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 Nalvaiz
Universidad de Zaragoza

Índice

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 encuentra entonces 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 ésto 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 de 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 cen-

tral.

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 no permita 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.

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

Figura 1: Descripción de los datos suministrados

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	tiniyint(10) unsigned	NO	MUL	0

Figura 2: `EXPLAIN posicionesgps`

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*:

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

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

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

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

```
summary(dataBahia)
```

	latitud
1	Min. :-1.0103
2	1st Qu.:-0.2266
3	Median :-0.2259
4	Mean :-0.1995
5	3rd Qu.:-0.2248
6	Max. : 0.4956

	longitud
1	Min. :-1.4575
2	1st Qu.:-0.6720
3	Median :-0.6713
4	Mean :-0.6137
5	3rd Qu.:-0.6702
6	Max. : 2.4729

	velocidad
1	Min. : 0.000
2	1st Qu.: 0.000
3	Median : 0.000
4	Mean : 3.751
5	3rd Qu.: 0.000
6	Max. :255.000

	orientacion
1	Min. : 0.0
2	1st Qu.: 22.0
3	Median : 90.0
4	Mean :118.7
5	3rd Qu.:202.0
6	Max. :315.0

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

Figura 3: `summary` de los datos de Salvador

Las unidades en las que está tomada la velocidad son km/h, por lo que un máximo de 255 *es algo curioso. Realizando un conteo de datos, obtenemos que unas* 723277 *posiciones son distintas a* 0 *de un total de* 4599974*, por lo que aproximadamente un* 85 % *de las posiciones son* 0*. Esto es un dato a nombrar, ya que posteriormente usaremos la velocidad a la hora de definir distancias.*

2.2. Espacio en disco

Con la cantidad de posiciones suministradas, vamos a calcular cuánto ocupa una 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;
```

Size in KB	
rio	120564000
bahia	96142000

Figura 4: Tamaño de la base de datos

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:

$$\frac{120564000}{6928467} = 17{,}4012519653 \text{ KB}$$

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

$$\frac{96142000}{4599974} = 20{,}9005529162 \text{ KB}$$

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

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, lo que nos da una frecuencia de toma de:

$$\frac{6928467}{5014 \cdot 360} = 3{,}83 \text{ posiciones a la hora.}$$

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. Si multiplicamos por los 5014 sujetos que contiene la base de datos proporcionada, son casi 100GB por jornada laboral almacenados en la base de datos centralizada, por lo que una consolidación cada día de un tanto por ciento definido con el cliente, sería necesario (esto quedaría a criterio de las características del sistema donde se aloja la base de datos).

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 sobre 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 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 ε con respecto su centro p_0, es decir:

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

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 la velocidad

En el momento que se toma la posición p_0, aparte de la latidud 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:

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

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

Su implementación en Python es la siguiente:

```
def IsInNeighSpeedRelative(self, q, eps):
        if self.speed != 0:
            return self.distance_eu(q) < eps * self.speed
        else:
            return False
```

3.3. Vecindad t0-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 reducida que otro que se desplace a una velocidad superior. Redefiniremos el radio de nuestro vecindario a través de la velocidad instantánea que lleve nuesto sujeto, es decir, $vel_{p_0} \cdot t_0$.

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

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:

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

Su implementación en Python es la siguiente:

```
def is_neighboorhoudByTime(self, q, lapse):
        time1 = time.mktime(self.date.timetuple())
        time2 = time.mktime(q.date.timetuple())
        return abs(time1 - time2) < 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 ε_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 ε_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**[5] 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[4] en Python se puede encontrar en el apéndice B.

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 éstos.

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:

Algoritmo 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)
   then
12:                Remove position in DB
13:            else
14:                Maintain position in DB
15:            end if
16:        end if
17:        if typeOfDistance ==' t0reachable' then
18:            if pos.IsInNeighBorhoodT0Reachablee(next(pos), t0) then
19:                Remove position in DB
20:            else
21:                Maintain position in DB
22:            end if
23:        end if
24:    end for
25: end function
```

Una implementación simple se puede encontrar en el apéndice A

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 úlima 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, ya que no borraríamos un bloque de posiciones, sino que intercalaráimos el borrado, dejando una frecuencia constante.

Algoritmo 2 Algoritmo de consolidación por adelgazamiento

```
1: function ConsolidationByThinning(positions, j, k)        ▷ 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 A.

5.3. Consolidación por tiempo

Una alternativa a una técnia de consolidación por adelgazamiento sería una consolidación por tiempo. Es posible que la toma de posiciones se tome de manera muy próxima temporalmente, 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. 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.

Algoritmo 3 Algoritmo de consolidación por tiempo

```
1: function ConsolidationByTime(positions, lapse)
2:     for each pos in positions do
3:         nextpos = pos++
4:         if IsInNeighboorhodByTime(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 A.

6. Algoritmos de consolidación asociados a métodos de clustering

Un análisis clúster es un conjunto de técnicas multivariantes utilizadas para clasificar a un conjunto de individuos en grupos homogéneos. Hemos elegido una serie de técnicas de aprendizaje no supervisado ya que éste parte de que no hay un conocimiento a priori y es útil en técnicas de compresión de datos.

En nuestro problema, ésto nos va a resultar muy útil a la hora de encontrar ciertos patrones, o ciertos clústers que nos agruparán nuestros en 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 clústering 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[1] es un método eficiente de *clustering* que tiene como objetivo la partición de un conjunto de n elementos en k grupos distintos. Dado un cojunto de datos $(x_1, x_2, \ldots, x_n)$, **K-means** construye una partición de las observaciones en k conjuntos con $k \leq n$, $S = \{S_1, S_2, \ldots, S_k\}$ 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 procesado de datos en el cual la base de datos necesitara urgentemente un descenso de cantidad de posiciones almacenadas.

En segundo caso, no hay distintos ente 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[1] o **Density-based spatial clustering of applications with noise** es un algoritmo de *clustering* basado en la densidad por lo que encuentra el número de clústers comenzando por una estimación de la distribución de densidad de los nodos correspondientes. 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 ε de p. Este conjunto de puntos se considerarán *directamente alcanzables* desde p.
- Un punto q es considerado *alcanzable* de p si existe un camino $p_1, \ldots, p_n$ 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.

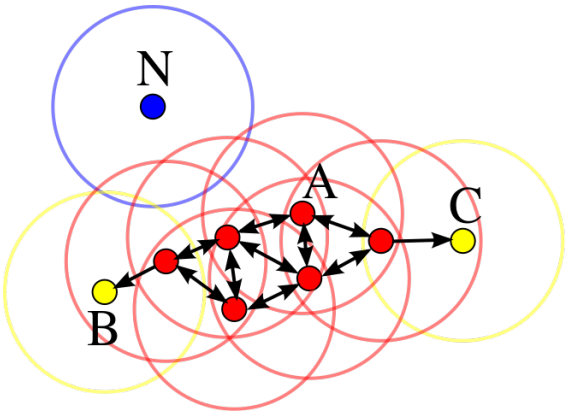


Figura 5: Diagrama **DBSCAN**

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 ε. 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 calificiado 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, ninguún 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: ε para la noción de vecindario y $minPts$ para el número mínimo de puntos necesario para formar un clúster. Se empieza tomando arbirariamente 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.

Algoritmo 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.[7]

```
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)
```

Figura 6: Ejemplo de uso de la implementación de **DBSCAN**

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.

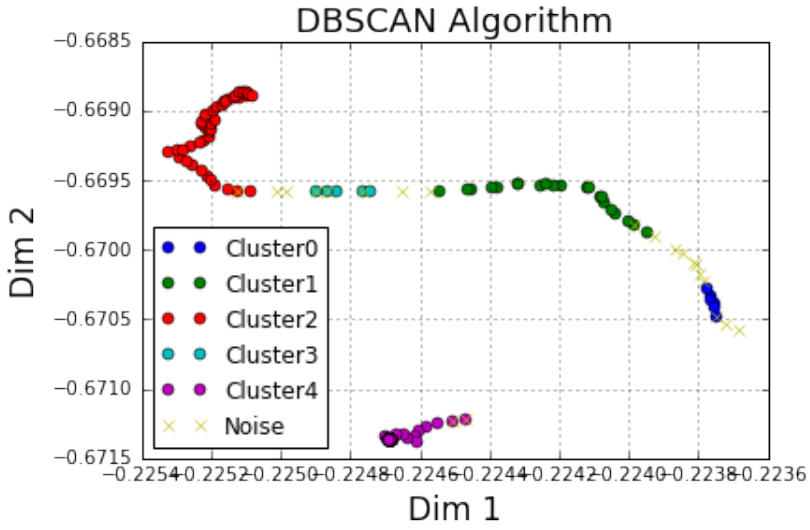


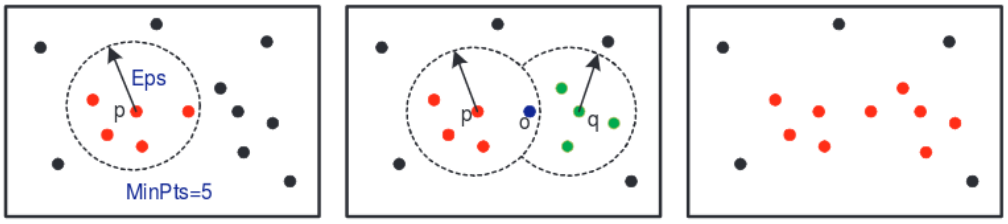
Figura 7: Resultado **DBSCAN**

6.3. DJ-Cluster

Density-Joinable Clúster[1] es un tipo de algoritmo de *clustering* basado en densidades de puntos que intenta solventar algunas de las limitaciones de **K-means**. 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 ε 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.



(a) $Density - N(p)$ (b) $Density - Joinable$ (c) $DJ - Cluster$

Figura 8: Diagrama **DJ-Clustering**

Algoritmo 5 Algoritmo DJ-Cluster

```
1: for each p in set S do
2:     Compute neighborhood N(p) for ε and MinPts
3:     if N(p) is null (|N(p)| < MinPts for ε) 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 ε. 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 ε 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 ε y $MinPts$ pequeños.[3]

El valor de la variable ε debe tomarse[3] en función de la precisión de los aparatos que toman las posiciones. Podemos estimar este parámero por unos 20 metros, que es la precisión de un GPS convencional.

Con respecto al valor de $MinPts$, un valor alto de esta parámetro implica que los clusters deben ser más densos a la hora de formarse, pero un valor razonable[3] estaría entre 3 y 10.

La complejidad de este algoritmo es $\mathcal{O}(n \log n)$.[1]

En los próximos resultados utilizaremos una implementación en Weka para lanzar un estudio, sin embargo, se ha desarrollado en Python el algoritmo de **DJ-Clúster** de manera parecida al de **DBSCAN**. Se puede consultar una implementación en el anexo B.

7. Aplicación de los algoritmos

Se va a realizar una comparativo de todos los métodos estudiados para dos sujetos en concreto de nuestra base de datos. Se han tomado los dos sujetos como más posiciones, y de cada uno de éstos, un muestreo de 2000 posiciones.

Antes de aplicar los métodos, es necesario aplicar un filtro de Normalización, dado que de no aplicarlo, la fecha pesaría sobre todos los demás y dejaría el resto de las variables sin valor (no hay que olvidar que la fecha es un `timestamp`, por lo que a efectos prácticos es un entero muy grande).

Sujeto 1	tetra:12082781
Sujeto 2	tetra:12082364

Antes de importar las variables de nuestra base de datos, 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

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;
```

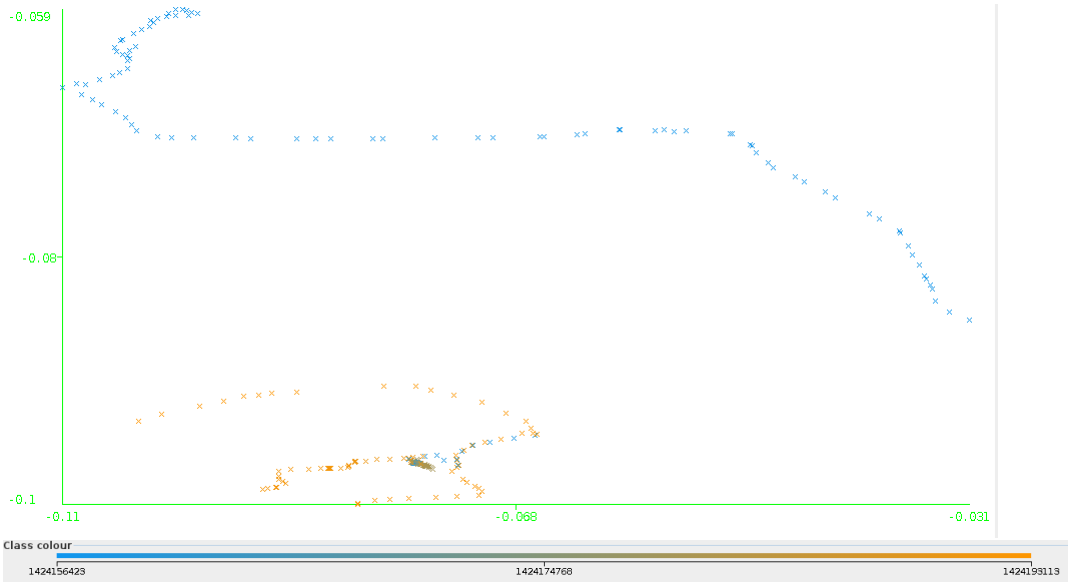


Figura 9: Distribución de las 2.000 posiciones del Sujeto 1

Se hace una distinción de colores azules y naranjas en función del tiempo. Se puede observar la traza de movimiento del sujeto, que empieza en la esquina superior izquierda y acaba en la parte inferior de la gráfica.

Calculamos la desviación típica y la media del segundo sujeto, del mismo modo que con el sujeto 1.

	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 partir de nuestra base de datos 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;
```

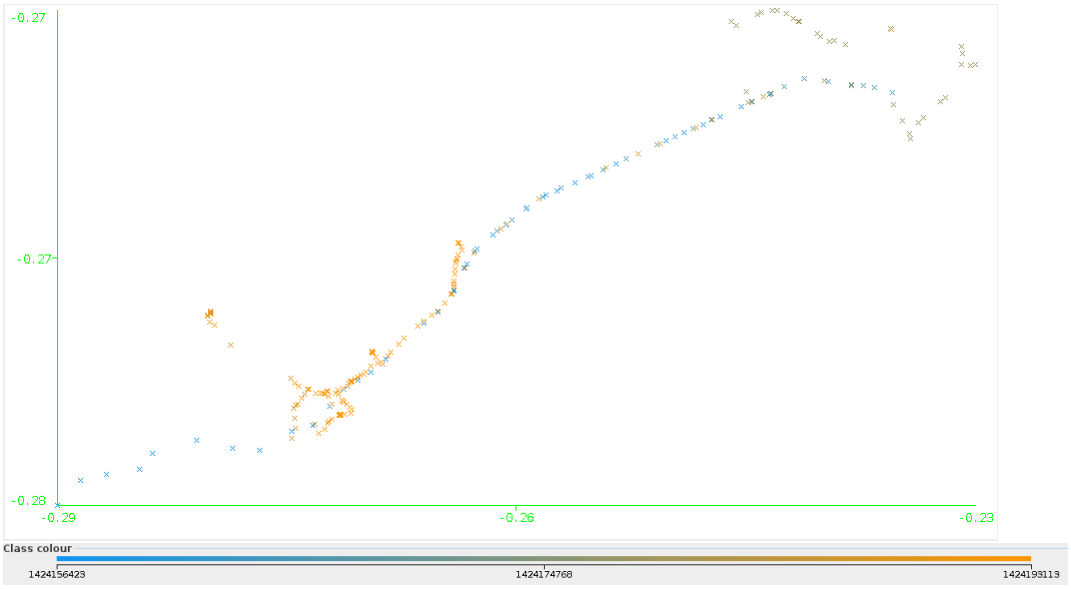


Figura 10: Distribución de las 2.000 posiciones del Sujeto 2

7.1. Resultados de algoritmos por consolidación simple

La importación de datos para poder utilizar los algoritmos que se han desarrollado en la sección 5 es parecida a la utilizada en el algoritmo **DBSCAN**.

Utilizaremos la clase `db` que hemos creado para conectarnos a la base de datos donde almacenamos las posiciones:

```
cur_sal = connect_db("bahia")
limit = 2000
cmd = "SELECT * FROM posicionesgps WHERE latitud<>0
                    AND longitud<> 0 AND recurso='tetra:12082781'
                    LIMIT {0};".format(limit)
cur_sal.execute(cmd)
```

Y crearemos una lista de posiciones con la implementación previa que hemos desarrollado:

```
list_pos = []
for row in cur_sal.fetchall():
        q = Position(row[0] # id
```

```
                , row[2] # resource
                , row[3] # lat
                , row[4] # lon
                , row[5] # speed
                , row[6] # track
                , row[10] # date
                )
        list_pos.append(q)
```

Tipificaremos las variables como anteriormente hemos hecho, calculando su media y su desviación típica:

```
listPosTyp = []
lats = []
longs = []

for pos in list_pos:
    lats.append(pos.lat)
meanLat = np.mean(lats)
for pos in list_pos:
    longs.append(pos.lon)
meanLon = np.mean(longs)
devLat = np.std(lats)
devLon = np.std(longs)

latsTyp = []
longsTyp = []
for pos in list_pos:
        q = Position(pos.id, pos.resource
        , (pos.lat - meanLat)/devLat
        , (pos.lon - meanLon)/devLon
        , pos.speed, pos.track, pos.date)
        listPosTyp.append(q)
        latsTyp.append(q.lat)
        longsTyp.append(q.lon)
```

Utilizaremos el código definido en el anexo B para realizar las consolidaciones por adelgazamiento, tiempo y distancia.

```
import consolidation as cs
```

7.1.1. Consolidación por adelgazamiento

Y vamos a realizar una primera consolidación por adelgazamiento, se mantienen 2 posiciones de cada 5:

```
result = cs.ConsolidationByThinning(list_pos, 2, 5)
```

Utilizando `matplotlib`, dibujamos los resultados.

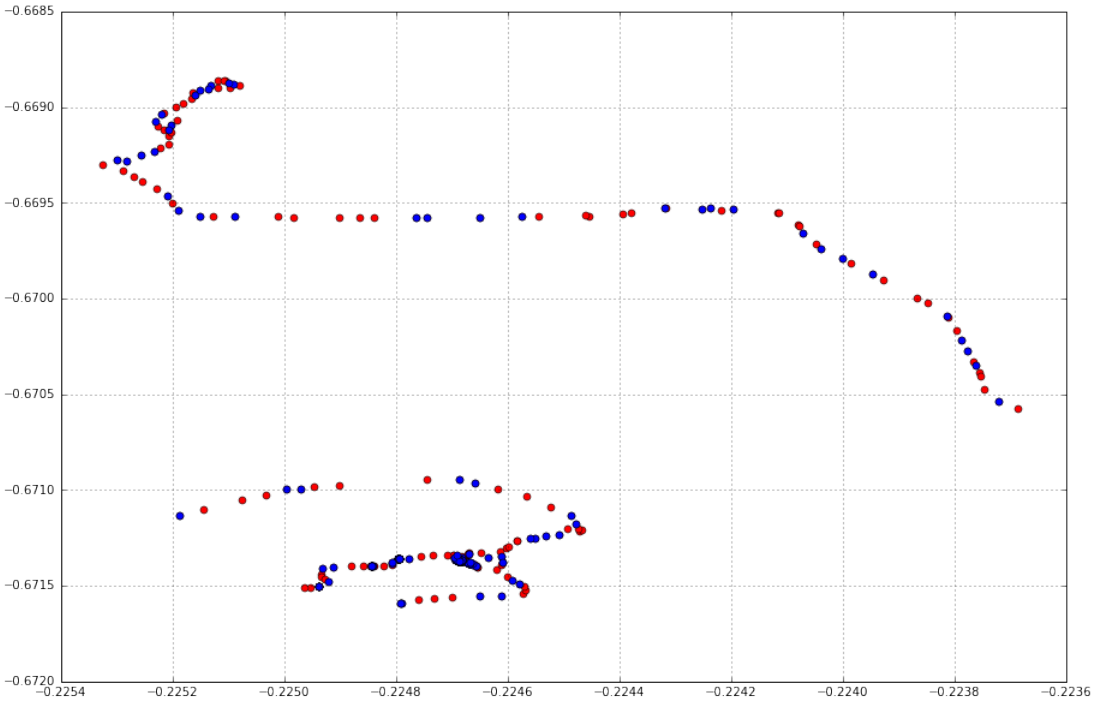


Figura 11: Resultado consolidación por adelgazamiento

Los puntos rojos son los eliminados porque se han consolidado y los azules son los que se han mantenido. En este caso, se mantienen 800 posiciones.

7.1.2. Consolidación por tiempo

Realizaremos una consolidación por tiempo, eliminaremos las posiciones que tengan entre ellas un lapso menor que 20:

```
result = cs.ConsolidationByTime(list_pos, 20)
```

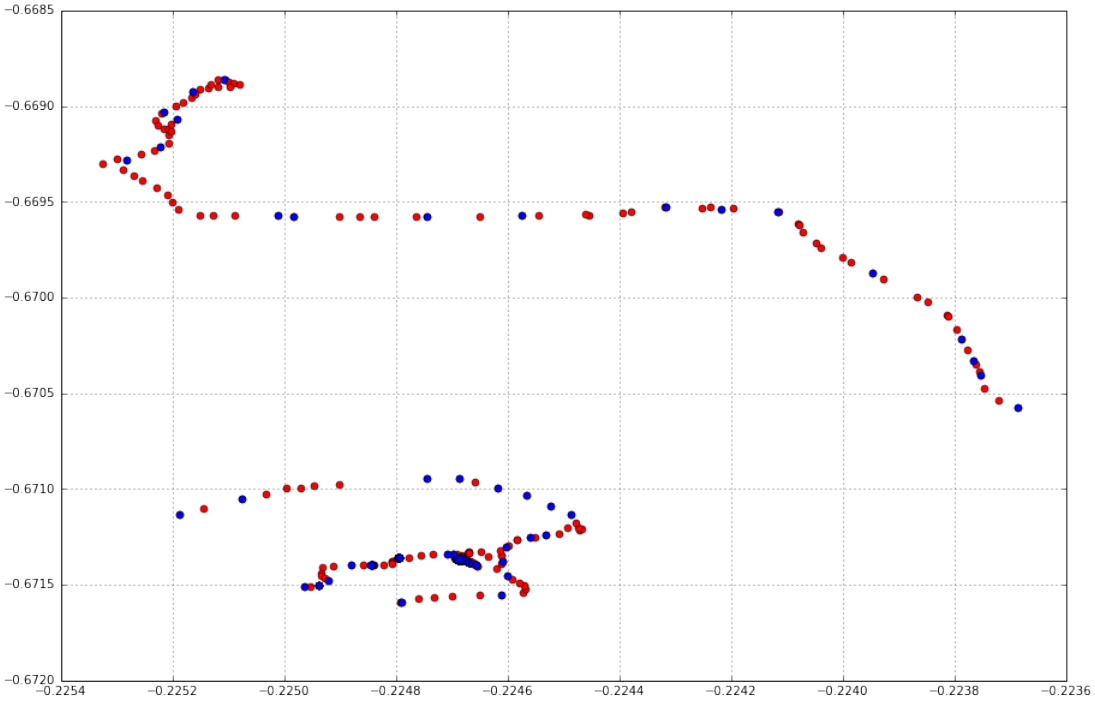


Figura 12: Resultado consolidación por tiempo

En este caso se mantienen 507 posiciones.

7.1.3. Consolidación por distancia simple

Se realiza una consolidación por distancia simple, es decir, por distancia euclídea dando un radio de $\varepsilon = 0{,}0001$:

```
result = cs.ConsolidationByDistance(list_pos, 0, 0.0001, 1)
```

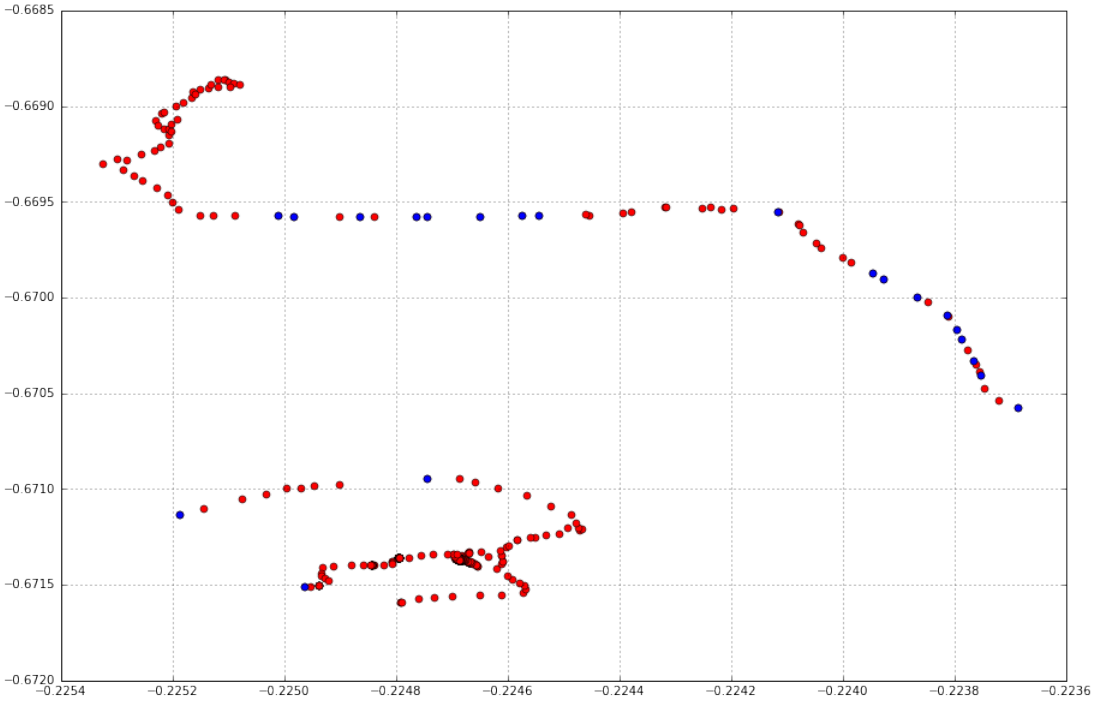


Figura 13: Resultado consolidación por distancia Simple

Se han reducido el número de puntos a 21.

7.1.4. Consolidación por distancia t0-alcanzable

```
cs.ConsolidationByDistance(list_pos, 2, 0.001, 1)
```

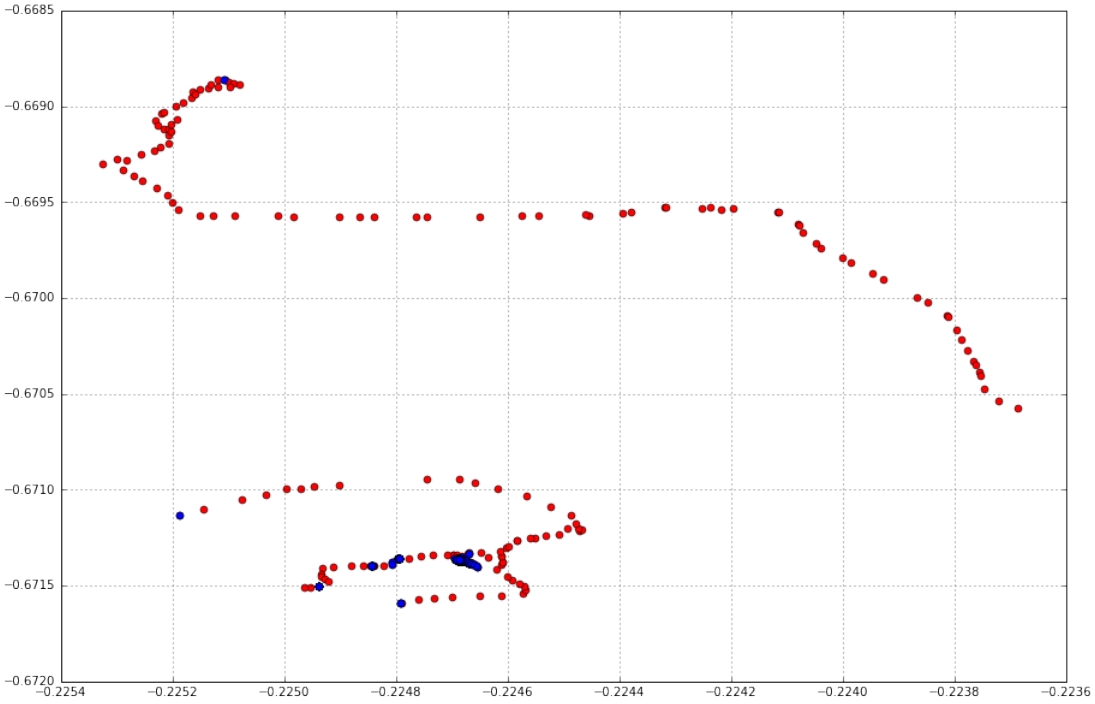


Figura 14: Resultado consolidación por distancia t_0-alcanzable

Esta consolidación, que tiene en cuenta la velocidad instantánea del sujeto, realiza una consolidación más severa en aquellos puntos en los cuales el sujeto posee velocidad, es decir, el vecindario de éstos puntos es superior al vecindario de los puntos donde no posee velocidad. Esta es la razón por la cual la traza superior (en la que el sujeto tiene una mayor velocidad) ha sido consolidada sólo al punto inicial.

Esta consolidación ha mantenido 1786 puntos.

7.2. Resultados con K-means

Se va a realizar un estudio **K-means** para ambos sujetos con los mismos parámetros que previamente habíamos aplicado, vamos a reducir el número de posiciones a 500, es decir, una consolidación al 25 %. En ambos experimentos se ha tomado una distancia euclídea por simplicidad. Se aplicará un preprocesado de *Canopy* y fijaremos el número de clústers a 500:

7.2.1. 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
```

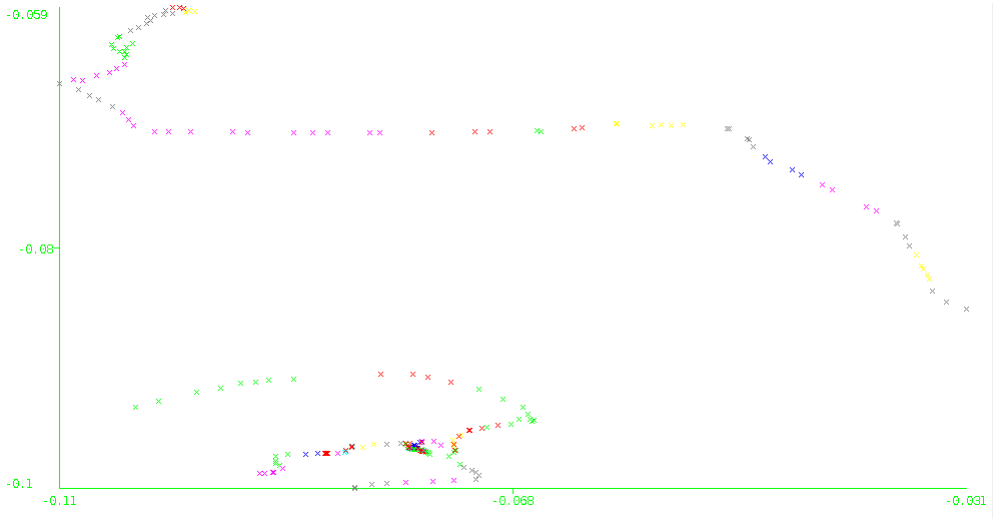


Figura 15: Resultado de **K-means** para el Sujeto 1

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. Debemos recordar que nuestro objetivo no es conseguir una cantidad de clústers muy pequeña, sino conseguir que cada clúster cuente con un número más o menos equilibrado de posiciones, para poder asignar cada posición a su centro del 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.

7.2.2. Sujeto 2

Realizamos el mismo estudio con el Sujeto 2:

```
=== Run information ===

Scheme:       weka.clusterers.SimpleKMeans -init 2 -max-candidates 500
-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: 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
```

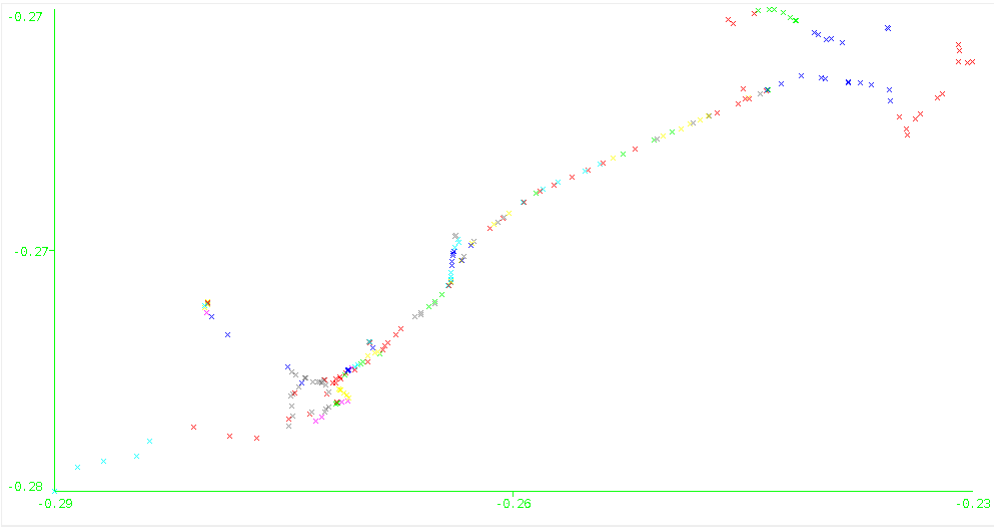


Figura 16: Resultado de **K-means** para el Sujeto 1

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 asingados 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 distantes 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.

7.3. 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:

7.3.1. Sujeto 1

```
cur= connect_db("bahia")
recurso = "tetra:12086044"
limit = 2000
cmd = "SELECT latitud, longitud, UNIX_TIMESTAMP(fecha)
        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], pos[2]])

Data = a
eps = 0.0001
MinPts=5

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

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.

El resultado de DBSCAN ha sido una división de nuestro conjunto inicial de 1000 posiciones en 6 clústers y algunas posiciones etiquetadas como ruido.

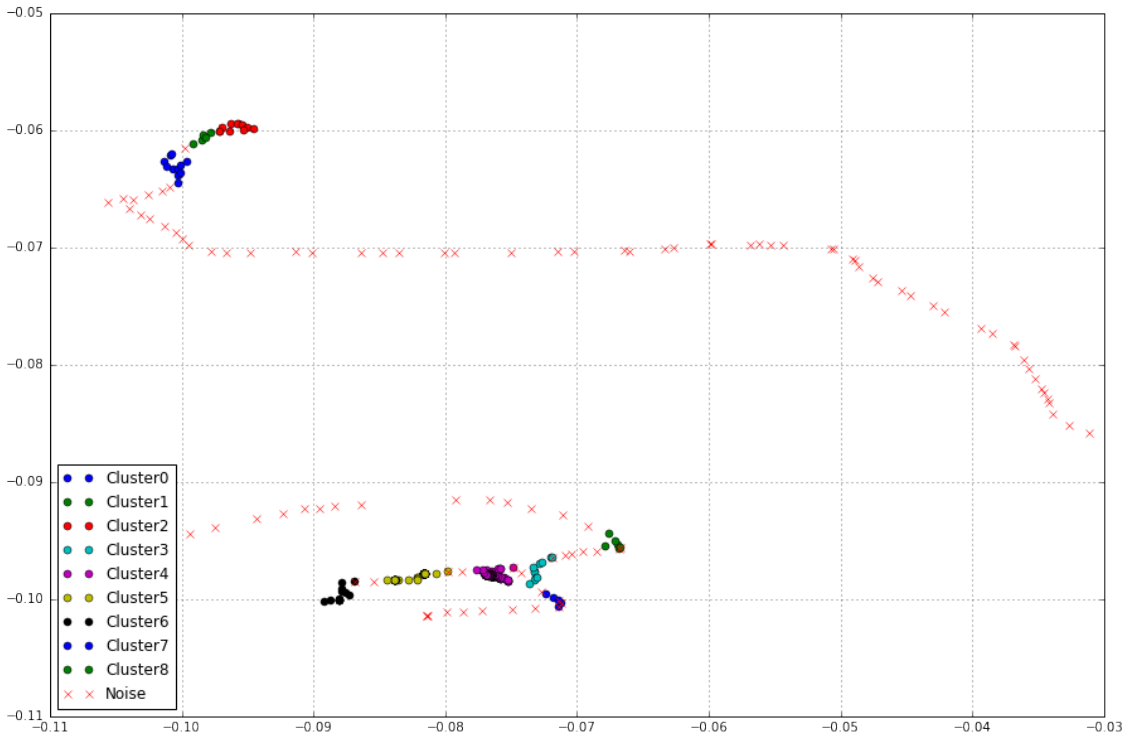


Figura 17: Resultado de **DBSCAN** para el Sujeto 1

7.3.2. Sujeto 2

Utilizaremos los mismos parámetros utilizados para el sujeto 1:

```
cur= connect_db("bahia")
recurso = "tetra:12082781"
limit = 2000
cmd = "SELECT latitud, longitud, UNIX_TIMESTAMP(fecha)
        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], pos[2]])

Data = a
eps = 0.0001
```

MinPts=5

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

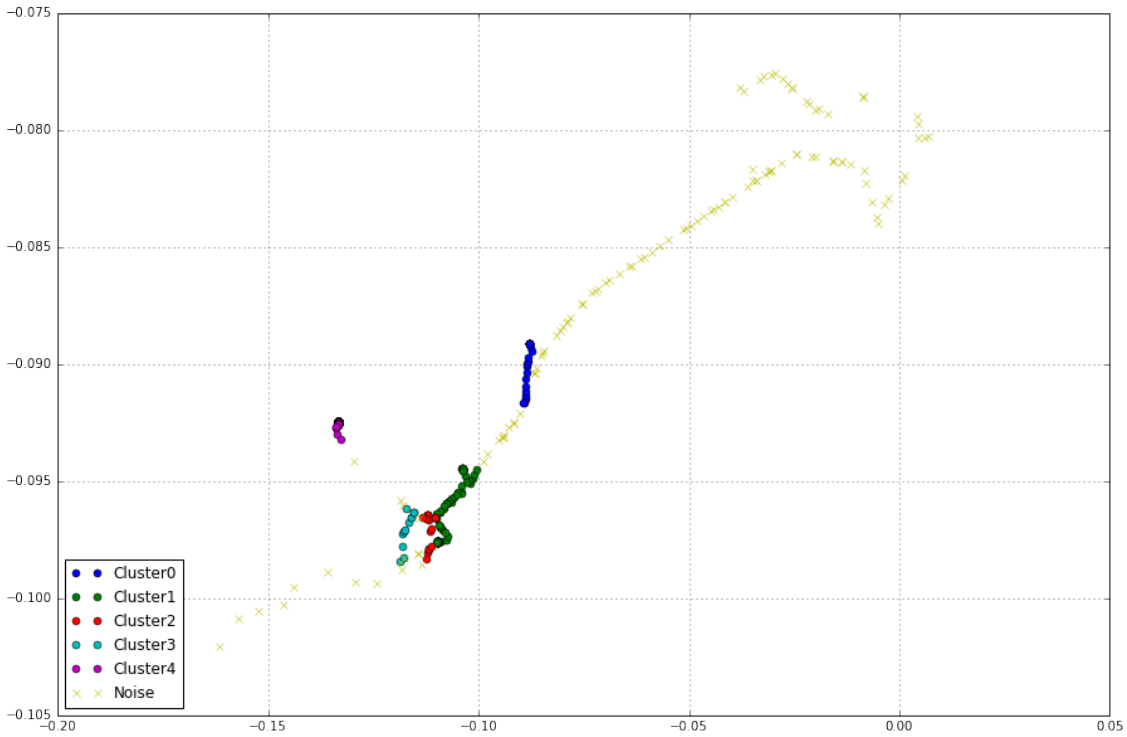


Figura 18: Resultado de **DBSCAN** para el Sujeto 2

Obtenemos 5 clústers en un tiempo de ejecución de 2 minutos, 20 segundos. En esta ejecucción, ha etiquetado muchas posiciones como ruido, probablemente son las posiciones en las cuales el sujeto se encuentra en movimiento. De haber utilizado aquí una consolidación en la cual se hubiera introducido una distancia que involucrara la velocidad, habríamos consolidado en esa zona superior derecha de ruido un nuevo clúster.

Haciendo un pequeño cambio en el código que podemos encontrar de la implementación de DBSCAN en el apéndice B.

```
def regionQuery(self,P,eps):
        P = Position(None, None, P.X, P.Y, P.Speed, None, None)
        for d in self.dataSet:
                d = Position(None, none, d.X, d.Y, d.Speed, None, None)
                if d.is_in_neighborhoodByEURelativeSpeed(P, 0.001):
                        result.append(d)
        return result
```

Así estaríamos utilizando la distancia que hemos implementado que involucra la velocidad.

7.4. Resultados con DJ-Cluster

Realizaremos un estudio **DJ-Cluster** utilizando un preprocesado de datos **Canopy** fijando una desviación mínima estándar de 0,001.

Se fija una distancia euclídea para ambos experimentos.

7.4.1. 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.clusterers.Canopy --
-N -1 -max-candidates 100
-periodic-pruning 10000
-min-density 2.0 -t2 -1.0 -t1 -1.25 -S 1
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
```

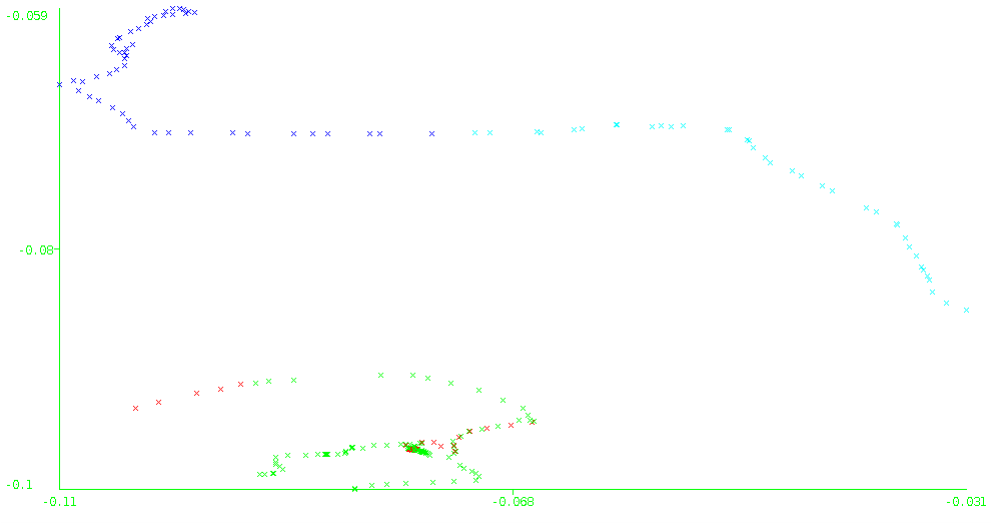


Figura 19: Resultado de **DBSCAN** para el Sujeto 2

Realizaremos un preprocesado *Canopy* de 500 clústers, con el fin de conseguir *tanta* consolidación.

```
=== Run information ===

Scheme:       weka.clusterers.MakeDensityBasedClusterer -M 0.001
-W weka.clusterers.Canopy --
-N 500 -max-candidates 100
-periodic-pruning 10000
-min-density 2.0 -t2 -1.0 -t1 -1.25 -S 1
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
```

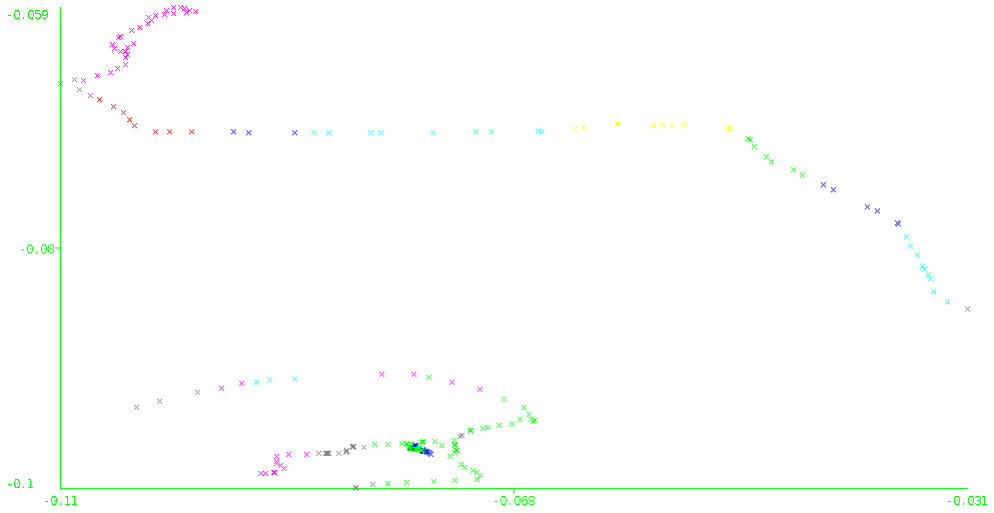


Figura 20: Resultado de **DJ-Cluster** para el Sujeto 1

Este resultado se puede interpretar como algo bastante positivo, ya que hemos reducido considerablemente el tamaño de nuestra base de datos, e incluso podemos etiquetar algunos clústers como ruido, es decir, aquellos que sólo se nos han producido a partir de una o dos posiciones como es el caso. Sería necessario hacer un *post-procesado* de datos para eliminar todas estas posiciones aisladas.

7.4.2. 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.

```
Time taken to build model (full training data) : 1.56 seconds
```

```
=== 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
```

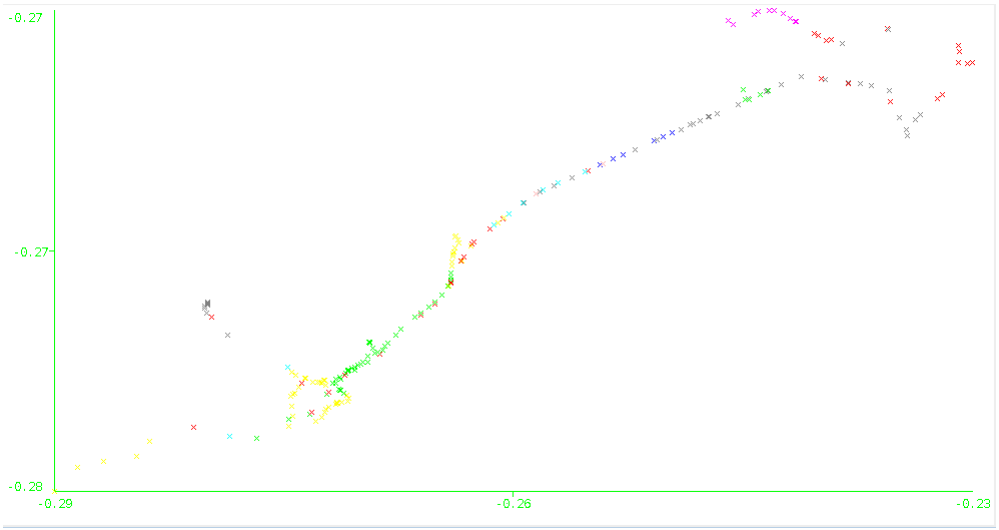


Figura 21: Resultado de **DJ-Cluster** para el Sujeto 2

8. Comparativa de resultados

Algoritmo	SUJETO 1		SUJETO 2	
K-**means**	Canopy N. Clústers Iteraciones Error cuadrático T. de ejecución	Sí 500 9 0.117363 0.69 secs	Canopy N. Clústers Iteraciones Error cuadrático T. de ejecución	Sí 500 24 0.208997 1.38 secs
DBSCAN	Canopy: Iteraciones: T. de ejecución: N. Clústers:	No 111 2 min 30 secs 9	Canopy: Iteraciones: T. de ejecución: N. Clústers:	No 111 2 min 20 secs 9
DJ-Clúster	Canopy: Iteraciones: Clústers: Error cuadrático: Verosimilitud: T. de ejecución	Sí 11 22 0.140929 9.34996 0.37 secs	Canopy: Iteraciones: N. Clústers: Error cuadrático: Verosimilitud: T. de ejecución	Sí 12 27 0.209090 10.83638 1.56 secs
Por adelgazamiento	Canopy: Iteraciones: Clústers: T. de ejecución:	No N. de posiciones 800 <0.01 sec		
Por tiempo	Canopy: Iteraciones: Clústers: T. de ejecución:	No N. de posiciones 507 <0.01 sec		
Por distancia simple	Canopy: Iteraciones: Clústers: T. de ejecución:	No N. de posiciones 21 <0.01 sec		
Por distancia t_0-alcanzable	Canopy: Iteraciones: Clústers: T. de ejecución:	No N. de posiciones 1786 <0.01 sec		

Figura 22: Comparativa entre los disintos métodos

Se puede observar que claramente el algoritmo **DBSCAN** es el más lento de todos, además del que más iteraciones realiza. Sin embargo, éste es el que menos clústers obtiene, es decir, el que realiza una consolidación mayor. Esto es debido a que no lleva un preprocesado *Canopy* previo, no como **K-means** y **DJ-Cluster**. Ambos llevan un preprocesado previo de *Canopy*, el cual les marca un número mínimo de clústers que dejar a la hora de realizarlo, sin embargo, podemos observar que **K-means** se queda en los 500 clústers que le hemos fijado desde el principio, mientras que **DJ-Cluster** consigue rebajar este número incluso más. Con respecto al tiempo de ejecución, **DJ-Cluster** es muchísimo más efectivo, aunque su error cuadrático es peor, comparado con **K-means**.

DBSCAN tiene algunas ventajas con respecto a sus demás compañeros:

- No necesita de prefijar el número de clústers.
- DBSCAN tiene la noción de ruido, así que no es necesario aplicar un post-filtro para encontrar aquellos puntos aislados.

Sin embargo, *DBSCAN* no es un algoritmo determinista, los puntos borde que son alcanzables desde más de un sólo clúster pueden asignarse a cualquiera de éstos. Sin embargo, esta situación no es usual, y en nuestro problema, sólo nos interesan realmente los puntos núcleo y los puntos aislados.

La comparativa de éstos métodos más avanzados con los propios simples definidos en 5 es algo difícil de realizar. Un primer punto sería hacer notar que la complejidad algorítmica de los algoritmos por consolidación simple vistos en 5 es mucho mayor que los algoritmos posteriormente estudiados. Sin embargo, la principal ventaja del uso de éstos sobre los avanzados sería la posibilidad de una utilización de otra definición de distancia, cosa que utilizando alguna implementación ya realizada en Weka es imposible.

9. Conclusiones y cuestiones abiertas

Entre los resultados de este texto, se encuentran dos tipos de algoritmos analizados e implementados. En primer lugar, 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 éstas. 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 mayor tanto computacional como de implementación que los anteriores. En un futuro, esto queda a criterio de la persona que desarrollara estos algoritmos directamente en la aplicación a utilizar.

Entre estos métodos de .ªlto nivel"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 comparaciń a la complejidad computacional 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 las implementaciones de Weka de **K-means** y **DJ-Cluster** no realizan, así que sería un punto a favor para utilizar **DBSCAN**.

A la hora de realizar una reconstrucción de la traza de movimiento a partir de los datos centralizados, es reseñable el uso del algoritmo **DJ-Cluster** ya que el algoritmo **K-means** no asegura encontrar los mejores centroides de los clústers. El resultado depende de la elección inicial de los centros de los clústers, la cual es al azar en el caso de **K-means**. Sería a lo mejor reco-

mendable lanzar el algoritmo varias veces con el fin de minimizar el error en cada una y elegir la que mejor se ajustara, pero en este caso es mejor utilizar un **DJ-Cluster**.

La definición de un nuevo concepto de distancia ha sido realizada para los algoritmos de consolidación simple, sin embargo, la realización de pruebas sobre los demás métodos ha sido realizada con el software Weka, el cual no permite el uso de distancias personalizable. El número de iteraciones en este caso es lineal, es decir, es el número de posiciones iniciales que mandamos consolidar.

Quedaría como trabajo a futuro la inclusión de la orientación en los algoritmos de **K-means**, **DJ-Cluster** y **DBSCAN** como un elemento más a la hora de definir las nociones de vecindario.

10. 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.[2]

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.[6]

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. Implementación de algoritmos de consolidación simple

Código 1: 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
```

Código 2: 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
```

Código 3: 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_neighboorhoudByTime(listPositions[i+1], lapse):
                        result.append(listPositions[i])
                i=i+1
        result.append(listPositions[len(listPositions) - 1])

        return result
```

B. Implementación de algoritmos de consolidación asociados a métodos de clustering

Código 4: Preprocesado 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
```

Código 5: 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) # etiquetamos el punto como ruido
                else:
                        result = np.isDensityJoinable(listClusters)

                        if result is None:
                                listClusters.append(np) # creamos un nuevo cluster
                        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_neighborhoodEURelativeSpeed(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)
```

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

        "Cluster is density Joinable with list of clusters?"
        def isDensityJoinable(self, listClusters):
                for cluster in listClusters:
                        for p in self.points:
                                if p.isInCluster(cluster):
                                        return cluster

                return None

        "Merge current cluster with another"
        def mergeCluster(self, cluster) :
                for p in cluster.points:
                        if not p.isInCluster(cluster):
                                self.points.append(p)

                return self
```

Código 6: DBSCAN

```
'''
Created on Feb 13, 2014
@author: sushant
'''
from cluster import *
from pylab import *

class dbscanner:

        dataSet = []
        count = 0
        visited = []
        member = []
        Clusters = []

        def dbscan(self,D,eps,MinPts):
                self.dataSet = D

                title(r'DBSCAN Algorithm', fontsize=18)
                xlabel(r'Dim 1',fontsize=17)
                ylabel(r'Dim 2', fontsize=17)
                plt.figure(figsize=(15,10))

                C = -1
                Noise = cluster('Noise')
                numit = 0
                for point in D:
                        if point not in self.visited:
                                self.visited.append(point)
                                NeighbourPoints = self.regionQuery(point,eps)

                                if len(NeighbourPoints) < MinPts:
                                        Noise.addPoint(point)
                                else:
                                        name = 'Cluster'+str(self.count)
                                        C = cluster(name)
                                        self.count+=1
                                        self.expandCluster(point,NeighbourPoints,C,eps,MinPts)

                                        plot(C.getX(),C.getY(),'o',label=name)
                                        hold(True)
```

```
        if len(Noise.getPoints())!=0:
                plot(Noise.getX(),Noise.getY(),'x',label='Noise')

        hold(False)
        legend(loc='lower left')
        grid(True)
        show()

def expandCluster(self,point,NeighbourPoints,C,eps,MinPts):

        C.addPoint(point)

        for p in NeighbourPoints:
                if p not in self.visited:
                        self.visited.append(p)
                        np = self.regionQuery(p,eps)
                        if len(np) >= MinPts:
                                for n in np:
                                        if n not in NeighbourPoints:
                                                NeighbourPoints.append(n)

                for c in self.Clusters:
                        if not c.has(p):
                                if not C.has(p):
                                        C.addPoint(p)

                if len(self.Clusters) == 0:
                        if not C.has(p):
                                C.addPoint(p)

        self.Clusters.append(C)


def regionQuery(self,P,eps):
        result = []
        for d in self.dataSet:
                if (((d[0]-P[0])**2 + (d[1] - P[1])**2)**0.5)<=eps:
                        result.append(d)
        return result
```

Índice de figuras

Índice de algoritmos

Referencias

[1] Terveen L. Bhatnagar N, Shekhar S. and Zhou C. Mining personally important places from gps tracks. In *Istanbul, Data Engineering Workshop, 2007 IEEE 23rd International Conference*, 2007.

[2] Frank E. and Witten I.H. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.

[3] Terveen L. Frankowski D., Ludford P. Shekhar S. and Zhou C. Discovering personal gazetteers: An interactive clustering approach. In *In Proc. ACMGIS*, pages 266–273. ACM Press, 2004.

[4] Gabe. Efficient python implementation of canopy clustering. https://gist.github.com/gdbassett/528d816d035f2deaaca1, 2014.

[5] Nigam K. McCallum A. and Ungar L. H. Efficient clustering of high-dimensional data sets with application to reference matching, 2000.

[6] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008.

[7] Kafle S. Implementation of dbscan algorithm in python. https://github.com/SushantKafle/DBSCAN, 2014.