TFM

Pilar Barbero Iriarte

November 25, 2015

Contents

1 Planteamiento del problema

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	timestamp	NO	MUL	CURRENT_TIMESTAMP
automático	tiniyint(10) unsigned	NO	MUL	0

Para este estudio se ha trabajado sólo con los siguientes datos,

1. Id
2. Recurso
3. Latitud
4. Longitud
5. Velocidad
6. Fecha

2.1 Análisis de los datos

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, conseguri°iamos 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 `Posición` 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 ϵ con respecto su centro p_0, es decir,

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

donde p es un punto con latitud lat_p y longitud $long_p$.
Su implementación en Python es la siguiente,

```
def is_in_neighborhoodByEUSimple(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 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:

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

donde vel_{p_0} es la velocidad instantánea de nuestro punto centro.
Su implementación en Python es la siguiente,

```
def is_in_neighborhoodT0Reachable(self, q, eps):
        return self.distance_eu(q) < eps * self.speed
```

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 reducido que otro que se desplace a una velo$vel_{p_0} \cdot t_0$.

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

Éste es un caso concreto del vecindario involucrando la velocidad.
Su implementación en Python es la siguiente,

```
def is_in_neighborhoodT0Reachable(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_neighboorhoudByTime(self, q, lapse):
        foo = time.mktime(self.date.timetuple())
        bar = time.mktime(q.date.timetuple())
        return abs(foo - bar) < lapse
```

3.5 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.

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 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 insercción 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.

4.1 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.IsInNeighBorhoodForEURelativeSpeed(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.IsInNeighBorhoodForT0Reachablee(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
```

4.2 Consolidación cada cierto número de posiciones

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.

Algorithm 2 Algoritmo de consolidación cada cierto número

```
1: function ConsolidationByEachJinK(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
```

5 Algoritmos de consolidación asociados a métodos de clustering

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 *clúser*.

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
```

5.1 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 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.

5.2 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 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 quizs 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.

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.

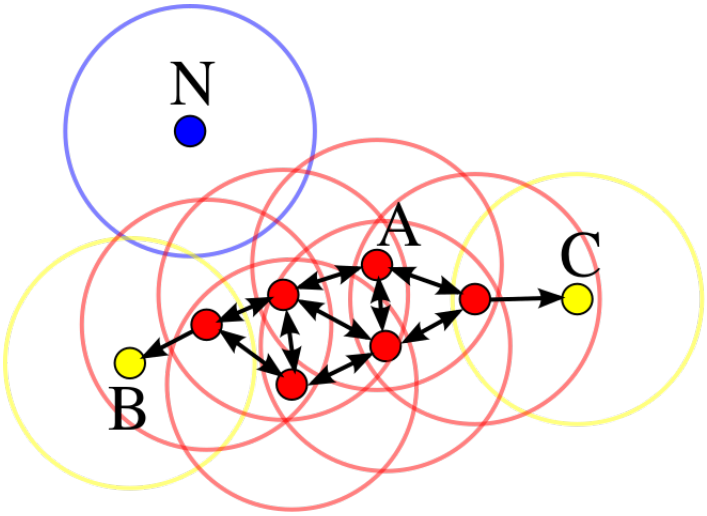


Figure 1: Diagrama DBSCAN

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.

Algorithm 3 Algoritmo DBSCAN

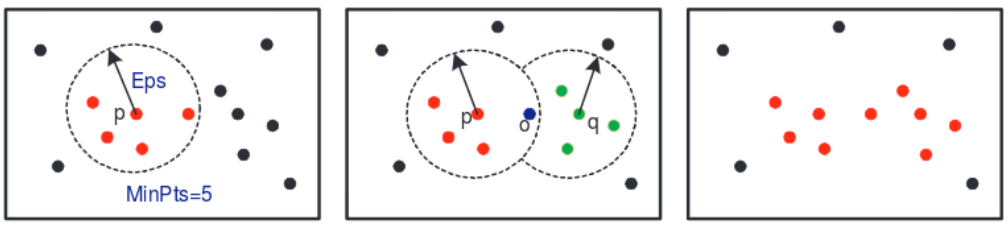
```
1: function DBSCAN(positions, eps, minPts)
2:     C = 0
3:     for each pos in positions do
4:         if pos has been visited then
5:             Continue next position
6:         else
7:             Mark pos as visited
8:             N(pos) = NeighborPts(pos, eps)
9:             if length(N(pos)) < 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
```

5.3 DJ-Cluster

Density-Joinable Clúster2 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 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) $DJ - Cluster$

Figure 2: DJ-Clustering

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.

Algorithm 4 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
```

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 en función de la precisión de los aparatos que toman las posiciones.2. 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 estaría entre 3 y 10.3.

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

5.3.1 Comparación de K-Means con DJ-Cluster

5.4 Canocopy

6 Conclusiones

7 Herramientas utilizadas

A Impementación de consolidación por distancia

```
"Consolidation By distance"
def ConsolidationByDistance(listPositions, typeOfDistance, eps, t0):
        print "Starting consolidation by: {0}".format(str(typeOfDistance))
        i = 0
        result = []
    while i < len(listPositions) - 1:
        # Neighborhood: Distance EU simple
        if typeOfDistance == 0:
                if not listPositions[i].is_in_neighborhoodByEUSimple(listPositions[i+1], eps):
                        result.append(listPositions[i])
                # Neighborhood: Distance EU relative to speed
                elif typeOfDistance == 1:
                if not listPositions[i].is_in_neighborhoodByEURelativeSpeed(listPositions[i+1], eps):
                        result.append(listPositions[i])
                # Neighborhood t0 reachable
        elif typeOfDistance == 2:
                        if not listPositions[i].is_in_neighborhoodT0Reachable(listPositions[i+1], t0):
                        result.append(listPositions[i])
                else:
                raise ValueError('That distance does not exist')
        i=i+1

        #Por defecto anadiremos la ultima posicion, ya que no tiene siguiente con quien comparar
        result.append(listPositions[len(listPositions) - 1])

        return result
```

B Impementación de consolidación por tiempo

```
"Deletes one position every k positions."
def ConsolidationEachNumber(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
```

References

[1] Mining Individual Life Pattern
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.361.9085&rep=rep1&type=pdf

[2] Mining Personally Important Places from GPS Track
http://www-users.cs.umn.edu/~czhou/pub/place-important_v3.pdf

[3] Discovering Personal Gazetteers: An Interactive Clustering Approach
http://files.grouplens.org/papers/zhou-acmgis04.pdf