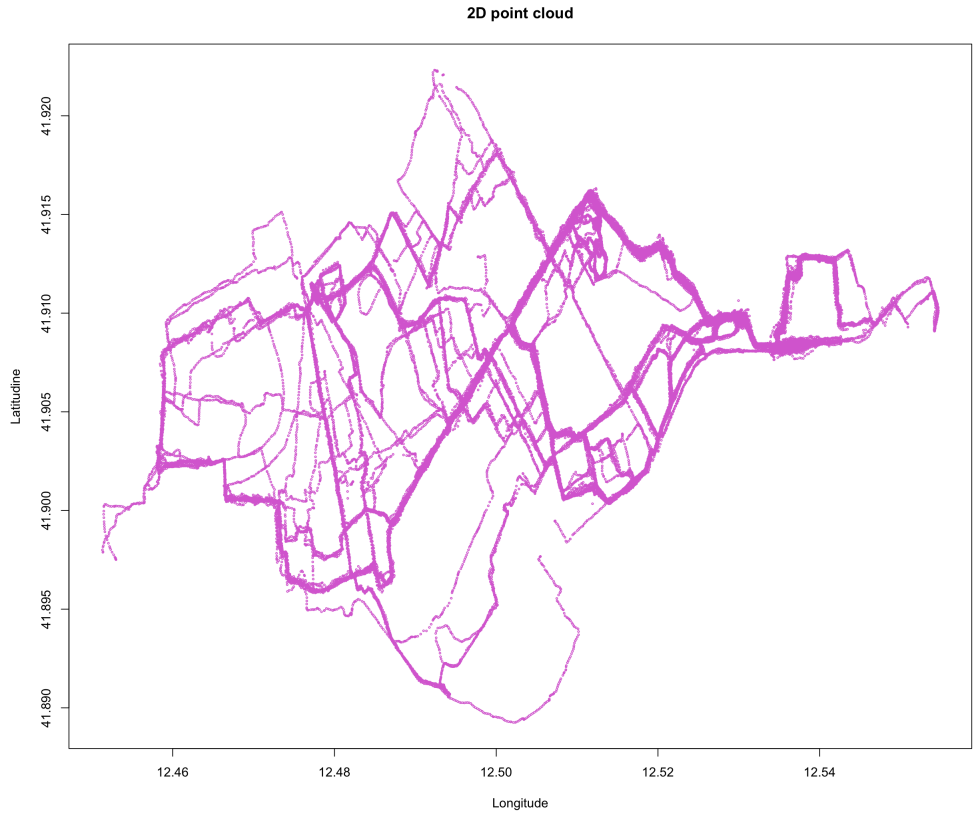
```
kable(head(runtrack))
```

| | lon | lat | ele | time | id |
|---|---|---|---|---|---|
| 64 | 12.53330 | 41.90820 | 43.3966 | 2014-01-12 06:35:08 | run_1 |
| 65 | 12.53326 | 41.90820 | 42.9160 | 2014-01-12 06:35:09 | run_1 |
| 66 | 12.53317 | 41.90822 | 41.4740 | 2014-01-12 06:35:12 | run_1 |
| 67 | 12.53315 | 41.90823 | 40.9934 | 2014-01-12 06:35:13 | run_1 |
| 68 | 12.53298 | 41.90825 | 39.0707 | 2014-01-12 06:35:18 | run_1 |
| 69 | 12.53293 | 41.90826 | 38.5901 | 2014-01-12 06:35:19 | run_1 |

For having an overview of what we have to handle, we're going to consider only the latitude and longitude variables, treating them like a **2D point cloud in the Euclidean space**.

Hide

```
plot(runtrack$lon,runtrack$lat,main='2D point cloud',col='orchid',cex=.3,xlab = "Longitude",ylab="Latitudine")
```




However, in this way we would lose information about lat-lon. Therefore, using a **3D plot** and an "heating-map" we'll be able to evaluate the frequency on the *z* axis. Before plotting we need to grab the variables of our interest.

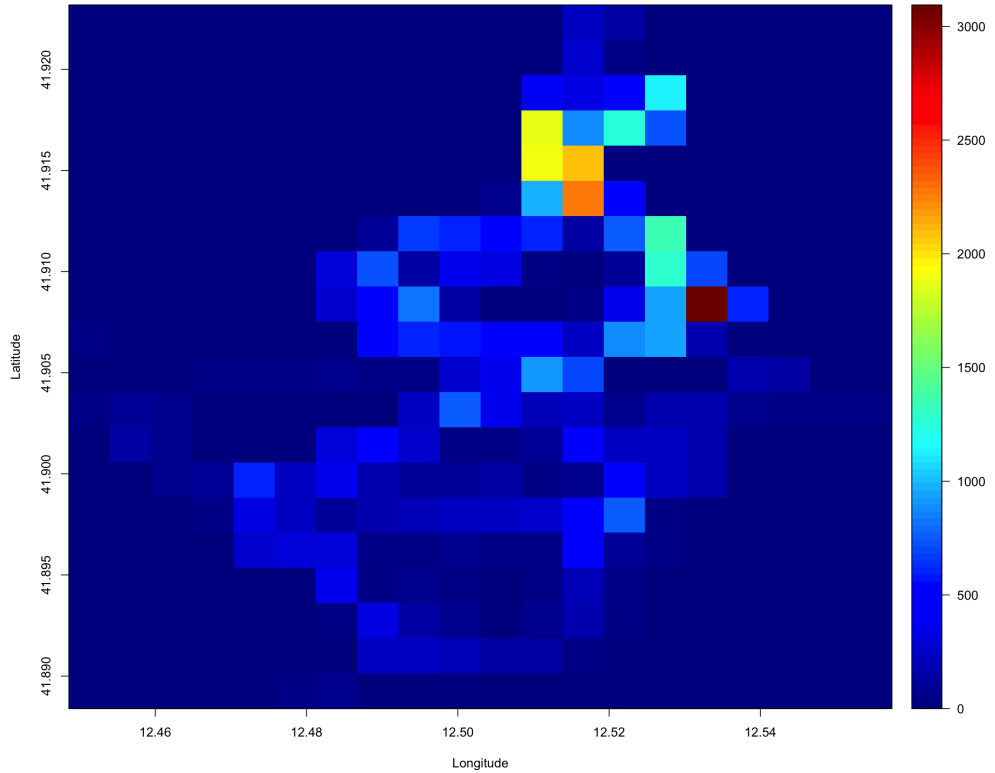
Hide

```
# Divide data (vector) according to the parameters breaks which indicates the number of
 chunks in which the vector is divided.

# The smaller the value of breaks the bigger the histogram bars. So if breaks parameter
 is small we obtain an oversmoothed representation.
# Undersmoothed in the opposite way.

lat_c <- cut(runtrack$lat,breaks = 20)
long_c <- cut(runtrack$lon,breaks = 20)
z <- table(lat_c,long_c)
```

Visualization time! The **heatmap** below shows us which are the areas of the tracks (defined by lat and lon) having the highest density.

Hide

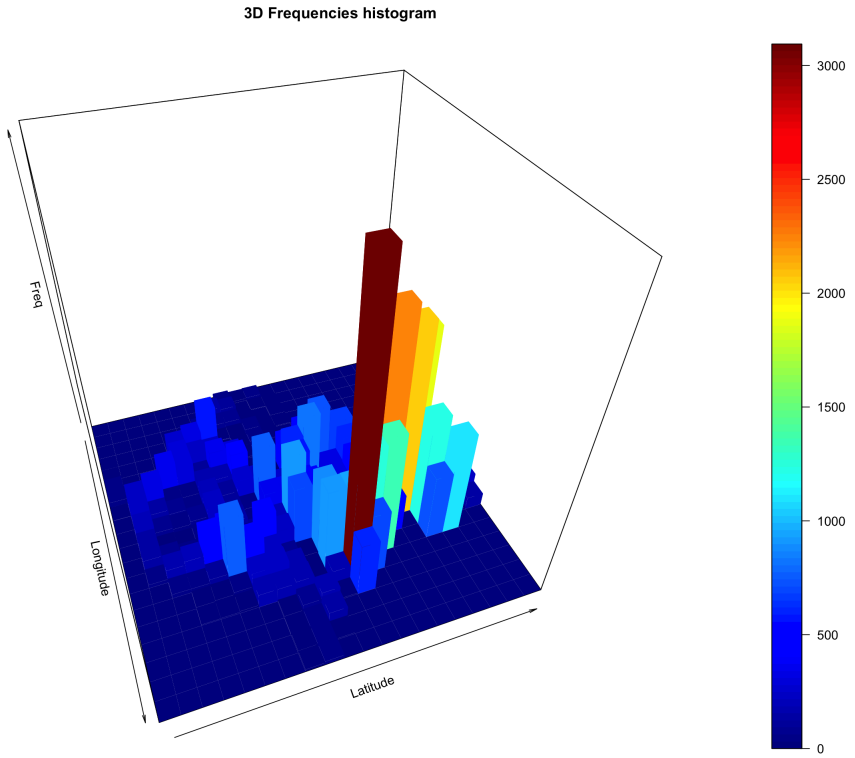
```
image2D(x = seq(min(runtrack$lon),max(runtrack$lon), length.out = nrow(z)),y = seq(min(runtrack$lat),max(runtrack$lat), length.out = ncol(z)),z = z,xlab='Longitude',ylab='Latitude')
```




Let's do a step forward. Using a **3D version** of our heating map, we're able to have a more detailed representation of our data.

Hide

```
# 3D histogram
hist3D(z = z, theta = 70,xlab='Longitude',ylab='Latitude',zlab='Freq',main = '3D Frequencies histogram')
```




At this point, to simplify our code we create a matrix with only the latitude-longitude values.

Hide

```
data_cluster <- data.matrix(runtrack[1:2])
```

# Density estimation

Now let's start our real work: **estimate the points density** through the **Kernel Density Estimator**.

## A bit of theory:

KDE is a **non-parametric** way to estimate the probability density function of a random variable and it is defined as follow:

$$\hat{p_h}(\underline{x}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{h^d}K\left(\frac{\|x - X_i\|}{h}\right)$$

The $K(\cdot)$ function is any function such that:

1. $\int K(x)dx = 1$

2. $\int xK(x)dx = 0$
3. $\sigma_k^2 = \int x^2 K(x)dx \in \left(0, +\infty\right)$

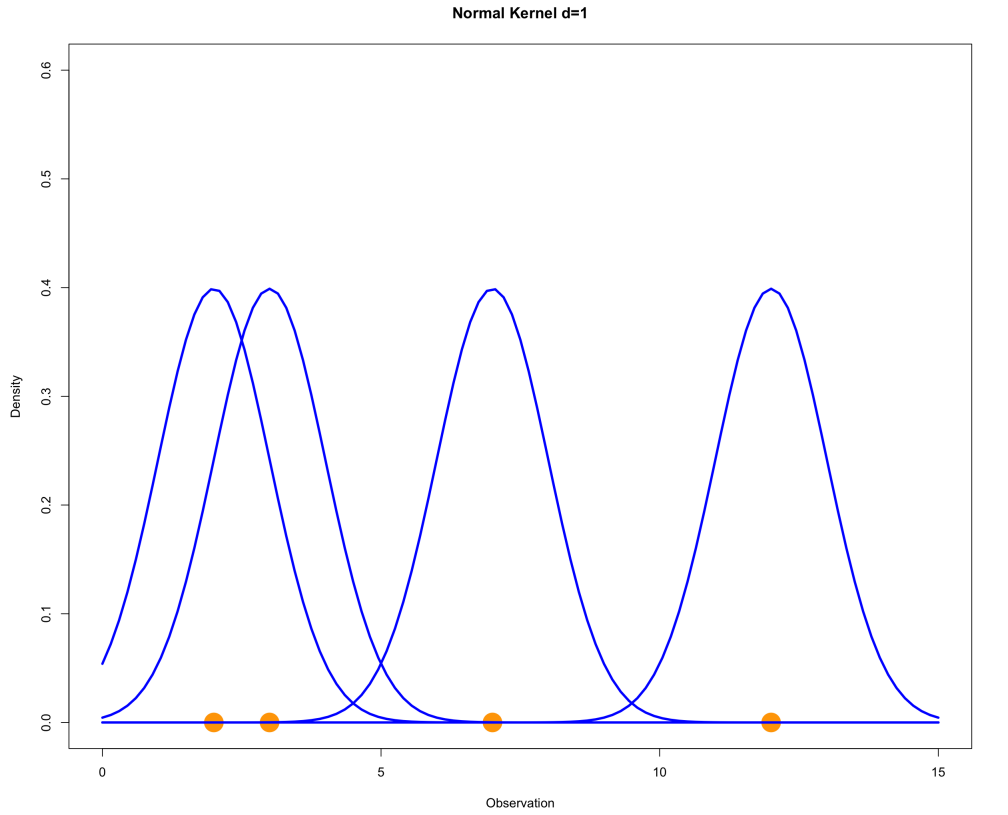
There are many different kernel estimators. In our work we decide to use the **Gaussian kernel**, defined as:

$$K(x) = \frac{1}{\sqrt{2\pi}} exp\left\{\frac{-x^2}{2}\right\}$$

The Normal Kernel density estimator evaluates a Normal density (with sd=1) on each observation. Below is showed how it works with $d = 1$:

Hide

```
punti <- c(2,3,7,12)
plot(x=punti,y=c(0,0,0,0),ylim=c(0,.6),col='orange',pch=16,xlim=c(0,15),cex=3.4,main='No
rmal Kernel d=1',xlab='Observation',ylab='Density')
curve(dnorm(x,2,1),add=T,col='blue',lwd=3)
curve(dnorm(x,3,1),add=T,col='blue',lwd=3)
curve(dnorm(x,7,1),add=T,col='blue',lwd=3)
curve(dnorm(x,12,1),add=T,col='blue',lwd=3)
```



However, this is the kernel definition for the **univariate** case, but our data are **bivariate**, so $\underline{x} \in \mathbb{R}^{\mathbb{d}=\mathbb{2}}$.

The multivariate kernel has different definitions:

1. $K(\underline{x}) = \prod_{j=1}^{d} K(x_j)$
2. $K(\underline{x}) = K(\|\underline{x}\|)$

Now come back to our $\hat{p}_h(\underline{x})$. The KDE strongly depends on the *h* parameter. If we call $p$ the true density, we can evaluate the density estimator performance using the **risk function**, defined as:

$$R(p, \hat{p}_h) = \mathbb{E}_{data}\left(L(p, \hat{p}_h)\right) = MSE(\hat{p}_h) = \mathbb{V}(\hat{p}_h) \; + \; bias^2(\hat{p}_h)$$

where $L(\cdot)$ is the **loss function** defined as:

$$L_2(\hat{p}_h, p) = \int_0^1 (\hat{p}_h(x) - p(x))^2 dx = \|\hat{p}_h - p\|^2 L_2$$

Even though an alternative **loss function** ($L_1$) it's better than $L_2$, we prefer to use this one because it is easy to deal with mathematically appending.

For minimizing $bias$ we want $h \to 0$ while for minimizing $\mathbb{V}$ we want $h \to +\infty$.

So we're facing on a **bias-variance tradeoff** problem.

Using the previous formula we find the **optimal bandwidth value** deriving the risk function:

$$h^* \asymp \frac{1}{n}\left(\frac{1}{d+2}\right)$$

We are almost there…but we have to keep in mind that we are speaking about the *order of*, and not the exact value. As it is possible to see in the last formula, if $n \to +\infty$ then $h \to 0$.

Hence, in according to what we said up to now, the KDE could approximate well the data or it could **undersmooth** or **oversmooth** them.

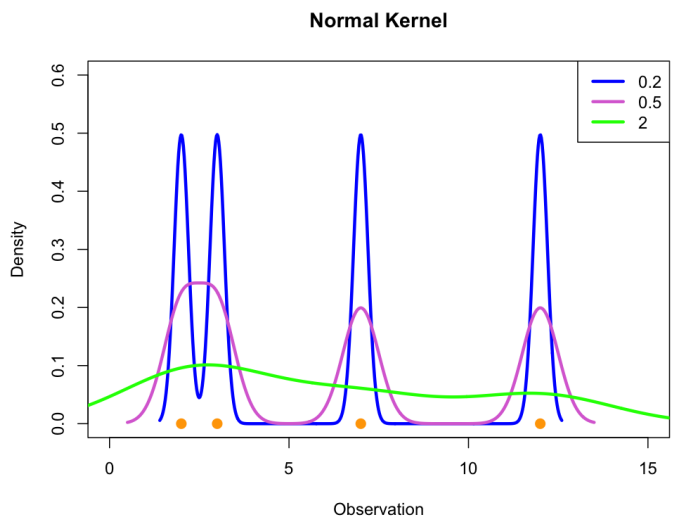
A long time ago a man said: "If I don't see the *h*, I do not believe it", so….in the next plot it's possible to have a visual representation of what happens using different values of *h* in a *1-d* example.

Hide

```
ui <- fluidPage(
  titlePanel("Interactive bandwidth plot"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("h", "h", 0.01, 2, 0.5,step=0.1)
    ),
    mainPanel(
      tabsetPanel(
        tabPanel("Plot",plotOutput("alpha"))
      )

    )
  )
)


server <- function(input, output) {
  output$alpha <- renderPlot({

    plot(x=c(2,3,7,12),y=c(0,0,0,0),ylim=c(0,.6),col='orange',pch=16,xlim=c(0,15),cex=1.
4,main='Normal Kernel',xlab='Observation',ylab='Density')

    lines(density(c(2,3,7,12),bw=.2),col='blue',lwd=3)
    lines(density(c(2,3,7,12),bw=input$h),col='orchid',lwd=3)
    lines(density(c(2,3,7,12),bw=2),col='green',lwd=3)
    legend(x='topright',legend=c("0.2",input$h,"2"),col=c("blue","orchid","green"),lwd=3
)

  })

}

# Set options for plot size
options=list(height=500,width=800)

# Create the shiny app
shinyApp(ui = ui, server = server,options = options)
```

# Interactive bandwidth plot


Plot




We've seen how to build a KDE. Now we have to choose **the right bandwidth value** using our data.

We wanna try to use different approaches:

## Method 1: Rule of thumb

The **Rule-of-thumb** method is usually used on **univariate data**, and in particular it has a good behaviour with **unimodal** data. The formula is the following:

$$h^* = \left( \frac{4\hat{\sigma}^5}{3n} \right)^{\frac{1}{5}}$$

We use *bandwidth.nrd()* from *MASS* package on lat and lon data separately.

Hide

```
# Use  bandwidth.nrd for both dimensions
h_est <- bandwidth.nrd(runtrack$lat)
h_est_2 <- bandwidth.nrd(runtrack$lon)

# Save results in a vector
h_ott <- c(h_est_2,h_est)
```

As we expected the *h* values are really small, due to our really big *n*:

Hide

```
h_ott
```

```
## [1] 0.010272817 0.002449489
```

### Method 2: Smoothed cross-validation

The main idea of cross-validation is to build the estimator and the risk function **using different sources of information**, that is to say different datasets.

We've tried to compute the LOO (Leave One Out) CV, but it turned out to be too slow in relation of our dataset size. Therefore we use a **smoothed version** which gives us a good result in a reasonable time.

To do this we use **Hscv** function from package *ks*. This function is a multivariate generalization of *hscv* function.

Hide

```
# The Hscv function doesn't return the bandwidth value in output.
# We need to use verbose = T, get inside the print of the iterations and copy the resul
t.
Hscv(x=data_cluster,verbose=T)
```

To keep the "beauty"" of our code we don't print the long output of the previous chunk, but we directly save the results in a vector:

Hide

```
# Save h value in a vector
h_cv <- c(0.0004048887,0.001224336)
```

## KDE estimation

Now we have two alternatives for bandwidth values, and we can evaluate the KDE using them. To build up our Normal Kernel Density Estimator, in according to the previous formulas, we use the *kde2d* function from *MASS* package.

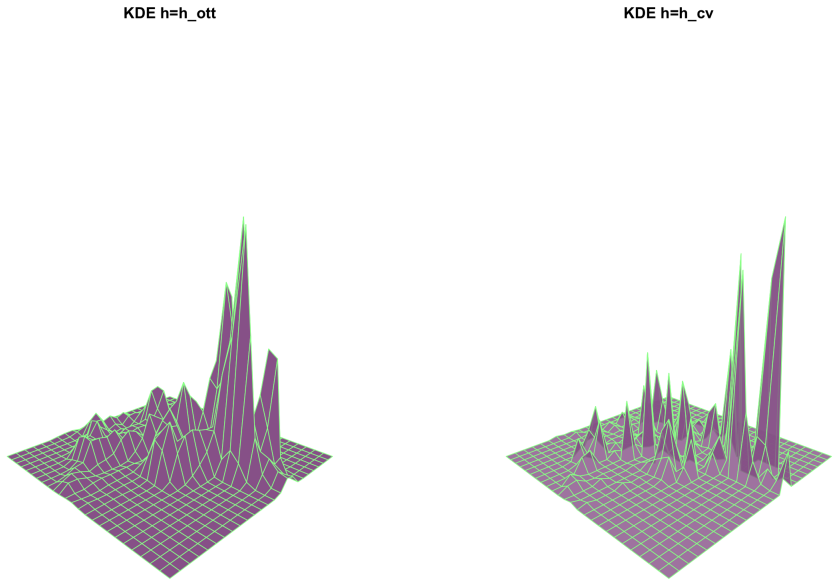
Hide

```
# Firstly we use h from rule of thumb
kd <- kde2d(runtrack$lon,runtrack$lat,h = h_ott)

# Then we use h from smoothed cross validation
kd_cv <- kde2d(runtrack$lon,runtrack$lat,h = h_cv)
```

Take a look at our densities, comparing them using *persp* function:

Hide

```
par(mfrow=c(1,2))

# Plot the KDE using h from rule of thumb
persp(kd$x,kd$y,kd$z,box=F,main='KDE h=h_ott',theta=45,col=rgb(.6,0.4,0.6,1),border='pal
egreen')

# Plot the KDE using h from smoothed CV
persp(kd_cv$x,kd_cv$y,kd_cv$z,box=F,main='KDE h=h_cv',theta=45,col=rgb(.6,0.4,0.6,0.8),b
order='palegreen')
```



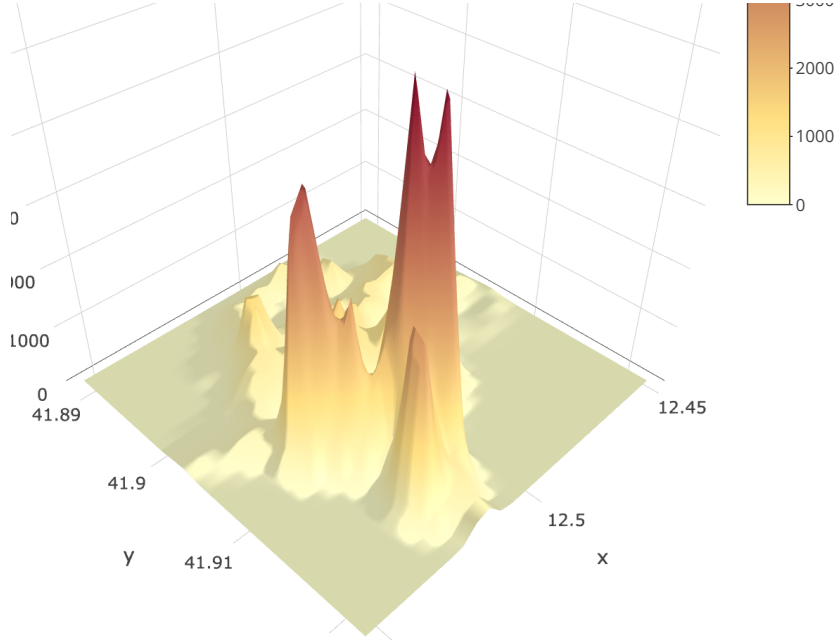

Hide

```
par(mfrow=c(1,1))
```

We decide to focus our attention on the first KDE because its behaviour seems to estimate better the density. Furthermore the *h_cv* bandwidth, having values smaller than the other one, give us an **undersmoothed KDE**. Thanks to the *plotly* function let's go to see our *rule-of-thumb* KDE.

Hide

```
p <- plot_ly(x = kd$x, y = kd$y, z = kd$z,colors="YlOrRd") %>% add_surface()
p
```



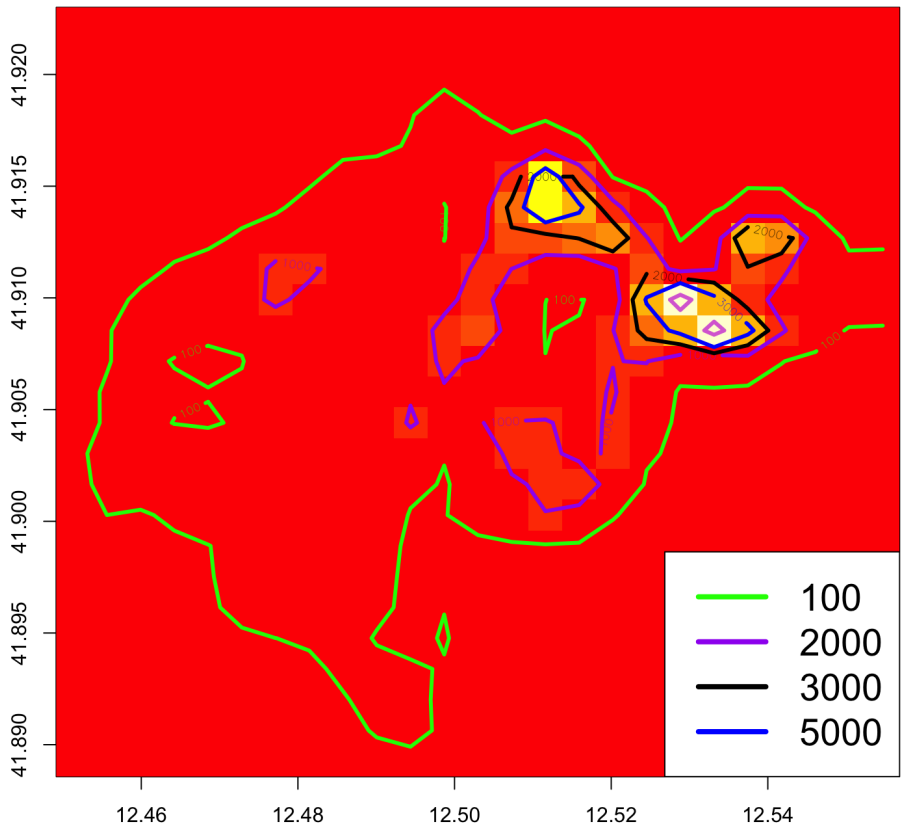

Let's continue to plot. As showed in the ***level-set*** plot below, we can see how the "zones" of the tracks which have the highest frequency (as we expected), are the ones which have the highest density. The *level-set* plot is explained by the following formula:

$$L_t = \{\underline{x} : \hat{p_h}(\underline{x}) > t\}$$

The *contour* function shows what we're saying, evaluating the KDE density at different levels.

Hide

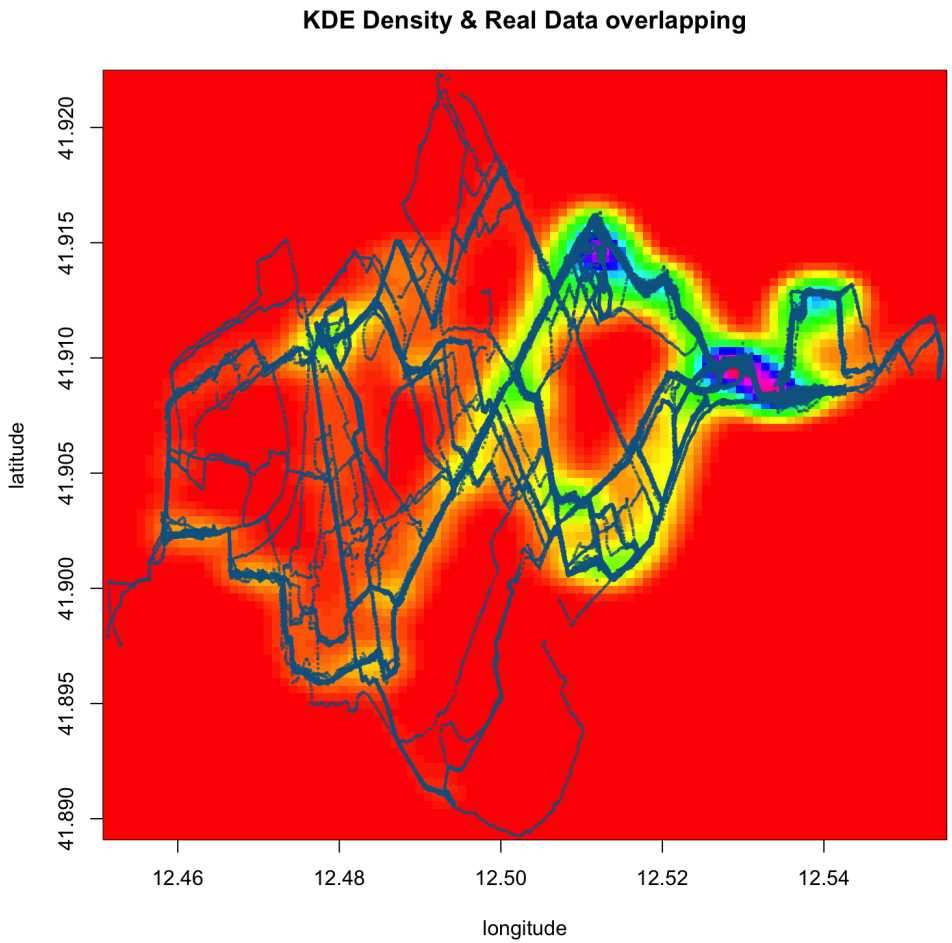
```
# 2D Estimate Density
image(kd)

contour(kd, levels = c(100,1000,2000,3000,5000),add=T,col=c("green","purple","black","bl
ue","orchid"),lwd=3)
legend(x='bottomright',legend=c(100,2000,3000,5000),col=c("green","purple","black","blu
e"),lwd=4,cex=1.8)
```




In according to the previous plot, we can see how a relationship persists between the KDE density and the observed data. In fact the “zones” with highest density are perfectly superimposed.

Hide

```
# 2D Estimate Density
image(kde2d(runtrack$lon,runtrack$lat, n = 100,h = h_ott),col = rainbow(100), xlab = "lo
ngitude", ylab = "latitude", main = "KDE Density & Real Data overlapping")

# Overlapping points
points(runtrack$lon,runtrack$lat,cex=.2,col=rgb(10, 100, 145, max = 255, alpha = 200, na
mes = "blue50"))
```




# Meanshift clustering algorithm

The Mean-Shift algorithm is a **non-parametric** technique for locating the maxima of a density function, a so-called **mode-seeking** algorithm.

Be patient now… we got a lot to explain!

At the beginning we have a bunch of points (our 54000 observations) and the estimated KDE $\hat{p}_h(\underline{x})$. The algorithm follows the following steps:

1. At the zero iteration we have a cluster for each point. So, for a generic point $m(x_i^{(0)}) \rightarrow x_i$.
2. In the next iteration we evaluate the $x_i^t$ equal to

$$m(x_j^{t+1}) = \frac{\sum_{i=1}^{n} K\left(\frac{1}{h}(x_j^t - x_i)\right) x_i}{\sum_{i=1}^{n} K\left(\frac{1}{h}(x_j^t - x_i)\right)}$$

The following quantity is only a weight: $\frac{K\left(\frac{1}{h}(x_j^t - x_i)\right)}{\sum_{i=1}^{n} K\left(\frac{1}{h}(x_j^t - x_i)\right)} = w_j \in [0, 1].$

Using this notation, we can rewrite:

$$m(x_j^{t+1}) = \sum_{i=1}^{n} x_i \, w_{i,t}$$

3. The algorithm repeats the estimation until $m(x_i^j)$ converges.
4. The output will be $\hat{M}_h = \left\{ m(x_1^\infty), m(x_2^\infty), \ldots, m(x_n^\infty) \right\}$. In this notation $\infty$ is an ipotetic number. In fact it represents the iteration at which the algorithm ends.

**The set of estimated modes, $\hat{M}_h$, is a function of the bandwidth *h*.**

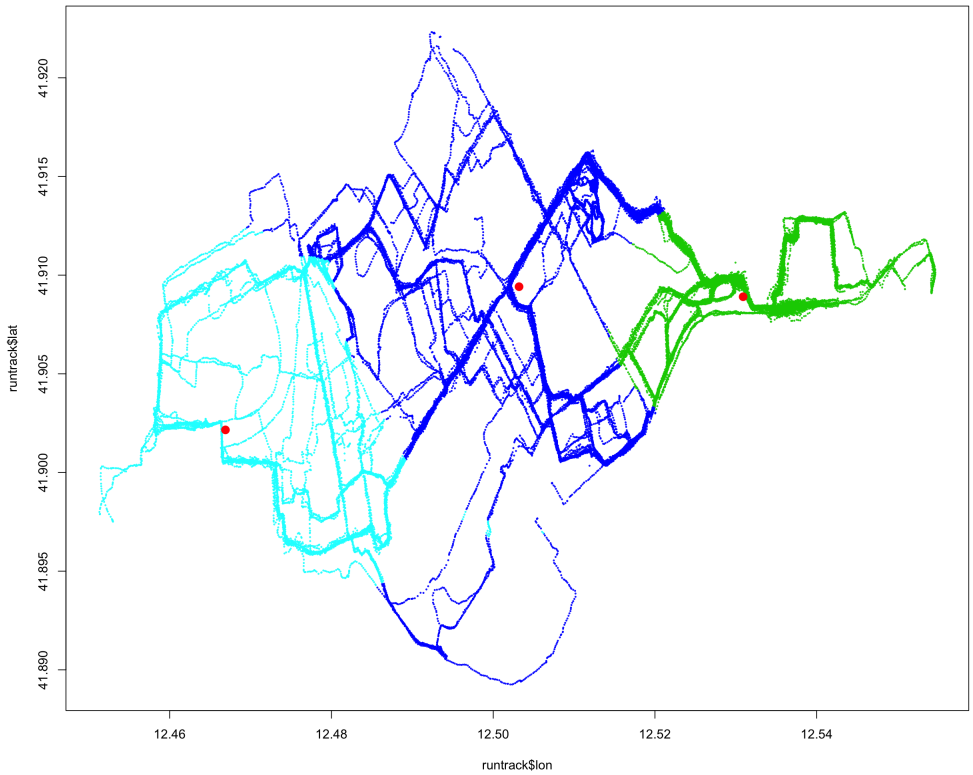
Now back to our mission!

We have to figure out the places with the highest density.

From the package *meanShiftR* we implemented the algorithm using the "optimal" bandwidth calculated previously with the rule of thumb.

Hide

```
# Parameters overview:
# nNeighbors is a scalar indicating the number neighbors to consider for the kernel dens
ity estimate. The default value is the full data set, but to save time we decided to red
uce the value of the parameter and we played around the value for having an accettable t
ime speed without losing too much information.

# The epsilonCluster parameter is a scalar used to determine the minimum distance betwee
n distinct clusters. We decided to choose that value because our points are close to eac
h other. However, with the default 0.000001 value we obtain an unique cluster.

meanshif_cl <- meanShift(queryData = as.matrix(data_cluster),bandwidth = h_ott,epsilonCl
uster = 0.0000004,nNeighbors = 10000)

# Plot the observed data point assigning different color to each cluster.
plot(runtrack$lon,runtrack$lat,col=meanshif_cl$assignment+2,cex=.2)

# These points represent the mode estimated.
points(unique(meanshif_cl$value),col='red',pch=16,cex=1.5)
```

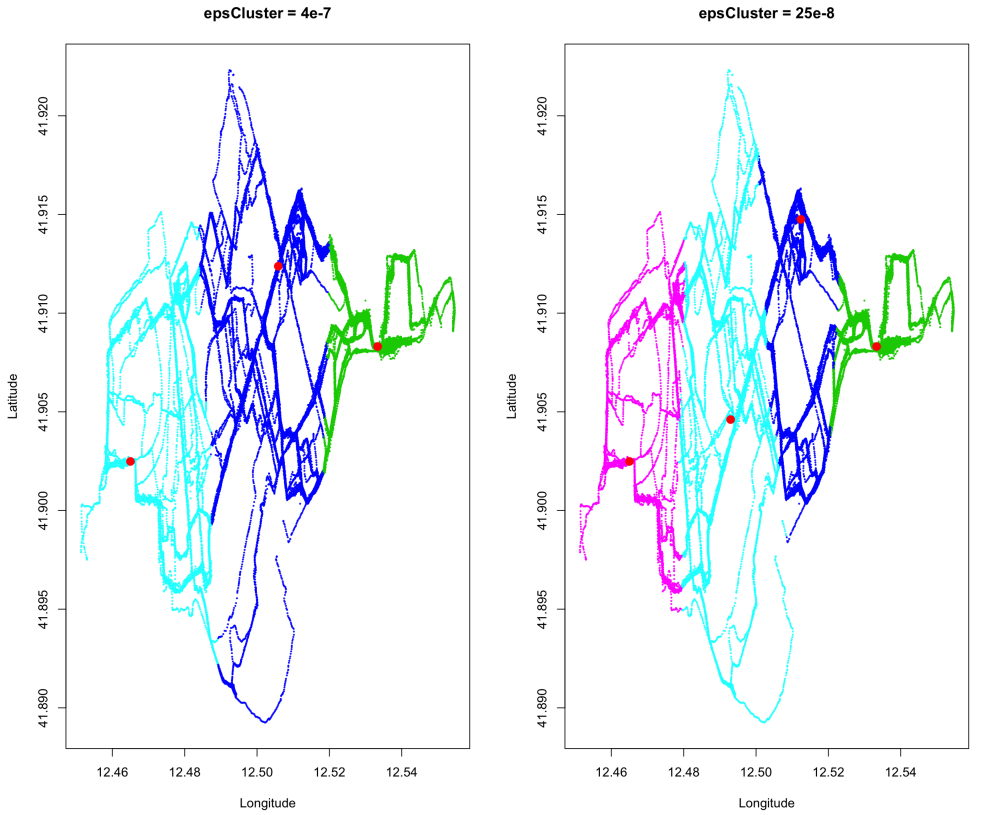



Playing around the epsilonCluster parameter we choose the value which allows us to visualize our clusters in a good way. The plot seems to show us something related to the analysis made before. We expected (more or less) 3 modes and as we can see, the clusters builded reflect our expectations.

In the following part we implement the same algorithm using a different “optimal” bandwidth extracted from the *smoothed cross-validation*. As before we played around the same parameters.

Hide

```
# Meanshift using smoothed cross validation bandiwidth and epsilonCluster=0.0000004
meanshif_cl_hcv_1 <- meanShift(queryData = as.matrix(data_cluster),bandwidth = h_cv,epsi
lonCluster = 0.0000004,nNeighbors = 10000)

# Meanshift using smoothed cross validation bandiwidth and epsilonCluster=0.00000025
meanshif_cl_hcv_2 <- meanShift(queryData = as.matrix(data_cluster),bandwidth = h_cv,epsi
lonCluster = 0.00000025,nNeighbors = 10000)

# Set plot parameters
par(mfrow=c(1,2))

# FIRST PLOT
# Plot the observed points, using different colours for each cluster
plot(runtrack$lon,runtrack$lat,col=meanshif_cl_hcv_1$assignment+2,cex=.2,xlab='Longitud
e',ylab='Latitude',main='epsCluster = 4e-7')
# Add the modes points
points(unique(meanshif_cl_hcv_1$value),col='red',pch=16,cex=1.5)

# Plot the observed points, using different colours for each cluster

# SECOND PLOT
plot(runtrack$lon,runtrack$lat,col=meanshif_cl_hcv_2$assignment+2,cex=.2,xlab='Longitud
e',ylab='Latitude',main='epsCluster = 25e-8')
# Add the modes points
points(unique(meanshif_cl_hcv_2$value),col='red',pch=16,cex=1.5)
```




Hide

```
par(mfrow=c(1,1))
```

We can see how changing the *epsCluster* parameter, and making it smaller, we obtain one more cluster, as we expect from the parameter description.

Now focusing on the left plot, we can check its similarity with the one made before (when *h* was chosen with the *rule-of-thumb*), in fact in both of them we obtain three clusters.

# Clustering everywhere... DBSCAN!

DBSCAN (Density-Based Spatial Clustering and Application with Noise), is another **density-based clustering** algorithm.

Let's go a bit deeper into DBSCAN world...

**Clusters are dense regions in the data space**, separated by regions of lower density of points. The key idea of DBSCAN is that for each point of a cluster, **the neighborhood of a given radius has to contain at least a minimum number of points**. To perform this algorithm we use the *dbscan* function from *dbscan* package. This function is a fast reimplementation of the DBSCAN clustering algorithm using a kd-tree that is possible to find in the *fpc* package.

Two important parameters are required for this function: $\epsilon$ ("*eps*") and minimum points ("*MinPts*").

The parameter *eps* defines the radius of neighborhood around a generic point *x*. It's called the $\epsilon$-neighborhood of *x*. The parameter *MinPts* is the minimum number of neighbors within "*eps*" radius.

Any point *x* in the data set, with a count of neighbor greater or equal to *MinPts*, is marked as a **core point**. We say that *x* is a **border point**, if the number of its neighbors is less than *MinPts*, but it belongs to the $\epsilon$-neighborhood of some core point. Finally, if a point is neither a core nor a border point, then it's called a **noise point** or outlier.

One of the most important advantages of *DBSCAN* cluster algorithm is that it doesn't require in input the number of clusters, but it automatically find it. Another important aspect is that **the cluster doesn't have to be circular**.

There are also some disadvantages using this algorithm. One of the most important is that it's sensitive to parameters. It means that **could be hard to determine the correct set of parameters**…. (as always!)

Now it's time to analyze in detail how we choose *minPts* and *eps* parameters:

- MinPts: The larger the data set, the larger the value of minPts should be chosen
- eps: If *eps* is too small, sparser clusters will be defined as noise. If *eps* is too large, denser clusters may be merged together. It's possible to find a value for *eps* using k-dist plot.

In our work we tried to use a k-dist plot (using *kNNdistplot* function, that is a fast calculation of the k-nearest neighbor distances in a matrix of points).

However, using the "eps" value from this function we obtain only one big cluster and many noise points. For this reason we decided to play a bit with parameters.

We also decided to work with **normalized data**, so that is possible to play easier with *eps* values.

Eventually, we reached our goal: for each point we defined the radius of neighborhood around a generic point as 0.241. By this way we find many clusters, but we consider only the one with more than 1700 points inside.

In this way we have **5 clusters** which are the zones with the highest density (remind: DBSCAN is a density-based clustering). As we expected the zones with the bigger clusters (the red and green ones) are the ones that we found in the previous analysis (using the KDE).

In the following chunk we normalize our dataset.

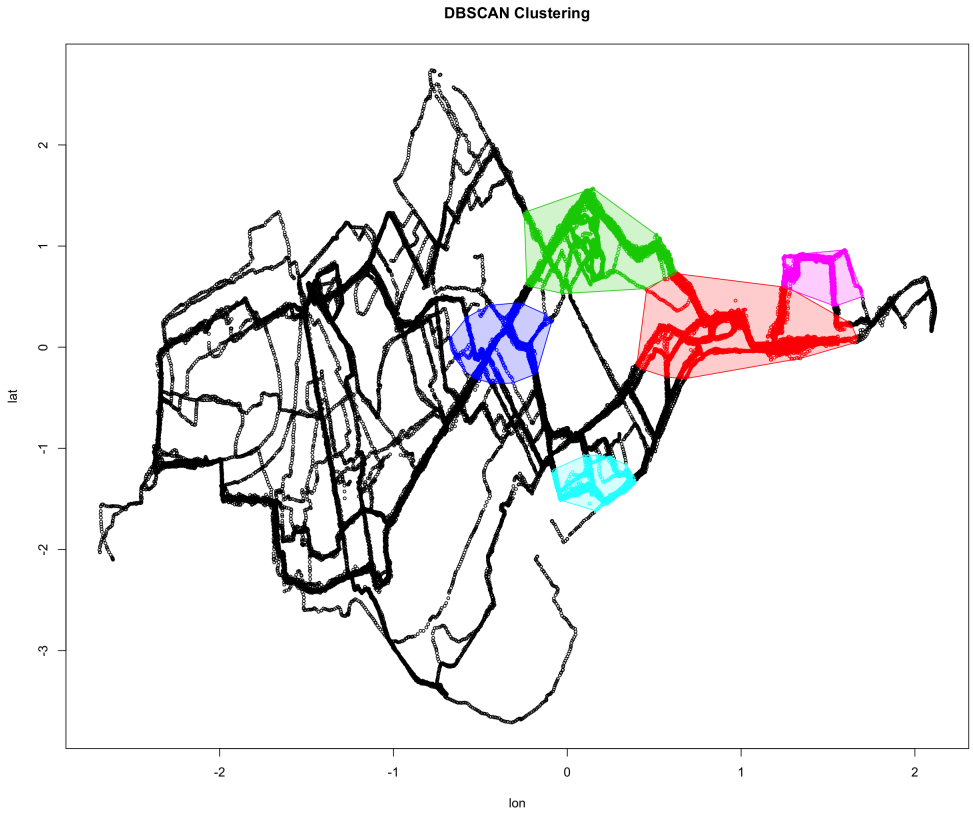
Hide

```
data_scaled <- scale(data_cluster)
```

Now we apply the DBSCAN algorithm, using the parameters grabbed before, and we plot the result using an *hullplot*:

Hide

```
dbscan_1 <- dbscan(data_scaled,eps=0.241,minPts = 1700)

# Plot of clusters
hullplot(data_scaled,cl=dbscan_1$cluster,main = "DBSCAN Clustering")
```




# Part 2

## Functional Data Analysis

Now let's start looking at the data from another point of view. Up to now we've considered our data as single points/observations (and that what they are), but they are more… they are part of a single track, **a curve in the plane**.

So far we have worked with a probability function: $\mathbf{X} : \Omega \to \mathbb{R}^2$ and we have estimated the density, based on the principle of local averaging, appling the **Lebesgue Density Theorem**:

$$p(x) = \lim_{\epsilon \to 0} \frac{Pr\{\|\mathbf{X} - x\|\} < \epsilon}{|B_\epsilon(x)|}$$

where $|B_\epsilon(x)|$ is the *Lebesgue's measurement* of the sphere of radius $\epsilon$ centered at $x$.

But now we are working on this :

$$\mathbf{X} : \Omega \to \mathbb{S}$$

and for $\mathbb{S}$ we mean the space of curves defined as $\mathbb{S} = \{\phi_i \mid \phi_i : [0, T_i] \to \mathbb{R}^2\}$ with $\phi_i \in C^0([0, T_i])$. Remember that $C^0$ is the class of continuos functions. Now the question is: are we still working on a vector space with the vector space operations defined as before? The following operations are always verified both in $R^2$ an in our new space $\mathbb{S}$. Let's show them:

1. If we took two distinct curves and added together we get a curve.
2. If we multiply a curve by scalar we always obtain a curve equal to the product between the previous curve and the scalar.
3. Finally, verifying the two previous operations, the Linear Combinations are valid.

So, taking two scalar $\alpha, \beta \in \mathbb{R}$ and two curves $\phi_1, \phi_2 \in \mathbb{S}$ we can see that $\alpha\,\phi_1 + \beta\,\phi_2$ returns a curve that belongs to $\mathbb{S}$.

$\mathbb{S}$ is also a metric space because in general we'll be able to use any **metric distance** (we will use the Hausdorff's distance ASAP) to evaluate the distance between two curves. Thanks to it we'll be able to obtain the topology of our space.

However, a problem arises.

- Can we apply the Lebesgue density theorem?
- Furthermore...what is a Lebesgue measure in $\mathbb{S}$?

Let's start with the last one question that is the core of our discussion:

Roughly speaking (many maaaaany mmmmaaaaaaaanyyyy roughly speaking) the Lebesgue's measurement ( $|E|$) is defined as the *n-dimensional* volume of *E*, where *E* is a subset of an Euclidean space $E \subseteq \mathbb{R}^n$.

Now that we know what a Lebesgue measure is, we can answer our first question:

The answer is no! Because in our case $n$ is $\infty$. So we can no longer calculate the previous limit because we don't know the nature of $|B_\epsilon(x)|$.

$$p(x) = \lim_{\epsilon \to 0} \frac{Pr\{dist(x, \mathbf{X}) < \epsilon\}}{???}$$

Then we need to work with a fixed $\epsilon > 0$ and for any planar curve $\phi$, we will evaluate an unnormalized kernel estimator.

In particular we can consider each curve/track as $\boldsymbol{X_i} : \ [0, T_i] \to \mathbb{R}^2$ where $\boldsymbol{X_i} = (X_{i1}, X_{i2})$ represents the (lat-lon) position at time $t$ and $T_i$ is the duration of the $i^{th}$ run.

So for any $i \in \{1, \ldots, 60\}$ we associate the set:

$$G_i \ = \{\boldsymbol{X}_i(t) = (X_{i1}, X_{i2}) \ s.t. \ \boldsymbol{X_i} : \ [0, T_i]\}$$

In the chunk below we "extracted" each curve from the dataset.

Hide

```
for (i in (unique(runtrack$id)))
{
  ll <- cbind(runtrack$lon[which(runtrack["id"] ==i)],runtrack$lat[which(runtrack["id"]
== i)])
  ll <- as.matrix(ll)
  names(ll) <- c("lon","lat")
  # Use function assign, that create a new variable (which name is i) with values of ll
  # At each iteration of the for loop the values of i and ll will be different
  assign(i, ll)
}
```

Once we have the set of curves, we can compute the distance using the **Hausdorff metric**. It's defined as:

$$\mathrm{d}_H(A,B) = \max\left\{ \sup_{a\in A}\inf_{b\in B}\mathrm{d}(a,b), \sup_{b\in B}\inf_{a\in A}\mathrm{d}(a,b) \right\}$$

What does it mean?

It measures how far two subsets of a metric space are from each other. In other words it's the greatest of all the distances from a point in one set (A) to the closest point in the other set (B).

Now the problem is: **how we evaluate the distance between two points in the Haussdorf metric?**

Let *a* and *b* be two points of sets *A* and *B* respectively, we define $d(a,b)$ any metric between these points. We are going to use the **Euclidian distance** for an easier manage.

We tried to implement the Hausdorff distance by ourself, and it's possibile to find the code below (note: the next chunk is not going to be run).

However… our job is good but not enough. In fact is too slow computationally speaking. So…a package is in the air!

Hide

```
h = 0
delta <- foreach (i = 1:length(run_1[,1])) %dopar%
{
  shortest = Inf
  for (j in 1:length(run_2[,1])){
    d_ij <- sqrt( (run_1[i,][1] - run_2[j,][1])^2 + (run_1[i,][2] - run_2[j,][2])^2 )
    if (d_ij < shortest){
      shortest = d_ij
    }

  }
  if (shortest > h){
    h = shortest
  }
  h
}
```

At this point our goal is to evaluate the Haussdorf distance between all the runs as faster as possible. Thanks to the parallelization we've been able to compute the symmetric matrix of Haussdorf distances in a reasonable way (one minute vs five… not so bad!).

For each pair of sets, we have $\frac{n(n-1)}{2} = 59 + 58 + 57 + \ldots + 1 = 1770$ total comparisons.

In the next chunk we set the parameters for the **parallel computing** loop that we'll use later.

Hide

```
cl <- makeCluster(3)
registerDoParallel(cl,cores=2)

#This command is used to "export on all the cores" the variables we wanna use.

clusterExport(cl, c("run_1", "run_2", "run_3", "run_4", "run_5", "run_6", "run_7", "run_
8", "run_9", "run_10", "run_11", "run_12", "run_13", "run_14", "run_15", "run_16", "run_
17", "run_18", "run_19", "run_20", "run_21", "run_22", "run_23", "run_24", "run_25", "ru
n_26", "run_27", "run_28", "run_29", "run_30", "run_31", "run_32", "run_33", "run_34",
"run_35", "run_36", "run_37", "run_38", "run_39", "run_40", "run_41", "run_42", "run_43"
, "run_44", "run_45", "run_46", "run_47", "run_48", "run_49", "run_50", "run_51", "run_5
2", "run_53", "run_54", "run_55", "run_56", "run_57", "run_58", "run_59", "run_60"),envi
r = environment())
```

Now we are ready to build our **distance matrix** using the *hausdorff_dist* function from the *pracma* package:

Hide

```
runn <- unique(runtrack$id)

# Parallel loop
ris_mmm <- foreach(i = 1:length(runn), .packages='pracma') %dopar%
{
  # Create an empy vector in which store information evaluated in the nested loop
  ss <- rep(99,length(runn))
  for (j in 1:length(runn))
  {
    kk=0
    # Beeing our output matrix a square matrix, we prefer to evaluate only the upper tri
angle
    if (i < j)
    {
      # Evaluate the Hausdorff distance between two sets
      # We have to use the get function to find the runn[i] variable, stored in RStudio
 Global Enviroment
      kk <- hausdorff_dist(get(toString(runn[i])),get(toString(runn[j])))
    }

    # Sotre the Haussdirf distance value (0 if i>j) in the j^th position of the ss vecto
r
    ss[j] = kk
  }

  # Store ss information in ris_mm vector
  # At the end of the loop ris_mm will be a list of list
  ss
}
```

OK, the most is done! **We evaluated the Haussdorf distance between each pairs of run**. Now we have just to transform our variable **ris_mm** (a list of list) in a matrix to obtain a square symmetric matrix from a triangular matrix. A breeze!

Hide

```
# transform a list of list in a matrix
ris <- do.call(rbind,ris_mmm)

# Obtain a square symmetric matrix
ris[lower.tri(ris)] = t(ris)[lower.tri(ris)]
```

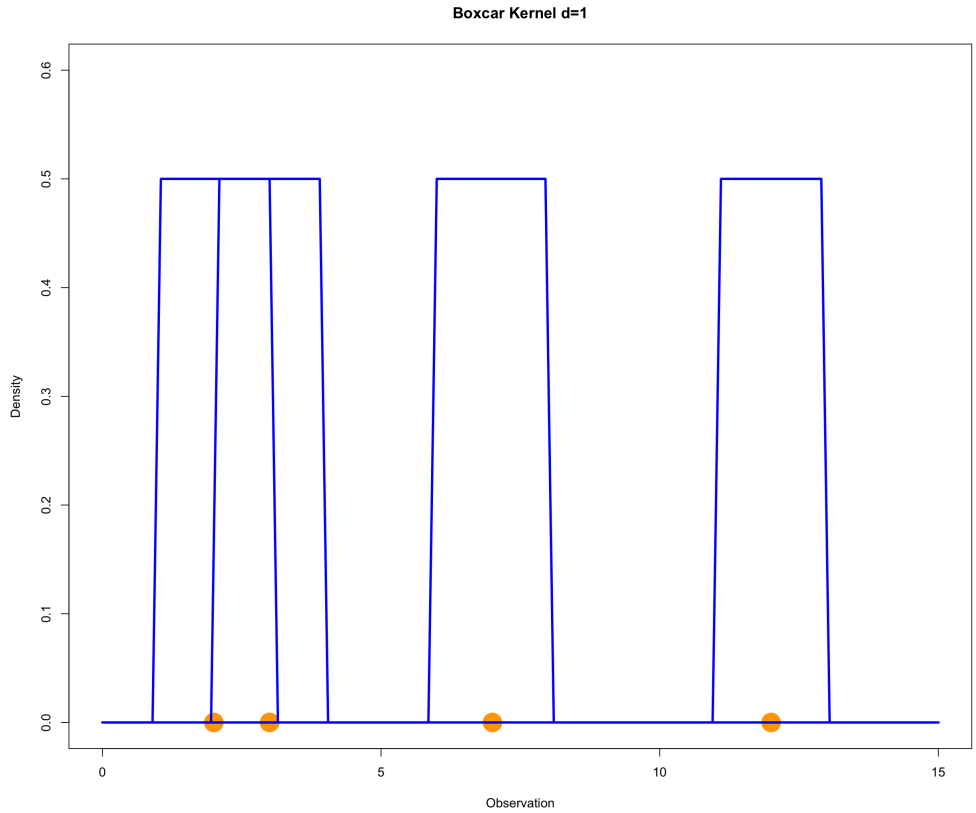
Now we have our matrix of distance. At this point we have to choose a value $\epsilon$ such that for any planar curve $\gamma$, we can evaluate our approximate unnormalized boxcar-kernel estimator Kernel defined as:

$$\hat{q_\epsilon}(\gamma) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(\text{dist}_H(\gamma, G_i) \ \leq \epsilon)$$

In detail a **boxcar Kernel** evaluate a Uniform density on each observation. Below there is an easy example with $d = 1$, as made before for the Normal Kernel:

Hide

```
punti <- c(2,3,7,12)
plot(x=punti,y=c(0,0,0,0),ylim=c(0,.6),col='orange',pch=16,xlim=c(0,15),cex=3.4,main='Bo
xcar Kernel d=1',xlab='Observation',ylab='Density')
curve(dunif(x,1,3),add=T,col='blue',lwd=3)
curve(dunif(x,2,4),add=T,col='blue',lwd=3)
curve(dunif(x,6,8),add=T,col='blue',lwd=3)
curve(dunif(x,11,13),add=T,col='blue',lwd=3)
```




However, we want to observe how our boxcar KDE changes in according to the $\epsilon$ change. For this reason we decide to set as values our first, second and third quantile from the distance distribution.

Hide

```
# We do not consider the values on the diagonal, that are always 0's
dist_vett <- ris[ris!=0]

# Vector of quantiles for epsilon
eps_value <- quantile(dist_vett,probs=c(0.25,0.5,0.75))
```

In the next chunk we create two functions:

1. The first one is *q_cap_fun* and will be used to evalute the estimate density $\hat{q}_\epsilon$, assigning $1$ each time the Haussdorf distance between a pairs of run is bigger than $\epsilon$, and $0$ otherwise. Then a sum will be applied and the result will be divided for *N=60*.
2. The second one, *fun_mat_10*, evaluates only the 0's and 1's matrix.

Hide

```
# FUNCTION 1

# The boxcar-KDE for any eps and matrix values
q_cap_fun <- function(eps,mat)
{
  # Create an empty matrxi
  eps_mat <- matrix(0,nrow=60,ncol=60)

  # Assign 1 in each position in which the matrix passed in input has values smaller or
 equal to eps
  eps_mat[mat<eps] = 1

  # Create an empty vector (with all values -1) having as length the number of column in
 the input matrix
  q_cap <- rep(-1,length(mat[1,]))

  for (i in 1:length(ris[1,]))
  {
    # Add in the i^th position of the q_cap vector the mean of the i^th row of eps_mat m
atrix
    q_cap[i]=round(sum(eps_mat[,i])/60,2)
  }

  # Return the q_cap vector
  return(q_cap)
}

##############################################################################################
##
# FUNCTION 2

# Matrix of 1's and 0's only about the Kernel
fun_mat_10 <- function(eps,mat)
{
  eps_mat <- matrix(0,nrow=60,ncol=60)

  # Assign 1 in each position in which the matrix passed in input has values smaller or
 equal to eps
  eps_mat[mat<eps] = 1
  return(eps_mat)
}
```

Here we calculate the $\hat{q_\epsilon}$ using the above function. We will have three different variables, one for each value of $\epsilon$ that we chose.

Hide

```
# Evaluate q_cap using the first quantile
q_cap_1 <- q_cap_fun(eps_value[1],ris)

# Evaluate q_cap using the second quantile (median)
q_cap_2 <- q_cap_fun(eps_value[2],ris)

# Evaluate q_cap using the third quantile
q_cap_3 <- q_cap_fun(eps_value[3],ris)
```

Ok, now we're ready to **find the top-5 paths with the highest local density**.

First of all we need to order the *q_cap* vector and mantain the information about the ID of each runtrack. To make this we create a new dataframe in which each value of $\hat{q_\epsilon}$ is associate to the relative ID. Then, without any loss of information, we can order the $\hat{q_\epsilon}$ values and easly find the *top 5* and *low 5*.

Hide

```
# Create a new data frame with q_cap value and run ID, using as q_cap the values obtaine
d using epsilon=1^th quantile
q_index_1 <- cbind(seq(1,60),q_cap_1)
q_index_1 <- as.data.frame(q_index_1)

# Make the same thing, using as q_cap the values obtained using epsilon=median
q_index_2 <- cbind(seq(1,60),q_cap_2)
q_index_2 <- as.data.frame(q_index_2)

# Make the same thing, using as q_cap the values obtained using epsilon=3^rd quantile
q_index_3 <- cbind(seq(1,60),q_cap_3)
q_index_3 <- as.data.frame(q_index_3)

# Now we have to find the top-5
q_index_1[order(q_index_1$q_cap_1,decreasing = T),]$V1[1:5]
```

```
## [1] 11 18 30 31 32
```

Hide

```
q_index_2[order(q_index_2$q_cap_2,decreasing = T),]$V1[1:5]
```

```
## [1]  6 10  1 51 40
```

Hide

```
q_index_3[order(q_index_3$q_cap_3,decreasing = T),]$V1[1:5]
```

```
## [1]  6 10  1 21 25
```

Hide

```
# Find the low-5
q_index_1[order(q_index_1$q_cap_1,decreasing = T),]$V1[55:60]
```

```
## [1] 55  4  3 36 53  2
```

Hide

```
q_index_2[order(q_index_2$q_cap_2,decreasing = T),]$V1[55:60]
```

```
## [1] 52 55 21 36  4  3
```

Hide

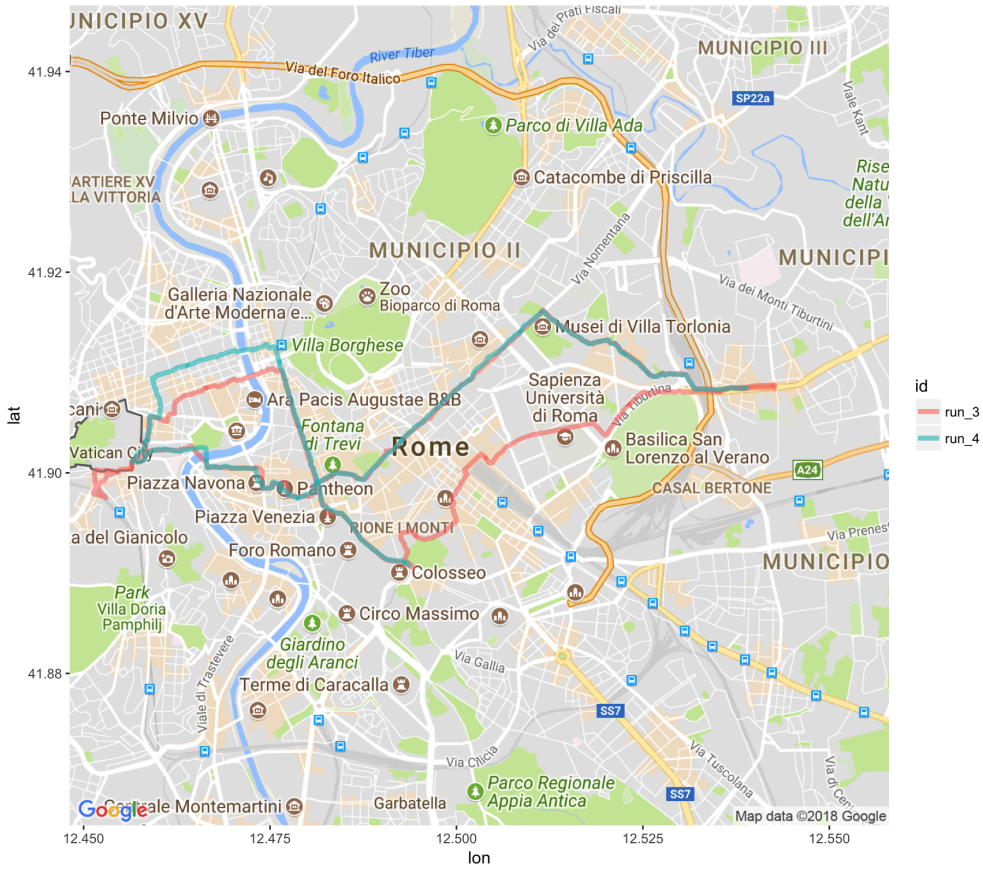
```
q_index_3[order(q_index_3$q_cap_3,decreasing = T),]$V1[55:60]
```

```
## [1] 46 48 52 55  4  3
```

As we expected the results are different. Howewer there are some persistent results. In particular in the *low-5* paths there are always $run_4$ and $run_3$. We can have an idea of this two paths in the next *Google Maps* plot.

Hide

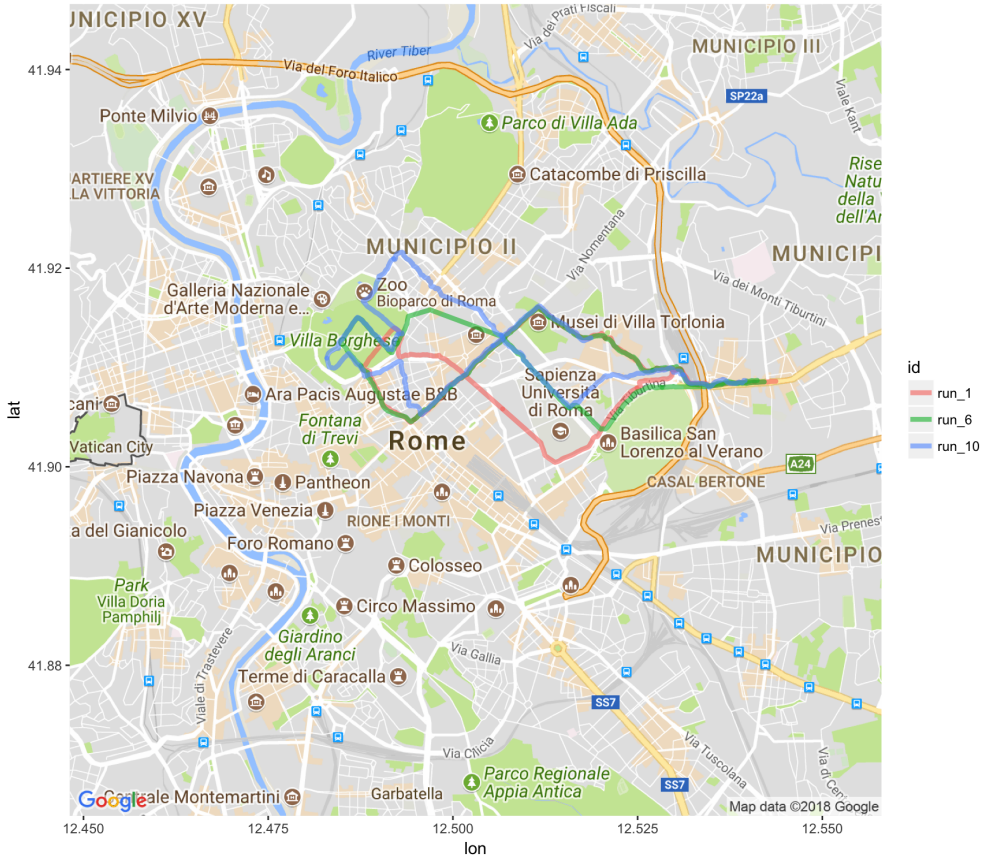
```
myLocation <- c(min(runtrack$lon, na.rm = T),
min(runtrack$lat, na.rm = T),
max(runtrack$lon, na.rm = T),
max(runtrack$lat, na.rm = T))
# Get the map from Google (default) and plot
myMapInD <- get_map(location = myLocation, maptype = "roadmap", zoom = 13)

runsmall <- subset(runtrack, id %in% c("run_4","run_3"))
gp2 <- ggmap(myMapInD) + geom_path(data = runsmall,
aes(x = lon, y = lat, col = id),
size = 1.5, lineend = "round",
alpha = .6)
# Take a look
print(gp2)
```




Regarding the *top-5* the results are a bit different using different values of $\epsilon$. However in the *q_cap_2* and *q_cap_3* variables there are $3$ runs that are repeated in both $(6, 10, 1)$. We can check their similarity as before:

Hide

```
myLocation <- c(min(runtrack$lon, na.rm = T),
min(runtrack$lat, na.rm = T),
max(runtrack$lon, na.rm = T),
max(runtrack$lat, na.rm = T))
# Get the map from Google (default) and plot
myMapInD <- get_map(location = myLocation, maptype = "roadmap", zoom = 13)

runsmall <- subset(runtrack, id %in% c("run_6", "run_10","run_1"))
gp2 <- ggmap(myMapInD) + geom_path(data = runsmall,
aes(x = lon, y = lat, col = id),
size = 1.5, lineend = "round",
alpha = .6)
# Take a look
print(gp2)
```



As we can see the tracks with the highest local density are those with the shortest path among the 60 analyzed. On the other hand those with lower density have longer tracks.

However, our analysis gives us some important results: in the run 1,6 and 10 (high density) the starting point and the end point are always the same. On the other hand in the $3^{rd}$ and $4^{th}$ tracks the starting or ending points are different.

# Cluster the tracks!

Now we want to cluster the tracks. The **mean shift algorithm** will be our friend again.

We've already computed an estimate of the density, using an approximation of the **boxcar Kernel**. Howewer, we'll proceed using another Kernel: the Normal one.

This choice has been forced because unfortunately there is not the boxcar Kernel into the package **meanShiftR**. Note that, generally, the Kernel choice, is not crucial in the meanshift or KDE result. As said before, it's really more crucial the choice of the bandwidth *h*.

So, let's start to estimate *h* for this new density!

Remember that we're dealing with a **distances matrix**, so our dataset has 60 observations, and each observation has 60 dimensions. In according to the theory mentioned before, we have to estimate a bandwidth value for each dimension. We'll use the *bandwidth.nrd* function again, that provides us an estimate through the already mentioned *rule of thumb*.

Let's start:

Hide

```
ris_h <- as.data.frame(ris)

# Create an empty vector
vett_bandwidth = rep(999,nrow(ris_h))

for ( i in 1:nrow(ris_h)){
  # Estimate the bandowth for the i^th row (or column, it's the same) of the ris_h data
 frame
  vett_bandwidth[i] <- bandwidth.nrd(ris[,i])

}
```

Now apply the meanshift algorithm. Being our dataset composed by $60x60$, we don't need to set the extra parameters *epsilonCluster* and *nNeighbors*.

Howewer, we have to play a bit with the bandwidth: using the values stored in the *vett_bandwidth* vector we obtain $5$ clusters. Since two of them have only $1$ observation, we decide to increment a bit (10%) our bandiwidth size for each dimension. By this way we obtain 4 clusters. The two "1-element" clusters are now merged.

Hide

```
meanshif_cl_fda <- meanShift(queryData = ris,bandwidth = vett_bandwidth*1.1)
```

We can analyze in detail how the cluster are composed in the next chunk:

Hide

```
table(meanshif_cl_fda$assignment)
```

```
##
##  1  2  3  4
## 19 11  2 28
```

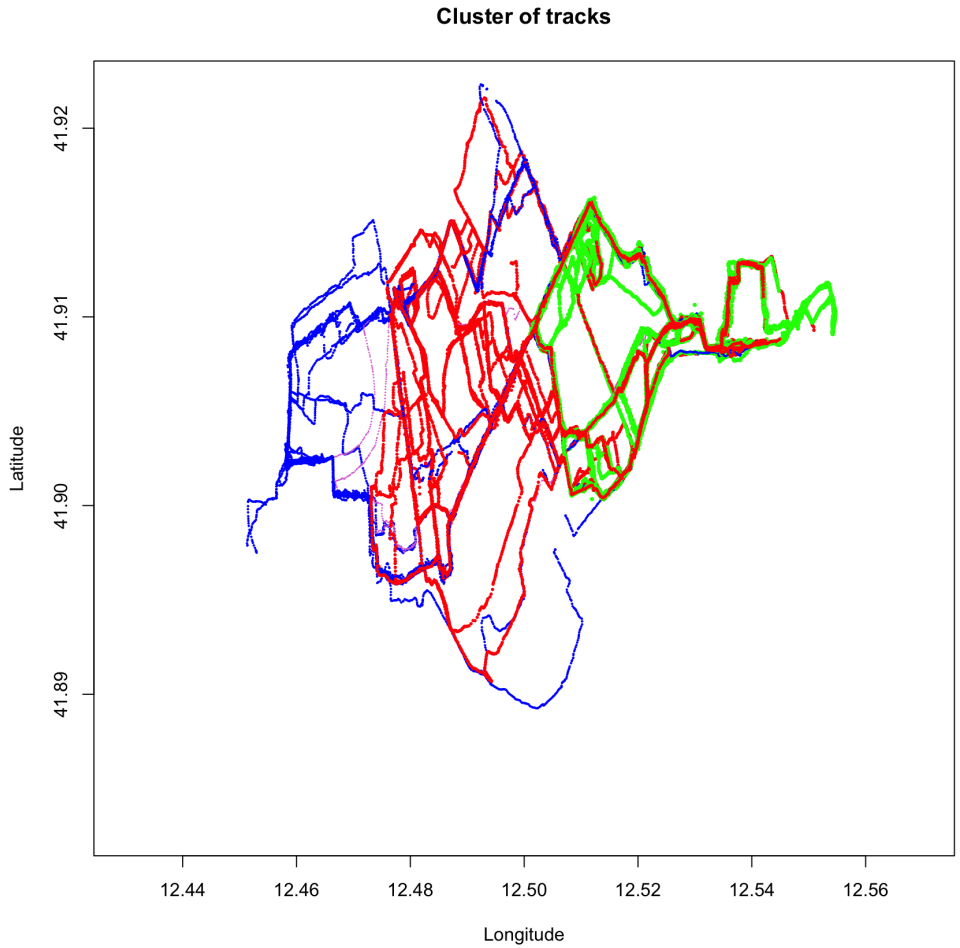
How we can see there is a cluster (the third one) with only two elements, so probably two **outliers**.

To have **an idea of the goodness of our clustering**, we want to plot each cluster and check if the tracks in the same cluster are really close to each other.

Let's start with a simple plot in a 2D space:

Hide

```
# Set an index equal to 1
indice = 1
# Make an empty plot, and use the right scale on X and Y axes
plot(0,0,xlim=c(12.43,12.57),ylim=c(41.883,41.922),xlab="Longitude",ylab="Latitude",main
="Cluster of tracks")

# Set graphical parameters that will be used later
point_dim=c(0.4,0.3,0.2,0.5)
colori <- c("red","blue","orchid","green")

# Start a loop that iterates on each runs
for (idx in meanshif_cl_fda$assignment)
{
  auxx <- paste("run_",indice,sep="")
  # Plot that runs using the right colour (one colour for each cluster)
  points(runtrack$lon[runtrack$id==(toString(auxx))],runtrack$lat[runtrack$id==(toString
(auxx))],col=colori[meanshif_cl_fda$assignment[indice]],cex=point_dim[meanshif_cl_fda$as
signment[indice]],pch=16)

  # Increment the index
  indice = indice + 1
}
```




Looking at the chart, it seems that **the clusters are well defined**, but we're not completely satisfied yet. We want to observe them above the **map** to get a much more realistic idea.

First of all we create a vector, one for each cluster, with inside the ID of the run that belongs to that cluster.

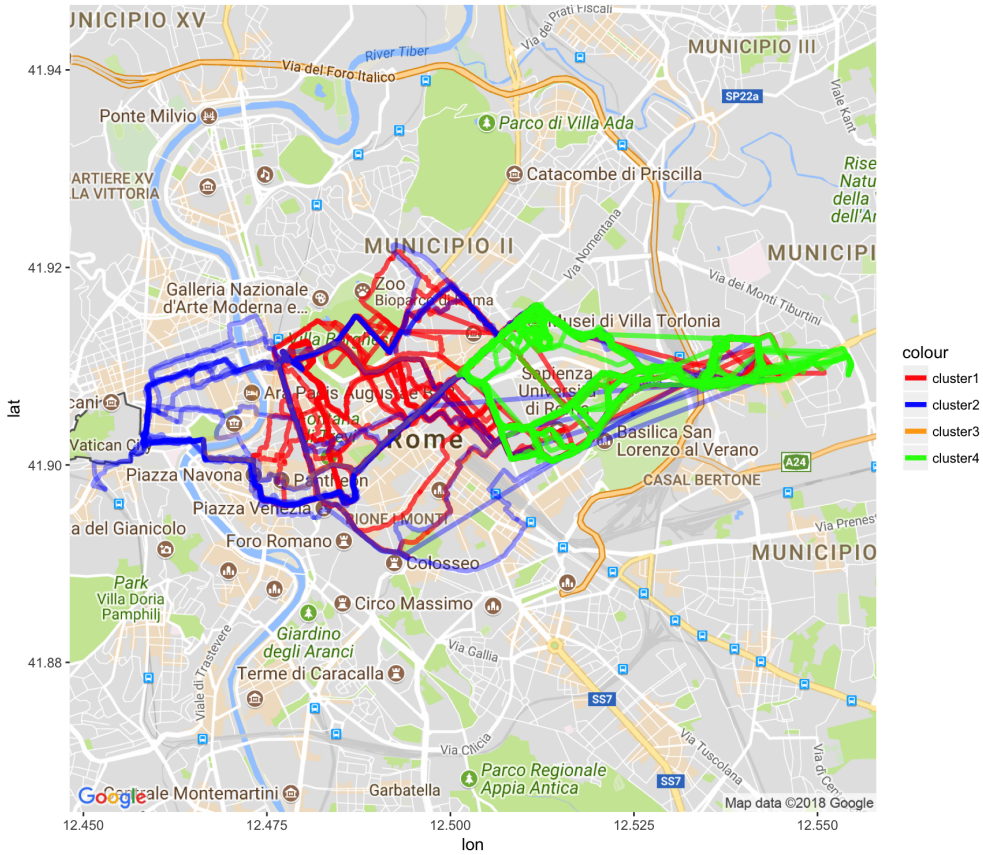
Hide

```
new_mean_s <- as.data.frame(cbind(seq(1,60),meanshif_cl_fda$assignment))
names(new_mean_s)=c("id","cluster")

# Inizialize empty vector
cluster1 <- NULL
cluster2 <- NULL
cluster3 <- NULL
cluster4 <- NULL

# Inizialize four different indexes (will be the vector position)
idx1=1
idx2=1
idx3=1
idx4=1

# Start a loop that iterates on the run_id
for (i in new_mean_s$id)
{
  # Check if the assigned cluster for the i^th run is the first one
  if (new_mean_s$cluster[i]==1)
  {
    # Add to the first cluster vector the string with the id of the tracks
    cluster1[idx1]=paste("run_",i,sep="")
    idx1 = idx1 +1
  }

  # Check if the assigned cluster for the i^th run is the second one
  else if (new_mean_s$cluster[i]==2)
  {
    cluster2[idx2]=paste("run_",i,sep="")
    idx2 = idx2 +1
  }

  # Check if the assigned cluster for the i^th run is the third one
  else if (new_mean_s$cluster[i]==3)
  {
    cluster3[idx3]=paste("run_",i,sep="")
    idx3 = idx3 +1
  }

  # Check if the assigned cluster for the i^th run is the fourth one
  else if (new_mean_s$cluster[i]==4)
  {
    cluster4[idx4]=paste("run_",i,sep="")
    idx4 = idx4 +1
  }
}
```

OK, now we're ready to plot all the runs, using different colours for different clusters, on a Google Map:

Hide

```
myLocation <- c(min(runtrack$lon, na.rm = T),
min(runtrack$lat, na.rm = T),
max(runtrack$lon, na.rm = T),
max(runtrack$lat, na.rm = T))
# Get the map from Google (default) and plot
myMapInD <- get_map(location = myLocation, maptype = "roadmap", zoom = 13)

cluster11 <- subset(runtrack, id %in% cluster1)
cluster22 <- subset(runtrack, id %in% cluster2)
cluster33 <- subset(runtrack, id %in% cluster3)
cluster44 <- subset(runtrack, id %in% cluster4)
gp2 <- ggmap(myMapInD) + geom_path(data = cluster11,aes(x = lon, y = lat, col = 'cluster
1'),size = 1.5, lineend = "round",alpha = .7) + geom_path(data = cluster22,aes(x = lon,
y = lat, col = 'cluster2'),size = 1.5, lineend = "round",
alpha = .4)+ geom_path(data = cluster33,aes(x = lon, y = lat, col = 'cluster3'),size =
1.5, lineend = "round",
alpha = .2)+ geom_path(data = cluster44,aes(x = lon, y = lat, col = 'cluster4'),size =
1.5,alpha=0.7)+
  scale_colour_manual(values=c("red","blue","orange","green"))
# Take a look
print(gp2)
```




Now we are able to observe how the denser cluster concentrates on shorter distances while the less dense cluster, not considering the two outlier traces, is the one that covers the greatest displacements.

# STATS time!

It's not enough!! We want to provide some **statistics for the tracks in each cluster**. In detail we can provide some statistics as the average time of physical activity for each run in a specific cluster (assuming that in the most of cases the runner will not do more than one times the same path in a single runtrack, but goes back home (https://www.google.it/search?q=FORREST+GUMP+BACK+HOME&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiKw6j3wLraAhVI6qQKHeZACPIQ_AUICigB&biw=1318&bih=7

Hide

```
# inizialize an empty vector
time1 <- NULL
time2 <- NULL
time3 <- NULL
time4 <- NULL

# Inizialize 4 indexes
j1 = 1
j2 = 1
j3 = 1
j = 1

# Start a loop in first cluster
for (i in cluster1){
  # For each track in this cluster, evaluate the difference (expressed in minutes) betwe
en the first observation and the last one
  time1[j1]<-(as.numeric(as.POSIXct(runtrack[which(runtrack$id==i),4][length(which(runtr
ack$id==i))])) - as.numeric(as.POSIXct(runtrack[which(runtrack$id==i),4][1])))/60
  j1 = j1+1
}

# Start a loop in the second cluster
for (i in cluster2){
  time2[j2]<-(as.numeric(as.POSIXct(runtrack[which(runtrack$id==i),4][length(which(runtr
ack$id==i))])) - as.numeric(as.POSIXct(runtrack[which(runtrack$id==i),4][1])))/60
  j2 = j2+1
}


# Start a loop in the third cluster
for (i in cluster3){
  time3[j3]<-(as.numeric(as.POSIXct(runtrack[which(runtrack$id==i),4][length(which(runtr
ack$id==i))])) - as.numeric(as.POSIXct(runtrack[which(runtrack$id==i),4][1])))/60
  j3 = j3+1
}


# Start a loop in the fourth cluster
for (i in cluster4){
  time4[j]<-(as.numeric(as.POSIXct(runtrack[which(runtrack$id==i),4][length(which(runtra
ck$id==i))])) - as.numeric(as.POSIXct(runtrack[which(runtrack$id==i),4][1])))/60
  j = j+1
}

# Evaluate the mean in each vector
medie1 = mean(time1)
medie2 = mean(time2)
medie3 = mean(time3)
medie4 = mean(time4)
```

Let's see the results:

Hide

```
c(medie1,medie2,medie3,medie4)
```

```
## [1] 77.51930 84.80758 86.24167 50.94881
```

Taking a look at the results, we have a confirm about the clusters obtained before.

In fact, as we expected, the $4^{th}$ cluster, which contains the most runtracks, has the lowest average running time.

*Are you getting lazy (or older)? :)*

The $2^{nd}$ cluster has the highest average of running time. For this reason it's got the smallest number of runtracks. We've escluded the $3^{rd}$ cluster because it contains only 2 tracks, so it would not been significative to analyze.