

---

# Identificación de patrones y algoritmos de consolidación en bases de datos de posicionamiento

---

TRABAJO FIN DE MÁSTER  
MÁSTER EN MODELIZACIÓN E INVESTIGACIÓN MATEMÁTICA,  
ESTADÍSTICA Y COMPUTACIÓN

PILAR BARBERO IRIARTE

DIRIGIDO POR

TOMÁS ALCALÁ  
*Universidad de Zaragoza*



**Universidad**  
Zaragoza



# Bibliografía

<b>1</b>	<b>Introducción</b>	<b>5</b>
<b>2</b>	<b>Datos y estructura de los datos suministrados</b>	<b>7</b>
2.1	Análisis de los datos . . . . .	8
2.2	Espacio en disco . . . . .	9
2.3	Implementación de los datos en clases de Python . . . . .	10
<b>3</b>	<b>Nociones de vecindario</b>	<b>12</b>
3.1	Vecindario simple . . . . .	12
3.2	Vecindario involucrando el módulo de la velocidad . . . . .	12
3.3	Vecindad $t_0$ -alcanzable . . . . .	13
3.4	Vecindario involucrando el tiempo . . . . .	13
<b>4</b>	<b>Preprocesado de datos</b>	<b>14</b>
4.1	Canopy . . . . .	14
<b>5</b>	<b>Algoritmos de consolidación simples</b>	<b>16</b>
5.1	Consolidación por distancia . . . . .	17
5.2	Consolidación por adelgazamiento . . . . .	18
5.3	Consolidación por tiempo . . . . .	19
<b>6</b>	<b>Algoritmos de consolidación asociados a métodos de clustering</b>	<b>21</b>
6.1	K-means . . . . .	21
6.2	DBSCAN . . . . .	22
6.2.1	Implementación en Python . . . . .	26
6.3	DJ-Cluster . . . . .	27
<b>7</b>	<b>Comparativa de resultados</b>	<b>30</b>
7.1	Resultados con K-means . . . . .	32
7.1.1	Sujeto 1 . . . . .	32
7.1.2	Sujeto 2 . . . . .	34
7.2	Resultados con DBSCAN . . . . .	36
7.2.1	Sujeto 1 . . . . .	36
7.2.2	Sujeto 2 . . . . .	37
7.3	Resultados con DJ-Cluster . . . . .	38
7.3.1	Sujeto 1 . . . . .	38
7.3.2	Sujeto 2 . . . . .	42
<b>8</b>	<b>Notas y conclusiones</b>	<b>44</b>

9 Herramientas utilizadas	45
A Impementación de consolidación por distancia	46
B Impementación de consolidación por adelgazamiento	47
C Impementación de DJ-Cluster	48
D Impementación de Canopy	49
E Impementación de consolidación por tiempo	50
Bibliografía	51

# 1 Introducción

Hoy en día, muchos dispositivos cuentan con un sistema de geolocalización GPS que nos permite conocer la localización de un sujeto en tiempo real. Con el fin de obtener la mayor información posible en todo momento, estas posiciones recogidas se guardan en una base de datos que puede ser temporal o permanente. En el caso de ser permanente, nos encontraremos con el problema de que la base de datos puede crecer hasta un límite desmesurado en el que dispositivo que recoge y almacena esta información llene su memoria, impidiendo almacenar posiciones nuevas.

En este momento, es necesario tomar la decisión de borrar parte de las posiciones almacenadas, según algún criterio. La dificultad en este momento es elegir el criterio con el cual eliminaremos este exceso de datos, por ejemplo, borrando posiciones repetidas o posiciones que no aporten la suficiente eficiencia en relación al espacio que ocupan en memoria. Esto introduce el concepto de función de consolidación o compactación, es decir una función que elimine un exceso de datos permitiéndonos conservar el máximo de información posible.

Contamos con datos proporcionados por una empresa de telecomunicaciones de sede en Zaragoza obtenidas de una base central. Se observa que esta empresa provee un servicio a sus clientes que permite que periódicamente se reciban posiciones de unos sujetos portadores de una terminal que transmita su posición GPS. Esta posición se inserta en una base de datos centralizada. Dichas posiciones son tomadas por la terminal de cada operativo, almacenadas localmente en esta terminal de manera temporal y enviadas a la central en el momento de conectividad con ésta.

Se observa un problema de almacenamiento de datos. Estos datos, cada vez más numerosos, empiezan a poblar la base de datos de una manera errática, es decir, un sujeto puede permanecer mucho tiempo en un sitio y seguir transmitiendo una posición constante a base. Nos lleva a plantearnos la siguiente pregunta, ¿es esto necesario?. ¿No sería más eficiente almacenar sólo una muestra de ésta? Al fin y al cabo, el objetivo del almacenamiento de estas posiciones es el ser posicionadas en un mapa, por lo que no necesitamos varias instancias de una misma.

Surge el concepto de *consolidación*. Este concepto nos lleva a que si un sujeto se ha movido muy poco o nada en una zona del espacio, sea posible eliminar de nuestra base de datos estas posiciones, quedándonos con una

central. A este proceso lo llamaremos consolidación y deberemos averiguar, dados los datos suministrados, las variables que nos son útiles a la hora de realizar este estudio.

Nos planteamos que tanto la terminal personal que lleva cada sujeto como la base centralizada pueden llegar a límite no deseado, provocando que este se sature e impida la inserción de nuevos datos. Con el fin de impedir esto, se va a realizar un estudio de distintas técnicas de *consolidación* con el fin de almacenar el mínimo de datos pero con la máxima información posible.

Este trabajo realiza una comparación entre diferentes técnicas de *clustering* y algoritmos diseñados propios con el fin de encontrar un método eficiente que evite el problema anteriormente explicado.

El código está disponible para bajarse y utilizarse bajo una licencia GNU GPL en,

<http://pbarbero.github.io/TFM/>

## 2 Datos y estructura de los datos suministrados

Se nos suministran dos bases de datos correspondientes a dos ciudades brasileñas distintas, **Salvador de Bahía** y **Río de Janeiro**. En cada de una de ellas encontramos posiciones de distintos sujetos estudiados identificados a través de un código. Cada base de datos contiene una tabla llamada *posicionesgps* en la que encontramos un registro por cada posición tomada por cada sujeto entre los días 2015-02-17 08:00:05 y 2015-03-04 08:18:05.

La estructura de los registros es la siguiente:

Parámetros	
Id	Identificador numérico de la posición (clave primaria)
IdServidor	Identificador numérico del servidor que realiza la inserción (PK)
Recurso	Nombre del recurso (tetra:1234567)
Latitud	Real que representa la latitud GPS
Longitud	Real que representa la longitud GPS
Velocidad	Entero que representa la velocidad instantánea
Orientación	Entero que representa la orientación respecto al norte en grados
Cobertura	Booleano que indica si hay cobertura
Error	Booleano que nos indica si ha habido algún error en la toma de la posición

En base de datos, el tipo de datos guardado es:

```
mysql> EXPLAIN posicionesgps;
```

Field	Type	Null	Key	Default
id	bigint(10)	NO	PRI	0
idServidor	int(10) unsigned	NO	PRI	0
recurso	varchar(100)	YES	MUL	NULL
latitud	double	YES		NULL
longitud	double	YES		NULL
velocidad	tinyint(10) unsigned	YES		NULL
orientacion	smallint(10) unsigned	YES		NULL
cobertura	tinyint(10) unsigned	YES		NULL
error	tinyint(10) unsigned	YES		NULL
antigua	tinyint(10) unsigned	YES		0
fecha	timestamp	NO	MUL	CURRENT_TIMESTAMP
automático	tinyint(10) unsigned	NO	MUL	0

## 2.1 Análisis de los datos

Vamos a utilizar **R** con el IDE *Rstudio* para realizar un análisis previo de los datos recibidos. Para ellos necesitamos de algunas librerías a la hora de conectarnos a la base de datos importada,

```
devtools::install_github("rstats-db/RMySQL")
devtools::install_github("rstats-db/DBI")
library(RMySQL)
library(DBI)
```

Importamos los datos haciendo una consulta sobre cada base de datos. Cada base de datos que se nos ha proporcionado cuenta con una tabla llamada *posicionesgps*,

```
conSalvador <- dbConnect(RMySQL::MySQL()
  , group = "posiciones"
  , user="root"
  , password="****"
  , dbname="posicionesSalvador")

dataquery=dbSendQuery(conSalvador
  , "SELECT latitud, longitud, velocidad, orientacion, fecha
    FROM posicionesgps")

dataSalvador = fetch(dataquery, n=-1)
```

Analicemos las columnas que más nos interesan, es decir, la latitud, longitud, la velocidad, la orientación y la fecha,

```
library(xtable)
valuesRio <- data.frame(x=dataRio)
xtable(summary(valuesRio))
```

latitud		longitud		velocidad	
1	Min. :-1.0103	1	Min. :-1.4575	1	Min. : 0.000
2	1st Qu.: -0.2266	2	1st Qu.: -0.6720	2	1st Qu.: 0.000
3	Median :-0.2259	3	Median :-0.6713	3	Median : 0.000
4	Mean :-0.1995	4	Mean :-0.6137	4	Mean : 3.751
5	3rd Qu.: -0.2248	5	3rd Qu.: -0.6702	5	3rd Qu.: 0.000
6	Max. : 0.4956	6	Max. : 2.4729	6	Max. :255.000



orientacion		fecha	
1	Min. : 0.0	1	Min. :2015-02-17 08:00:05
2	1st Qu.: 22.0	2	1st Qu.:2015-02-19 21:41:13
3	Median : 90.0	3	Median :2015-02-26 01:40:02
4	Mean :118.7	4	Mean :2015-02-24 19:55:44
5	3rd Qu.:202.0	5	3rd Qu.:2015-03-01 03:49:20
6	Max. :315.0	6	Max. :2015-03-04 08:18:05

Figure 1: summary de los datos de Salvador

## 2.2 Espacio en disco

Con la cantidad de posiciones suministradas, cuánto ocupa cada posición en disco, para hacernos una idea de cuántas posiciones sería posible acumular en función de la frecuencia de éstas sobre un espacio en disco finito.

En nuestra base de datos llamada **Río de Janeiro** contamos con **6928467** posiciones y en **Salvador de Bahía** contamos con **4599974** posiciones.

El tamaño en disco de nuestras bases de datos es,

```
mysql> SELECT table_schema as 'Database',
        table_name AS 'Table',
        round((((data_length + index_length) / 1024 / 1024), 2)
        FROM information_schema.TABLES
        ORDER BY (data_length + index_length) DESC;
```

Database	Table	Size in MB	Size in KB
rio	posicionesgps	1205.64	120564000
bahia	posicionesgps	961.42	96142000

Lo cual nos da una idea de cuánto puede ocupar una toma de posición en disco.

El total de posiciones almacenadas en río es de 6928467 luego podemos estimar el tamaño de una posición en,

$$\frac{120564000}{6928467} = 17.4012519653KB$$

El total de posiciones almacenadas en bahía es de 4599974, luego

$$\frac{96142000}{4599974} = 20.9005529162KB$$

Podemos aproximar el tamaño de una posición por unos 19 KB.

Supongamos que una consola tiene unos 1GB de almacenamiento. Podemos almacenar unas 52631 posiciones en estos 30GB.

Los datos han sido recogidos entre las fechas 2015-02-17 08:00:05 y 2015-03-04 08:18:05, lo que hace una diferencia de 360 horas.

Tenemos 5014 distintos tipos de sujetos a estudiar en la base de datos de río:

```
mysql> USE rio;
mysql> SELECT COUNT(distinct(recurso))
        FROM posicionesgps;
```

count(distinct(recurso))
5014

Lo que nos da una frecuencia de toma de:

$$\frac{6928467}{5014 \cdot 360} = 3.83$$

posiciones a la hora.

Si aumentáramos esta frecuencia a una posición cada 30 segundos, conseguiríamos una frecuencia de 120 posiciones a la hora, luego un único sujeto, en una jornada laboral de 8 horas, ocuparía en espacio de 19.2 MB.

## 2.3 Implementación de los datos en clases de Python

La estructura de los datos es implementable en diversos lenguajes, pero se elige Python por su simplicidad y ya que es el lenguaje científico más usado hoy en día.

Se define la clase `Position` de la siguiente manera,

---

```
class Position:
    def __init__(self, id, resource, lat
                  , lon, speed, track, date):
        self.id = id
        self.resource = resource
        self.lat = lat
        self.lon = lon
        self.speed = speed
        self.track = track
        self.date = date
```

---

A partir de esta clase definiremos una serie de métodos propios a ésta que nos permitirán saber si un punto está en un vecindario asociado a la posición. Vamos a utilizar la noción de distancia euclídea como concepto en el que apoyarnos.

```
def distance_eu(self, q):
    return math.sqrt((self.lat - q.lat)**2
                     + (self.lon - q.lon)**2)
```

### 3 Nociones de vecindario

Con el fin de realizar los algoritmos de consolidación, hemos realizado un estudio acerca de distintos tipos de vecindarios a utilizar para los algoritmos de consolidación propios y los algoritmos de *clustering* utilizados que usaremos más adelante.

#### 3.1 Vecindario simple

Utilizando la distancia euclídea, definimos un vecindario como aquel conjunto de puntos que se encuentran a una distancia euclídea menor que  $\epsilon$  con respecto su centro  $p_0$ , es decir,

$$d_E(p_0, p) = \sqrt{(lat_p - lat_{p_0})^2 + (long_p - long_{p_0})^2} < \epsilon$$

donde  $p$  es un punto con latitud  $lat_p$  y longitud  $long_p$ .

Su implementación en Python es la siguiente,

---

```
def IsInNeighEUSimple(self, q, eps):  
    return self.distance_eu(q) < eps
```

---

#### 3.2 Vecindario involucrando el módulo de la velocidad

En el momento que se toma la posición  $p_0$ , aparte de la latitud y su longitud, se toma la velocidad instantánea del sujeto. Podemos considerar en este caso que, dado que nuestro sujeto se encuentra a mayor velocidad, puntos más alejados de lo que consideraríamos en el primer caso (fuera de nuestro vecindario simple), podrían estar dentro de nuestro nuevo radio, que dependería de la velocidad instantánea. Así, definimos nuestro nuevo vecindario:

$$d_E(p_0, p) = \sqrt{(lat_p - lat_{p_0})^2 + (long_p - long_{p_0})^2} < \epsilon \cdot vel_{p_0}$$

donde  $vel_{p_0}$  es la velocidad instantánea de nuestro punto centro.

Su implementación en Python es la siguiente,

---

```
def IsInNeighSpeedRelative(self, q, eps):  
    return self.distance_eu(q) < eps * self.speed
```

---

### 3.3 Vecindad $t_0$ -alcanzable

Si fijamos un intervalo de tiempo  $t_0$ , podemos definir una vecindad  $t_0$ -alcanzable como aquellos puntos que nuestro sujeto puede alcanzar en un tiempo  $t_0$ . Un sujeto que se desplace a velocidad reducida, tendrá una vecindad  $t_0$ -alcanzable más reducido que otro que se desplace a una velocidad  $vel_{p_0} \cdot t_0$ .

$$d_E(p_0, p) = \sqrt{(lat_p - lat_{p_0})^2 + (long_p - long_{p_0})^2} < vel_{p_0} \cdot t_0$$

Éste es un caso concreto del vecindario involucrando la velocidad.

Su implementación en Python es la siguiente,

---

```
def IsInNeighT0Reachable(self, q, t0):  
    return self.distance_eu(q) < t0 * self.speed
```

---

### 3.4 Vecindario involucrando el tiempo

Las posiciones de nuestros sujetos vienen muestreadas además con el instante en el que fueron tomadas. Podemos considerar que el tiempo entre tomas también es una distancia y definir un vecindario. Definimos esta distancia temporal como la resta de ambos instantes, y el vecindario como:

$$d_T(p_0, p) = time_p - time_{p_0} < \delta$$

---

```
def is_neighborhoodByTime(self, q, lapse):  
    foo = time.mktime(self.date.timetuple())  
    bar = time.mktime(q.date.timetuple())  
    return abs(foo - bar) < lapse
```

---

## 4 Preprocesado de datos

Antes de empezar a realizar un algoritmo que nos realice una consolidación de los datos, es conveniente realizar un preprocesado de éstos.

Un primer procesado consistiría en la eliminación de todos aquellos registros que tienen como latitud y longitud 0 ya que son datos tomados por error que lo único que harían sería conseguir un clúster centrado en (*latitud* = 0, *longitud* = 0)

Vamos a fijar una cantidad mínima de distancia, un  $\varepsilon_0$ , y compararemos una posición con la última leída para decidir si la insertamos en base de datos o no. Si la distancia del nuevo muestreo con la última es menor que este  $\varepsilon_0$  fijado, desecharemos esta nueva posición. Esto permite que más adelante nuestro algoritmo de consolidación sea mucho más rápido.

### 4.1 Canopy

El algoritmo de clustering de **Canopy** se usa generalmente como un preprocesado de los datos para posteriormente aplicar un clustering de tipo **K-means** o alguna técnica de agrupamiento jerarquizado

La idea se basa en el uso de una medida de distancia aproximada para dividir el conjunto de los datos en subconjuntos que se superponen. A estos subconjuntos los llamaremos *canopies*. Un *canopy* es un subconjunto de puntos que yacen bajo el vecindario de un punto central. Un punto puede pertenecer a varias *canopies* distintas. Los *canopies* son creados con la intención de que si dos puntos no pertenecen a un *canopy* en común, están bastante lejos de pertenecer a un mismo clúster.

Debido a que **Canopy** no es más que un preprocesado de los datos, se fija una distancia sencilla con el fin de reducir drásticamente el número de puntos y posteriormente aplicar una técnica mejor. En nuestro caso, utilizaremos la distancia euclídea como distancia para realizar este proceso.

Dada una distancia euclídea, se crean los *canopies* como sigue,

1. Sea  $S$  nuestro conjunto de puntos.
2. Se fijan dos umbrales para  $T_1, T_2$  tal que  $T_1 > T_2$ .

3. Se toma un punto  $p \in S$ , éste será nuestro primer *canopy*.
4. Se colocan todos los puntos  $q \in S \setminus p$  tal que  $d_E(p, q) < T_1$  en el mismo *canopy*.
5. Se eliminan del conjunto inicial  $S$  aquellos puntos que estén dentro del umbral de distancia  $T_2$ .
6. Se repite hasta que el conjunto inicial esté vacío.

La implementación en Python se puede encontrar en el apéndice [D](#).

## 5 Algoritmos de consolidación simples

Utilizando las nociones de vecindario definidas en la sección anterior, nos planteamos la idea de definir unos algoritmos de consolidación simples con el fin de mantener la base de datos en un tamaño más o menos estable.

Una primera aproximación sería una creación de un trigger o un pequeño programa en el momento de inserción en base de datos que comparara la última posición recibida para ese sujeto con la nueva a insertar. Se compararía la distancia entre éstas con una distancia euclídea simple, y si ésta estuviera bajo el límite permitido (es decir, muy próxima), se obviaría.

Una segunda aproximación será definir una tarea programada `cron` (ya que nuestros dispositivos están basados en una distribución de Linux) que cada cierto tiempo ejecutara una consolidación sobre estos.

Estas consolidaciones menos avanzadas se realizarán sobre posiciones antiguas, es decir, según el tamaño de la base de datos y el nivel crítico al que puede llegar a estar, mandaremos un cierto número de posiciones a realizar la consolidación.



## 5.1 Consolidación por distancia

Utilizando los tres tipos de vecindarios que hemos definido, definimos el siguiente método que realizará la consolidación del tipo que le indiquemos,

---

**Algorithm 1** Algoritmo de consolidación simple por distancia

---

```
1: function CONSOLIDATIONBYDISTANCE(positions, typeOfDistance, eps, t0)
2:   for each pos in positions do
3:     if typeOfDistance == 'distanceEUSimple' then
4:       if pos.IsInNeighBorhood(next(pos), eps) then
5:         Remove position in DB
6:       else
7:         Maintain position in DB
8:       end if
9:     end if
10:    if typeOfDistance == 'DistanceEUrelativetospeed' then
11:      if pos.IsInNeighBorhoodRelativeSpeed(next(pos), eps)
12:    then
13:      Remove position in DB
14:    else
15:      Maintain position in DB
16:    end if
17:  end if
18:  if typeOfDistance == 't0reachable' then
19:    if pos.IsInNeighBorhoodT0Reachablee(next(pos), t0) then
20:      Remove position in DB
21:    else
22:      Maintain position in DB
23:    end if
24:  end if
25: end for
26: end function
```

---

## 5.2 Consolidación por adelgazamiento

Se puede dar el caso que la consolidación por distancia no sea lo suficientemente eficaz y no de los resultados necesarios de liberación de espacio, ya que las posiciones estén muy lejos entre sí. Como última opción, se puede recurrir a un tipo de consolidación en la cual dada una lista de posiciones normalmente antiguas, se elimine un subconjunto de estas, por ejemplo, 3 de cada 5. Así aseguraríamos una pérdida mínima de información.

---

**Algorithm 2** Algoritmo de consolidación por adelgazamiento

---

```
1: function CONSOLIDATIONBYTHINNING(positions, j, k)  $\triangleright j < k$ 
2:   for each pos in positions do
3:     if position.Index%k == 0 then
4:       for i = 0; i < k; i ++ do
5:         Remove position with index == position.Index
6:       end for
7:     end if
8:   end for
9: end function
```

---

Una sencilla implementación en Python se encuentra en el apéndice [B](#).

### 5.3 Consolidación por tiempo

Es posible que la toma de posiciones se tome de manera muy próxima en el tiempo, o simplemente que sea necesaria hacer una consolidación más drástica de las posiciones y se tome la decisión de reducir de un modo más severo la base de datos. Otra opción a la reducción por adelgazamiento sería una consolidación por tiempo. Se fija un lapso de tiempo que se debe cumplir entre posición y posición, y se eliminan todas aquellas que estén cuya distancia temporal con su siguiente esté por debajo de este lapso fijado.

---

**Algorithm 3** Algoritmo de consolidación por tiempo

---

```
1: function CONSOLIDATIONBYTIME(positions, lapse)
2:   for each pos in positions do
3:     nextpos = pos ++
4:     if IsInNeighborhoodByTime(nextpos, pos, lapse) then
5:       Remove pos
6:     end if
7:   end for
8: end function
9:
10: function ISINNEIGHBOORHODBYTIME(pos1, pos2, lapse)
11:   if  $|pos1.time - pos2.time| < lapse$  then
12:     Return true
13:   else
14:     Return false
15:   end if
16: end function
```

---

Una implementación en Python se puede encontrar en el apéndice [E](#).

## 6 Algoritmos de consolidación asociados a métodos de clustering

Un análisis cluster es un conjunto de técnicas multivariantes utilizadas para clasificar a un conjunto de individuos en grupos homogéneos. En nuestro problema, ésto nos va a resultar muy útil a la hora de encontrar ciertos patrones, o ciertos clústers que nos agruparán nuestros datos en subconjuntos de éstos, con el fin de identificar ése subconjunto con su centro y poder eliminar el resto de puntos.

En la sección 2.3 hemos definido una implementación en Python para el concepto de posición. Si queremos utilizar métodos de clustering más avanzados, se ha de definir el concepto de *clúster*.

Definimos un clúster de posiciones como un conjunto de posiciones agrupado en torno a una posición singular, llamada posición central del clúster.

Realizando una sencilla implementación en Python,

---

```
class Cluster:
    "Cluster of points"
    def __init__(self, center, points):
        self.center = center
        self.points = points
```

---

### 6.1 K-means

**K-means** es un método eficiente de *clustering* que tiene como objetivo la partición de un conjunto de  $n$  elementos en  $k$  grupos distintos. Dado un conjunto de datos  $(x_1, x_2, \dots, x_n)$ ,  $K$ -means construye una partición de las observaciones en  $k$  conjuntos con  $k \leq n$ ,  $S = \{S_1, S_2, \dots, S_k\}$  con el fin de minimizar el término de error que es la suma de las distancias al cuadrado de cada punto al centro de su clúster, es decir,

$$E = \sum_{i=1}^n \sum_{x \in S_i} d(x, m_i)$$

donde  $m_i$  es el centro de cada clúster  $S_i$  y  $d(x, m_i)$  es la distancia definida entre el punto  $x$  y  $m_i$ .

Inicialmente, el algoritmo asigna cada punto a su clúster de manera aleatoria. Posteriormente, itera sobre cada punto, encuentra el centro de

clúster más cercano y asigna el punto al clúster cuyo centro está más cercano. Esa iteración se repite hasta que el error es pequeño o se estabiliza.

Este algoritmo, aunque eficiente, tiene algunos inconvenientes con respecto a la consolidación de datos que se busca.

La primera de todas, es que se debe fijar un número de clústers a obtener desde el principio, lo que a priori no sería malo en nuestro caso, no es interesante en términos de eficiencia y de mantener la máxima información posible. En todo caso,  $K$ -means sería interesante para un primer procesamiento de datos en el cual la base de datos necesitara urgentemente un descenso de cantidad de posiciones almacenadas.

En segundo caso, no hay distinción entre puntos considerados "ruido", ya que todos los puntos se consideran en los clústers resultado. Esto introduciría muchos errores a la hora de intentar minimizar el término del error, ya que fácilmente se podrían etiquetar posiciones no significativas como ruido y no introducirlas en el proceso.

Además,  $K$ -means es un algoritmo no determinístico, debido a la primera fase de asignación de centros de clústers de manera aleatoria, por lo que no sería muy fiable.

## 6.2 DBSCAN

**DBSCAN** o **Density-based spatial clustering of applications with noise** es un algoritmo de *clustering* que dado un conjunto de puntos en un espacio, los agrupa en función de la densidad de puntos que tengan a su alrededor, dejando a un lado aquellos que tienen una densidad baja.

Se considera un conjunto de puntos a aplicar la técnica. El algoritmo clasificará los puntos en tres grupos,

- Un punto  $p$  es considerado *núcleo* si al menos un número de puntos mínimo (al que denotaremos por  $minPts$  están a una distancia menor que  $\varepsilon$  de  $p$ . Este conjunto de puntos se considerarán *directamente alcanzables* desde  $p$ .
- Un punto  $q$  es considerado *alcanzable* de  $p$  si existe un camino  $p_1, \dots, p_n$  tal que  $p_1 = p$  y  $p_n = q$ , donde cada  $p_{i+1}$  es directamente alcanzable desde  $p_i$  (todos los puntos del camino son puntos núcleo, excepto quizás  $q$ ).

- Todos los puntos que no son considerados ni núcleos ni alcanzables son considerados *aislados*.

Ahora, si  $p$  es un punto núcleo, entonces forma un clúster con aquellos puntos que sean alcanzables desde  $p$ . Cada clúster contiene al menos un punto núcleo; y puntos no núcleo pueden formar parte de éste, pero formaran lo que parten del *borde*, ya que no permiten *alcanzar* más puntos.

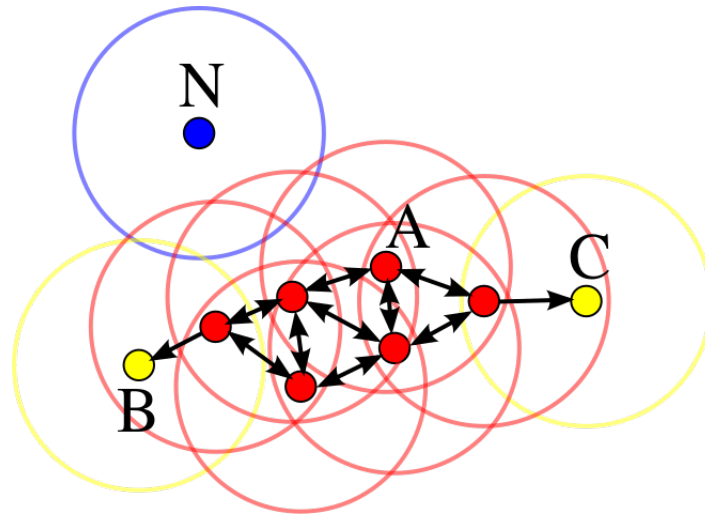


Figure 2: Diagrama DBSCAN

En el diagrama, se puede observar que si fijamos la variable *minPts* a 3, el punto  $A$  y los demás puntos rojos son puntos núcleo, ya que al menos están rodeados de 3 puntos en su vecindario de radio  $\epsilon$ . Como son densamente alcanzables unos con otros, forman un clúster. Los puntos  $B$  y  $C$  no son puntos núcleo, pero sí que son alcanzables desde  $A$ , por lo que también pertenecen al clúster. El punto  $N$  es calificado como aislado o *ruido* ya que no es ni punto núcleo ni densamente alcanzable.

La alcanzabilidad no es una relación simétrica ya que, por definición, ningún punto puede ser alcanzable por un punto no núcleo (un punto no núcleo puede ser alcanzable, pero no puedo "*alcanzar*"). Es necesario definir una noción más fuerte de *conectividad*. Decimos que  $p$  y  $q$  están densamente conectados si existe un punto  $o$  tal que  $p$  y  $q$  son densamente alcanzables. Esta noción de *densamente conectados* sí que es simétrica.

Redefinimos la noción de clúster que previamente habíamos definido. Un clúster debe satisfacer dos propiedades:

1. Todos los puntos deben estar mutuamente *densamente conectados*.
2. Si un punto  $q$  es densamente alcanzable desde un punto  $p$  del clúster,  $q$  es parte del clúster también.

**DBSCAN** requiere de dos parámetros para empezar:  $\varepsilon$  para la noción de vecindario y *minPts* para el número mínimo de puntos necesario para formar un clúster. Se empieza tomando arbitrariamente un punto del conjunto que no haya sido visitado. Se obtiene su vecindario, en el caso de que no exista, este punto se marca como ruido y se pasa al siguiente. Si no es nulo y tiene un número de puntos mayor que *minPts*, se crea un clúster.

Si uno de los puntos del proceso resulta que es parte de un clúster, su vecindario también se añade a éste. Se reitera este proceso, ya que todos los puntos nuevos añadidos del vecindario anterior, son parte del clúster, luego el vecindario de cada uno es añadido. Este proceso se continúa hasta que se obtiene el clúster densamente conectado.



---

**Algorithm 4** Algoritmo DBSCAN

---

```
1: function DBSCAN(positions, eps, minPts)
2:   C = 0
3:   for each pos in positions do
4:     if pos has been visited then
5:       Continue next position
6:     else
7:       Mark pos as visited
8:       N(pos) = NeighborPts(pos, eps)
9:       if length(N(pos)) < MinPts then
10:        Mark pos as noise
11:       else
12:        C = next Cluster
13:        expandCluster(pos, N(pos), C, eps, MinPts)
14:       end if
15:     end if
16:   end for
17: end function
18:
19: function EXPANDCLUSTER(P, NeighborPts, C, eps, MinPts)
20:   add P to cluster C
21:   for each P' in NeighborPts do
22:     if P' is not visited then
23:       Mark P' as visited
24:       NeighborPts' = regionQuery(P', eps)
25:       if length(NeighborPts') >= MinPts then
26:        NeighborPts = NeighborPts joined with NeighborPts'
27:       end if
28:     end if
29:     if P' is not yet member of any cluster then
30:       add P' to Cluster C
31:     end if
32:   end for
33: end function
34:
35: function NEIGHBORPTS(P, eps)
36:   return all points within P's eps-neighborhood (also P)
37: end function
```

---

### 6.2.1 Implementación en Python

Se encuentra una implementación bastante eficaz y sencilla en el repositorio de [Sushant Kafle 6](#).

---

```
from dbscanner import dbscanner
from algorithms.db import connect_db

cur= connect_db("bahia")
recurso = "tetra:12082781"
limit = 1000
cmd = "SELECT latitud, longitud
      FROM posicionesgps
      WHERE latitud <> 0 and longitud <> 0
      AND recurso=\"{0}\"
      LIMIT {1};".format(recurso, limit)
cur.execute(cmd)

a=[]
for pos in cur.fetchall():
    a.append([pos[0], pos[1]])

Data = a
eps = 0.0001
MinPts= 5

dbc = dbscanner()
dbc.dbscan(Data, eps, MinPts)
```

---

Notar que hemos tomado como valor de  $\varepsilon = 0.0001$  ya que es una aproximación de la distancia media de toma entre posiciones y 5 es un buen valor a la hora de hacer una consolidación. El resultado consiste en algunas posiciones marcadas como ruido y 5 clústers. Debido a que es un proceso muy costoso, nos hemos limitado en este caso a hacer la consolidación en unas 1000 posiciones.

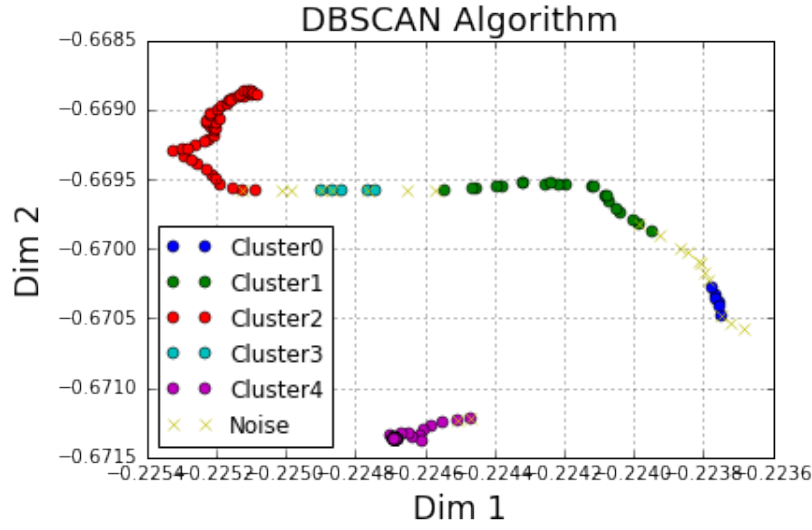


Figure 3: Resultados del algoritmo **DBSCAN** con  $\varepsilon = 0.0001$  y  $minPts = 5$

### 6.3 DJ-Cluster

**Density-Joinable Clúster**[2] es un tipo de algoritmo de *clustering* cuya realización depende de la distancia elegida, la cual nos generará un tipo de vecindario en concreto. Este algoritmo localiza puntos significativos sobre el conjunto de todos los puntos, es decir, el centro del clúster. No debemos olvidar que nuestro objetivo es encontrar posiciones significativas en todo nuestro conjunto de posiciones GPS, y éstos centros de clúster que nos generará este algoritmo nos servirán para tal propósito.

La idea del algoritmo es la siguiente, para cada punto, calculamos su vecindario. Este vecindario dependerá de la distancia elegida entre todas las anteriores definidas, y según cuál sea la elegida, dependerá de una variable  $\varepsilon$  o un instante  $t_0$  escogido. Se impone la condición de que el número de puntos conseguido al computar su vecindario sea al menos un *MinPts* definido previamente. Si esta condición no se cumple, se marca la posición actual como *ruido* y se prosigue con la siguiente. En el caso de cumplirse, este nuevo punto es el centro del clúster, junto a su vecindario.

Con este nuevo clúster creado, el siguiente paso es comprobar que este clúster no sea *densamente acoplable* con los que ya llevamos computados. Un clúster es *densamente acoplable* a otro clúster si existe un punto común entre ambos.

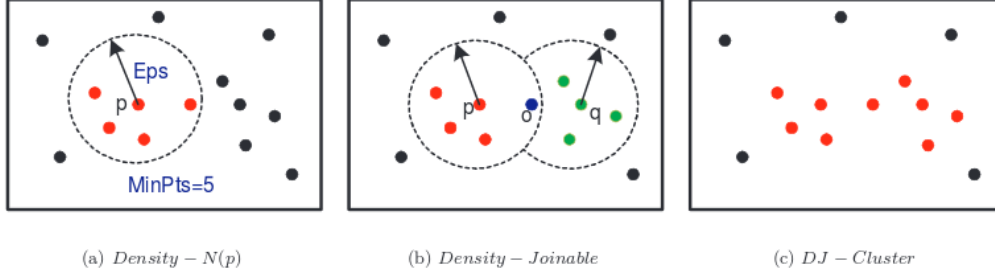


Figure 4: DJ-Clustering

---

**Algorithm 5** Algoritmo DJ-Cluster

---

```

1: for each  $p$  in set  $S$  do
2:   Compute neighborhood  $N(p)$  for  $\varepsilon$  and  $MinPts$ 
3:   if  $N(p)$  is null ( $|N(p)| < MinPts$  for  $\varepsilon$ ) then
4:     Label  $p$  as noise
5:   else if  $N(p)$  is density-joinable to an existing cluster then
6:     Merge  $N(p)$  with the cluster which is density-joinable
7:   else
8:     Create a new cluster  $C$  based on  $N(p)$ 
9:   end if
10: end for

```

---

Durante el proceso, se recorren todos los puntos del conjunto a analizar, calculando cada vecindario de cada punto con un centro  $p$  y un radio  $\varepsilon$ . Si el número de puntos del vecindario excede esta cantidad mínima  $MinPts$ , entonces es un vecindario a considerar. Este clúster es posteriormente *mergeado* con otros posibles clústers densamente acoplables.

Al final de cada iteración puede ser que el número de clústers no cambie, porque no existe un nuevo clúster o porque el nuevo clúster sea mergeado con alguno de los ya existentes.

El valor de los parámetros  $\varepsilon$  y  $MinPts$  es el que determina el tamaño de nuestros clusters. En nuestro caso, no buscamos grandes números de clústers, sino perder el mínimo de información posible, por lo que nos convendría tomar unos valores de  $\varepsilon$  y  $MinPts$  pequeños. 3

El valor de la variable  $\varepsilon$  debe tomarse en función de la precisión de los aparatos que toman las posiciones.<sup>2</sup> Podemos estimar este parámetro por unos 20 metros, que es la precisión de un GPS convencional.

Con respecto al valor de *MinPts*, un valor alto de este parámetro implica que los clusters deben ser más densos a la hora de formarse, pero un valor razonable estaría entre 3 y 10.<sup>3</sup>

La complejidad de este algoritmo es  $\mathcal{O}(n \log n)$  <sup>2</sup>.

## 7 Comparativa de resultados

Vamos a realizar un estudio de los métodos estudiados para dos sujetos en concreto. De dos sujetos de nuestra base de datos, tomaremos 2.000 posiciones y veremos cómo responden a cada uno de los algoritmos. A la hora de importar los datos a Weka, aplicamos un filtro de Normalización, dado que un futuro vamos a estar utilizando la distancia euclídea en nuestros algoritmos, debemos normalizar los datos para que ninguno de estos pese sobre los demás. Como la fecha está representada por un `timestamp`, éste tendría muchísimo más valor a la hora de realizar la consolidación.

Sujeto 1	tetra:12082781
Sujeto 2	tetra:12082364

Antes de importar las variables a Weka, necesitamos hacer el cálculo de su media y su desviación típica con el fin de tipificar cada una de las variables y dar la misma importancia a cada una de ellas.

Para el primer sujeto, contamos con lo siguiente,

	latitud	longitud	fecha
Media	-0.223	-0.665	1424174494.89
Desv. Típica	0.022	0.065	104277.37

Para el primer sujeto, hacemos una importación de datos a Weka de la siguiente forma,

```
mysql > SELECT (latitud + 0.223)/0.022 as 'latitudT',  
(longitud + 0.665)/0.065 as 'longitudT'.  
(UNIX_TIMESTAMP(fecha) - 1424174494.89)/104277.37 as 'time'  
FROM posicionesgps  
WHERE latitud <> 0 AND longitud <> 0 AND recurso='tetra:12082781'  
LIMIT 2000;
```

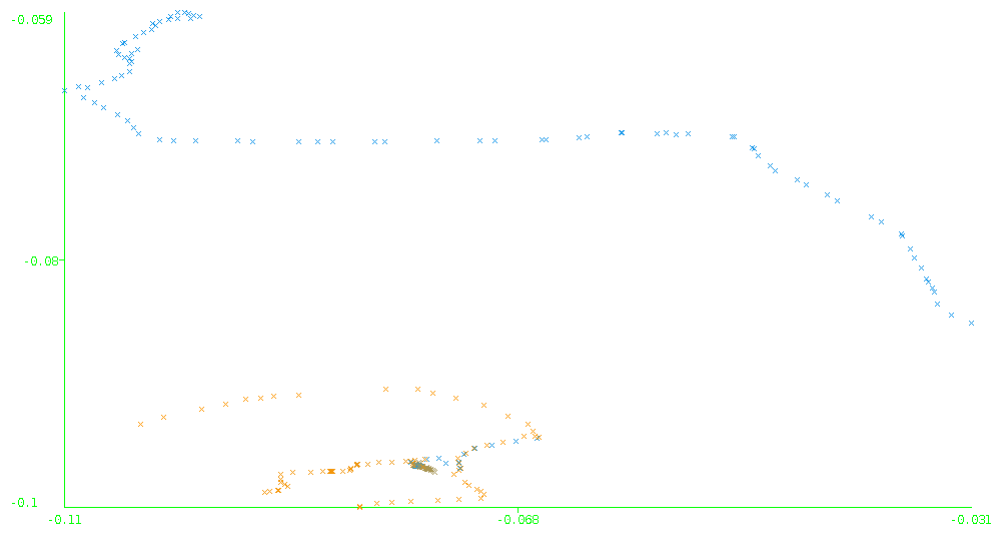


Figure 5: Distribución de las 2.000 posiciones del Sujeto 1

Calculamos la desviación típica y la media del segundo sujeto,

	latitud	longitud	fecha
Media	-0.21	-0.625	1424350386.203
Desv. Típica	0.057	0.169	41234.453

Para el segundo sujeto, hacemos una importación de datos a Weka de la siguiente forma,

```
mysql > SELECT (latitud + 0.21)/0.057 as 'latitudT',
(longitud + 0.625)/0.169 as 'longitudT',
(UNIX_TIMESTAMP(fecha) - 1424350386.203)/41234.453 as 'time'
FROM posicionesgps
WHERE recurso='tetra:12082364' AND latitud<>0 AND longitud<>0
ORDER BY 3 ASC
LIMIT 2000;
```

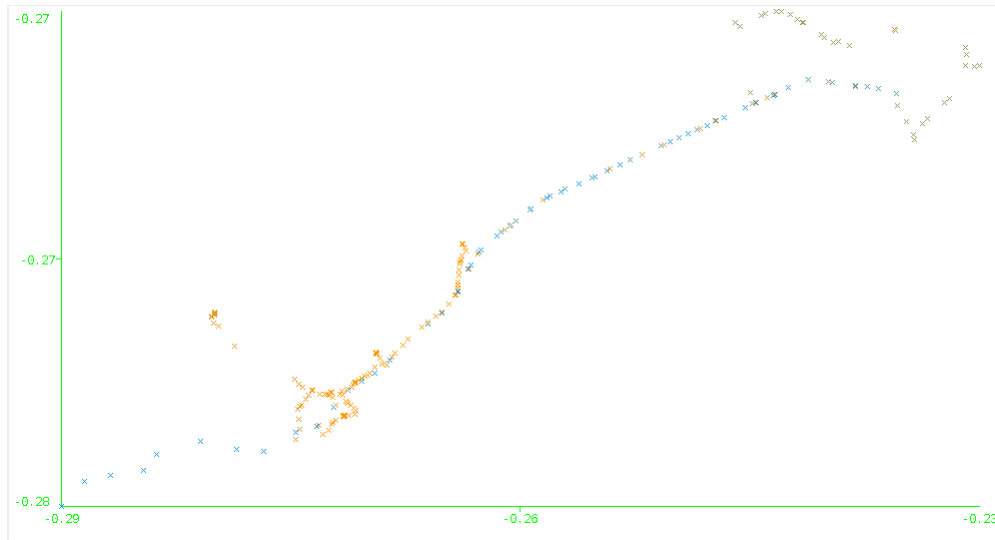


Figure 6: Distribución de las 2.000 posiciones del Sujeto 2

## 7.1 Resultados con K-means

Vamos a realizar un estudio **K-means** para ambos sujetos con los mismos parámetros que previamente habíamos aplicado, pero en vez de realizar una consolidación al 20%, vamos a reducir el número de posiciones a 500, es decir, una consolidación al 25%.

Realizaremos un preprocesado por *Canopy* y fijaremos el número de clústers 500,

### 7.1.1 Sujeto 1

=== Run information ===

```

Scheme:      weka.clusterers.SimpleKMeans -init 2 -max-candidates 100
-periodic-pruning 10000 -min-density 2.0
-t1 -1.25 -t2 -1.0 -N 500 -A "weka.core.EuclideanDistance
-R first-last" -I 500 -num-slots 1 -S 10
Relation:    QueryResult
Instances:    2000
Attributes:   3
              latitudT
              longitudT
              time

```



Test mode:      evaluate on training data

=== Clustering model (full training set) ===

kMeans

=====

Number of iterations: 9

Within cluster sum of squared errors: 0.11736312393686821

Initial starting points (canopy):

T2 radius: 0,504

T1 radius: 0,631

Cluster 0: -0.077067,-0.09795,0.073287

Cluster 1: -0.076555,-0.097964,-0.092632

Cluster 2: -0.044869,-0.075594, -0.166324

...

Time taken to build model (full training data) : 0.69 seconds

Se han realizado 10 iteraciones y se ha llegado a un error cuadrático de 0.117363. Observando el número de posiciones que ha agrupado por clúster, podemos observar que varían entre 1 y 9, lo cual es una buena media, ya que no ha agrupado demasiadas posiciones en un mismo clúster.

Se puede observar mejor en la gráfica. Cada clúster está representado por un color distinto. En la parte superior de esta gráfica, podemos observar que ha seguido un camino con respecto al tiempo en una dirección, mientras que en la parte inferior ha pasado varias veces por un mismo sitio, y aparece una aglomeración de colores en un punto. Esto se debe a que para nuestro estudio hemos introducido también la variable temporal, por lo que no puede agrupar todas esas posiciones en un mismo clúster, ya que no lo están, debido a que vienen de varios instantes distintos.

El total de clústers es de 500, ya que con  $K$ -means es necesario prefijarlos.

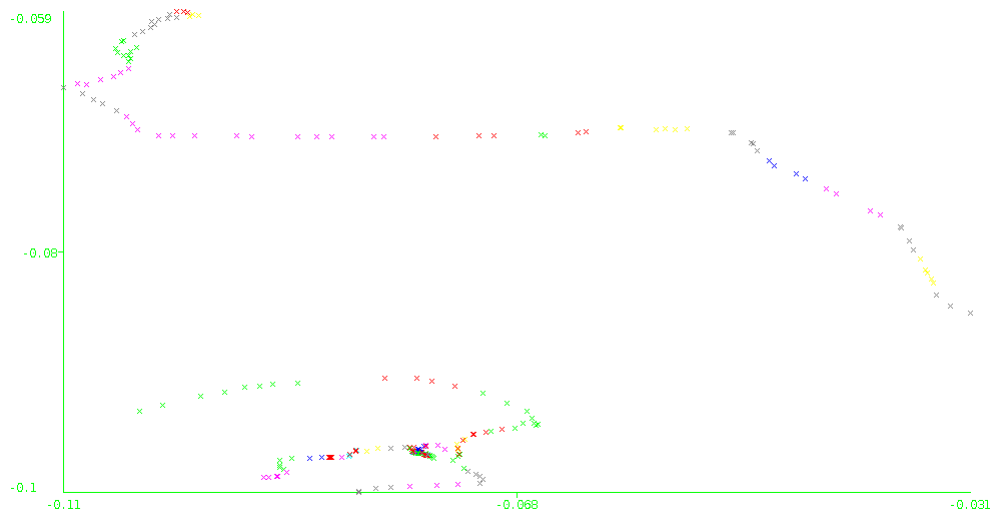


Figure 7: Estudio K-means para el Sujeto 1

### 7.1.2 Sujeto 2

Realizamos el mismo estudio con el Sujeto 2,

=== Run information ===

```

Scheme:      weka.clusterers.SimpleKMeans -init 2 -max-candidates 500 -periodic
Relation:    QueryResult
Instances:   2000
Attributes:  3
              latitudT
              longitudT
              time
Test mode:   evaluate on training data

```

=== Clustering model (full training set) ===

kMeans

=====

Number of iterations: 24

Within cluster sum of squared errors: 0.2079975699953984

Initial starting points (canopy):

T2 radius: 0,439

T1 radius: 0,548

Cluster 0: -0.278213,-0.272406,0.032468,

Cluster 1: -0.236123,-0.267776,-0.148367

.....

Time taken to build model (full training data) : 1.38 seconds

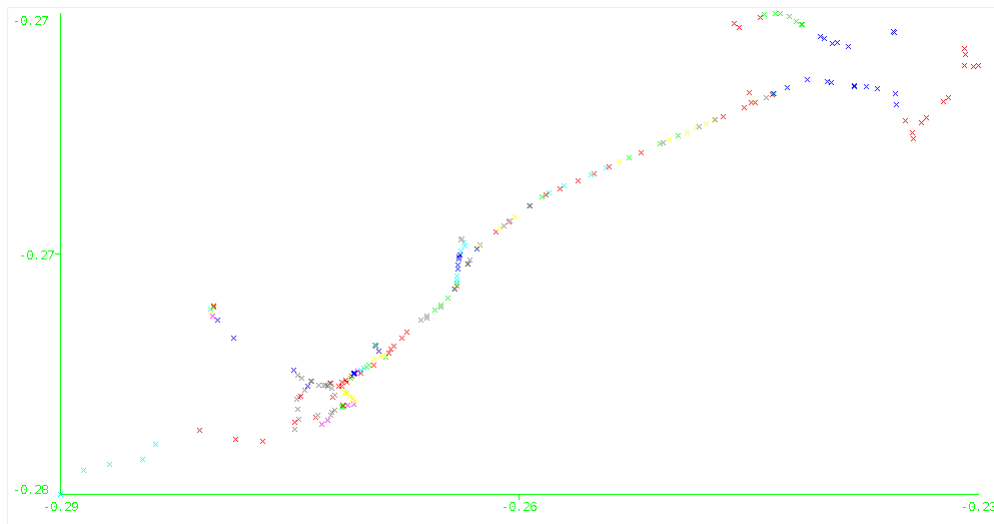


Figure 8: Estudio K-means para el Sujeto 1

En la siguiente figura, se observa una asignación un poco rara de los clústers. En la sección de abajo se pueden observar distintos colores, como si varios puntos muy cercanos se hubieran asignados a clústeres distintos, pero esto es debido a que no son seguidos en el tiempo, es decir, el sujeto está volviendo a pasar por el mismo sitio. Si nos volvemos a fijar en la representación sin clusterizar, se puede observar que existe un trozo donde el naranja y el azul se superponen, es decir, son instantes dinstantes temporalmente.

El error cuadrático es de 0.207997, un poco peor que con el sujeto 1, y el tiempo de ejecución es de 1.38 segundos.

## 7.2 Resultados con DBSCAN

Utilizaremos los mismos parámetros utilizados para el método **DBSCAN** previamente estudiado, un valor de  $\varepsilon = 0.0001$  y un valor de  $minPts = 5$ . Tomaremos 1000 datos de cada sujeto y los compararemos,

### 7.2.1 Sujeto 1

---

```
cur= connect_db("bahia")
recurso = "tetra:12082781"
limit = 2000
cmd = "SELECT latitud, longitud
      FROM posicionesgps
      WHERE latitud <> 0 and longitud <> 0 and recurso=\"{0}\"
      LIMIT {1};".format(recurso, limit)
cur.execute(cmd)

a=[]
for pos in cur.fetchall():
    a.append([pos[0], pos[1]])

Data = a
eps = 0.0001
MinPts=5

dbc = dbscanner()
dbc.dbscan(Data, eps, MinPts)
```

---

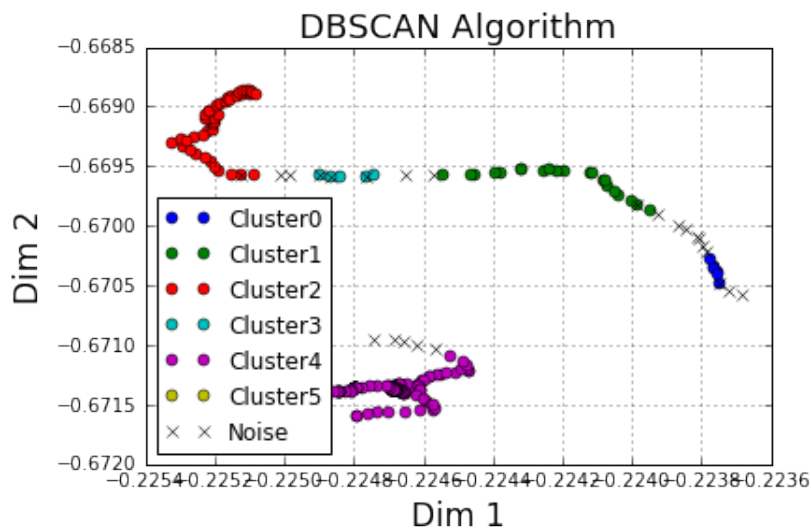


Figure 9: Estudio DBSCAN para el Sujeto 1

Una de las cosas positivas que se puede decir del algoritmo **DBSCAN** es que identifica puntos como *ruido*, cosa que los algoritmos de **K-means** y **DJ-Cluster** no hacen, ya que asignan simplemente esos puntos a un clúster de un único punto.

### 7.2.2 Sujeto 2

---

```

cur= connect_db("bahia")
recurso = "tetra:12082781"
limit = 2000
cmd = "SELECT latitud, longitud
      FROM posicionesgps
      WHERE latitud <> 0 and longitud <> 0 and recurso=\"{0}\"
      LIMIT {1};".format(recurso, limit)
cur.execute(cmd)

a=[]
for pos in cur.fetchall():
    a.append([pos[0], pos[1]])

Data = a
eps = 0.0001
MinPts=5

```

```
dbc = dbscanner()
dbc.dbscan(Data, eps, MinPts)
```

---

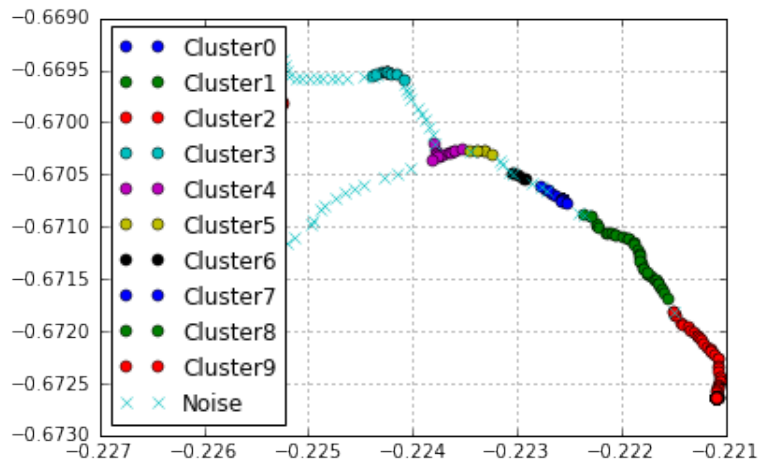


Figure 10: Estudio DBSCAN para el Sujeto 2

Otro de los problemas encontrados a la hora de utilizar el algoritmo **DBSCAN** es que la implementación utilizada sólo involucra las variables latitud y longitud, por lo que la consolidación no tendrá en cuenta si un sujeto pasa repetidas veces por un mismo lugar.

## 7.3 Resultados con DJ-Cluster

Realizaremos un estudio **DJ-Cluster** utilizando un preprocesado de datos **Canopy** fijando una desviación mínima estándar de 0.001.

### 7.3.1 Sujeto 1

Notar que si utilizamos un preprocesado de *Canopy* sin fijar el número de clústers previo, éste nos consolida demasiado la información (tal y como pasa en el DBSCAN), lo cual no es muy interesante.

=== Run information ===

```
Scheme:      weka.clusterers.MakeDensityBasedClusterer -M 0.001 -W weka.cluster
Relation:    QueryResult
```

Instances: 2000  
Attributes: 3  
          latitudT  
          longitudT  
          time  
Test mode: evaluate on training data

=== Clustering model (full training set) ===

MakeDensityBasedClusterer:

Wrapped clusterer:

Canopy clustering

=====

Number of canopies (cluster centers) found: 4

T2 radius: 0,504

T1 radius: 0,631

Cluster 0: -0.094874,-0.065178,-0.166183,{55} <0>

Cluster 1: -0.076573,-0.097976,-0.028496,{1474} <1,2>

Cluster 2: -0.077869,-0.097886,0.137391,{438} <1,2>

Cluster 3: -0.044869,-0.075594,-0.166324,{33} <3>

Time taken to build model (full training data) : 0.03 seconds

=== Model and evaluation on training set ===

Clustered Instances

0        51 ( 3%)

1       1170 ( 59%)

2       742 ( 37%)

3       37 ( 2%)

Log likelihood: 9.34996

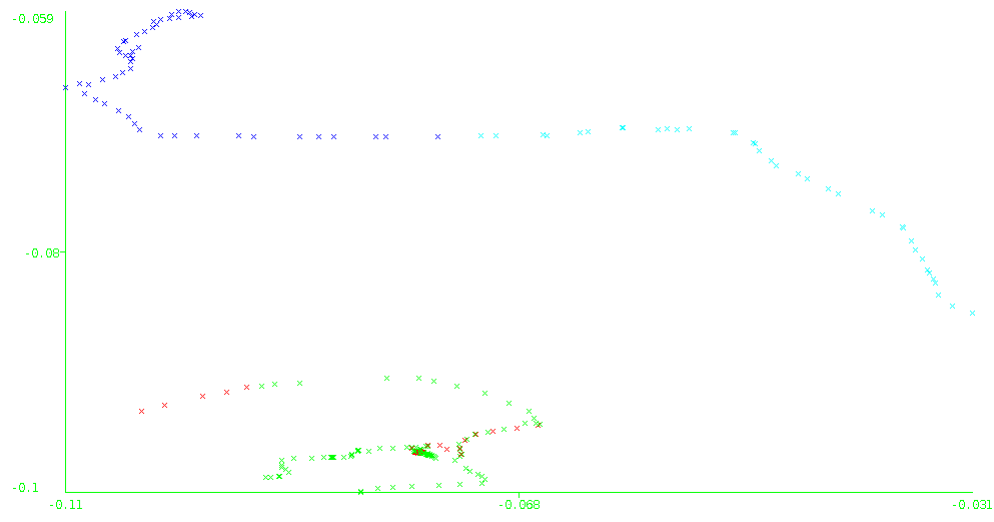


Figure 11: Estudio DBSCAN para el Sujeto 2

Realizaremos un preprocesado *Canopy* de 500 clústers,

=== Run information ===

```
Scheme:      weka.clusterers.MakeDensityBasedClusterer -M 0.001 -W weka.cluster
Relation:    QueryResult
Instances:   2000
Attributes:  3
              latitudT
              longitudT
              time
Test mode:   evaluate on training data
```

=== Clustering model (full training set) ===

MakeDensityBasedClusterer:

Wrapped clusterer:

Canopy clustering

=====

Number of canopies (cluster centers) found: 500

T2 radius: 0,504



T1 radius: 0,631

Cluster 0: -0.094874,-0.065178,-0.166183,

Cluster 1: -0.076573,-0.097976,-0.028496

...

El cual nos consigue un preprocesado de 500 clústers con Canopy, sin embargo, al aplicar DJ-Cluser, se nos reduce a 22 clústers

,

Time taken to build model (full training data) : 0.37 seconds

=== Model and evaluation on training set ===

Clustered Instances

0	3 ( 0%)
12	605 ( 30%)
58	7 ( 0%)
142	449 ( 22%)
165	3 ( 0%)
209	57 ( 3%)
230	188 ( 9%)
287	8 ( 0%)
295	32 ( 2%)
345	17 ( 1%)
353	9 ( 0%)
369	68 ( 3%)
373	3 ( 0%)
380	6 ( 0%)
385	2 ( 0%)
386	10 ( 1%)
458	7 ( 0%)
459	3 ( 0%)
464	3 ( 0%)
473	9 ( 0%)
474	1 ( 0%)
493	510 ( 26%)

Log likelihood: 9.05128

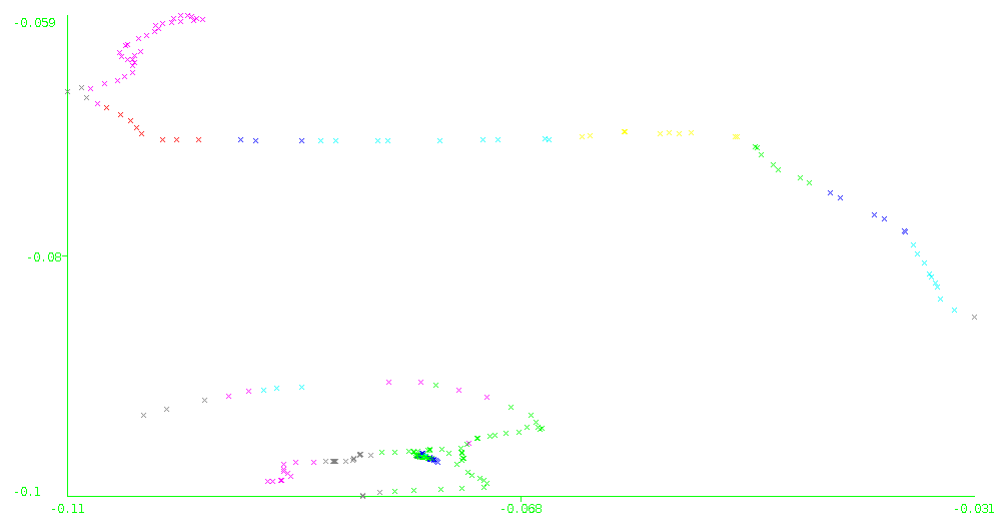


Figure 12: Estudio DJ-Cluster para el Sujeto 1

TODO COMENTAR

### 7.3.2 Sujeto 2

Con el sujeto 2 ocurre lo mismo que con el sujeto 1, con el preprocesado de *Canopy* obtenemos una consolidación en 500 clústers, sin embargo, al aplicar DJ-Cluster, obtenemos una consolidación a 27 clústers.

=== Model and evaluation on training set ===

Clustered Instances

0	6 ( 0%)
2	5 ( 0%)
3	1 ( 0%)
4	2 ( 0%)
76	5 ( 0%)
91	6 ( 0%)
96	36 ( 2%)
107	1 ( 0%)
133	6 ( 0%)
142	91 ( 5%)
146	68 ( 3%)
159	4 ( 0%)

166	19 ( 1%)
173	1 ( 0%)
211	17 ( 1%)
214	2 ( 0%)
288	8 ( 0%)
329	1669 ( 83%)
354	5 ( 0%)
370	1 ( 0%)
381	1 ( 0%)
386	5 ( 0%)
387	10 ( 1%)
391	1 ( 0%)
459	4 ( 0%)
474	16 ( 1%)
475	10 ( 1%)

Log likelihood: 10.83917

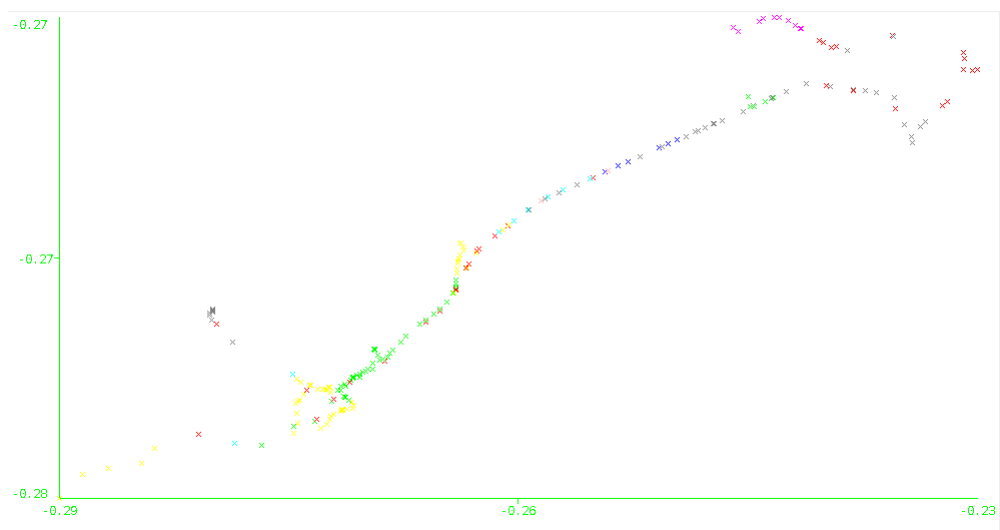


Figure 13: Estudio DJ-Cluster para el Sujeto 2

## 8 Notas y conclusiones

Se debe notar que entre los resultados de este texto, se encuentran dos tipos de algoritmos analizados e implementados. Los algoritmos de consolidación que hemos definido como "simples", que no realizan un estudio a partir de los datos, sino que parten de distintas nociones de distancia tanto espacial como temporal y realizan una consolidación en función de ésta. La ventaja de estos algoritmos es que son eficaces, cumplen el papel que prometen y liberan la memoria necesaria en disco para poder seguir con una inserción en base de datos. La desventaja principal de éstos es que a nivel estadístico no realizan un estudio de las propias características del dato en sí, como los algoritmos de *clustering*.

Los segundos algoritmos expuestos son de un nivel superior, ya que están pensados para cualquier tipo de dato, no necesariamente ordenado en una magnitud temporal. Con éstos aseguramos una menor pérdida de información, aunque sí que resultan de mayor coste computacional que los anteriores. En un futuro, esto queda a criterio de la persona que implementaría estos algoritmos directamente en la aplicación a utilizar.

Entre estos métodos también ha sido valorado el estudiar métodos de *clustering* jerarquizados. Sin embargo, la razón principal que desechó el estudio de éstos fue que la complejidad de un *clustering* aglomerativo jerarquizado es de  $\mathcal{O}(n^3)$  y la de *clustering* divisivo también jerarquizado es de  $\mathcal{O}(2^n)$ , bastante grandes en comparación a la complejidad computacional de  $\mathcal{O}(n \log n)$  de **DJ-Cluster**.

Entre estos últimos, **K-means**, **DJ-Cluster** y **DBSCAN** no se encuentran muchas diferencias. Obviamente, **K-means** es un algoritmo de mayor simpleza, pero se puede comprobar que es bastante eficaz a la hora de resolver nuestro problema. Notar que **DBSCAN** es un algoritmo que tiene la propiedad de etiquetar algunas posiciones como *ruido*, lo cual ni **K-means** ni **DJ-Cluster** realizan, así que sería un punto a favor para utilizar **DBSCAN**.

Otro punto a notar en favor de **DJ-Cluster** es a la hora de realizar una reconstrucción de la traza de movimiento a partir de los datos centralizados. TODO

## 9 Herramientas utilizadas

1. **Weka** (Waikato Environment for Knowledge Analysis, en español «entorno para análisis del conocimiento de la Universidad de Waikato») es una plataforma de software para el aprendizaje automático y la minería de datos escrito en Java y desarrollado en la Universidad de Waikato. Weka es software libre distribuido bajo la licencia GNU-GPL.<sup>7</sup>
2. **Python** se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.
3. **R** es un lenguaje y entorno de programación para análisis estadístico y gráfico. **R** se distribuye bajo la licencia GNU GPL y está disponible para los sistemas operativos Windows, Macintosh, Unix y GNU/Linux.<sup>8</sup>
4. **GitHub** es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git.
5. **MySQL** es un sistema de gestión de bases de datos relacional, multihilo y multiusuario bajo una licencia GNU GPL para uso no comercial.

## A Impementación de consolidación por distancia

---

"Consolidation By distance"

```
def ConsolidationByDistance(listPositions, typeOfDistance, eps, t0):
    i = 0
    result = []
    while i < len(listPositions) - 1:
        # Neighborhood: Distance EU simple
        if typeOfDistance == 0:
            if not listPositions[i].IsInNeighEUSimple(listPositions[i+1], eps):
                result.append(listPositions[i])
        # Neighborhood: Distance EU relative to speed
        elif typeOfDistance == 1:
            if not listPositions[i].IsInNeighSpeedRelative(listPositions[i+1], eps):
                result.append(listPositions[i])
        # Neighborhood t0 reachable
        elif typeOfDistance == 2:
            if not listPositions[i].IsInNeighT0Reachable(listPositions[i+1], t0):
                result.append(listPositions[i])
        else:
            raise ValueError('That distance does not exist')

        i=i+1

    result.append(listPositions[len(listPositions) - 1])

return result
```

---

## B Implementación de consolidación por adelgazamiento

---

"Consolidation by thinning."

```
def ConsolidationByThinning(listPositions, k, j):  
    if k >= j:  
        raise ValueError('K tiene que ser menor que J')  
  
    i = 0  
    result = []  
    while i < len(listPositions) - 1:  
        if i%j == 0:  
            l = 0  
            while l < k:  
                result.append(listPositions[i - l])  
                l = l+1  
            i = i+1  
  
    return result
```

---

## C Impementación de DJ-Cluster

---

```
from position import Position, Cluster
```

```
"Dj-Clustering Algorithm"
```

```
def DjCluster(setPoints, typeDistance, eps, minPoints, t0):
```

```
    listClusters = []
```

```
    listNoises = []
```

```
    for p in setPoints:
```

```
        np = computeNeighborhood(p, setPoints, typeDistance, minPoints, eps, t0)
```

```
        if np is None:
```

```
            listNoises.append(p)
```

```
        else:
```

```
            result = np.isDensityJoinable(listClusters)
```

```
            if result is None:
```

```
                listClusters.append(np)
```

```
            else:
```

```
                result.mergeCluster(np)
```

```
    return [listClusters, listNoises]
```

```
"Compute Neighborhood"
```

```
def computeNeighborhood(p, setPoints, typeDistance, minPoints, eps, t0):
```

```
    pointsOfCluster = []
```

```
    for q in setPoints:
```

```
        if typeDistance == 0:
```

```
            if p.is_in_neighborhoodByEUSimple(q, eps):
```

```
                pointsOfCluster.append(q)
```

```
        elif typeDistance == 1:
```

```
            if p.is_in_neighborhoodEUSpeed(q, eps):
```

```
                pointsOfCluster.append(q)
```

```
        elif typeDistance == 2:
```

```
            if p.is_in_neighborhoodT0Reachable(q, t0):
```

```
                pointsOfCluster.append(q)
```

```
    if len(pointsOfCluster) < minPoints:
```

```
        return None
```

```
    else:
```

```
        return Cluster(p, pointsOfCluster)
```

---



## D Impementación de Canopy

---

```
from sklearn.metrics.pairwise import pairwise_distances
import numpy as np

# T1 > T2 for overlapping clusters
# T1 = Distance to centroid point to not include in other clusters
# T2 = Distance to centroid point to include in cluster
# T1 > T2 for overlapping clusters
# T1 < T2 will have points which reside in no clusters
# T1 == T2 will cause all points to reside in mutually exclusive clusters

def canopy(X, T1, T2, distance_metric='euclidean', filemap=None):
    canopies = dict()
    X1_dist = pairwise_distances(X, metric=distance_metric)
    canopy_points = set(range(X.shape[0]))
    while canopy_points:
        point = canopy_points.pop()
        i = len(canopies)
        canopies[i] = {"c":point, "points": list(np.where(X1_dist[point] < T2)[0])}
        canopy_points = canopy_points.difference(set(np.where(X1_dist[point] < T1)[0]))
    if filemap:
        for canopy_id in canopies.keys():
            canopy = canopies.pop(canopy_id)
            canopy2 = {"c":filemap[canopy['c']], "points":list()}
            for point in canopy['points']:
                canopy2["points"].append(filemap[point])
            canopies[canopy_id] = canopy2
    return canopies
```

---

## E Impementación de consolidación por tiempo

---

"Consolidation by time. Deletes positions too close by time."

```
def ConsolidationByTime(listPositions, lapse):  
    i = 0  
    result = []  
    while i < len(listPositions) - 1:  
        if not listPositions[i].is_neighborhoodByTime(listPositions[i+1], lapse):  
            result.append(listPositions[i])  
            i=i+1  
    result.append(listPositions[len(listPositions) - 1])  
  
    return result
```

---

## Bibliografía

- [1] Yang Ye, Yu Zheng, Yukun Chen, Jianhua Feng, Xing Xie  
*Mining Individual Life Pattern Based on Location History*
- [2] Changqing Zhou, Nupur Bhatnagar, Shashi Shekhar, Loren Terveen  
*Mining Personally Important Places from GPS Track*
- [3] Changqing Zhou, Dan Frankowski, Pamela Ludford, Shashi Shekhar, Loren Terveen  
*Discovering Personal Gazetteers: An Interactive Clustering Approach*
- [4] Andrew McCallum, Kamal Nigam, Lyle H. Ungar  
*Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching*
- [5] *Gabe's Gist*  
Efficient python implementation of canopy clustering.
- [6] *Sushant Kafle's* GitHub Repository  
Implementation of DBSCAN Algorithm in Python.
- [7] Ian H. Witten and Eibe Frank (2005). Data Mining: Practical Machine Learning Tools and Techniques. 2nd Edition, Morgan Kaufmann, San Francisco.  
[Weka Project](#)
- [8] R Development Core Team (2008). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0