UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERÍA INFORMÁTICA DE SEGOVIA

Grado en Ingeniería Informática de Servicios y Aplicaciones

TRACLUS: Clustering de trayectorias

Mario Vázquez Onrubia

Dirigido por:
Anibal Bregón Bregón
Miguel Angel Martínez Prieto

"Son nuestras decisiones las que muestran lo que podemos llegar a ser. Mucho más que nuestras propias habilidades."

Joanne Rowling

"*De todos los animales de la creación, el hombre es el único que bebe sin tener sed, come sin tener hambre y habla sin tener nada que decir.*"

John Earnest Steinbeck

Agradecimientos

Mi más sincero agradecimiento a mis tutores, Anibal y Miguel Ángel por la ayuda y por la presencia durante este tiempo, pero sobre todo, por brindarme la oportunidad de realizar este proyecto; a toda la gente que me ha acompañado durante este tiempo y sobre todo a mi familia, por estar ahí en los momentos buenos y en los no tan buenos, con el "ánimo" y el "tú puedes" constante.

Resumen

TRACLUS: Clustering de trayectorias surge a partir de la investigación de técnicas de clustering, con las diferentes particularidades que tiene *TRACLUS* respecto a otros algoritmos de clustering que son aplicados a trayectorias. *TRACLUS* ha sido implementado utilizando tecnologías y herramientas que están actualmente en auge en el campo de la minería y la ciencia de datos, aplicando una optimización posterior a la implementación y obteniendo como cómputo global del proyecto los mismos resultados de implementación que sin optimizar, pero en un tiempo de computación mucho menor.

Abstract

TRACLUS: Clustering de trayectorias arises from research in some clustering techniques, with the *TRACLUS* differences over other trajectory clustering algorithm. *TRACLUS* has been deployed using technologies and tools which are rising in data mining anda data science domain, applying a optimization after implementation, getting same results than the implementation without optimization but much lower computational time.

Índice general

1. Introducción **1**
- 1.1. Motivación . . . 1
- 1.2. Objetivos . . . 2
- 1.3. Estructuración de la memoria . . . 3
- 1.4. Contenido del CD . . . 3

2. Clustering de trayectorias **5**
- 2.1. K-means clustering . . . 6
- 2.2. Fuzzy c-means clustering . . . 8
- 2.3. Traclus . . . 9
  - 2.3.1. Introducción a Traclus . . . 10
  - 2.3.2. Clustering de trayectorias con *TRACLUS* . . . 12
  - 2.3.3. Particionado de trayectorias . . . 15
  - 2.3.4. Clustering de los segmentos . . . 18

3. Gestión del proyecto **25**
- 3.1. Planificación temporal . . . 25
- 3.2. Presupuestos . . . 26
- 3.3. Costes temporales . . . 27
- 3.4. Costes económicos . . . 28

4. R y su entorno **31**
- 4.1. ¿Qué es R? . . . 31
  - 4.1.1. Historia de R . . . 31
  - 4.1.2. Características de R . . . 32
  - 4.1.3. Objetos en R . . . 34
  - 4.1.4. ¿Cómo trabaja R? . . . 34
- 4.2. Paquetes en R . . . 36
  - 4.2.1. ¿Qué son los paquetes de R? . . . 36
  - 4.2.2. Repositorios . . . 37
  - 4.2.3. Estructura de un paquete . . . 37
- 4.3. ¿Qué es Shiny? . . . 38
  - 4.3.1. Arquitectura de Shiny . . . 39
  - 4.3.2. Componentes en Shiny . . . 40

5. Implementación **43**
- 5.1. RStudio . . . 43
  - 5.1.1. ¿Qué es RStudio? . . . 43
  - 5.1.2. Interfaz de RStudio . . . 44

5.2. Traclus en R . . . 45
5.2.1. Particionado de trayectorias en R . . . 45
5.2.2. Agrupamiento de trayectorias en R . . . 46
5.2.3. Obtención de la trayectoria representativa en R . . . 47
5.3. Creación de un paquete con R y C . . . 47
5.3.1. Observaciones . . . 47
5.3.2. Particionado en C . . . 48
5.3.3. Agrupamiento en C . . . 49
5.3.4. Estructura final del paquete . . . 51

6. Manual del paquete para R **53**
6.1. Creación, instalación y uso . . . 53
6.2. Integración del paquete en una aplicación Shiny . . . 56
6.2.1. Arquitectura de la aplicación . . . 56
6.2.2. Parte de interfaz de usuario . . . 57
6.2.3. Parte servidor . . . 60

7. Benchmarking **65**
7.1. Resultados obtenidos . . . 65
7.1.1. Datos escalados . . . 66
7.1.2. Datos sin escalar . . . 66
7.2. Tiempos de ejecución . . . 69

8. Conclusiones y Trabajo Futuro **71**
8.1. Conclusiones . . . 72
8.2. Trabajo futuro . . . 72
8.3. Aprendizajes . . . 72

Índice de figuras

2.1. Ejemplo de clustering 5
2.2. Conjunto de trayectorias 7
2.3. Clusters obtenidos para diferentes valores de k y $nstart$ 7
2.4. Resultados para diferentes valores de c 9
2.5. Ejemplo de Traclus 10
2.6. Sub-trayectoria común 10
2.7. Ejemplo de aplicación del método DBSCAN 11
2.8. Ejemplo de particionado y agrupamiento 13
2.9. Componentes de la distancia entre dos segmentos L_i y L_j 14
2.10. Particionado de una trayectoria y sus particiones 15
2.11. Ejemplo de aplicación del principio MDL al particionado de trayectorias 16
2.12. Ejemplo de aplicación del algoritmo de particionado aproximado 18
2.13. Segmentos conectados en densidad y/o alcanzables en densidad 19
2.14. Influencia de un segmento muy corto al hacer clustering 20
2.15. Ejemplo de un cluster y su trayectoria representativa 21
2.16. Rotación de los ejes X e Y 23

3.1. Diagrama de Gantt para la planificación temporal 26
3.2. Diagrama de Gantt para los costes temporales 28

4.1. Ejecución de una función en R 35
4.2. Esquema de funcionamiento de R 35
4.3. Ejemplo de un fichero `DESCRIPTION` de un paquete en R 38
4.4. Ejemplo de estructura de un paquete 38
4.5. Aproximación de arquitectura de una aplicación Shiny 39
4.6. Algunos componentes de entrada para Shiny 40
4.7. Ejemplo de aplicación en Shiny 40
4.8. Ejemplo de aplicación en Shiny 41
4.9. Ejemplo de aplicación en Shiny 41

5.1. Interfaz de RStudio 44

6.1. Ejemplo de particionado de una trayectoria 55
6.2. Ejemplo de clustering de los segmentos 56
6.3. Arquitectura final de la implementación 57
6.4. Elementos de entrada para seleccionar aeropuertos 57
6.5. Elementos de entrada para seleccionar parámetros 58
6.6. Elementos de entrada para establecer gráficas 58
6.7. Elementos de visualización de segmentos por cluster 58
6.8. Elementos de entrada del tipo de mapa 59

6.9. Botón de descarga de la gráfica 59
6.10. Gráfica principal de la aplicación 59
6.11. Barra de progreso para la carga de datos de los aeropuertos 60
6.12. Ejemplo de ejecución de la aplicación 61
6.13. Ejemplo de ejecución de la aplicación 62
6.14. Ejemplo de ejecución de la aplicación 63

7.1. Clusters y subtrayectorias para la implementación de R escalando los datos 67
7.2. Clusters y subtrayectorias para la implementación de C escalando los datos 67
7.3. Clusters y subtrayectorias para la implementación de R sin escalar los datos 68
7.4. Clusters y subtrayectorias para la implementación de C sin escalar los datos 68
7.5. Ejemplo de comparación de tiempos para C y R 70

Índice de algoritmos

1. **TRACLUS** (TRAjectory CLUStering) 13
2. Particionado de trayectorias aproximado 17
3. Clustering de los segmentos particionados 22
4. Obtención de la trayectoria representativa 23

Capítulo 1

Introducción

Actualmente, cuando se habla de generación de datos seguramente todo el mundo piense en internet. Según un estudio realizado en 2016 por Excelacom [9], en un minuto se envían 20,8 millones de mensajes por WhatsApp, se alcanzan las 2,4 millones de búsquedas en Google, se generan 347222 nuevos Tweets y se realizan 701389 logins en Facebook, lo que supone una gran cantidad de datos a almacenar. Sin embargo, esta no es la única gran fuente de datos.

Los dispositivos de seguimiento y satélites están en continuo desarrollo, lo que permite recolectar millones de datos de objetos en movimientos, con diferentes fines. Por ejemplo, se puede determinar la posición de un vehículo con dispositivos "in situ", con bucles magnéticos instalados en las carreteras (*magnetic loops*), formando un campo magnético y mandando los datos a un dispositivo que cuenta los vehículos que pasan por ese campo magnético, o el envío de datos por GPS, enviando ambos tipos de dispositivos datos útiles para el desarrollo de Sistemas Inteligentes de Transporte, (*ITS - Intelligent Transportation Systems*). Otro tipo de seguimiento que está evolucionando es el de los huracanes, que permitirá determinar qué zonas serán afectadas, así como las que ya lo han sido, recogiendo todos estos datos desde diferentes satélites. Así mismo, se puede recoger el movimiento de animales. Por otra parte, y según Virgin Atlantic [10] un Boeing 787s crea medio terabyte de datos por cada vuelo.

Una vez se tienen estas recolecciones, la tarea a llevar a cabo es aportar valor a los datos. Para ello, se han de agregar y procesar dichos datos, de manera que se puedan generar predicciones que lleven a facilitar la toma de decisiones. Por ejemplo, en el caso del Boeing 787s, se podrían combinar los datos provenientes de los sensores del avión durante el vuelo con los datos disponibles de las reparaciones de las incidencias sucedidas, o bien de las tareas de mantenimiento o inspecciones, encontrando patrones que, dadas unas condiciones, han llevado a una situación de fallo, prediciendo los mismos antes de que ocurran.

1.1. Motivación

Como ya se ha mencionado, existen diferentes recolecciones de datos recogidos con diferentes fines, pero, en este caso, se va a trabajar con datos, concretamente, trayectorias. Este tipo de recolección de datos está aumentando de cara a analizar las diferentes recorridos que han realizado los objetos, siendo un análisis típico la obtención de similitudes entre objetos que han realizado trayectorias parecidas. Por lo tanto, la manera elegida de aportar valor a las grandes cantidades de datos de trayectorias, en este caso ha sido el clustering, para lo que se deberá tener un algoritmo que agregue, manipule y transforme los datos con el fin de obtener unas conclusiones finales.

Siguiendo con el ejemplo de los vuelos de aviones, estos están continuamente enviando datos a las diferentes torres de control. En ese envío se incluyen datos de ubicación y de tiempo, es decir, el punto exacto en longitud y latitud en el que se encuentra el avión en el momento del envío, y en qué momento se produce. Con la unión de estos puntos, se construirá la trayectoria que ha realizado.

Para un día dado, se producen numerosos vuelos entre un origen y un destino en concreto, siendo estos, tanto de diferentes compañías aéreas, como de diferentes aeronaves. La probabilidad de que dos de estos vuelos realizados entre un origen y un destino coincida es ínfima, por lo que, a partir de los datos de las trayectorias, se pueden plantear similitudes y diferencias, viendo así que zonas del espacio aéreo tienen más densidad de trayectorias y en qué zonas se dispersan más.

Combinando los resultados obtenidos con otro tipo de datos, como podrían ser datos meteorológicos, se podrían sacar conclusiones de zonas problemáticas en el espacio aéreo debido a que en ese momento del día hay una alta probabilidad de que en una zona en concreto las condiciones climatológicas no sean las adecuadas para atravesarla. Además, a partir de los patrones obtenidos, se podrían sacar trayectorias alternativas óptimas (por ejemplo, cuando el consumo de combustible haya sido menor) que se han realizado cuando se daban condiciones adversas.

La obtención de *sub*-trayectorias comunes a todas las trayectorias presenta bastantes utilidades, sobre todo si se tiene un especial interés en analizar un espacio en concreto por el que han pasado algunas trayectorias. Para ello, habría que poner especial atención al comportamiento de las trayectorias en una región precisa para obtener similitudes.

Como se mencionaba antes, con la ayuda de satélites se puede obtener el comportamiento de huracanes y las trayectorias que estos dibujan. Aprovechando estos datos, los meteorólogos podrán mejorar sus habilidades de predecir dónde y cuándo un huracán alcanza la tierra, de manera que sabiendo esto se reducirán los daños que pueda causar. Por ello, los meteorólogos estarán interesados en zonas de costa o cercanas a ella cuando un huracán haya alcanzado la tierra, o en las zonas de mar cuando un huracán no haya llegado a la tierra. Por lo tanto, esto se traduce en que los meteorólogos estarán interesados en las *sub*-trayectorias formadas por los huracanes con el fin de mejorar la precisión de sus pronósticos.

Para obtener los resultados esperados se deberá utilizar un algoritmo de clustering que sea aplicable a trayectorias, como podría ser k-means o c-means [15], DBSCAN [8] o *TRACLUS* [14], que a diferencia de los anteriores, proporcionará una subtrayectoria común a todas las trayectorias.

1.2. Objetivos

El objetivo principal del proyecto es obtener una implementación eficiente del algoritmo *TRACLUS* en lenguaje R, de manera que se pueda utilizar con grandes conjuntos de datos. La consecución de este objetivo permitirá:

1. Obtener clusters o grupos de trayectorias cuando presenten similitudes, pudiendo ver así dónde se concentran y se dispersan en mayor o menor número

2. Obtener una subtrayectoria común al conjunto de trayectorias al cual se le aplique el algoritmo, observando el patrón que están siguiendo las trayectorias, en este caso, entre dos aeropuertos.

1.3. Estructuración de la memoria

En esta sección se va a describir la estructura del documento actual de modo que sirva de ayuda al lector. Se va a dividir en los siguientes capítulos:

- **Capítulo 1. Introducción.** En este capítulo se va a presentar el proyecto, la motivación para llevarlo a cabo y los objetivos que se persiguen con él.
- **Capítulo 2. Clustering de trayectorias.** En este capítulo se van a introducir diferentes algoritmos de clustering de trayectorias, centrando la explicación en TRACLUS.
- **Capítulo 3. Gestión del proyecto.** Este capítulo aborda las planificaciones de tiempo y presupuesto, así como los costes temporales y económicos finales.
- **Capítulo 4. R y su entorno.**. En este capítulo se va a presentar el lenguaje de programación R junto a algunas de las tecnologías que le rodean.
- **Capítulo 5. Implementación.**. En esta parte del documento se va a describir la implementación realizada del algoritmo en R y del paquete para R obtenido.
- **Capítulo 6. Manual del paquete para R.** En esta parte del documento se va a explicar como crear, instalar y usar el paquete creado en el anterior capítulo, así como se va a indicar cómo integrarlo en una aplicación web.
- **Capítulo 7. Benchmarking.** En este apartado se va a realizar una comparación de las diferentes implementaciones realizadas.
- **Capítulo 8. Conclusiones y Trabajo Futuro.** En este capítulo se va a realizar una valoración del trabajo realizado, así como las posibles mejoras e introducciones al proyecto.

1.4. Contenido del CD

Junto al actual documento se entregará un CD-ROM, el cual tendrá los siguiente directorios:

- **Implementaciones**, en el que estarán las diferentes implementaciones del algoritmo, y se dividirá en diferentes directorios:
  - **R**, con el código fuente del algoritmo en lenguaje R
  - **C**, en el que estará el código fuente del paquete para R en C del algoritmo
  - **Aplicación Shiny**, con el código de la aplicación en Shiny.
- **Memoria**, con una copia en formato PDF del actual documento.

Capítulo 2

Clustering de trayectorias

Agrupar el contenido de un conjunto de datos es una tarea realmente útil para clasificar y realizar un etiquetado de los datos según ciertos parámetros similares entre ellos. Esta tarea se conoce como *clustering*, la cual da lugar a grupos o *clusters*, formados por objetos similares. Para generar dichos grupos, existen algoritmos de *clustering*, que siguen un procedimiento de agrupación de los datos según un criterio. En la figura 2.1 se puede ver el resultado que se obtendría al realizar clustering de un conjunto de puntos, obteniendo tres grupos coloreados en diferentes colores.

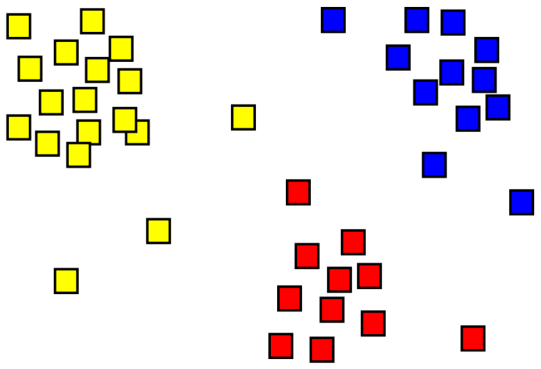


Figura 2.1: Ejemplo de clustering

Para la obtención de estos grupos, se podría seguir como criterio determinante, para formar parte de un cluster u otro, la distancia euclídea entre ellos. Por lo tanto el cálculo de los grupos, teniendo dicho criterio, se presenta relativamente fácil. Cabe plantearse la posibilidad de realizar clustering de otro conjunto de objetos diferentes a los puntos, como podrían ser las trayectorias. Agrupar trayectorias permitirá obtener patrones de movimiento, siendo los clusters resultantes clave para el análisis de las trayectorias. Para obtener los diferentes clusters, y como se mencionaba antes, se ejecuta un algoritmo de clustering. En los próximos apartados se explicarán diferentes algoritmos de clustering: en los dos primeros apartados del actual capítulo se explicarán algoritmos no especializados en trayectorias y se aplicarán a ellas, y en el tercer apartado se explicará de manera más detallada un algoritmo de clustering para trayectorias, el cual será el grueso del capítulo.

2.1. K-means clustering

K-means clustering, también conocido como *algoritmo de Lloyd* [15], dice que dado un conjunto de n observaciones $\boldsymbol{X} = \{x_1, x_2, \ldots, x_n\}$, se agruparán los datos en k grupos ($k \leq n$), obteniendo el conjunto $\boldsymbol{S} = \{S_1, S_2, \ldots, S_k\}$, con el objetivo de minimizar la suma de los cuadrados dentro de cada grupo de manera que:

$$argmins \sum_{i=1}^{k} \sum_{x_j \in S_i} \|x_j - \mu_i\|^2 \tag{2.1}$$

Obteniendo con *argmins* el conjunto de valores para los cuales el sumatorio de la ecuación 2.1 toma su valor mínimo, se establecen inicialmente un conjunto inicial de k centroides $\boldsymbol{(M)}^{(1)} = (m_1^{(1)}, m_2^{(1)}, \ldots, m_k^{(1)})$ de manera aleatoria sobre los datos y se continúa iterando sobre las dos siguientes fases:

1. **Fase de asignación**, en la que se asigna cada observación x_i a un grupo perteneciente a $\boldsymbol{S}$ cuyo valor menos la media sea la mínima de entre todos los k grupos de $\boldsymbol{S}$. Puesto que la suma de los cuadrados es el cuadrado de la distancia euclídea, esta será la media más cercana, por lo que cada uno de los k grupos se formará como sigue:

$$S_i^{(t)} = x_p : \|x_p - m_i^{(t)}\|^{2\leq} \|x_p - m_j^{(t)}\|^2 \forall j, 1 \leq j \leq k \tag{2.2}$$

2. **Fase de actualización**, en la que se calculan las medias que serán los nuevos centroides de las observaciones en los nuevos clusters:

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j \tag{2.3}$$

El algoritmo converge cuando se realiza una asignación y esta es igual a la anterior, es decir, cuando las asignaciones dejan de ser diferentes entre ellas. Normalmente, las asignaciones se realizan según la menor suma de los cuadrados, lo que se corresponde exactamente con la menor distancia euclídea. A continuación se va a aplicar el algoritmo al conjunto de trayectorias que se puede observar en la figura 2.2 considerando x_i una trayectoria. Para esto, se ha dado valor al parámetro *nstart*, el cual nos permite realizar las configuraciones iniciales que le indiquemos para que comience a iterar sobre la más precisa de todas ellas. Por ejemplo, si establecemos un valor de $nstart = 25$, k-means realizará 25 configuraciones iniciales e iterará sobre la que haya sido más precisa de todas ellas, entendiendo la más precisa como la que cumpla en mayores ocasiones la ecuación (2.1).

A continuación, se mostrarán varios ejemplos para los cuales la elección del parámetro k se ha realizado de manera arbitraria, al igual que el parámetro *nstar*t. El valor de k se ha establecido visualizando los datos y deduciendo los posibles clusters. La elección correcta de k es a menudo ambigua, dependiendo de la forma y la escala del conjunto de observaciones, así como del nivel de precisión deseado por el usuario. Aumentando el valor de k se reduce el número de errores en cuanto a que una observación forme parte de un cluster, hasta llegar al nivel extremo de que no existan errores, en el que cada trayectoria sea considerada como su propio cluster. Por lo que, el valor óptimo de k se encontrará en el balance entre la máxima compresión de los datos, en la que se considerarán todos los datos como un único cluster, y entre la máxima precisión, en la que se considerará cada trayectoria un cluster.

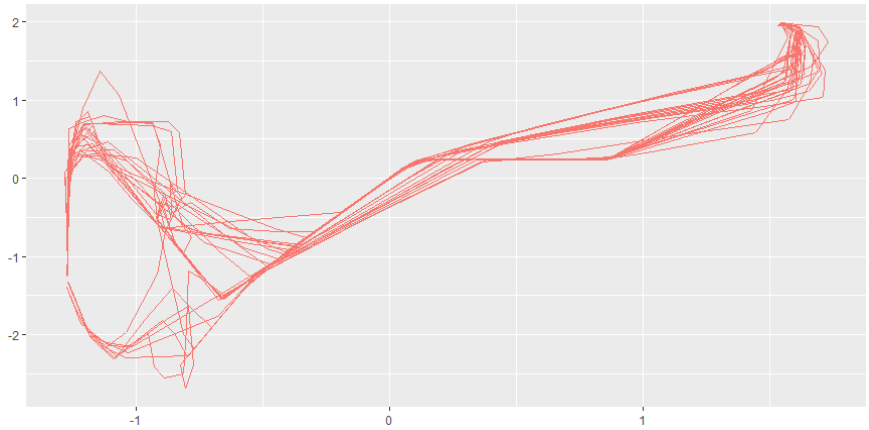
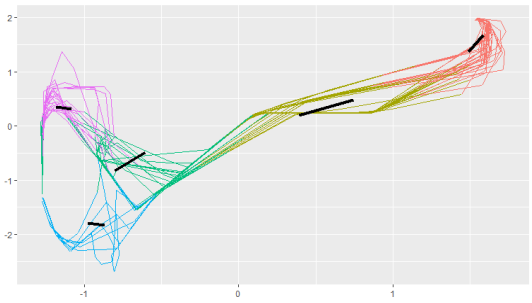
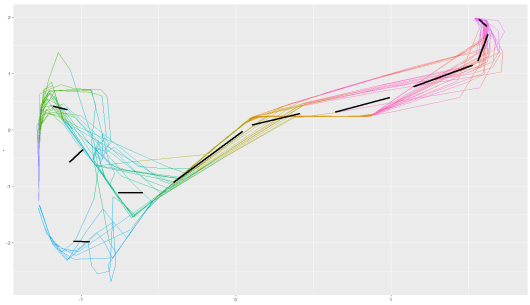


Figura 2.2: Conjunto de trayectorias

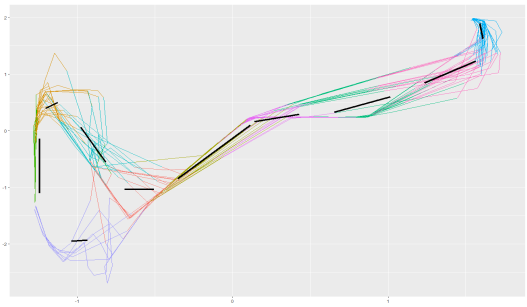
En la figura 2.3 se pueden observar los diferentes clusters y centroides obtenidos para diferentes valores de k y $nstart$, diferenciando cada k cluster por colores.



(a) $k = 5, nstart = 20$



(b) $k = 10, nstart = 1$



(c) $k = 10, nstart = 20$

Figura 2.3: Clusters obtenidos para diferentes valores de k y $nstart$

En resumen, k-means clustering es un algoritmo iterativo de clustering con dos pasos:

1. Asignación de los clusters, en la cual inicialmente se establecen aleatoriamente k centroides sobre las observaciones y se calcula la distancia a cada centroide, y en función de esto se lleva a cabo el siguiente paso.

2. Actualización de los centroides, en este paso se mueven los centroides a la media de las observaciones pertenecientes a cada cluster, es decir, k-means calcula la media de todas las observaciones y traslada el centroide a la dicha ubicación media para cada uno de los k grupos.

En conclusión, se puede observar la variabilidad de los resultados, como para un valor de k más pequeño, se ha de establecer un número elevado de configuraciones iniciales ($nstart$) para obtener unos resultados aceptables. Los k centroides iniciales se establecen de manera aleatoria y si la dispersión de los datos es elevada a su vez, la precisión disminuirá. por lo tanto cuanto mayor sea la dispersión de los datos y menor sea el valor de k, mayor ha de ser el número de configuraciones iniciales a realizar.

En lo que respecta a la dispersión, se puede observar en los resultados obtenidos en (b) y en (c) de la figura 2.3, para las cuales en la parte izquierda de las figuras, donde los datos están más dispersos, cuando se establece una única configuración inicial ($nstart = 1$ en (b)), los resultados obtenidos son menos precisos que en (c), donde se puede observar, que en la misma parte de la figura donde los datos están más dispersos, los resultados obtenidos respecto a los clusters y los centroides establecidos poseen una mayor precisión, siguiendo los centroides el movimiento de las trayectorias.

2.2. Fuzzy c-means clustering

Fuzzy c-means clustering [5] es uno de los principales algoritmos de fuzzy clustering donde cada observación x_i, pertenece de manera difusa a uno de los k grupos. A diferencia del anterior algoritmo k-means clustering, donde cada una de las observaciones pertenece exclusivamente a un cluster, fuzzy c-means clustering se basa en la probabilidad que tiene cada punto de pertenecer a cada uno de los clusters. Este último trata de enfrentarse a la problemática que surge cuando una observación se encuentra entre dos centroides.
Dado un conjunto $\boldsymbol{X}$ con n observaciones tal que $\boldsymbol{X} = x_1, x_2, \ldots, x_n$, con un conjunto de grupos $\boldsymbol{C} = c_1, c_2, \ldots, c_c$, se asociará cada elemento de $\boldsymbol{X}$ con todos los clusters. Tras esto, se tendrá una matriz de particionado $\boldsymbol{W}$, tal que $\boldsymbol{W} = w_{(i,j)} \in [0,1], i = 1, \ldots, n, j = 1, \ldots, c$, donde cada uno de los elementos de $\boldsymbol{W}$, $w_{(i,j)}$ indicará el grado de pertenencia del elemento x_i al cluster c_j.
El principal objetivo de fuzzy c-means será minimizar la siguiente función objetivo:

$$argmins \sum_{i=1}^{k} \sum_{j=1}^{n} w_{i,j}^{m} \|x_i - c_j\|^2 \tag{2.4}$$

Una vez establecida la matriz de pertenencias W, puesto que se está de nuevo ante un algoritmo iterativo, se llevarán a cabo las siguientes fases:

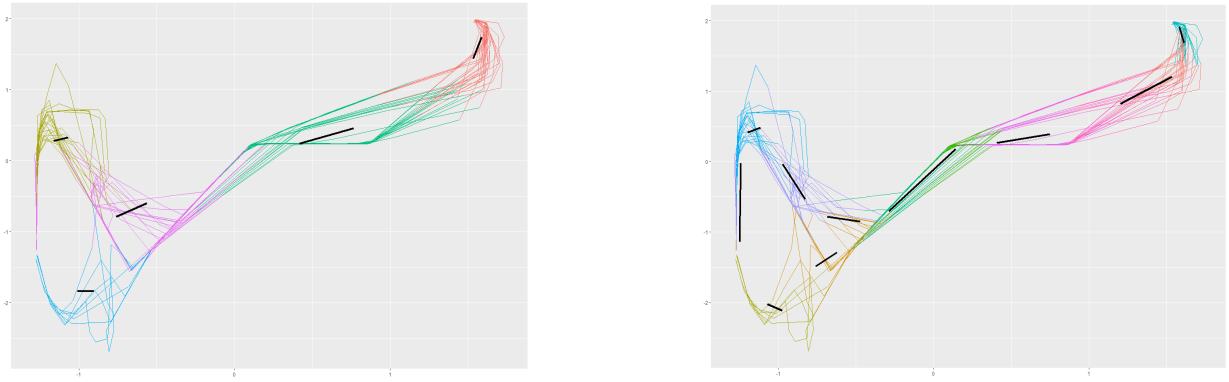
1. **Cálculo de los centros de los clusters** para cada una de las observaciones de $\boldsymbol{X}$ mediante la siguiente fórmula:
$$c_i = \frac{\sum_{k=1}^{n} (w_{i,k}^{)} m x_k}{\sum_{j=1} n(w_{i,k}^{m})}; 1 \leq i \leq c \tag{2.5}$$

2. **Actualizar los valores de la matriz de pertenencia $\boldsymbol{W}$**, según la siguiente fórmula:
$$w_{i,j} = \frac{1}{\sum_{k=1}^{c} (\frac{\|x_i - c_j\|}{\|x_i - c_k\|})^{\frac{2}{m-1}}} \tag{2.6}$$

En todos los casos, $m \in \mathbb{R}$; $m \leq 1$ es el grado de difusión o difusor, que determina el grado de difusión de los clusters. Si m tiene un valor elevado, los valores de $w_{(i,j)}$ serán más pequeños, lo que significa que el valor de pertenencia es más pequeño y esto se traduce en clusters más difusos. Si $m = 1$, toma los valores 0 o 1, lo que se traduce en el algoritmo k-means, en el que teníamos

pertenencia exclusiva a cada cluster. El valor de m suele establecerse en 2. Así mismo, el resultado final depende de los valores iniciales establecidos, como, por ejemplo, la elección de la cantidad de clusters c.
Al igual que se hacía con el algoritmo k-means, se va a aplicar fuzzy c-means al conjunto de trayectorias de la Figura 2.2. El número de clusters serán los mismos que para k-means en cada uno de los casos.

En la figura 2.4 se pueden observar los centroides obtenidos y los clusters obtenidos.



(a) $c = 5$ (b) $c = 10$

Figura 2.4: Resultados para diferentes valores de c

En resumen, la principal diferencia entre k-means y fuzzy c-means es que este último, establece un peso para cada una de las observaciones respecto a cada cluster que será determinante a la hora de observar qué trayectorias pertenecen a qué cluster.

Otros ejemplos de algoritmos de clustering son BIRCH [27], que se caracteriza por realizar agrupaciones jerárquicas sobre conjuntos de datos particularmente grandes, DBSCAN [8], basado en densidad ya que agrupa los datos en clusters partiendo de una estimación de la distribución de densidad de las observaciones correspondientes, u OPTICS [2], conocido como una generalización de DBSCAN, estableciendo como parámetro que determina la existencia de clusters, el máximo radio de búsqueda. Todos estos algoritmos se aplican a la hora de realizar clustering sobre diferentes observaciones y son aplicables a trayectorias, pero con ninguno de ellos podemos obtener sub-trayectorias comunes. Descubrir sub-trayectorias comunes a un grupo de trayectorias puede ser de gran utilidad, por ejemplo, a la hora de analizar una zona del mapa que atraviesan diversas trayectorias con patrones comunes no esperados.

2.3. Traclus

Para poner fin al problema de la ausencia de sub-trayectorias comunes, *TRACLUS* [14] propone un algoritmo para clustering de trayectorias basado en el particionado y la agrupación de trayectorias en clusters. La principal ventaja y objetivo de este algoritmo es obtener sub-trayectorias comunes de un conjunto de trayectorias. Para la primera parte del algoritmo, en la cual se particionan las trayectorias, *TRACLUS* presenta un algoritmo de particionado usando el principio de la Longitud Mínima Descriptiva *(MDL - Minimum Description Length)* [22]. Para la segunda fase, en la que se agrupan las particiones de las trayectorias, *TRACLUS* utiliza un algoritmo de clustering

de trayectorias basado en la densidad. Los resultados experimentales demuestran que *TRACLUS* obtiene de manera correcta sub-trayectorias, como se puede ver en la figura 2.5, en la cual se han utilizado las mismas observaciones de trayectorias que en las aplicaciones de k-means y c-means. En los siguientes apartados se plantea una descripción detallada de *TRACLUS y su uso para el clustering de trayectorias*

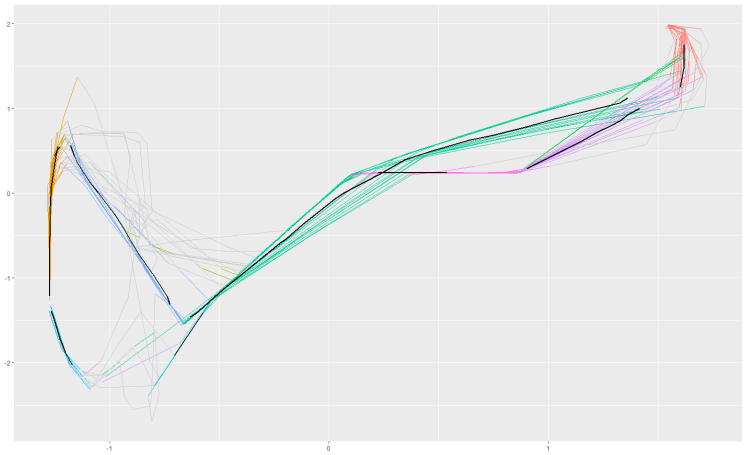


Figura 2.5: Ejemplo de Traclus

2.3.1. Introducción a Traclus

En el algoritmo de clustering de trayectorias propuesto por Scott Gaffney y Padhraic Smyth [12], consideran como unidad básica de cluster toda la trayectoria, lo que no permite obtener particiones cómunes. Las trayectorias suelen dibujar una ruta compleja, por lo que, si se particionan, las porciones obtenidas de cada una de las trayectorias podrán presentar un comportamiento similar, lo que será menos probable considerando cada trayectoria como un cluster.

Considerando un conjunto de trayectorias $\boldsymbol{T} = TR_1, TR_2, ..., TR_{num_{tra}}$, siendo en este caso el valor de $num_{tra} = 5$, se obtiene un comportamiento similar de las trayectorias (rectángulo amarillo de la figura 2.6). Si se considerase $TR_i : i \in [1, num_{tra}] : i \in \mathbb{N}$ como unidad básica para aplicar clustering a $\boldsymbol{T}$, no se podría obtener la subtrayectoria común, ya que cada TR_i toma una dirección.

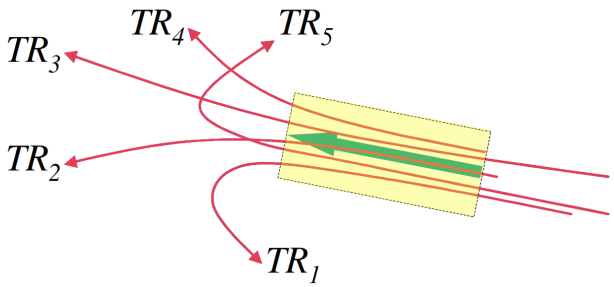


Figura 2.6: Sub-trayectoria común

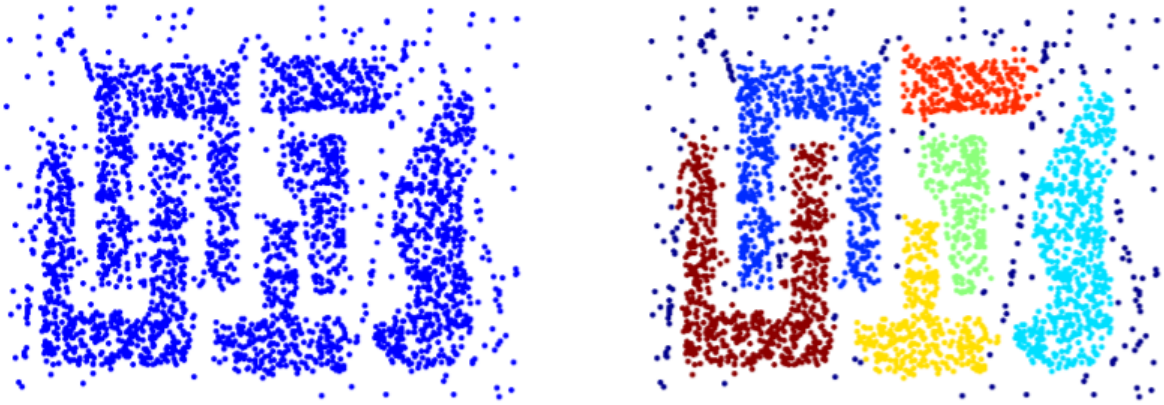
La solución que propone *TRACLUS* a esto es particionar cada valor de $TR_i : i \in [1, num_{tra}] : i \in \mathbb{N}$, en un conjunto de segmentos y aplicar clustering a cada uno de los conjuntos de segmentos

obtenidos. Este framework se considera un framework de *partition-group*, es decir, un framework de particionado y agrupamiento. La utilización de este framework tiene una clara ventaja: el descubrimiento de *sub*-trayectorias de $\boldsymbol{T}$. Esta es la principal razón del particionado de una trayectoria en segmentos.

Por lo tanto, *TRACLUS* propone un marco de trabajo de particionado y agrupamiento de trayectorias, que, como bien se indica, consistirá en dos fases:

1. La fase de ***particionado***, en la que cada trayectoria se particionará de manera optima en un conjunto de segmentos que serán utilizados en la siguiente fase.
2. La fase de ***agrupamiento*** o *clustering*, en la que se obtendrán similitudes entre los segmentos particionados y se agruparán en un cluster, utilizando para ello un método basado de densidad.

La utilización de un método de clustering basado en densidad se debe a que es el tipo de método más apropiado para este caso, ya que, considerando un cluster como un conjunto denso de puntos (en este caso, trayectorias), el cual está separado de otro conjunto denso de puntos por otro menos denso, tiene la capacidad de obtener clusters cuando estos tienen formas irregulares y actúa de manera robusta ante la presencia de ruido.



(a) Puntos con ruido

(b) Clusters con ruido

Figura 2.7: Ejemplo de aplicación del método DBSCAN

Como se puede observar en la figura 2.7, se tiene un conjunto de puntos al cual se ha aplicado el método de clustering basado en densidad DBSCAN, obteniendo los clusters con diferentes formas y de manera robusta a pesar de la existencia de ruido alrededor de los mismos.

Debido a que las trayectorias suelen tener formas totalmente arbitrarias, al disponer de un conjunto de trayectorias y aplicar clustering sobre el mismo, se observará que una parte será ruido. Debido a la presencia de este ruido, el método de clustering por densidad será el más adecuado. Finalmente, *TRACLUS* se presentará como sigue:

- En un entorno de trabajo de *particionado* y *agrupamiento* para realizar clustering de las trayectorias, permitiendo este entorno la obtención de subtrayectorias a diferencia de otros entornos de trabajo descritos previamente.
- Aplicando el principio de mínima distancia descriptiva [22] para particionar las trayectorias (se describirá en secciones posteriores).

- Utilizando un método basado en la densidad para agrupar los segmentos obtenidos en el particionado (como se ha descrito anteriormente). Además, se ha definido una función de distancia para determinar la densidad de los segmentos, así como la heurística para definir los parámetros que recibe el algoritmo.

- Utilizando un conjunto de trayectorias reales, se aplicará el algoritmo para obtener trayectorias representativas.

2.3.2. Clustering de trayectorias con *TRACLUS*

En este apartado se va a presentar un resumen del diseño principal de *TRACLUS*, describiendo el planteamiento del problema en el que se establecerán diferentes términos a utilizar durante el resto del presente documento, la estructura de algoritmo con sus correspondientes algoritmos y tareas internas, y la función desarrollada para calcular las distancias entre segmentos

Planteamiento del problema

Como se ha comentado, el algoritmo se basa en un entorno de trabajo de particionado y agrupamiento. Para un conjunto de trayectorias $\mathcal{I} = \{TR_1, ..., TR_{num_{tra}}\}$, *TRACLUS* genera un conjunto de clusters $\mathcal{O} = \{C_1, ..., C_{num_{clus}}\}$ y un conjunto de trayectorias representativas para cada uno de los cluster C_i. A continuación se van a aclarar estos términos:

- Una *trayectoria* es un conjunto de puntos situados de manera secuencial, ya sean *bi*dimensionales o *multi*-dimensionales. Considerando una trayectoria TR_i como $TR_i = p_1p_2p_3...p_j...p_{len_i} (1 \leq i \leq num_{tra})$. El valor de len_i puede diferir para cada una de las $TR_i; i \in [1, num_{tra}]$. Una trayectoria $p_1p_2p_3...p_j...p_{c_k} (1 \leq c_1 \leq c_2 \leq ... \leq c_k \leq len_i)$ se considerará una *sub*-trayectoria de TR_i.

- Un *cluster* es un conjunto particiones de una trayectoria. Una *partición* de una trayectoria es un segmento $p_ip_i (i < j)$, siendo p_i y p_j los puntos seleccionados para que formen el segmento pertenecientes ambos a una misma trayectoria. Los segmentos formarán parte de un mismo cluster según la distancia que los separe, la cual se calculará con la función de distancia. Por lo tanto, una trayectoria puede formar parte de varios clusters, ya que el clustering se aplicará a los segmentos obtenidos en el particionado.

- Una *trayectoria representativa* es un conjunto de puntos dispuestos de la misma manera que cualquier otra trayectoria. Es una trayectoria imaginaria obtenida a partir del conjunto de trayectorias reales, y que indicará como se comportan el mayor número de trayectorias del conjunto, es decir, una trayectoria representativa indica una *sub*-trayectoria común a todas la trayectorias.

En la figura 2.8 se puede visualizar en un esquema el entorno de trabajo de particionado y agrupamiento. Para un conjunto de trayectorias $\mathcal{I} = \{TR_1, ..., TR_{num_{tra}}\}$, siendo en este caso $num_{tra} = 5$, se obtiene un conjunto de segmentos a partir del paso (1), que es el particionado. Tras esto, se procede al paso (2), que es el agrupamiento, del que se obtiene un cluster y una subtrayectoria común a los segmentos pertenecientes a ese mismo cluster (en color rojo en la foto).

El algoritmo *TRACLUS*

A continuación, se va a mostrar la estructura del algoritmo de clustering de trayectorias *TRACLUS*, que a su vez está formado por tres algoritmos que llevan a cabo el particionado, el clustering o agrupamiento y la obtención de las subtrayectorias o trayectorias representativas.

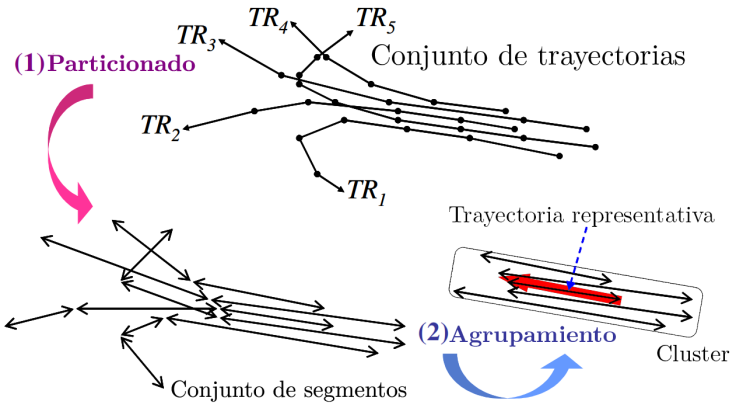


Figura 2.8: Ejemplo de particionado y agrupamiento

Algoritmo 1 TRACLUS (TRAjectory CLUStering)

INPUT: Un conjunto de trayectorias $\mathcal{I} = \{TR_1, ..., TR_{num_{tra}}\}$
OUTPUT: Un conjunto de clusters $\mathcal{O} = \{C_1, ..., C_{num_{clus}}\}$ y un conjunto de trayectorias representativas
ALGORITMO:

```
/* FASE DE PARTICIONADO */
for each (TR ∈ I) do
    Ejecutar la función ApproximatePartitioningTrajectory; /* Algoritmo 2 */
    Obtener un conjunto de segmentos L utilizando el resultado;
    Almacenar L en un conjunto D;
end for
/* FASE DE AGRUPAMIENTO */
Ejecutar LineSegmentClustering for D; /* Algoritmo 3 */
Obtener un conjunto de clusters O;
for each (C ∈ O) do
    Ejecutar RepresentativeTrajectoryGeneration; /* Algoritmo 4 */
    Obtener una trayectoria representativa de C como resultado;
end for
```

Cálculo de la distancia

En este apartado se va a describir la metodología seguida para obtener la distancia entre dos segmentos obtenidos de particionar dos trayectorias diferentes. Para la obtención de la distancia final, es necesario el cálculo previo de tres tipos de distancias: *perpendicular*$(d_\perp)$, *paralela* $(d_\parallel)$ y *angular* (d_θ). Para comenzar, en la figura 2.9 se van a representar gráficamente estas distancias. Para ello, se plantean dos segmentos L_i y L_j formados por los puntos (s_i,e_i) y (s_j,e_j) respectivamente, entendiendo por los puntos s los puntos de comienzo y los puntos e como los de final. Además, para la representación se ha planteado el segmento L_j como un segmento de longitud menor que la del segmento L_i.

Por lo que, por definición, los tres componentes quedan como sigue:

- La *distancia perpendicular* $(d_\perp)$ entre L_i y L_j vendrá dada por la fórmula 2.7, la cual es en

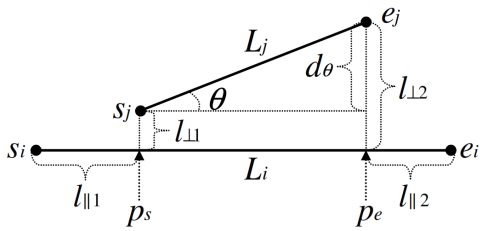


Figura 2.9: Componentes de la distancia entre dos segmentos L_i y L_j

realidad la media de Lehmer[1] de orden $n = 2$.

$$d_\perp(L_i, L_j) = \frac{l_{\perp 1}^2 + l_{\perp 2}^2}{l_{\perp 1} + l_{\perp 2}} \tag{2.7}$$

Entendiendo los puntos p_s y p_e como la proyección de los puntos s_j y e_j en L_i, $l_{\perp 1}$ es la distancia entre p_s y s_j, y $l_{\perp 2}$ es la distancia entre p_e y e_j.

- La *distancia paralela* ($d_\parallel$) entre L_i y L_j se puede ver definida en la fórmula 2.8. Siendo la proyección de los puntos s_j y e_j sobre L_i los puntos p_s y p_e respectivamente, $l_{\parallel 1}$ será el valor mínimo de la distancia euclídea del punto p_s a los puntos s_i y e_i, de la misma manera $l_{\parallel 2}$ será el mínimo de las distancias euclídeas de p_e a los puntos s_i y e_i.

$$d_\parallel(L_i, L_j) = MIN(l_{\parallel 1}, l_{\parallel 2}) \tag{2.8}$$

Esta distancia paralela está diseñada para que el algoritmo actúe de manera robusta ante la detección de errores, ya que si se utilizase $MAX(l_{\parallel 1}, l_{\parallel 2})$ en lugar de $MIN(l_{\parallel 1}, l_{\parallel 2})$, el cálculo de este componente se vería alterado por alguno de los segmentos.

- La *distancia angular* (d_θ) entre L_i y L_j se obtiene mediante la fórmula 2.9. En este caso, se tiene el parámetro $\|L_j\|$, que es el tamaño de L_j, y θ ($0^o \leq \theta \leq 180^o$), que es el menor ángulo de la intersección entre L_i y L_j.

$$d_\theta(L_i, L_j) = \begin{cases} \|L_j\| \text{x} \sin(\theta) & \text{si } (0^o \leq \theta \text{ ¡}90^o) \\ \|L_j\| & \text{si } (90 \leq \theta \leq 180^o) \end{cases} \tag{2.9}$$

El cálculo de este componente está diseñado para trayectorias con direcciones. La presencia de esta distancia tiene sentido cuando se calcula entre dos segmentos que difieren en su dirección.

Los tres componentes anteriores se pueden calcular usando operaciones vectoriales. Siendo $\overrightarrow{ab}$ un vector formado por los puntos a y b, los puntos de proyección p_s y p_e representados en la figura 2.9 se calculan con la formula 2.10 y el ángulo θ se calcula utilizando la fórmula 2.11.

$$p_s = s_i + u_1 \cdot \overrightarrow{s_i e_i}, p_e = s_i + u_2 \cdot \overrightarrow{s_i e_i}, \text{siendo} u_1 = \frac{\overrightarrow{s_i s_j} \cdot \overrightarrow{s_i e_i}}{\|\overrightarrow{s_i e_i}\|^2}, u_2 = \frac{\overrightarrow{s_i e_j} \cdot \overrightarrow{s_i e_i}}{\|\overrightarrow{s_i e_i}\|^2} \tag{2.10}$$

$$\cos(\theta) = \frac{\overrightarrow{s_i e_i} \cdot \overrightarrow{s_j e_j}}{\|\overrightarrow{s_i e_i}\| \|\overrightarrow{s_j e_j}\|} \tag{2.11}$$

[1]La media de Lehmer de un conjunto de n números viene definida por $L_p(a_1, a_2, ..., a_n) = \frac{\sum_{k=1}^n a_k^p}{\sum_{k=1}^n a_k^{p-1}}$

2.3.3. Particionado de trayectorias

En esta sección, se va a presentar el algoritmo de particionado de las trayectorias para esta fase de particionado. En primer lugar se discutirán las propiedades deseables del algoritmo, y con estas, establecidas se tratarán de obtener. Tras esto, se optimizará el particionado aplicando el principio de MDL (*Minimum Description Length*), y debido al alto coste computacional de dicha aplicación, se propondrá una aproximación a la solución.

Propiedades ideales

El principal objetivo de la partición es la obtención de los puntos en los cuales las trayectorias cambien de manera característica o rápida. Estos puntos en los cuales la trayectoria cambia y a partir de los cuales se particionará la trayectoria se llamarán, de ahora en adelante, *puntos característicos*. Por lo tanto, para una trayectoria $TR_i = p_1p_2p_3...p_j...p_{len_i}$ se determina un conjunto de puntos $(p_{c_1}, p_{c_2}, p_{c_3}, ..., p_{c_{par_i}},)$ $((c_1 < c_2 < c_3 < c_{par_i})$. Por lo tanto, la trayectoria TR_i se particionará en cada uno de los puntos característicos, y cada partición estará formada por la unión de dos puntos característicos consecutivos. Por lo tanto, la trayectoria TR_i se particionará en un conjunto de $(par_i - 1)$ segmentos, siendo este conjunto $\{p_{c1}, p_{c2}, p_{c3}...p_{c_{par_i-1}}, p_{c_{par_i}}\}$ y considerando cada uno de los segmentos p_i como una partición de la trayectoria. En la figura 2.10 se muestra un ejemplo de una trayectoria y como procedería su particionado.

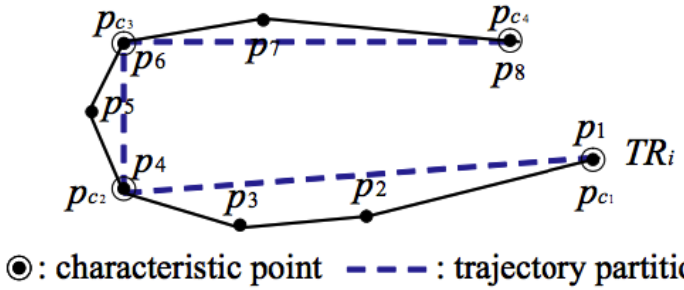


Figura 2.10: Particionado de una trayectoria y sus particiones

Con esto, se puede considerar que una trayectoria se particionará de manera óptima si posee dos propiedades ideales: *precisión*, en lo que refiere a que la diferencia entre una trayectoria y sus particiones sea mínima, y *concisión*, en lo que refiere que el número de particiones de una trayectoria sea mínimo.

Para conseguir la precisión máxima, una trayectoria debería particionarse en cada uno de los puntos en los que su comportamiento cambie de manera ínfima. Por lo tanto, la consecución de la precisión máxima se considera un hecho deseable. Si en la trayectoria TR_i de la figura 2.10, no se hubiese considerado el punto p_{c_2} como punto característico, la precisión de la partición disminuiría de manera considerable, debido a la notable diferencia entre TR_i y $p_{c_1}p_{c_3}$. Por otra parte, la obtención de la precisión máxima y la concisión máxima en el particionado es contradictoria. Si todos los puntos de una trayectoria son considerados como puntos característicos, es decir, $par_i = len_i$, se estaría ante una situación con precisión máxima y concisión mínima, ya que como todos los puntos son diferentes de su siguiente (mínimo cambio en el comportamiento de la trayectoria) son considerados como puntos característicos. Por lo contrario, y en lo que respecta a la concisión, si se considerasen como puntos característicos exclusivamente los puntos de inicio y final de la trayectoria, es decir, $par_i = len_i$, se estaría ante una situación de concisión máxima y precisión

mínima. Por lo tanto, se necesita encontrar un punto intermedio entre ambos extremos de precisión y concisión del particionado, lo cual se va a tratar en la siguiente sección.

Formalización con el uso del principio MDL

Como bien se comentaba en la sección anterior, es preciso encontrar un punto intermedio entre la precisión y la concisión del particionado. Para ello, se ha tomado el principio del mínimo tamaño descriptivo o *Minimun Description Length (MDL)*, el cual es muy utilizado en la teoría de la información.

El principio MDL está formado por dos componentes principales: $L(H)$ y $L(D|H)$, de tal manera que H es la hipótesis y D los datos. De esta manera, $L(H)$ es el coste en bits de describir la hipótesis H, y $L(D|H)$ es el tamaño en bits de la descripción de los datos cuando estos se codifican con la ayuda de la hipótesis H. Por lo tanto, la mejor hipótesis H que explica D será aquella que minimice la suma de $L(H)$ y $L(D|H)$.

Al trasladar esto al particionado de las trayectorias, la hipótesis H será un conjunto de particiones de una trayectoria, ya que lo que se buscará será encontrar la partición más óptima de una trayectoria. Así pues, encontrar un particionado óptimo de las trayectorias equivale a encontrar la mejor hipótesis al utilizar el principio MDL.

Por lo tanto, con la fórmula 2.13 se obtendrá el valor de $L(D|H)$, lo que representa la suma de la diferencia entre una trayectoria y un conjunto de particiones de la misma trayectoria. Para cada partición de la trayectoria, denotada por $p_{c_j}p_{c_{j+1}}$, se calculará la diferencia entre esta y un segmento $p_kp_{k+1}(c_j \leq k \leq c_{j+1} - 1)$, el cual pertenece a esta misma partición. Para medir la diferencia, se utiliza la suma de la distancia perpendicular y la distancia angular. La distancia paralela no se ha considerado ya que una trayectoria contiene sus propias particiones.

$$L(H) = \sum_{j=1}^{par_i - 1} \log_2(len(p_{c_j}p_{c_{j+1}})) \tag{2.12}$$

$$L(D|H) = \sum_{j=1}^{par_i - 1} \sum_{k=c_j}^{c_{j+1}-1} \{\log_2(d_\perp(p_{c_j}p_{c_{j+1}}, p_kp_{k+1})) + \log_2(d_\theta(p_{c_j}p_{c_{j+1}}, p_kp_{k+1}))\} \tag{2.13}$$

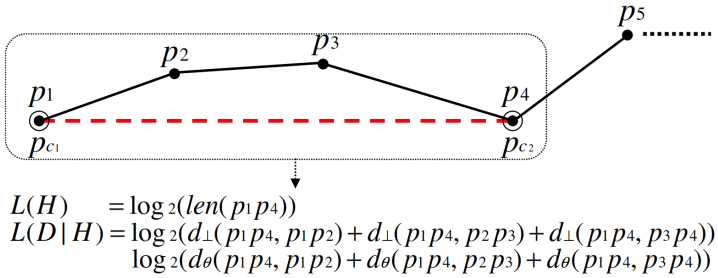


Figura 2.11: Ejemplo de aplicación del principio MDL al particionado de trayectorias

Con esto se puede decir, que con $L(H)$ se medirá el grado de concisión y con $L(D|H)$ el grado de precisión. $L(H)$ será directamente proporcional al número de particiones de una trayectoria, a diferencia de $L(D|H)$, que se minimizará cuanto menos se desvíen las particiones de la trayectoria. En la figura 2.11 se puede visualizar un ejemplo de cómo actuará la aplicación del principio MDL al particionado de trayectorias calculando los valores con las formulas 2.12 y 2.13.

Como se mencionaba antes, lo que se busca es una partición óptima que minimice el valor de la suma de $L(H)$ y $L(D|H)$. Debido al coste que supondría realizar este cálculo sobre cada uno de los subconjuntos de puntos que formen segmentos, se va a proponer una solución aproximada en la siguiente sección.

Solución aproximada

Se considerará $MDL_{par}(p_i, p_j)$ como el coste de aplicar el principio MDL anterior a una trayectoria entre p_i y p_j $(i < j)$ considerando p_i y p_j como puntos característicos. A su vez, se considerará $MDL_{nopar}(p_i, p_j)$ como el coste de aplicar el principio MDL a una trayectoria entre p_i y p_j $(i < j)$ la cual no incluye puntos característicos como podría ser la propia trayectoria completa (en este caso $L(D|H)$ será cero, ya que las particiones no se desviarían de la trayectoria original en ningún momento). En base a esto, una partición óptima $p_i p_j$ será aquella que cumpla la condición de $MDL_{par}(p_i, p_k) \leq MDL_{nopar}(p_i, p_k)$ para todo k tal que $i < k \leq j$. Si para un punto anterior se cumple la condición de manera menor a la actual, habrá que considerar el punto anterior como punto característico. A su vez, se tratará de aumentar dicha partición hasta el punto siguiente con el fin de conseguir la menor concisión.

En el algoritmo 2 se puede observar el *Particionado de trayectorias aproximado*. Se ejecuta MDL_{par} y MDL_{nopar} para cada uno de los puntos de la trayectoria. Si MDL_{par} es mayor que MDL_{nopar}, se inserta el punto anterior al actual en el conjunto de puntos característicos CP_i, ya que particionar por el punto actual supondría un coste mayor. Una vez se inserta el punto, se comienza de nuevo desde dicho punto. Si no se considera un punto característico, se aumenta el tamaño de la partición actual.

Algoritmo 2 Particionado de trayectorias aproximado

INPUT: Una trayectoria $TR_i = \{p_1 p_2 p_3 ... p_j ... p_{len_i}\}$
OUTPUT: Un conjunto de puntos característicos CP_i
ALGORITMO:

1: Añadir p_1 a CP_i /* el primer punto */
2: $puntoInicial := 1, tamanio := 1$;
3: **while** $(puntoInicial + tamanio \leq len_i)$ **do**
4: $puntoActual := puntoInicial + tamanio$;
5: $cost_{par} := MDL_{par}(p_{puntoInicial}, p_{puntoActual})$;
6: $cost_{nopar} := MDL_{nopar}(p_{puntoInicial}, p_{puntoActual})$;
7: /* Se comprueba si el coste de particionar la trayectoria en el punto actual es más alto que si no se particiona*/
8: **if** $(cost_{par} > cost_{nopar})$ **then**
9: /*Se particiona en el punto anterior porque el actual incrementa el coste de particionar*/
10: Añadir $p_{puntoInicial-1}$ al conjunto de puntos representativos CP_i;
11: $puntoInicial := puntoActual - 1, tamanio := 1$;
12: **else**
13: $tamanio := tamanio + 1$;
14: Añadir p_{len_i} a CP_i;
15:

Este algoritmo no pueda obtener el particionado óptimo. En la figura 2.12 se puede ver un ejemplo. Suponiendo que la partición óptima con el coste mínimo es $p_1 p_5$, el algoritmo no puede encontrar la solución más óptima, ya que parará de escanear en p_4, dónde se cumple la condición

de que MDL_par es mayor que MDL_nopar, por lo que se particionará por p_3. De todas maneras, la precisión del algoritmo es bastante alta.

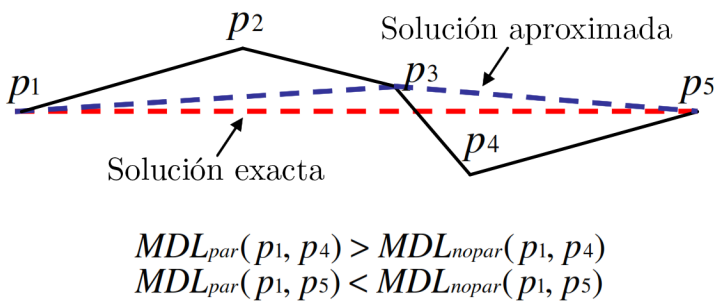


Figura 2.12: Ejemplo de aplicación del algoritmo de particionado aproximado

2.3.4. Clustering de los segmentos

En esta sección se va a explicar el algoritmo diseñado para realizar el clustering de los segmentos para la fase de agrupamiento. En primer lugar, se va a describir el concepto de densidad de segmentos, después se va a presentar el algoritmo de clustering basado en la densidad. Tras esto, se va a presentar un método para obtener la trayectoria representativa de un cluster, para terminar estableciendo una heurística de la elección de los parámetros para los cuales varían los resultados obtenidos a partir del algoritmo.

Densidad de los segmentos

En primer lugar, se va a repasar la función de distancia diseñada. Esta función estaba compuesta por la suma de tres tipos de distancias (perpendicular, paralela y angular). La distancia perpendicular mide principalmente la distancia entre segmentos extraídos de diferentes trayectorias. La distancia paralela mide la distancia entre segmentos extraídos de una misma trayectoria. Cabe recordar que la distancia paralela de dos segmentos cercanos obtenidos de una misma trayectoria será siempre igual a 0. Por último, la distancia angular medirá la diferencia en la dirección entre dos segmentos.
En lo que respecta a la simetría de la función, la distancia entre dos segmentos L_i y L_j, definida por $dist(L_i, L_j)$ será simétrica, es decir, $dist(L_i, L_j) = dist(L_j, L_i)$

Nociones necesarias para el clustering basado en densidad

A continuación, se describirán las principales nociones a tener en cuenta para comprender como se procederá con el clustering de las trayectorias basado en la densidad:

- El valor de $\epsilon - neighborhood N_\epsilon(L_i)$ de un segmento $L_i \in \mathcal{D}$ vendrá definido por $N_\epsilon(L_i) = \{L_j \in \mathcal{D} | dist(L_i, L_j) \leq \epsilon\}$.
- Un segmento $L_i \in \mathcal{D}$ se le denominará *segmento principal* respecto a ϵ y $MinLns$ si $|N_\epsilon(L_i)| \geq MinLns$
- Un segmento $L_i \in \mathcal{D}$ es directamente alcanzable en densidad por otro segmento $L_j \in \mathcal{D}$ respecto a ϵ y $MinLns$ si $L_i \in N_\epsilon(L_j)$ y $|N_\epsilon(L_i)| \geq MinLns$
- Un segmento $L_i \in \mathcal{D}$ es alcanzable en densidad por otro segmento $L_j \in \mathcal{D}$ respecto a ϵ y $MinLns$ si hay un conjunto de segmentos $L_j, L_{j-1}, ..., L_{i+1}, L_i \in \mathcal{D}$ tal que un segmento L_k es directamente alcanzable en densidad por $L_k + 1$ respecto a $\epsilon y MinLns$

- Un segmento $L_i \in \mathcal{D}$ está conectado en densidad con un segmento $L_j \in \mathcal{D}$ respecto a ϵ y $MinLns$ si existe un segmento $L_k \in \mathcal{D}$ para el cual L_i y L_j son alcanzables desde L_k para los valores de ϵ y $MinLns$
- Un subconjunto no vacio $\mathcal{C} \subseteq \mathcal{D}$ se denomina conjunto conectado en densidad respecto a ϵ y $MinLns$ si $\mathcal{C}$ cumple las siguientes condiciones:
  1. *Conectividad*: $\forall L_i, L_j \in \mathcal{C}$, L_i está conectado con L_j para los valores de ϵ y $MinLns$
  2. *Máxima*: $\forall L_i, L_j \in \mathcal{D}$, si $L_i \in \mathcal{C}$ y L_j es alcanzable en densidad por L_i para los valores de ϵ y $MinLns$, entonces $L_j \in \mathcal{C}$

Si dos segmentos son directamente alcanzables en densidad, serán también alcanzables de manera simple, pero esta relación no es simétrica. Únicamente los segmentos principales son alcanzables en densidad mutuamente. Sin embargo, si están conectados en densidad nos encontraríamos ante una relación simétrica. En la figura 2.13 se ilustran las nociones anteriores para un valor de $MinLns = 3$.

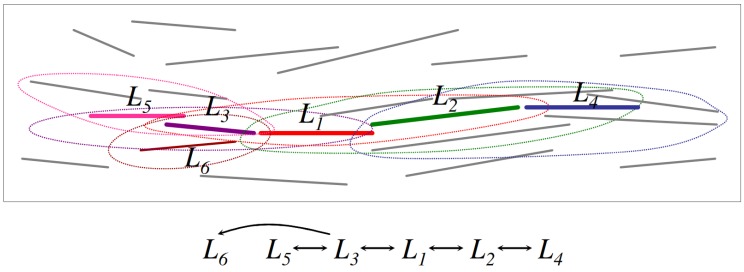


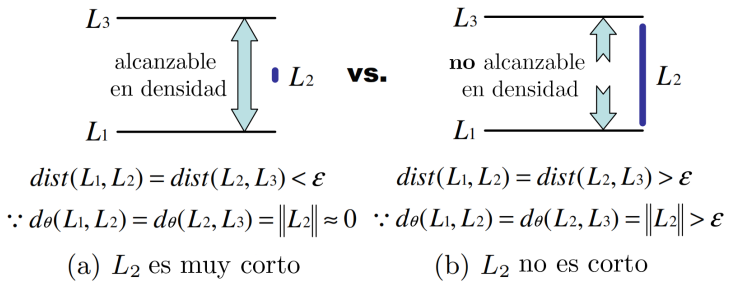
Figura 2.13: Segmentos conectados en densidad y/o alcanzables en densidad

- L_1,L_2,L_3,L_4 y L_5 son segmentos principales, ya que para cada uno de ellos, se cumple la condición de que $|N_\epsilon(L_i)| \geq MinLns$
- L_2 (o L_3) son directamente accesibles en densidad por L_1, ya que tanto L_2,$L_3 \in N_\epsilon(L_1)$ y $|N_\epsilon(L_i)| \geq MinLns$
- L_1,L_4 y L_5 están conectados en densidad, ya que forman un conjunto que cumplen las condiciones de Conectividad y Máxima.

Observaciones
Puesto que los segmentos poseen dirección y tamaño, tendrán características interesantes de cara a realizar el clustering basado en la densidad. En la figura 2.13 se podía observar como la forma de los $\epsilon - neighborhood$ en los segmentos no es un círculo ni una esfera, si no que esta depende de los datos y en la mayoría de los casos tomará la forma de una elipse o un elipsoide. Esta forma dependerá sobre todo de la dirección y la longitud del segmento principal así como de los segmentos adyacentes.

Por otra parte, otra observación interesante es que un segmento más corto podría decrementar la calidad del agrupamiento realizado sobre los segmentos. Se puede observar que la longitud de un segmento representa su *fuerza direccional*, es decir, cuánto de definida está la dirección de un segmento, por lo que cuanto menor sea la longitud menor será la fuerza direccional. A consecuencia de esto, la distancia angular será pequeña independientemente del ángulo, si uno de los dos segmentos es muy corto. Como se puede ver en la figura 2.14, varía la longitud de L_2 entre (a) y (b). En (a),

L_1 puede ser alcanzable en densidad por L_3 (o viceversa) a través de L_2, pero no puede serlo en (b). En (a) el resultado es contrario a lo que se intuiría, ya que en ese caso L_1 y L_3 estarían muy separados. Por lo tanto, los segmentos cortos pueden conllevar a la aparición de cluster excesivos.



(a) L_2 es muy corto (b) L_2 no es corto

Figura 2.14: Influencia de un segmento muy corto al hacer clustering

Una solución a esto es ajustar los criterios de partición en la figura 2. Para esto, basta con añadir una pequeña constante a la variable $cost_{nopar}$ de manera que en ciertos casos se anulará la partición, obteniendo así segmentos más largos y aumentando por lo tanto la calidad de los clusters obtenidos.

Algoritmo de clustering

A continuación se va a presentar el algoritmo de clustering basado en densidad. Para un conjunto de segmentos $\mathcal{D}$, el algoritmo generará un conjunto de clusters $\mathcal{O}$. Este algoritmo toma dos parámetros ϵ y $MinLns$. En este caso, y con lo explicado anteriormente, se entenderá un cluster como un conjunto de segmentos conectados en densidad.

A diferencia de otros algoritmos de clustering, como por ejemplo DBSCAN, no todos los segmentos conectados en densidad formaran un cluster. Cabe considerar el numero de trayectorias de las que se obtienen los segmentos, el cual será normalmente menor que el de segmentos. En el extremo, se podría dar el caso de que todos los segmentos en un subconjunto de segmentos conectados en densidad fuesen de una misma trayectoria. Estos clusters se evitarán ya que no son representativos porque no explican el comportamiento de un número considerable de trayectorias (en el extremo, el número de trayectorias de un cluster sería 1). Por lo tanto, un conjunto de trayectorias participantes en un cluster C_i vendrá definido por $PTR(C_i) = \{TR(L_j)|\forall L_j \in C_i)\}$, siendo $TR(L_j)$ la trayectoria de la que se ha extraído el segmento L_j. A su vez, $|PTR(C_i)|$ dará lugar a la *cardinalidad* del cluster C_i.

En el algoritmo 3 se puede ver el código del clustering de los segmentos. En un comienzo, todos los segmentos se establecen como no clasificados. Según va transcurriendo el algoritmo, clasifican o bien como un cluster o bien como ruido. El algoritmo consiste en tres pasos principales:

1. El algoritmo calcula el valor de $\epsilon - neighborhood$ para cada segmento L sin clasificar. Si L se clasifica como segmento principal, el algoritmo procede con el segundo paso para ampliar el cluster. El cluster actual para el segmento L únicamente contiene $N_\epsilon(L)$

2. El algoritmo calcula el conjunto de segmentos conectados en densidad con cada segmento considerado como principal. El método *ExpandCluster()* (línea 27) calcula los segmentos directamente alcanzables en densidad por el segmento principal y los añade al cluster actual de la iteración. Si un segmento nuevo está sin clasificar, se añade a la cola $\mathcal{Q}$ para expandir el

cluster ya que dicho segmento puede ser un segmento principal, si no está sin clasificar, no se añade a la cola porque ya se sabe que no va a ser un segmento considerado principal

3. El algoritmo comprueba la cardinalidad de cada cluster ($|PTR(C_i)|$). Si la cardinalidad está por debajo del umbral establecido (en este caso, del valor de $MinLns$), se desestima el cluster para el que se está comprobando.

Trayectoria representativa de un cluster

Una trayectoria representativa de un cluster describe el movimiento que han seguido las particiones pertenecientes a dicho cluster. Interesa obtener información cuantitativa del movimiento dentro de un mismo cluster, es decir, cuantas particiones se han movido de la misma manera en un cluster, de esta manera se explotará el potencial que posee el agrupamiento de trayectorias. Para ello, se calculará una trayectoria representativa del cluster.

En la figura 2.15 se puede visualizar cómo se genera la trayectoria representativa de un cluster. Una trayectoria representativa es un conjunto de puntos $RTR_i = p_1p_2p_3...p_j...p_{len_i} (1 \leq i \leq num_{clus})$. Estos puntos se determinan haciendo un barrido vertical de los segmentos que forman el cluster en la dirección del eje mayor del cluster (más adelante se explicará cómo se obtiene este eje), contando los segmentos que se cruzan con la linea de barrido. Cuando la línea de barrido se cruza simultáneamente con un número de segmentos mayor o igual al valor del umbral (en este caso $MinLns$), se obtienen los puntos en los que la linea se cruza con dichos segmentos y se calcula la coordenada media de todos los puntos respecto al eje mayor, insertando el punto con la coordenada media en la trayectoria representativa. Si el número de particiones que cruza la línea de barrido es menor que $MinLns$, se desestiman los cortes realizados y se continúa con el barrido, así como si un punto se encuentra muy próximo a otros, se desestima dicho punto de cara a representar la trayectoria (punto 3 de la figura 2.15.

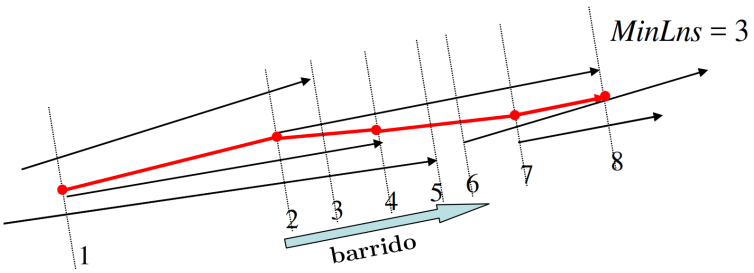


Figura 2.15: Ejemplo de un cluster y su trayectoria representativa

A continuación se va a explicar el anterior enfoque de obtención de la trayectoria representativa más detalladamente. Para el cálculo del eje mayor del cluster, se definirá el *vector de dirección media*, suponiendo un conjunto de vectores $\mathcal{V} = \{\overrightarrow{v_1}, \overrightarrow{v_2}, \overrightarrow{v_3}, ..., \overrightarrow{v_n}\}$, el *vector de dirección media* $\overrightarrow{\mathcal{V}}$ de $\mathcal{V}$ vendrá definido por la fórmula 2.14, dónde $|\mathcal{V}|$ es la cardinalidad de $|\mathcal{V}|$

$$\overrightarrow{\mathcal{V}} = \frac{\overrightarrow{v_1}, \overrightarrow{v_2}, \overrightarrow{v_3}, ..., \overrightarrow{v_n}}{\mathcal{V}} \tag{2.14}$$

Como se indicaba, se calculará la coordenada media de los puntos de corte de la linea de barrido con los segmentos utilizando para ello el *vector de dirección media*. Para facilitar dicho cálculo, se rotarán los ejes de manera que el eje X esté paralelo al *vector de dirección media*. Para ello, se

Algoritmo 3 Clustering de los segmentos particionados

INPUT: Una trayectoria $\mathcal{D} = \{L_1, ..., L_{num_{ln}}\}$ y dos parámetros ϵ y $MinLns$
OUTPUT: Un conjunto de clusters $\mathcal{O} = \{C_1, ..., C_{num_{clus}}\}$
ALGORITMO:

```
1: /* PASO 1 */
2: Establecer clusterId a 0 /* id inicial */
3: Se establecen todos los segmentos de D como sin clasificar;
4: for each (L ∈ D) do
5:     if (L está sin clasificar) then
6:         Calcular N_ε(L);
7:         if (|N_ε(L)| ≥ MinLns) then
8:             Establecer el clusterId a ∀X ∈ N_ε(L);
9:             Insertar N_ε(L) − {L} en la cola Q;
10: /* PASO 2 */
11:             ExpandCluster(Q, clusterId, ε, MinLnss);
12:             Incrementar clusterId en una unidad; /* un nuevo id */
13:         else
14:             Marcar L como ruido;
15:         end if
16:     end if
17: end for
18: /* PASO 3 */
19: Asignar ∀L ∈ D a su cluster C_clusterId;
20: /* A CONTINUACIÓN SE COMPRUEBA LA CARDINALIDAD */
21: for each (L ∈ D) do /* Se puede utilizar un umbral diferente de MinLns */
22:     if (|PTR(C_i)| < MinLns) then
23:         Borrar el cluster C del conjunto de clusters O;
24:     end if
25: end for
26: /* PASO 2: cálculo de un conjunto conectado en densidad */
27: ExpandCluster(Q, clusterId, ε, MinLnss){
28: while (Q ≠ ∅) do
29:     Establecer M como el primer segmento en Q;
30:     Calcular N_ε(M);
31:     if (|N_ε(L)| ≥ MinLns) then
32:         for each (X ∈ N_ε(M)) do
33:             if (X es sin clasificar o ruido) then
34:                 Asignar clusterId
35:             end if
36:             if (X es sin clasificar) then
37:                 Insertar X en la cola Q
38:             end if
39:
40:         end for
41:         Borrar M de la cola Q
42:         }
```

ha utilizado la fórmula 2.15. El ángulo θ se puede obtener de hacer el producto entre el *vector de dirección media* y el vector unidad. Una vez que es calculada la coordenada media en las coordenadas $X'Y'$, como en la figura 2.16 se traslada el punto de nuevo al sistema de coordenadas inicial XY.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.15}$$

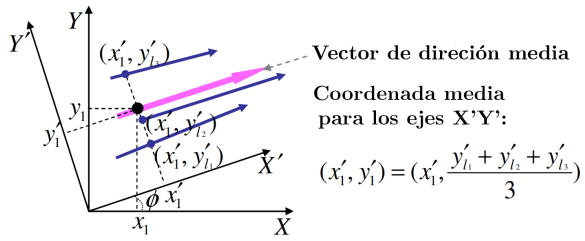


Figura 2.16: Rotación de los ejes X e Y

En el algoritmo 4 se puede observar la obtención de la trayectoria representativa. En primer lugar se obtiene el *vector de dirección media*. Después se ordenan los puntos de comienzo y fin por su coordenada en el eje rotado. Cuando se van escaneando los puntos ya ordenados, se cuenta cuantos segmentos se barren en cada punto y si es mayor que el valor de $MinLns$ se calcula la coordenada media avg_p en dicho punto.

Algoritmo 4 Obtención de la trayectoria representativa

INPUT: Un cluster C_i de segmentos, $MinLns$ y un parámetro de suavizado γ
OUTPUT: Una trayectoria representativa RTR_i de C_i ALGORITMO:

1: Calcular el *vector de dirección media* $\overrightarrow{\mathcal{V}}$;
2: Girar los ejes para hacer el eje X paralelo a $\overrightarrow{\mathcal{V}}$;
3: En la variable $\mathcal{P}$ se introducen los puntos de comienzo y fin de los segmentos del cluster C_i
4: /* X' es el valor de la coordenada en el eje X' (rotado)*/
5: Se ordenan los puntos de $\mathcal{P}$ por su valor de x';
6: **for each** $(p \in \mathcal{P})$ **do**
7: /* En num_p se almacena el numero de puntos obtenidos de hacer el barrido de los segmentos*/
8: Se establece en num_p el numero de segmentos que contienen el valor x' del punto p;
9: **if** $(num_p \geq MinLns)$ **then**
10: $diff$:= diferencia en el eje x' entre el punto p donde el barrido corta los segmentos y el anterior punto p;
11: **if** $(diff \geq \gamma)$ **then**
12: Se calcula la coordenada media avg_p' de los puntos en los que corta p en el eje x';
13: Se deshace la rotación de los ejes para obtener avg_p;
14: Se añade avg_p al final de RTR_i;
15: **end if**
16: **end if**
17: **end for**=0

Capítulo 3

Gestión del proyecto

En este capítulo se va a presentar la planificación del proyecto. Se describirá la metodología a seguir, así como la planificación temporal y la estimación presupuestaria sobre la cual se llevará a cabo el proyecto

3.1. Planificación temporal

En este caso, y debido a la lógica del proyecto, se utilizará un modelo de desarrollo en cascada. Para ello, se identificará un conjunto de fases que se seguirán de manera secuencial, estableciendo en cada una de las etapas un conjunto de metas, así como un conjunto de actividades, realizando una única iteración de cada una de ellas. Estas fases serán:

- **Investigación**: durante esta fase, el investigador, en este caso, un ingeniero informático junior, obtendrá el contexto necesario sobre el algoritmo de trayectorias para conocer este de manera precisa.

- **Análisis**: será llevada a cabo por un analista de software, quien planteará como deben funcionar las implementaciones y qué resultados se deben obtener de las mismas, así como los requisitos que debe cumplir el proyecto.

- **Programación**: en la cual se realizará la implementación del código y será realizada también por un ingeniero informático junior.

- **Pruebas**: en esta etapa volverá a el programador comprobará que los resultados obtenidos de la implementación son los esperados en el análisis y que la implementación se ha realizado de manera correcta.

A continuación, se podrá observar en la tabla 3.1, la planificación temporal establecida para el proyecto, desglosada por fases, así como los días en los que se estima empiece y acabe cada fase.

Sumando los días de la planificación, se estima una duración del proyecto de 160 días (20 días al mes y 8 horas al día, es decir, a tiempo completo). Para esta planificación temporal se ha obtenido el diagrama de Gantt de la figura 3.1, con una fecha inicial del día 14/10/2016 y de fin del día 25/05/2017.

Nombre	Duración	Inicio	Fin	Recurso
Investigación	80 días	14/10/2016 08:00 h	02/02/2017 17:00 h	Investigador
Análisis	30 días	03/02/2017 08:00 h	16/03/2017 17:00 h	Analista de software
Programación	30 días	17/03/2017 08:00 h	27/04/2017 17:00 h	Ingeniero informático
Pruebas	20 días	28/04/2017 08:00 h	25/05/2017 17:00 h	Ingeniero informático

Cuadro 3.1: Planificación temporal del proyecto

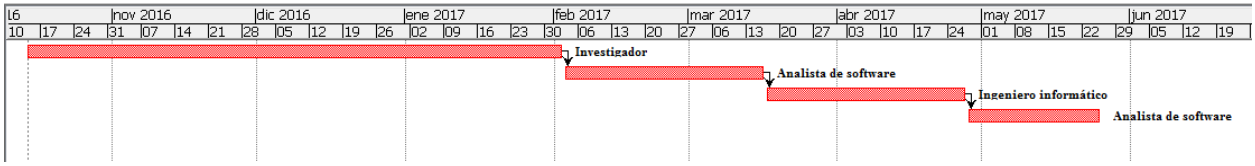


Figura 3.1: Diagrama de Gantt para la planificación temporal

3.2. Presupuestos

Puesto que este proyecto tiene una gran parte de investigación y de observación de resultados obtenidos, no se va a realizar un presupuesto basado en líneas de código o centrado en la programación, sino en las horas trabajadas por cada recurso, así como en el material utilizado. En primer lugar, en la tabla 3.8 se tiene el presupuesto del software. Para calcular este, se ha establecido un periodo de amortización del mismo para cada uno de ellos.

Herramienta	Precio total	Periodo de amortización	Tiempo de uso	Coste
Windows 10	135 €	24 meses	1280 h = 8 meses	45 €
RStudio	0 €	-	240 h = 1.5 meses	0 €
ShareLatex	0 €	-	120 h = 0.75 meses	0 €
			Total:	**45 €**

Cuadro 3.2: Presupuesto software

Por otra parte, en la figura 3.3, se puede encontrar el presupuesto del proyecto para el hardware, a los cuales se le ha añadido también un periodo de amortización para calcular de manera precisa el coste que supondrá su utilización.

Herramienta	Precio total	Periodo de amortización	Tiempo de uso	Coste
Ordenador personal	1200 €	48 meses	1280 h = 8 meses	200 €
Ratón USB	20 €	24 meses	1280 h = 8 meses	6.67 €
Conexión a internet	9 € / mes	-	1280 h = 8 meses	72 €
			Total:	**278.67 €**

Cuadro 3.3: Presupuesto hardware

Una vez que se tienen los costes de hardware y software, se va a proceder a definir los costes de recursos humanos. En la tabla 3.5 se puede ver el total obtenido para el recurso humano en el proyecto. Así como en la tabla 3.4 se puede ver el cálculo del salario bruto, que es lo que costará finalmente el salario de esa persona. Este se ha calculado en base al tipo de contrato realizado

al personal, ya que este es un contrato temporal a tiempo completo durante 8 meses, por lo que hay que añadir al sueldo neto un 6,7 % al sueldo neto, así como un 24,5 % correspondiente a la Seguridad Social y la paga extraordinaria correspondiente a los 8 meses del contrato.

Rol	Salario mensual (neto)	Salario por horas (neto)	Salario por horas (bruto)
Investigador	1500 €/mes	9.4 €/h	13.50 €/h
Analista software	1900 €/mes	11.875 €/h	17.06 €/h
Ingeniero informático	1700 €/mes	10.625 €/h	15.27 €/h

Cuadro 3.4: Salario presupuestado por horas

Una vez se ha calculado lo que cobrará cada participante del proyecto, y puesto que se dispone de las horas estimadas de trabajo, se tendrá el presupuesto total que suponen los recursos humanos del proyecto, como se puede ver en la tabla 3.5.

Rol	Salario mensual (bruto)	Horas trabajadas	Salario total
Investigador	2160 €	640 horas	8640 €
Analista software	2730 €	240 horas	2850 €
Ingeniero informático	2442.9 €	400 horas	4250 €
			Total: **15740.00 €**

Cuadro 3.5: Salario presupuestado para los recursos humanos

Finalmente, el presupuesto total estimado será la suma del presupuesto de software, hardware y el de los recursos humanos:

	Presupuesto final
Hardware	278.67 €
Software	45 €
Recursos humanos	15740.00 €
	Total: **16363.67 €**

Cuadro 3.6: Presupuesto final

3.3. Costes temporales

En la sección 3.1 se ha planificado el desarrollo completo del proyecto, de manera que recordando lo estimado, se tenía un total de 1280 horas de trabajo como cómputo global del tiempo invertido en cada una de las fases: Investigación, Análisis, Programación y Seguimiento. A diferencia de lo estimado, en la tabla 3.7 se puede visualizar el tiempo final empleado para cada una de las fases. La fecha de inicio para el proyecto es la misma que la estimada, pero sin embargo, la fecha de fin ha aumentado al 22/06/2017, aumentando el número de horas a 1440 horas de proyecto, lo que supone un 11.1 % de aumento de horas. El diagrama de Gantt obtenido se puede visualizar en la figura 3.2.

Nombre	Duración	Inicio	Fin	Recurso
Investigación	85 días	14/10/2016 08:00 h	09/02/2017 17:00 h	Investigador
Análisis	30 días	10/02/2017 08:00 h	23/03/2017 17:00 h	Analista de software
Programación	40 días	24/03/2017 08:00 h	18/05/2017 17:00 h	Ingeniero informático
Pruebas	20 días	19/05/2017 08:00 h	22/06/2017 17:00 h	Ingeniero informático

Cuadro 3.7: Planificación temporal del proyecto

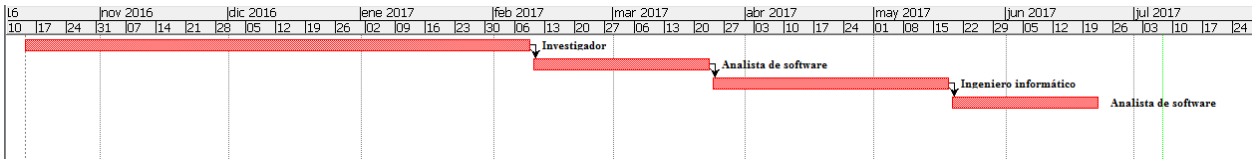


Figura 3.2: Diagrama de Gantt para los costes temporales

3.4. Costes económicos

A consecuencia del aumento temporal del proyecto, los costes económicos del proyecto han aumentado a su vez. A continuación, se pueden observar los costes que ha supuesto finalmente el proyecto para cada una de las partes: software, en la tabla 3.8, hardware en la tabla 3.9 y recursos humanos en la tabla 3.10 (teniendo en cuenta la tabla 3.4 para calcular los salarios).

Herramienta	Precio total	Periodo de amortización	Tiempo de uso	Coste
Windows 10	135 €	24 meses	1440 h = 9 meses	50.7 €
RStudio	0 €	-	320 h = 2 meses	0 €
ShareLatex	0 €	-	120 h = 0.75 meses	0 €
				Total: **50.7 €**

Cuadro 3.8: Costes de software

Herramienta	Precio total	Periodo de amortización	Tiempo de uso	Coste
Ordenador personal	1200 €	48 meses	1440 h = 9 meses	225 €
Ratón USB	20 €	24 meses	1440 h = 9 meses	7.5 €
Conexión a internet	9 € / mes	-	1440 h = 9 meses	81 €
				Total: **313.5 €**

Cuadro 3.9: Costes de hardware

Rol	Salario mensual (bruto)	Horas trabajadas	Salario total
Investigador	2160 €	680 horas	9180 €
Analista software	2730 €	280 horas	3325 €
Ingeniero informático	2442.9 €	480 horas	5100 €
			Total: **17605 €**

Cuadro 3.10: Coste de los recursos humanos

Por lo que finalmente en la tabla 3.11 se puede observar el coste total final del desarrollo del proyecto.

	Presupuesto final
Hardware	313.5 €
Software	50.7 €
Recursos humanos	17605 €
	Total: **17969.2 €**

Cuadro 3.11: Presupuesto final

Recordando el apartado de estimación de presupuestos, se había estimado un total de 16363.47 €, sin embargo el total real ha sido **17969.2 €**, lo que ha supuesto un aumento del 8.9 % respecto a lo estimado.

Capítulo 4

R y su entorno

Actualmente, debido al almacenamiento de grandes cantidades de datos en diferentes ámbitos tecnológicos, surge la necesidad de realizar análisis estadístico de los mismos, así como realizar diferentes tipos de gráficas, todo ello de la manera más simple y efectiva. Es en este punto donde nace este lenguaje de programación denominado R [25]. En este capítulo, en el primer apartado, se va a comenzar introduciendo R y explicando como funciona R, en qué se basa, que características tiene y por qué utilizarlo. En el segundo apartado se va a explicar en profundidad qué son los paquetes o librerías para R y, para finalizar el capítulo, se va a explicar cómo se pueden crear aplicaciones web interactivas en R, mediante una herramienta denominada "Shiny".

4.1. ¿Qué es R?

R es un lenguaje de programación que, como se mencionaba antes, surge debido a la necesidad de la transformación de cantidades ingentes de datos en información estructurada con la cual se pueda inferir algún conocimiento relacionado con el contenido de los datos. Para comenzar, se va a hablar de la historia de R y de cómo ha ido evolucionando a lo largo del tiempo.

4.1.1. Historia de R

R nace en 1992, en Nueva Zelanda, creado por Ross Ihaka y Robert Gentleman con la intención inicial de crear un lenguaje didáctico para ser utilizado en el curso *Introducción a la Estadística* de la Universidad de Nueva Zelanda. Ross y Robert decidieron adoptar la sintaxis del lenguaje S [19](creado en un principio por John Chambers, de los laboratorios Bell), por lo que la sintaxis de R es similar a la de S, pero la semántica, aunque es parecida, en ocasiones es diferente, sobre todo en detalles más específicos de programación.

Inicialmente y a modo de broma, Ross y Robert llamaron "R" al lenguaje que habían implementado debido a las iniciales de sus nombres, y desde entonces se le conoce así en la comunidad de dicho lenguaje.

Tras la creación de R en 1992, en 1993 se publica el software R. En el año 1995, Martin Mächler, de la Escuela Politécnica Federal de Zúrich, convence a Ross y Robert a usar la licencia GNU para que R sea un software libre. A partir de 1997, R pasa a formar parte del proyecto GNU (*GNU is Not Unix*, proyecto colaborativo de software libre, creado en 1983). Posiblemente, gracias a esta decisión actualmente se pueden encontrar numerosos cursos y tutoriales de R.

Versión	Descripción
0.16	Última versión alfa desarrollada exclusivamente por Ross y Robert
0.49 - 23 de abril de 1997	Versión más antigua de la que se conserva código. En esta misma fecha arranca CRAN, albergando 12 paquetes.
0.60 - 5 de diciembre de 1997	R se integra en el proyecto GNU.
1.0.0 - 29 de febrero de 2000	R se considera lo suficientemente estable para su uso en producción.
1.4.0	Se introducen métodos S4 y aparece la primera versión para Mac OS X.
2.0.0	Se introduce lazy loading, que permite una carga rápida de datos con un pequeño coste de memoria.
2.1.0	Aparece soporte para UTF-8 y comienzan los esfuerzos de internacionalización para distintos idiomas
2.9.0	El paquete 'Matrix' se incluye en la distribución básica de R.
2.11.0 - 22 de abril de 2010	Se incluye soporte para sistemas Windows de 64 bits
2.13.0 - 14 de abril de 2011	Se añade una nueva función al compilador que permite acelerar las funciones convirtiéndolas a byte-code.
2.14.0 - 31 de octubre de 2011	Se añaden espacios de nombres obligatorios para los paquetes. Se añade un nuevo paquete de paralelización a la distribución oficial
2.15.0 - 30 de marzo de 2012	Nuevas funciones de balanceo de carga. Mejorada la velocidad de serialización para grandes vectores.
3.0.0 - 3 de abril de 2013	Mejoras en GUI, funciones gráficas, gestión de memoria, rendimiento e internacionalización añadidos
3.4.0 - 21 de abril de 2017	Se habilita el compilador de código de bytes JTI ('Just In Time') en el nivel 3 de manera predeterminada

Cuadro 4.1: Evolución de R

Debido al auge que estaba teniendo R, se decidió crear una lista pública de correos 1996, a modo de soporte para el lenguaje. Puesto que la fama del lenguaje no dejaba de aumentar y los creadores no paraban de recibir correos, en 1997 decidieron crear dos listas de correos: R-help y R-devel, que son las que actualmente funcionan para responder las diferentes dudas que propongan los usuarios del lenguajes. Fue entonces el 29 de febrero del año 2000 cuando R es considerado un software completo y estable, liberando así la versión 1.0, teniendo actualmente a día 30 de junio de 2017 la versión 3.4.1.

En la tabla 4.1 se puede ver de manera más detallada cómo han ido evolucionando las versiones más estables con los cambios destacables en cada una de las versiones.

R actualmente es un proyecto vivo y sus capacidades no coinciden explícitamente con las de su padre S.

4.1.2. Características de R

Una vez se ha situado R en el tiempo, en esta sección se van a presentar las características más importantes de este lenguaje.

R es un conjunto de programas para la manipulación de datos, cálculo y gráficos. Entre otras características, destaca por que ofrece:

- almacenamiento y manipulación de los datos de manera efectiva,
- operadores para realizar cálculos sobre variables indexadas (arrays),
- una amplia colección de herramientas para el análisis de los datos,
- diferentes posibilidades gráficas para análisis de datos,
- un lenguaje de programación desarrollado, simple y efectivo, que incluye condiciones, ciclos, funciones recursivas y posibilidad de entradas y salidas.

Muchas veces se habla de R como un entorno, ya que este es un sistema diseñado, coherente y completo, a diferencia de otros entornos específicos para análisis de datos, que suelen ser agregaciones incrementales de diferentes herramientas específicas.

La utilización de R aporta muchas ventajas. Es software libre, lo que, por varios motivos, transmite valores socialmente positivos, como podría ser la libertad individual, el conocimiento compartido, la solidaridad y la cooperación, y además nos aproxima al método científico, porque permite el examen y mejora del código desarrollado por otros usuarios. Aparte de esta faceta de software libre, R tiene algunas ventajas específicas, por ejemplo, su sintaxis básica es sencilla e intuitiva, por lo que es más fácil familiarizarse con ella, lo que se traduce en un aprendizaje rápido y cómodo. Por otra parte, tiene una enorme comunidad de estructurada alrededor de la *Comprehensive R Archive Network*, más conocida como *CRAN* que desarrolla cada día nuevos paquetes con el fin de extender la funcionalidad y cubrir las necesidades computacionales y estadísticas de un científico o ingeniero.

Por lo que finalmente, se pueden concluir las siguientes características:

- Es un lenguaje robusto y efectivo, con una curva de aprendizaje medianamente compleja, pero cuando se aprende lo básico se puede proceder con materia más avanzada facilmente. Además está en constante evolución y se dispone de una amplia documentación del mismo.
- Facilita la preparación de los datos, tarea que requiere una amplia dedicación de cara a preparar la información para la visualización mediante la programación de *scripts*.
- Proporciona flexibilidad en lo que se refiere al tipo de archivo con el que puede funcionar: .txt,.csv,.json o EXCEL
- Gestiona un gran volumen de datos, ya que permite la implementación de paquetes que le aportan una capacidad de gestión de datos enorme. Esto es un aspecto importante en proyectos de gran volumen en los que la escalabilidad es un elemento clave.
- Es de código abierto y gratuito. R no tiene limitaciones.
- Capacidad de generar visualizaciones de información compleja de manera sencilla.

Sin embargo, R tiene una desventaja principal sobre la cual gira este proyecto, y esta es su velocidad de computación, ya que este es un lenguaje interpretado y debe traducirse cada vez que se ejecuta (este tema se trata en profundidad más adelante). Esto se debe a que R fue diseñado para hacer análisis de datos y estadísticas de manera más sencilla, sin tener en cuenta la computación.

4.1.3. Objetos en R

Ahora que se conocen las principales características de R, es conveniente presentar los objetos más simples que manipula R, a partir de los que posteriormente se generarán objetos más complejos. Estos objetos más simples son:

- `character` (cadenas de caracteres)
- `numeric` (números reales)
- `integer` (números enteros)
- `complex` (números complejos)
- `logical` (lógicos o booleanos, que solo toman valor *True* o *False*)

Sin embargo, R no manipula este tipo de datos de manera aislada, si no que lo hace dentro de los objetos más básicos denominados `vectores`. Un vector incluirá cero o más objetos pero todos de la misma clase, a diferencia de los objetos de tipo `list`, que pueden incluir componentes de clases distintas. A continuación, se describirán estos y otros tipos de objetos utilizados frecuentemente en R:

- **`Vectores`**: como se ha indicado, las clases de datos definidas antes, no se manejan de manera individual, por lo que cuando se declare una variable con un único número, R genera un vector de tamaño 1. Los vectores únicamente son unidimensionales.
- **`Matrices`**: al igual que los vectores, únicamente alojan objetos de una sola clase, y el acceso a los datos es similar a los vectores, sin embargo tienen dos dimensiones, con filas y columnas.
- **`Listas`**: que son objetos de la clase `list`, pueden contener cero o más elementos al igual que los vectores, pero con la diferencia de que pueden albergar diferentes clases de objetos con diferentes tamaños en su interior, ya sea una cadena de caracteres, un vector y un entero.
- **`Dataframes`**: son un tipo de objetos de la clase `list` pensados para trabajar con ellos a modo de tablas, por lo tanto todos los elementos que albergue este tipo de lista han de tener la misma longitud.
- **`Funciones`**: a diferencia de otros lenguajes de programación, como podría ser C o Java, en R las funciones constituyen una clase, por lo que se podría tener un vector de funciones. La lógica de las funciones sí que es la misma que en estos lenguajes, pueden recibir parámetros que realizan operaciones y pueden devolver objetos.

4.1.4. ¿Cómo trabaja R?

En esta sección se va a describir cómo funciona R.El hecho de que R sea un lenguaje de programación puede hacer pensar a algunos usuarios que no pueden utilizar R por lo que conlleva el concepto de "programación", pero con este lenguaje no se da el caso por las siguientes razones:

1. R es un lenguaje interpretado, no es un lenguaje compilado, lo que quiere decir que todos los comandos introducidos en un script de R se ejecutarán directamente a diferencia de otros lenguajes de programación, los cuales requieren la construcción de un programa completo (C, Fortran, Pascal...)
2. La sintaxis de R es muy simple e intuitiva. Por ejemplo, una regresión lineal se podría hacer con el comando `lm(x y)`, el cual dirá realízame la regresión lineal de y en base a x".

Cuando R se está ejecutando, las variables, datos, información, resultados, etc. se almacenan en la memoria principal del sistema en forma de objetos los cuales tienen un nombre. El usuario puede realizar acciones en dichos objetos utilizando diferentes operadores (aritméticos, lógicos, de comparación...) o funciones. En la figura 4.1 se puede ver una aproximación de como se ejecuta una función.

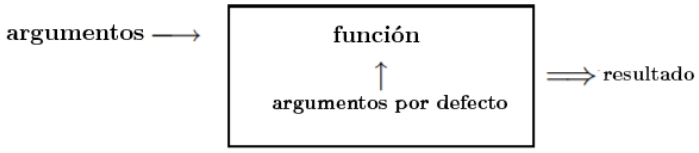


Figura 4.1: Ejecución de una función en R

Una función puede no requerir parámetros, de manera que el resultado de la función dependa de los argumentos definidos por defecto dentro de la función.

En R, todas las acciones se realizan sobre objetos cargados en la memoria principal del ordenador, no existen ficheros temporales para ello. El usuario ejecutará las funciones mediante determinados comandos que introducirá en R. De esta manera, los objetos resultantes de las funciones se mostrarán por pantalla, o se almacenarán en otro objeto de R de resultados, o se escribirán en disco (ya sea como fichero de datos o como gráfica). Puesto que los resultados siguen siendo objetos, estos mismos son considerados como datos sobre los que aplicar las funciones. Los ficheros de datos pueden ser leídos de manera local con ficheros locales de datos, o de manera remota desde internet. En la figura 4.2 se puede ver un esquema de lo anterior.

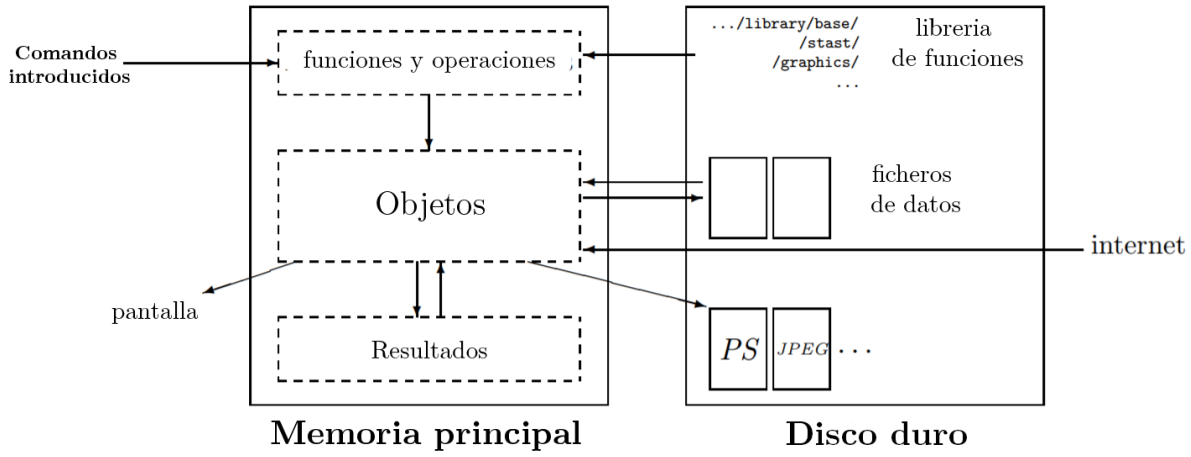


Figura 4.2: Esquema de funcionamiento de R

Las funciones disponibles se almacenan en una librería local en el disco duro en el directorio *RHOME/library*, donde *RHOME* es el directorio donde se ha instalado R. Este directorio incluirá los paquetes con las funciones, que a su vez estarán organizados en directorios.

4.2. Paquetes en R

4.2.1. ¿Qué son los paquetes de R?

Los paquetes, también conocidos como librerías, son colecciones de datos y funciones desarrollados por la comunidad de R. Con ellos, se incrementa el potencial de R, mejorando la funcionalidad que incluye R en su distribución base. Actualmente, el repositorio oficial de paquetes para R (*CRAN*) alcanza lo 10000 paquetes publicados, a los que hay que añadir el resto de paquetes en repositorios diferentes a *CRAN*.

Un paquete es una manera apropiada de organizar código, y si procede compartirlo con el resto de la comunidad. La unidad básica de compartir código la componen los paquetes, e incluirán código (no solo va a ser en R), documentación explicativa del código y de las funciones, algunas pruebas de como han de ejecutarse las funciones y los conjuntos de datos. Más adelante se hablará de la estructura de los directorios que ha de seguir un paquete en R.

Mediante el uso de paquetes podremos realizar acciones que quedarían lejanas si el uso de R se limitase únicamente a su distribución base. Algunas de estas acciones son:

- **Carga de datos**: de manera que se quieran obtener los datos con los que trabajar de diferentes fuentes de datos, como puede ser:
  - De una base de datos, para lo que utilizaremos paquetes como *ROBDC, RMySQL, RSQLite*
  - De una hoja de cálculo, los cuales nos permitirán leer y escribir archivos de Microsoft Excel o cualquier otro programa que permita manipular archivos en formato `.xls`, a su vez que permitirán exportarlos a formato `.csv`, para lo que existen paquetes como *XLConnect o xlsx*.
  - De un conjunto de datos en SAS o SPSS, para lo que existe *foreign*.
- **Manipulación de datos**: con el fin de obtener subconjuntos, o unir conjuntos de datos de manera rápida y eficaz (*dplyr*).
- **Visualización de datos**: con el fin de ofrecer gráficas lo más precisas posibles de la manera más eficiente posible. (*ggplot2, plotly, leaflet, DT, diagrammeR, network3D,googleVis*.
- **Creación de informes**: lo que permite crear un informe final con los resultados obtenidos, llegando a ser este interactivo *R Markdown, shiny*. De este último se hablará en el apartado siguiente.
- **Visualización de datos geográficos**: para obtener mapas, como pueden ser *ggmap, maps o maptools*.
- **Desarrollo de paquetes**: para realizar paquetes para R, por ejemplo *devtools, testthat o roxygen2*.

Como se puede observar, existen muchas posibilidades de utilización de R gracias a los paquetes desarrollados por la comunidad. Esto, como ya se ha comentado, surge de que R se caracteriza por ser un lenguaje de código abierto, sin limitaciones, lo que supone aportar facilidades a los desarrolladores.

4.2.2. Repositorios

Un repositorio es un lugar donde se alojan paquetes y desde el cual se pueden instalar. Algunos de los repositorios más famosos de paquetes para R son:

- **CRAN**, que es el repositorio oficial y es una red de FTP y servidores web mantenidos por la comunidad de R alrededor de todo el mundo. Está gestionado por la fundación de R, y para publicar un paquete en dicho repositorio es necesario pasar ciertos tests para asegurarse de que un paquete cumple las políticas de CRAN
- **Bioconductor**, el cual es un repositorio clásico, cuya intención es alojar codigo abierto para bioinformática. Al igual que CRAN, tiene un proceso de súbida y revisión de los paquetes que allí se alojan.
- **GitHub**: a pesar de que no es específico de R, es probablemente el mayor repositorio para proyectos de código abierto. Su popularidad parte de que no existen limitaciones de espacio para subir código abierto. Además, la integración con git (software para control de versiones) y la facilidad de compartir y colaborar con el resto de usuarios han influido en su fama.

4.2.3. Estructura de un paquete

Todo paquete de R sigue una estructura mínima, así como otra parte del mismo de carácter opcional. En la parte de aparición obligatoria se debe encontrar:

- Un directorio llamado `R/` que va a incluir los ficheros con los scripts de R en los que se escriban las funciones.
- Un fichero llamado `DESCRIPTION` con metadatos del paquete. Este debería tener la estructura de la figura 4.3.

En la parte de ficheros y directorios opcionales a incluir, podemos encontrar:

- El directorio `man`, el cual incluirá ficheros de documentación para los objetos creados en el paquete en formato de Documentación de R (.Rd). En caso de incluir ficheros que no fuesen de documentación, el paquete daría un error en la instalación.
- En el directorio `src` se incluirá código fuente realizado en otros lenguajes de programación, como puede ser C, C++ o FORTRAN. Los ficheros que aquí se incluyan tienen la particularidad de que necesitan ser compilados para su ejecución.
- El directorio `data` poseerá los ficheros con datos necesarios para la ejecución del paquete así como para las demostraciones.
- El directorio `demo` debe incluir los scripts de R que muestran alguna funcionalidad del paquete.
- El fichero `NAMESPACE` describe que funciones de las descritas estarán accesibles por el usuario final que vaya a utilizar el paquete.

En la figura 4.4 se puede observar la estructura interna de un paquete de ejemplo denominado *squaresPack*. Se puede ver como los ficheros `DESCRIPTION`, `NAMESPACE` y los directorios `R` y `man` están a la misma altura. Dentro de R encontramos los scripts `addsquares.R` y `substractSquares.R`, para los cuales, en el directorio `man` se encuentran sus archivos de documentación donde se informará sobre cómo utilizar dichos scripts.

```
Package: pkgname
Version: 0.5-1
Date: 2015-01-01
Title: My First Collection of Functions
Authors@R: c(person("Joe", "Developer", role = c("aut", "cre"),
                    email = "Joe.Developer@some.domain.net"),
             person("Pat", "Developer", role = "aut"),
             person("A.", "User", role = "ctb",
                    email = "A.User@whereever.net"))
Author: Joe Developer [aut, cre],
  Pat Developer [aut],
  A. User [ctb]
Maintainer: Joe Developer <Joe.Developer@some.domain.net>
Depends: R (>= 3.1.0), nlme
Suggests: MASS
Description: A (one paragraph) description of what
  the package does and why it may be useful.
License: GPL (>= 2)
URL: https://www.r-project.org, http://www.another.url
BugReports: https://pkgname.bugtracker.url
```

Figura 4.3: Ejemplo de un fichero `DESCRIPTION` de un paquete en R

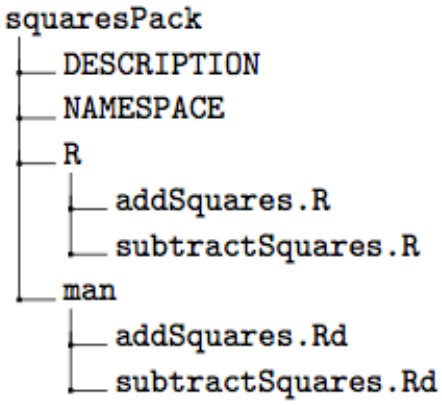


Figura 4.4: Ejemplo de estructura de un paquete

4.3. ¿Qué es Shiny?

Shiny es un paquete de R que se puede encontrar en CRAN y permite crear aplicaciones web de una manera sencilla, es decir, permite convertir aplicaciones clásicas de scripts de R en aplicaciones web interactivas para el uso de cualquiera sin necesidad de tener conocimiento de HTML o JavaScript para desarrollarlas.

Se caracteriza por:

- Estar desarrollado por RStudio (el *IDE* más conocido de desarrollo en R)

- La curva de aprendizaje es poco pronunciada, ya que está pensado para usuarios de R
- No es necesario conocer HTML, CSS o JavaScript
- R se encuentra completamente integrado en él.
- Utilizar la versión gratuita de código abierto de RStudio como servidor no dedicado y levanta las aplicaciones en local.

Puesto que se puede encontrar en CRAN, con el siguiente código de R estaría todo preparado para comenzar a utilizar Shiny:

```
install.packages("shiny")
library(shiny)
```

4.3.1. Arquitectura de Shiny

Shiny sigue la misma arquitectura que cualquier otra aplicación web, es decir, se tiene una parte de cliente y otra de servidor, conectadas entre sí bidireccionalmente. En este caso, se tendrá un archivo `server.R` que tendrá la parte de back de la aplicación, es decir, la parte servidor de la aplicación y un archivo `ui.R` que será la parte de front de la aplicación, es decir, la parte cliente.

Para levantar la aplicación se tienen tres opciones:

- En un servidor local con RStudio.
- En un servidor linux con un Servidor Shiny de RStudio instalado en él.
- En algún servicio de alojamiento especializado para Shiny, como por ejemplo ShinyApps.io[1]

Aunque no sea necesario conocer los lenguajes de desarrollo de aplicaciones web HTML, CSS o JavaScript, Shiny genera al final su parte frontal en este código desde el archivo `ui.R`. En la figura 4.5 se puede ver una aproximación de lo que sería la arquitectura de una aplicación Shiny.

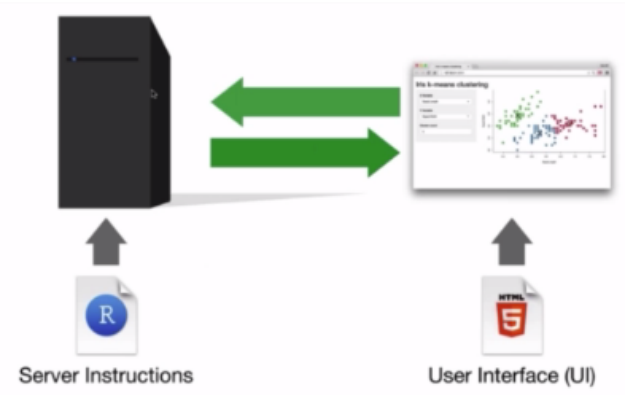


Figura 4.5: Aproximación de arquitectura de una aplicación Shiny

[1]ShinyApps.io - https://www.shinyapps.io/

4.3.2. Componentes en Shiny

Para el desarrollo de la aplicación, Shiny proporciona diferentes conjuntos de componentes que conforman la aplicación:

- **Componentes de Entrada**: que son aquellos mediante los cuales el usuario introduce los datos e interactúa con la aplicación en la parte de cliente y en función del valor que toman, se ejecutan las funciones de la parte de servidor (que mediante los componentes de salida mostrará datos). En la figura 4.6 se pueden observar diferentes tipos de componentes.

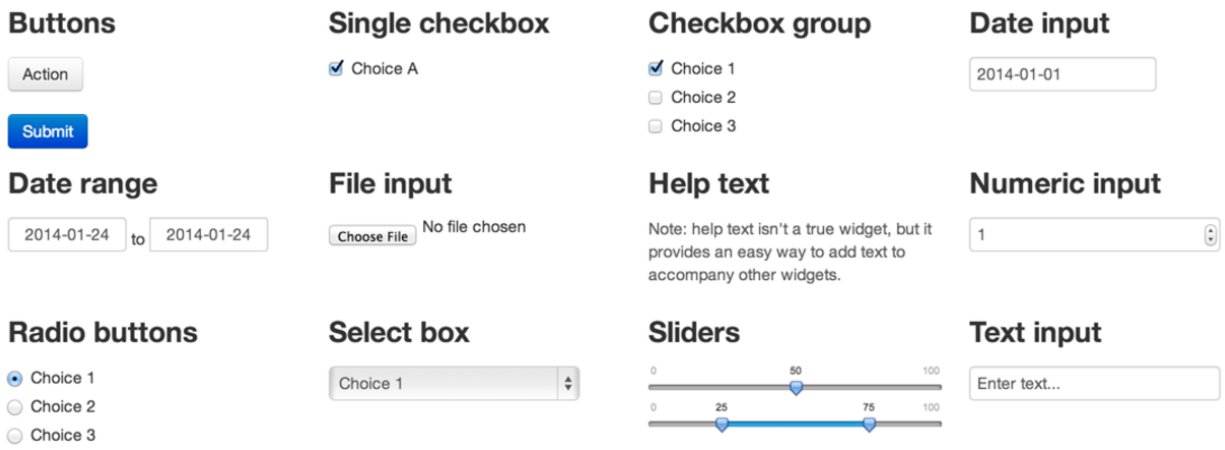


Figura 4.6: Algunos componentes de entrada para Shiny

- **Componentes de salida**: están presentes en la parte frontal de la aplicación (que es donde se renderizan) y su valor se determina desde la parte de servidor. Estos pueden ser gráficas, imágenes, texto, tablas, etc.

En las figuras 4.7, 4.8 y 4.9 se puede ver algún ejemplo de aplicación web Shiny con diferentes componentes de entrada y de salida.

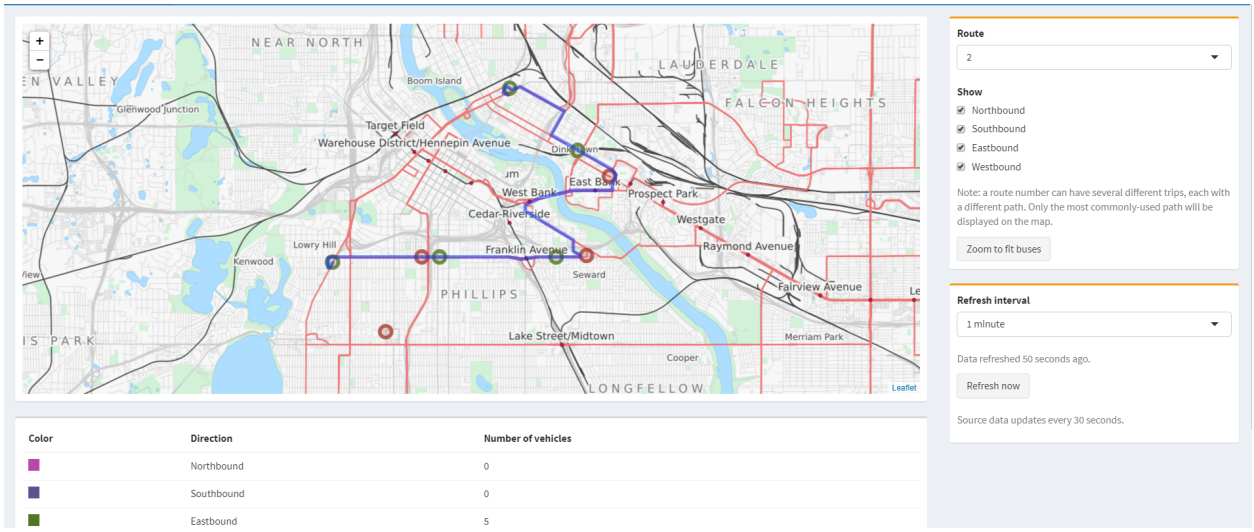


Figura 4.7: Ejemplo de aplicación en Shiny

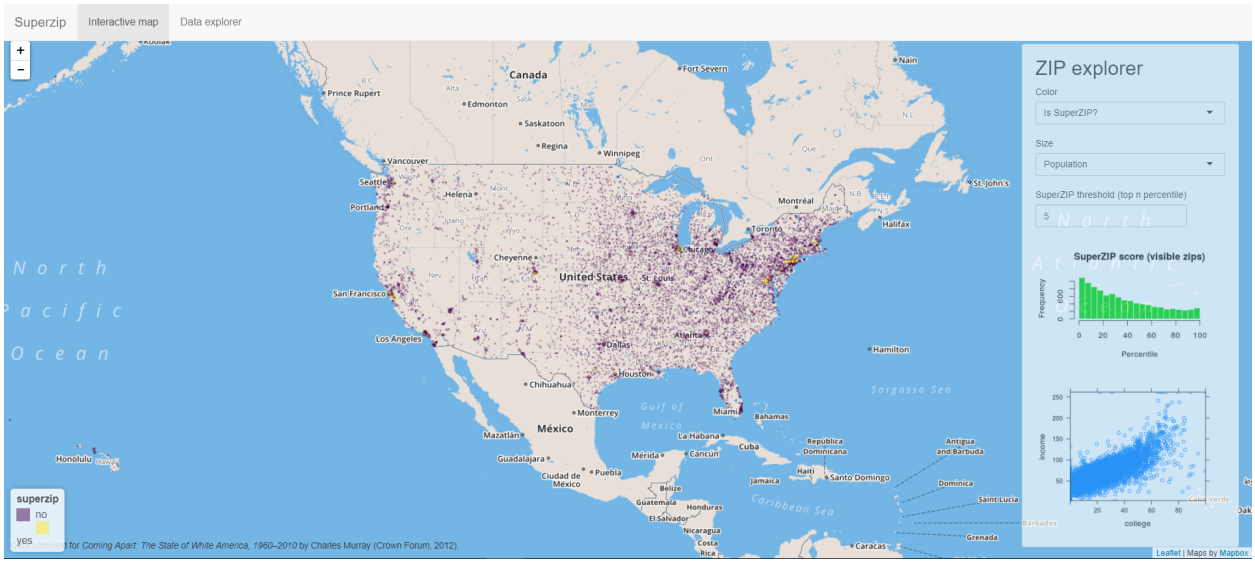


Figura 4.8: Ejemplo de aplicación en Shiny

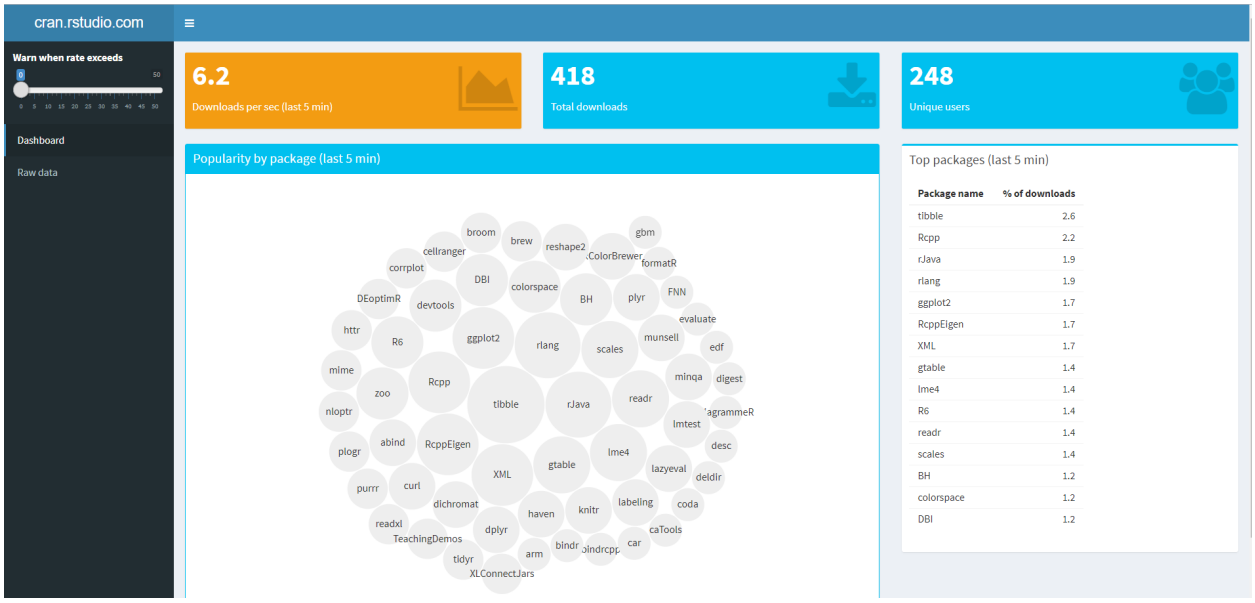


Figura 4.9: Ejemplo de aplicación en Shiny

Capítulo 5

Implementación

Una vez que se conoce qué se va a implementar y cuales van a ser las herramientas a utilizar, se va a proceder a la implementación. En esta sección se va a comenzar introduciendo brevemente el IDE que se ha utilizado para después, presentar, en el mismo orden en el que se presentaban las herramientas, cómo se han utilizado, siendo el orden: R, paquete de R y Shiny.

5.1. RStudio

A pesar de que la instalación de R nos genera una consola de ejecución de comandos, no ofrece facilidades a la hora de realizar debugging, así como a la hora de crear proyectos formados por diferentes scripts de R y a la de visualizar gráficas. Para ello RStudio aporta diferentes facilidades.

5.1.1. ¿Qué es RStudio?

RStudio es un Entorno Desarrollo Integrado (*Integrated Development Enviroment, (IDE)*) con el principal objetivo de facilitar el desarrollo en R. Fue desarrollada por Joseph J. Allaire y está implementado en los lenguajes de programación Java, C++ y JavaScript, utilizando el framework Qt para su interfaz de usuario. RStudio está disponible en dos versiones: versión de escritorio, en la cual RStudio se ejecuta de manera local como cualquier otra aplicación de escritorio, y en su versión servidor (*RStudio Server*), que permite acceder a RStudio a través de un navegador instalando el servidor en un servidor Linux remoto.

RStudio en su versión de escritorio está disponible para Windows, macOS y Linux, y se caracteriza por:

- Resaltar la sintaxis, auto completar el código y generar sangría inteligente en los scripts.
- Ejecutar código directamente desde el editor del que dispone.
- Saltar entre funciones definidas facilmente.
- Mostrar el historial de comandos ejecutados.
- Integrar directamente la ayuda de los paquetes.
- Visualizar los paquetes instalados y/o cargados.
- Permitir tener abiertos varios scripts
- Integrar control de veriones (git, svn, etc)

- Ser de código abierto a todo el mundo en el repositorio de GitHub.[1]

5.1.2. Interfaz de RStudio

En esta sección se va a explicar de manera breve la interfaz de RStudio. En la figura 5.1 se puede ver la interfaz.

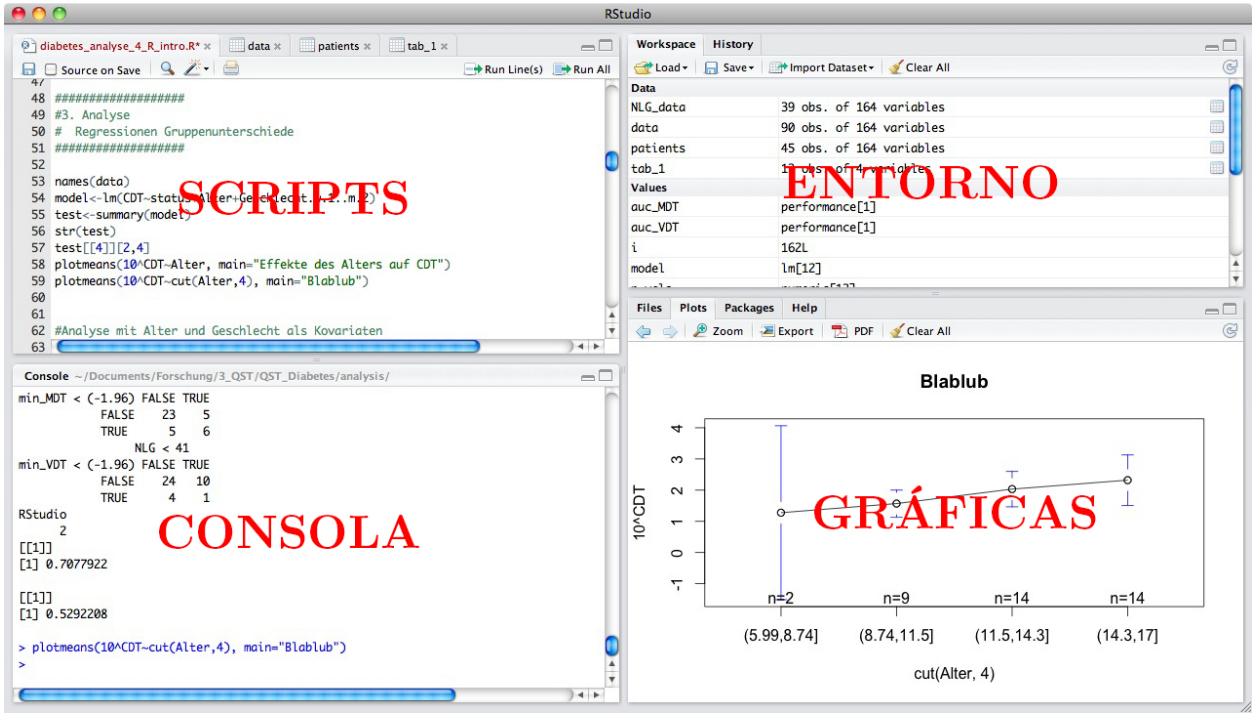


Figura 5.1: Interfaz de RStudio

Se pueden distinguir cuatro componentes principales:

- En la parte de los **scripts**, es donde RStudio permite generar varios y programar varios scripts en R, así como editar otro tipo de archivos (Rmarkdown, latex, css, etc.).
- En la parte de **entorno**, es donde se pueden ver las variables que se tienen declaradas en memoria principal, así como las funciones, es decir, aquí se pueden ver los objetos que se tienen en el entorno de la memoria principal de manera cómoda.
- En el componente de la **consola** se verán las salidas generadas por los scripts y dónde se podrán ejecutar comandos de R. Es similar a la consola que crea R al instalarlo en el el equipo.
- En la parte de **gráficas** se visualizarán las gráficas generadas desde R y aquí se podrá ver también que paquetes se tienen instalados y/o cargados en la memoria principal, ayuda de R con las funciones de los paquetes y los ficheros que se tienen en el directorio actual al igual que se podría visualizar en cualquier otro explorador de archivos.

Una vez se ha establecido y explicado qué herramientas se van a utilizar para la implementación (R, paquetes de R y Shiny) y dónde se va a realizar la implementación (RStudio), se va a proceder a explicar qué se va a implementar y cómo se va a realizar en las siguientes secciones del presente capítulo.

[1] https://github.com/rstudio/rstudio

5.2. Traclus en R

En el capítulo 2 se ha presentado el algoritmo de clustering de trayectorias *TRACLUS*, donde se ha podido ver en el algoritmo 1 en *pseudo*-código de *TRACLUS*, en el que se hacía mención a otros *sub*-algoritmos, que, recordando la descripción del algoritmo, se tenían tres métodos principales: particionado, clustering y obtención de la trayectoria representativa, que se corresponderán con los *sub*-algoritmos.

En esta sección, la tarea principal es transformar el algoritmo 1, que se tenía en *pseudo*-código en un script de R con las funciones necesarias para que la salida de particionar, agrupar y obtener la trayectoria representativa sea la esperada.

5.2.1. Particionado de trayectorias en R

Recordando el algoritmo de particionado, este recibía como parámetro una trayectoria TR_i formada por un conjunto de puntos $\{p_1p_2p_3...p_j...p_{len_i}\}$.

Para la implementación del método de particionado se ha implementado una función llamada `partition_flights_route`, que quedaría de la siguiente manera:

```
partition_flights_route<-function(dep,des, scale=FALSE)
```

Esta función recibe por parámetro:

- El aeropuerto de origen en formato ICAO (variable `dep`)
- El aeropuerto de destino en formato ICAO (variable `des`)
- Un booleano por si se desea escalar los datos [4] (variable `scale`), que inicialmente toma valor `FALSE`, es decir, que no se escalan los datos a no ser que se indique lo contrario.

Una vez que se le indican los aeropuertos, se filtran las trayectorias en función del vuelo indicado y se particiona cada una de ellas con la función:

```
ApproximateTrajectoryPartitioning(D, scale=FALSE)
```

Que recibe por parámetro:

- Un dataframe con la longitud y la latitud de cada una de las trayectorias TR_i filtradas (variable `D`)
- Un booleano que indica si se escalan los datos o no (variable `scale`)

Esta función es computacionalmente la más cara, ya que tarda una media de 30 minutos en ejecutarse. Devuelve un dataframe con los puntos característicos de cada una de las trayectorias, tal y como se indicaba en el apartado 2.3.3.[2]

Una vez se tienen los puntos característicos de la trayectoria, se generan los segmentos formado por dichos puntos característicos mediante la función:

```
GetLineSegments(CP)
```

Que recibe por parámetro un conjunto de puntos y devuelve un dataframe con los segmentos formados. Una vez formados, estos segmentos se unen a los de la trayectoria anterior, asignando un id de trayectoria a cada uno de los segmentos para diferenciarla de la siguiente trayectoria, de manera que la función final `partition_flights_route` devuelva todos los segmentos particionados de la trayectoria junto a su identificador.

[2]Para obtener el dataframe final se han implementado funciones auxiliares para obtener la distancia, así como para calcular la longitud mínima descriptiva (véase Algoritmo 2 del apartado 2.3.3.)

5.2.2. Agrupamiento de trayectorias en R

Ahora que se tiene el dataframe con los segmentos resultantes de particionar la trayectoria, se va a proceder a realizar el agrupamiento de los mismos. Para ello, se ha implementado la función siguiente:

```
LineSegmentClustering<-function(D, epsilon, MinLns, w_dist)
```

La cual recibirá como parámetros:

- Un dataframe con los segmentos con punto de inicio y fin de cada uno de ellos (variable `D`)
- Un número double que determinará el valor de epsilon (variable `epsilon`)
- Un número entero que determinará el valor de MinLns (trayectorias mínimas para formar un cluster) (variable `MinLns`)
- Un vector de números double con los pesos de cada una de las distancias y que entre los tres valores sume 1 (variable `w_dist`)

Esta función añadirá una columna al dataframe resultante de particionar, de manera un número a cada segmento en función del cluster al que pertenezca. Si un segmento se desestima como parte de un cluster, el valor de este en la nueva columna será `"desestimated"` y si se considera ruido será `"noise"`. Para llegar a obtener la nueva columna, han de seguirse algunos pasos.

En primer lugar se calcula el valor de $N_\epsilon(L)$ - *Epsilon Neighborhood*, para cada uno de los segmentos del dataframe `D` y se les asigna un identificador de cluster en el caso de que pase a formar parte de alguno de los cluster. A continuación, se expande el cluster como se indicaba en el algoritmo 3, en el método ExpandCluster.

Si el número de segmentos resultantes de calcular $N_\epsilon(L)$ - *Epsilon Neighborhood* es menor que el valor de la variable `MinLns`, se añade en el la posición pertinente en la nueva columna el valor `"noise"`.

Finalmente para cada uno de los clusters, con la siguiente función:

```
PTR(D, i)
```

Siendo el parámetro `D` el dataframe con todos los segmentos y la variable `i` el identificador del cluster, se calculará el número de trayectorias participantes en él. Esta devolverá un número entero. Si este número es menor que la variable `MinLns`, en todas las posiciones de los segmentos que forman parte de dicho cluster se insertará el valor `"desestimated"`.

Tras esto, se dispondría de un dataframe D, el cual es devuelto por la función `LineSegment-Clustering` y estaría formado por los segmentos con su punto de inicio y fin, cada uno con su id de trayectoria y su id de cluster correspondiente.[3]

Esta función es considerada computacionalmente cara, ya que tarda, aproximadamente, 13 minutos en completar su ejecución.

[3]Al igual que al particionar, se han implementado funciones auxiliares para calcular $N_\epsilon(L)$ - *Epsilon Neighborhood* y `ExpandCluster`

5.2.3. Obtención de la trayectoria representativa en R

Ya con el dataframe y con los clusters formados, se procede a obtener la trayectoria representativa para cada uno de los clusters. Para ello, se ha declarado la función:

```
RepresentativeTrajectoryGeneration<-function(cluster, MinLns, gamma)
```

Siendo los parámetros que recibe:

- `cluster` es un numero entero con el id del cluster que se va a calcular la trayectoria representativa
- `MinLns` es un numero entero con el mínimo de segmentos necesarios para formar cluster, que a su vez será utilizado para obtener los puntos de la trayectoria representativa del cluster.
- `gamma` que será un número double que determine el suavizado para la obtención de los puntos de la trayectoria

Esta función devolverá un dataframe con los puntos que forman la trayectoria representativa para el cluster indicado. Para obtener los puntos se transforman, los datos de acuerdo al algoritmo 4.

Esta función no se considera computacionalmente cara, puesto que no alcanza los 2 segundos de ejecución en total.

Con lo anterior se tendría *TRACLUS* implementado en lenguaje R, sin embargo, surge un problema fruto de que R es un tipo de lenguaje interpretado, esto quiere decir, necesita un interprete que traduzca el código cada vez que se ejecute, por lo que el tiempo de ejecución aumenta considerablemente. En este caso, la suma total del tiempo de ejecución asciende a 40 minutos.

5.3. Creación de un paquete con R y C

Ahora que se ha implementado el algoritmo y se ha visto que surgen problemas con las funciones de particionado y agrupamiento, y puesto que la creación de paquetes para R permite la introducción de código en lenguaje C como se podía ver en el apartado 4.2.3, se va a proceder por dicha vía.

5.3.1. Observaciones

A la hora de la implementación, y debido a ciertas limitaciones que presenta el lenguaje C, se han tenido que crear diferentes tipos de objetos con los que trabajar. Por motivos de comodidad a la hora de programar, se creo el objeto de tipo `struct` denominado `LS` (Line Segment).

```
typedef struct LineSegments{
  double start1;
  double start2;
  double end1;
  double end2;
  int TrajId;
  int cluster;
} LS;
```

De esta manera, se tendrá un objeto de tipo `LS` representando a los segmentos procedentes de particionar. El segmento estará formado por dos variables de tipo `double` que representan el punto de inicio, dos variables de tipo `double` que representan el punto de fin, un entero con el id de la trayectoria y un entero con el id del cluster al que pertenece.

Por otra parte, y siendo este mayor problema, C no dispone de arrays dinámicos, es decir, creando arrays por defecto se debe asignar un tamaño fijo, por lo que se ha creado un objeto de tipo `struct` que simula un array dinámico:

```
typedef struct {
  double *array;
  size_t used;
  size_t size;
} Array;
```

Por lo tanto, el array estará compuesto por un puntero a un objeto de tipo double y dos variables de tamaño: el tamaño usado y el tamaño real del array. Va acompañado de los siguientes métodos:

```
void initArray(Array *a, size_t initialSize) {
  a->array = (double *)malloc(initialSize * sizeof(double));
  a->used = 0;
  a->size = initialSize;
}

void insertArray(Array *a, double element) {
  // a->used is the number of used entries, because a->array[a->used++] updates
      a->used only *after* the array has been accessed.
  // Therefore a->used can go up to a->size
  if (a->used == a->size) {
    a->size *= 2;
    a->array = (double *)realloc(a->array, a->size * sizeof(double));
  }
  a->array[a->used++] = element;
}

void freeArray(Array *a) {
  free(a->array);
  a->array = NULL;
  a->used = a->size = 0;
}
```

De manera que para comenzar a utilizar el array ha de inicializarse con el método `initArray`, indicandole el array que se quiere inicializar en el parámetro `*a` e indicando en el parámetro `initialSize` el tamaño inicial que se desea que tenga el array. Para insertar un objeto en el array, se ha de utilizar el método `insertArray`, el cual tiene que recibir por parámetro el array en el que ha de insertarse un objeto (variable `*a`) y en la variable `element` el objeto a insertar. Al insertar el objeto, se comprueba si el tamaño usado es igual al tamaño que ocupa el array y si esto se cumple se duplica el tamaño del array. Con el método `freeArray`, se libera el espacio no utilizado.

Para explicar la creación de arrays dinámicos en R, se ha hecho con objetos de tipo double, pero este tipo de implementación ha sido escalada a objetos de tipo `int` y a objetos de tipo `LS`, teniendo así dos nuevos objetos más: `ArrayInt` y `ArrayLS`, que alojarán de manera dinámica objetos `int` y `LS`.

5.3.2. Particionado en C

Como buenas prácticas de implementación en C, por cada archivo con extensión `.c`, que incluirá el código, debe existir un fichero con extensión `.h`, que incluirá las cabeceras. En el caso del particionado, se ha creado un fichero denominado `approximate_partitioning.c` que va a acompañado de su archivo de cabecera `approximate_partitioning.h`.

Dentro del fichero `approximate_partitioning.c` se podrá encontrar, en primer lugar, la función:

```
double MDLpar_ls(double *TRpoints,int Index1, int Index2, int *len);
```

Con esta función se devolverá el coste de particionar, tal y como se indicaba en *pseudo*-código en el algoritmo 2. Este valor que devuelve la función será de tipo `double`, recibiendo este un array de `double` con los puntos de las trayectorias y tres enteros. Por otra parte, y siendo el grueso del archivo, se encuentra la función:

```
void ApproximateTrajectoryPartitioning(double *TRpoints, int *check_scale, int *len,
    double *lng_return,double *lat_return);
```

Este método en C es de tipo void ya que no devuelve ningún valor. La obtención de la transformación de los datos se realiza pasando esta por referencia en memoria a través del uso de punteros. De esta manera, se obtendrán los arrays con la longitud y latitud de los puntos característicos de cada una de las trayectorias. Teniendo el array con los valores de longitud y latitud de los puntos característicos, se procederá de igual manera que se hacía en la implementación únicamente con R.

Al llamar a la función de C desde R se ha hacer con la función `.C()`. La función `ApproximateTrajectoryPartitioning` de R quedaría de la siguiente manera utilizando la función de C:

```
ApproximateTrajectoryPartitioning <- function(TRpoints,scale = FALSE){
  a<- .C("ApproximateTrajectoryPartitioning",
         as.double(TRpoints),
         as.logical(scale),
         as.integer(length(TRpoints)/2),
         double(length = length(TRpoints)/2),
         double(length = length(TRpoints)/2))
  return(a)
}
```

De esta manera, `.C` devolverá un objeto de tipo `list` donde cada una de las columnas es cada una de las variable que toman valor por referencia dentro de la función `ApproximateTrajectoryPartitioning` de C. Por lo tanto, para acceder a la columna de la longitud de los puntos característicos, (variable `double *lng_return` recibida por parámetro), bastará con ejecutar el comando **`CP[[4]]`**, donde CP es la variable que devuelve `ApproximateTrajectoryPartitioning` en R en la anterior función.

Entonces, en el script de R que tenía la implementación anterior con la función `partition_flights_route`, bastaría con sustituir las declaraciones de `ApproximateTrajectoryPartitioning` por la que incluirá el paquete. [4]

5.3.3. Agrupamiento en C

En este caso, el fichero creado es `line_segment_clustering.c`, que va acompañado de su fichero correspondiente con las cabeceras, `line_segment_clustering.h`. En este archivo se han creado varias funciones, siendo las una de ellas:

```
ArrayInt eps_neighborhood(int LSId, ArrayLS D, double *epsilon, double *w_dist,
    int *numLS);
```

[4]Las funciones auxiliares que se utilizaban en R se han replicado en C, por ejemplo, para calcular la distancia y, en este caso, se ha replicado la función `scale` de la distribución base de R. [4]

Esta función recibe por parámetro un entero con el id del segmento sobre el que se va a calcular $N_\epsilon(L)$ - *Epsilon Neighborhood*, un objeto del tipo creado ArrayLS, con el conjunto de todos los segmentos en la variable D, el valor de epsilon, de w_dist y el número de segmentos en la variable `numLS`. Así mismo, `eps_neighborhood` devolverá un objeto de tipo ArrayInt, con un conjunto de enteros que se corresponderán con los ids de los segmentos que entran dentro de $N_\epsilon(L)$ (véase apartado 2.3.4).

Por otra parte, y siendo esta más importante, se ha declarado otra función:

```
void LineSegmentClustering(double *start1, double *start2, double *end1, double *
    end2, int *TrajId, double *epsilon, int *minLns, double *w_dist, int *numLS,
    int *clusterstoR);
```

Dentro de esta función se va a ejecutar la función anterior, `eps_neighborhood`, de cara a obtener los clusters según los ids de segmentos devueltos por la misma. En cuanto a los valores que recibe:

- Los puntos de longitud y latitud de inicio y fin de los segmentos, que se corresponden con `start1, start2` siendo estos los puntos de inicio y `end1, end2`, siendo estos los de fin.
- El valor de epsilon en la variable `epsilon`, siendo esta de tipo double.
- Un entero con el valor de MinLns en la variable `minLns`.
- Un vector con los pesos de las distancias en formato double en la variable `w_dist`.
- Un entero con el número de segmentos en la variable `numLS`.
- Un array de enteros el cual se cogerá luego desde R y que tomará el valor de los clusters a los que pertenece cada uno de los segmentos.

Puesto que en C el manejo de cadenas de caracteres resulta una tarea complicada, a la hora de clasificar los cluster (como se indicaba en la sección 5.2.2 con `"desestimated"`, `"noise"` y `ünclassified"`), se les ha asignado un valor númerico a cada uno de los diferentes valores que en R se trataban como cadenas de caracteres. Para ello, se han creado las siguientes constantes:

```
#define UNCLASSIFIED -3
#define DESESTIMATED -2
#define NOISE -1
```

De esta manera, cuando se obtenga el array con los clusters a los que pertenece, habrá que tener en cuenta dicha transformación de su valor entero proporcionado desde C a su valor de cadena de caracteres en R. Esta transformación ha de hacerse en R, y finalmente la función que se podrá ejecutar desde R, manipulando los datos en C, se ha declarado de la siguiente manera:

```
LineSegmentClustering<-function(datos, epsilon, MinLns, w_dist){
  A <- datos$start[,1]
  B <- datos$start[,2]
  C <- datos$end[,1]
  D <- datos$end[,2]
  y <- datos$TrajId

  a <- .C("LineSegmentClustering",as.double(A), as.double(B), as.double(C), as.
      double(D),as.integer(y),
          as.double(epsilon),
          as.integer(MinLns),
```

```
            as.double(w_dist),
            as.integer(length(A)),
            integer(length = length(A))
    )
    D<-setNames(data.frame(matrix(ncol = 2, nrow = length(A))),c("start", "end"))
    D$start <- matrix(ncol = 2, nrow = length(A))
    D$start[,1]<-datos$start[,1] #matrix containing the starting points of line
        segments
    D$start[,2]<-datos$start[,2]
    D$end <- matrix(ncol = 2, nrow = length(A))
    D$end[,1]<-datos$end[,1] #matrix containing the ending points of line segments
    D$end[,2]<-datos$end[,2]
    D$TrajId <- datos$TrajId
    D$class <- a[[10]]
    #Transformacion de los enteros recibidos desde C a las cadenas de caracteres
        segun corresponde
    D$class[which(D$class == "-1")] <- "noise"
    D$class[which(D$class == "-2")] <- "desestimated"
    return(as.data.frame(D))
}
```

Finalmente, una vez se ejecute dicha función, se tendrá un dataframe con los segmentos, su id de trayectoria y su id de cluster como se puede ver en la siguiente muestra de los 6 primeros segmentos:

```
     start.1   start.2     end.1     end.2 TrajId class
1 1.6815175 1.7463984 1.6983005 1.7422480      1     1
2 1.6983005 1.7422480 1.7568508 1.5832329      1     1
3 1.7568508 1.5832329 1.6244088 1.3505129      1    15
4 1.6244088 1.3505129 1.3998323 1.1992155      1    19
5 1.3998323 1.1992155 0.5942475 0.7890607      1 noise
6 0.5942475 0.7890607 0.2706854 0.5808388      1 noise
```

En comparación con la implementación del script de R con el *TRACLUS*, la función `RepresentativeTrajectoryClustering` no se va a implementar en C, ya que funciona correctamente en R. Por lo tanto, se va a incluir en el paquete de manera que pueda ejecutarse para obtener la trayectoria representativa.

Además, a parte de las funciones mencionadas en este apartado se han creado otros archivos con funciones auxiliares para la transformación de datos y medición de distancias entre trayectorias. En el siguiente apartado se introducirán dichos ficheros.

5.3.4. Estructura final del paquete

Recordando el apartado 4.2.3, finalmente, el paquete para *TRACLUS* queda formado por:

- El fichero `DESCRIPTION`, el cual incluye los metadatos sobre el paquete.
- El directorio `man`, que incluye un archivo descriptivo del paquete, así como las funciones accesibles y los valores que toman sus parámetros, y algún ejemplo de ejecución. Esto se incluye en un archivo llamado `traclus` en formato `.Rd` *(RMarkdown)* [17].
- El fichero `NAMESPACE`, que incluye información sobre que funciones estarán accesibles al instalar el paquete, así como importaciones de otros paquetes.
- El directorio `R`, en el que se encontrará un archivo llamado `traclus.R`, en el que se incluye las declaraciones de las funciones en R, desde las cuales se llamara a las funciones en C.

- El directorio `src`, que es la parte más densa del paquete y que incluye los siguientes ficheros:
  - `approximate_partitioning.c`, que, como se ha indicado en el apartado 5.3.2, incluye las funciones necesarias para realizar las particiones de las trayectorias.
  - `approximate_partitioning.h`, que es la cabecera del archivo `approximate_partitioning.c` donde se pueden encontrar las declaraciones de las funciones y las librerias incluidas.
  - `distance.c`, donde se declaran las funciones de distancia que se usarán en las funciones de particionado y clustering.
  - `distance.h`, que es la cabecera del fichero `distance.c`
  - `line_segment_clustering.c`, donde se encontrarán las funciones que realizarán el agrupamiento de los segmentos resultantes de particionar las trayectorias.
  - `line_segment_clustering.h`, que es la cabecera del archivo `line_segment_clustering.h`.
  - `utils_airports.c`, donde se encuentran funciones que son de utilidad para la manipulación y transformación de los datos (para escalar, obtener máximos o mínimos de un vector, etc.) y a su vez se declaran los tipos de objetos creados, explicados en la sección 5.3.1.
  - `utils_airports.h`, que es la cabecera del archivo anterior.

Capítulo 6

Manual del paquete para R

En este capítulo se va a indicar cómo crear, instalar y utilizar el paquete creado, así como se va a mostrar un ejemplo de utilización del paquete integrándolo con una aplicación Shiny.

6.1. Creación, instalación y uso

En primer lugar, para poder utilizar el paquete para R, habrá que instalar R. Para ello habrá que seguir los siguientes pasos:

1. Acceder a la página web del proyecto *CRAN* [24]: *http://cran.r-project.org*, y dependiendo del sistema operativo actual, elegir «*Download R for Windows*» o «*Download R for Mac OS X*»

2. - Para un usuario de Windows, una vez se ha hecho click en «*Download R for Windows*», se debe hacer click en el enlace «*base*» y seguir las instrucciones del documento «*Installation and other instructions*» en la página resultante
   - Para un usuario de Mac OS X, basta con descargar el fichero en formato `.pkg` correspondiente, abrirlo y seguir las instrucciones del Asistente de Instalación
   - Para un usuario de Ubuntu o Debian, para instalar R se tiene que ejecutar el comando **`sudo aptitude install r-base`** en una terminal.

Cuando se haya instalado R para Windows o Mac OS X, se instalará a su vez una interfaz gráfica que se abrirá al abrir la aplicación, en la cual se podrán introducir los comandos en lenguaje R. En cambio, para Linux no se instalará interfaz gráfica, de manera que habrá que ejecutar el comando **`R`** en un terminal para poder iniciar una sesión de R.

Una vez instalado R, se puede proceder a sacar el máximo partido del mismo, que como se ha visto en secciones anteriores, el abanico de posibilidades que ofrece, así como el potencial que tiene es bastante amplio, ya sea con la distribución inicial de R y los paquetes que aporta inicialmente, o instalando paquetes desarrollados por la comunidad alojados en diferentes repositorios. Debido al importante papel que cumplen los paquetes en R, se va a explicar como crear un paquete.

Aunque los directorios y archivos se pueden crear de forma manual, R proporciona una función llamada `package.skeleton` para automatizar el proceso y crear una estructura básica. La manera de ejecutar este comando es como sigue:

```
package.skeleton(name = "traclus", list = "source")
```

El resultado de este comando sería una carpeta con directorios y ficheros como los que podría tener cualquier paquete, todos ellos a rellenar: el fichero `DESCRIPTION`, el directorio man con los archivos .Rd pertinentes para la documentación, el directorio `R` con el código que formará parte del paquete, el directorio `src`, vacía, que contendrá el código compilado y un archivo *Read-and-delete-me* con instrucciones para completar los diferentes archivos creados.

Una vez se han completado los archivos obligatorios y redactado las ayudas, se tiene que abrir un terminal, dirigirse al directorio en el que se encuentra el paquete, (en este caso, *traclus*) y en primer lugar ejecutar el comando **`R CMD check traclus`** para comprobar que todo esté listo para ser compilado como paquete. Si el resultado es positivo, se ha de pasar a compilar el paquete mediante el comando **`R CMD build traclus`** dando lugar a un fichero `traclus.tar.gz`.

Ahora que se sabe como crear un paquete, se va a proceder a indicar cómo instalarlo. Existen diversas maneras de instalar un paquete, dependiendo estas de dónde esté alojado el paquete. Se va a utilizar como ejemplo el paquete `traclus` creado.

- En el caso de que este paquete se encontrase en CRAN, basta con ejecutar el siguiente comando de R:

```
install.packages("traclus")
```

- Suponiendo la ruta en la que se ha compilado el paquete es el escritorio, se ha de indicar al comando anterior la ruta del paquete, ya sea en la red o local (como es el caso), y a su vez indicarle que no lo busque en un repositorio:

```
install.packages("C:/Users/Desktop/traclus.tar.gz", repos = NULL)
```

- Otra manera es, desde un terminal, yendo a la carpeta donde se encuentra el paquete compilado, ejecutar el comando **`R CMD install traclus`**.

Una vez realizado cualquiera de los anteriores pasos, se tendría el paquete con las funciones instaladas en el disco duro del ordenador como se podía ver en la figura 4.2. Para cargar estas en la memoria principal, habría que ejecutar el siguiente comando:

```
library(traclus)
```

Y de esta manera ya estaría cargado el paquete en la memoria principal y las funciones de este listas para ejecutarse.

Para las pruebas de uso, se ha utilizado un conjunto de datos de diferentes vuelos de un solo día (entendiendo un vuelo como el par de origen y destino) y para cada vuelo, las trayectorias sucedidas con los puntos que conforman cada una de ellas y los aeropuertos del que proceden y al que se dirigen, en formato ICAO [7], todo ello en un fichero en formato `.RData` [18].

A continuación, se puede observar un ejemplo de uso de las funciones del paquete:

```
##Con las trazas de los vuelos transformadas como se ha indicado en la seccion
    de particionado en R
#Primer paso del algoritmo -> Particionado de trayectorias

D <- partition_flights_route("LEMD","LEBL",scale = TRUE)

#D tendra los segmentos formados por los puntos caracteristicos de los vuelos
    para el dia de las trazas para el vuelo entre los aeropuertos con ICAO LEMD
    y LEBL, escalando los datos
```

En la figura 6.1 se puede observar los segmentos resultantes de ejecutar la función anterior para una trayectoria.

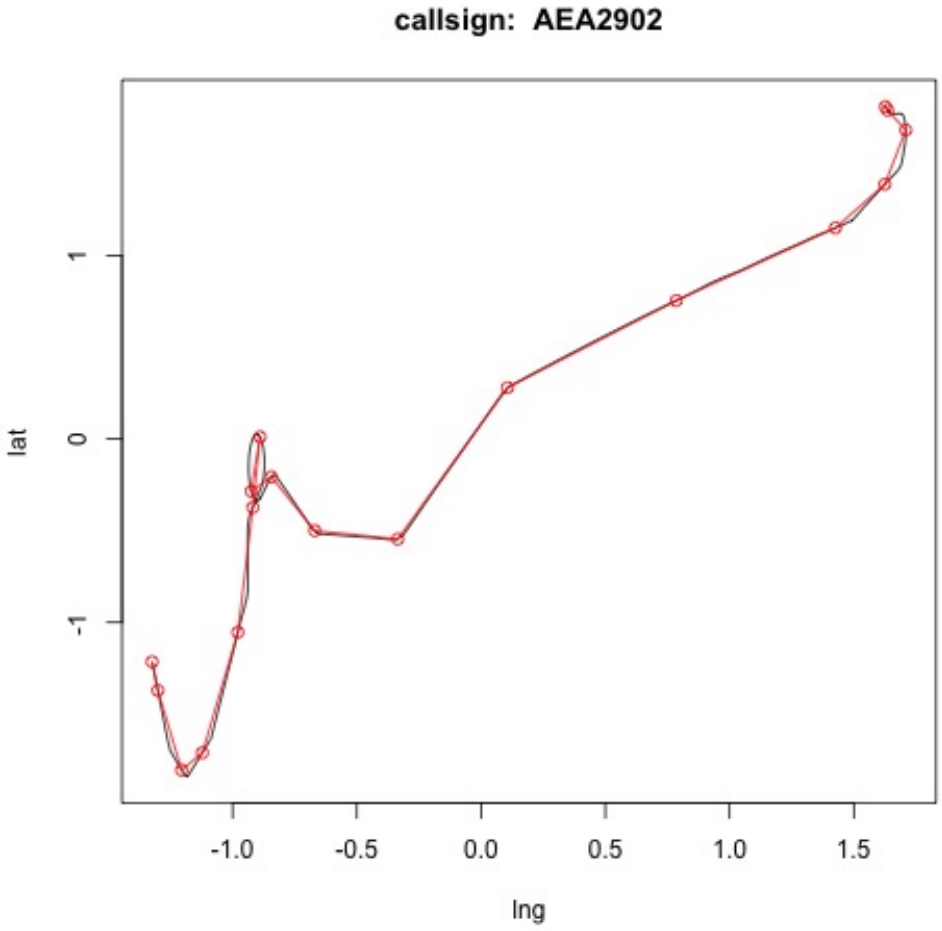


Figura 6.1: Ejemplo de particionado de una trayectoria

Y para la obtención de los clusters se procedería de la siguiente manera:

```
 #Declaracion de variables
 MinLns <- 5
epsilon <- 0.0215
gamma <- 0.045  ##needed to the generate subtrajectories
w_dist<-c(1/3,1/3,1/3)

#Segundo paso del algoritmo -> Agrupar los segmentos a partir del objeto D con
    las particiones
D <- LineSegmentClustering(datos = D, epsilon=epsilon ,MinLns=MinLns,w_dist=w_dist)

#Se pintan las trayectorias con sus clusters
plot_clustering(D)

#Se pinta la subtrayectoria comun en los clusters
plot_RTC(D, MinLns ,gamma)
```

En la figura 6.2 se puede observar el resultado con los clusters y sus subtrayectorias.

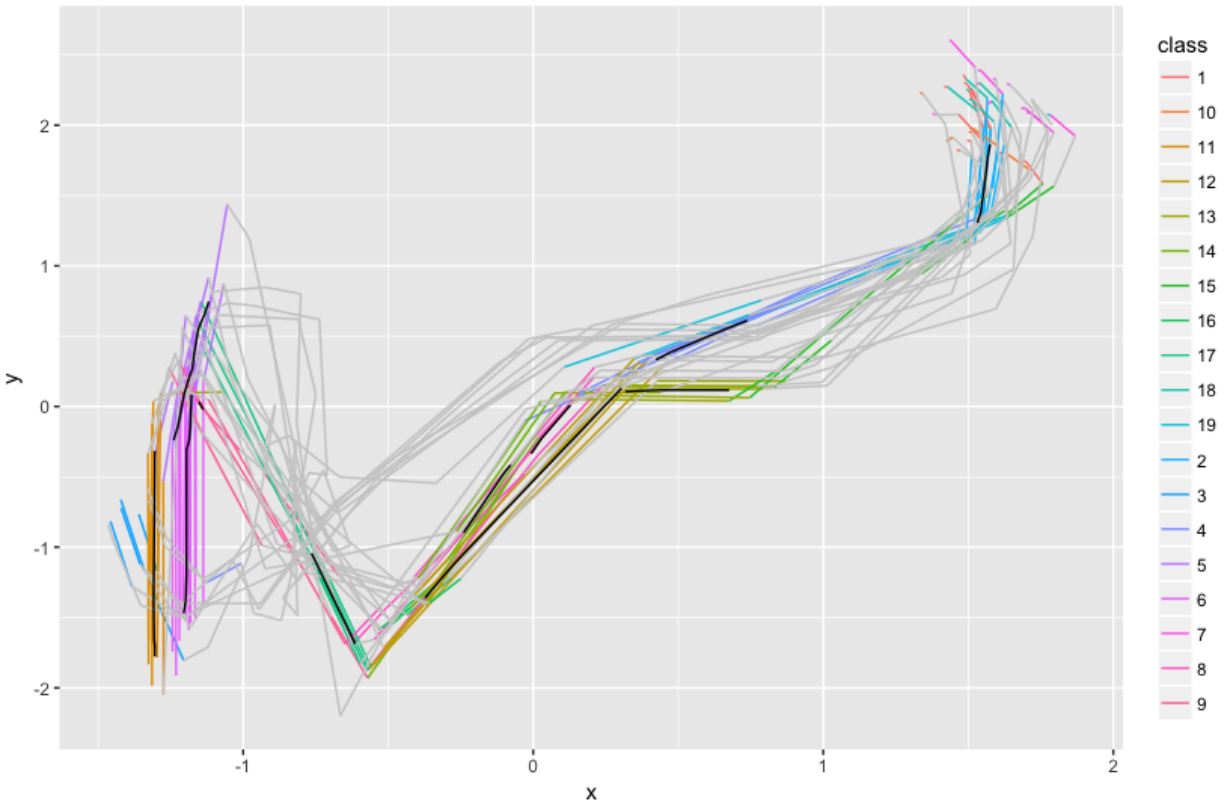


Figura 6.2: Ejemplo de clustering de los segmentos

6.2. Integración del paquete en una aplicación Shiny

En la sección 4.3 se introducía este paquete de R, viendo sus ventajas y características, así como su arquitectura típica. En este apartado se va a presentar una aplicación Shiny que se ha creado en la que se puede observar cómo funciona la implementación del paquete de *TRACLUS*, combinando este con otros paquetes, para, por ejemplo, mejorar la visualización de las gráficas.

6.2.1. Arquitectura de la aplicación

En la figura 6.3 se puede observar la arquitectura final de la implementación.

En resumen, toda la implementación se ha realizado en el IDE RStudio. La parte superior de la línea discontinua se refiere a la implementación del script de R con el algoritmo, y juntando este con R, surge el paquete traclus_1.0.0.tar.gz. Se ha ido realizando control de versiones del código a traves de git [1] y un repositorio de GitHub (véase la subsección 4.2.2). En la parte inferior de la línea, se puede ver la arquitectura de la implementación de la aplicación Shiny. Para ello, se ha utilizado el paquete Shiny junto con los scripts en R pertinentes. Además, aprovechando el paquete para R creado en la fase anterior, y combinando este con los scripts, y utilizando los datos disponibles, se ha obtenido la aplicación de Shiny.

Como se podía observar en la figura 4.5, la arquitectura de Shiny está formada por una parte de visualización con la interfaz de usuario, o *front* y una parte de servidor o *back*. A la hora de realizar la implementación se tendrá dos archivos principales: `ui.R`, donde se incorporarán elementos de entrada de datos [21] (véase la figura 4.6) y gráficas de visualización, y `server.R`, donde se

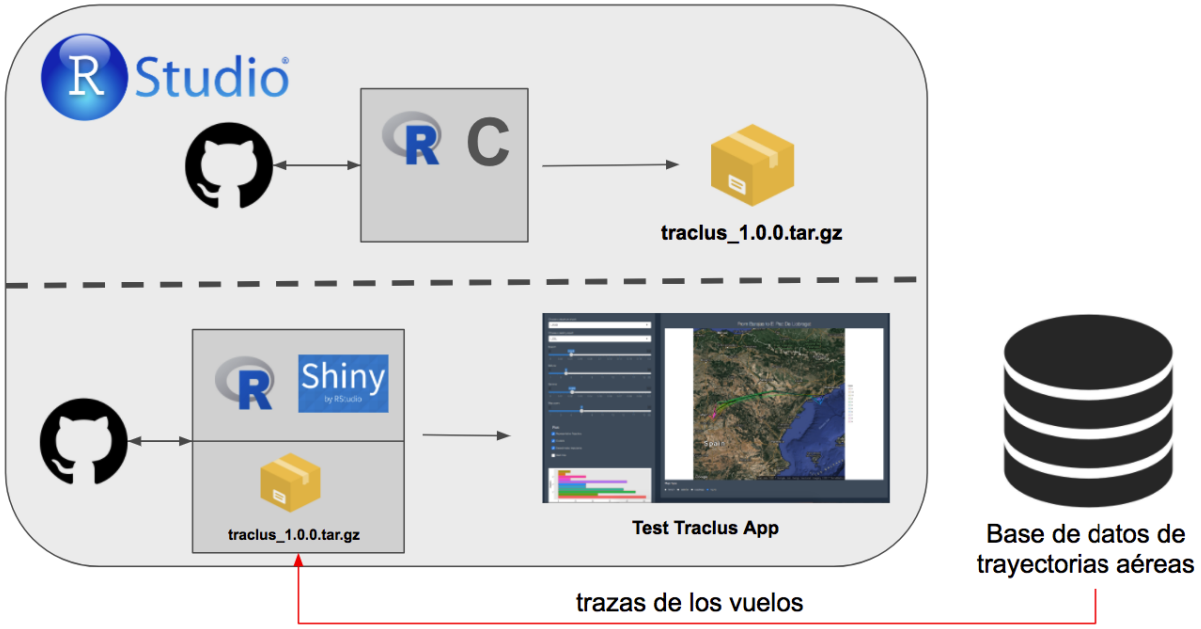


Figura 6.3: Arquitectura final de la implementación

manipularán las visualizaciones y gráficas según la entrada de datos desde `ui.R`. Estos archivos están directamente conectados y se corresponden respectivamente con la parte de front y back.

6.2.2. Parte de interfaz de usuario

En lo que a la parte de interfaz de usuario respecta, Shiny dispone de diferentes tipos de layouts para su implementación [3]. En esta implementación se ha elegido *sideBarLayout*, el cual dispone de una barra lateral y el contenido principal. En esta barra lateral se han ido introduciendo diferentes componentes. En la figura 6.4 se ha introducido un componente que incluye dos elementos de entrada de tipo `select`, mediante los cuales se podrá seleccionar los aeropuertos de destino y origen en formato ICAO de las trayectorias a las que se quiere aplicar el algoritmo.

Figura 6.4: Elementos de entrada para seleccionar aeropuertos

En la figura 6.5 se pueden ver cuatro elementos de entrada de tipo `slideInput`, en los cuales deslizando sobre la barra se podrá establecer los diferentes valores de los parámetros que recibe el algoritmo (epsilon, MinLns y gamma), así como el zoom que se le quiere aplicar al mapa.

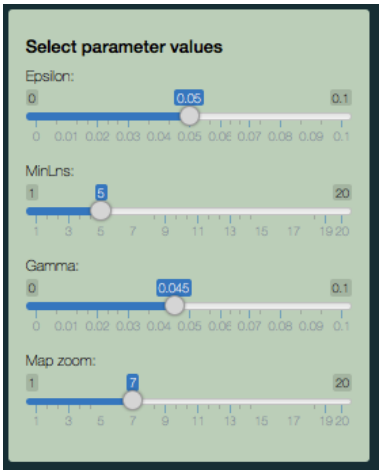


Figura 6.5: Elementos de entrada para seleccionar parámetros

Por otra parte, en la figura 6.6, se han establecido elementos de entrada de tipo `checkBoxInput`, de manera que al seleccionarlos se establecerá que se desea dibujar en la gráfica.

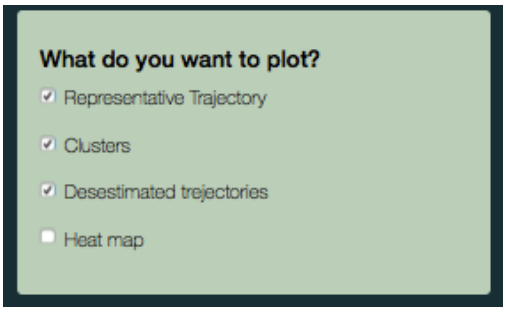


Figura 6.6: Elementos de entrada para establecer gráficas

Y como último componente de la barra lateral, se ha añadido un diagrama de barras con los clusters y el número de segmentos que conforma cada uno (véase figura 6.7).

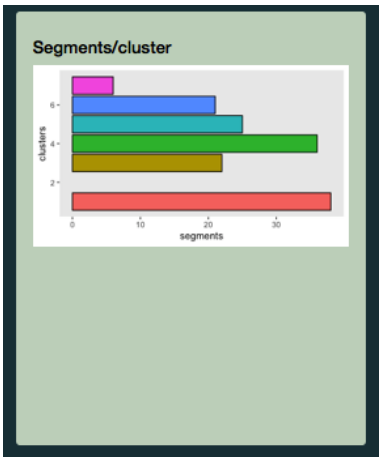


Figura 6.7: Elementos de visualización de segmentos por cluster

En la parte del contenido principal del layout se han añadido unos elementos de entrada para establecer el tipo de mapa que se quiere dibujar (ver la figura 6.8) de tipo `radioButtons`.

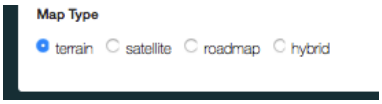


Figura 6.8: Elementos de entrada del tipo de mapa

Un botón para descargar la gráfica principal en formato `.png`. que se está visualizando en el momento de presionarlo (figura 6.9)

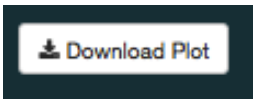


Figura 6.9: Botón de descarga de la gráfica

Y finalmente la gráfica principal con las parámetros establecidos en las figuras 6.6 y 6.8, la cual quedará como se ve en la figura 6.10.

Figura 6.10: Gráfica principal de la aplicación

6.2.3. Parte servidor

Por otra parte, en la parte de servidor se van a manipular los datos para poder realizar las visualizaciones como corresponde.

En primer lugar, cabe destacar la creación de un fichero denominado `"develop.R"`. Este fichero es el encargado de cargar los datos de las trazas de las trayectorias y transformarlos de manera que se puedan utilizar con el paquete `traclus`. Al ejecutar la aplicación se comprobará si los datos de los vuelos están cargados en memoria y en caso de no estarlo, se ejecutará dicho código y se cargaran.

Desde la parte del servidor, se cogeran los aeropuertos seleccionados en la figura 6.4 y se introducirá el valor en dos variables de la siguiente manera:

```
#Para coger el valor de los aeropuertos seleccionados
adep <- input$adep
ades <- input$ades
```

En base a estas variables, se filtrarán las trazas de los aeropuertos seleccionados y se obtendrán las longitudes y latitudes de origen y destino. Desde la parte de servidor se pueden añadir directamente algunos elementos de visualización, como por ejemplo una barra de progreso para determinadas sentencias:

```
withProgress(message = "Loading airports data",value=0,{
  #Para coger el valor de los aeropuertos seleccionados
  adep <- input$adep
  ades <- input$ades
  incProgress(0.2)
  #Para obtener las trazas de los aeropuertos seleccionados
  trazas <- trazas[which(trazas$adep == adep),]
  trazas <- trazas[which(trazas$ades == ades),]
  incProgress(0.2)
  #Para obtener los aeropuertos de destino y origen
  adep_lnglat <- as.double(geocode(paste(spanish_airports$name[which(spanish_
      airports$icao == adep)],"airport")))
  ades_lnglat <- as.double(geocode(paste(spanish_airports$name[which(spanish_
      airports$icao == ades)],"airport")))
  incProgress(0.2)
  #Vector con las coordenadas de destino y origen
  coordinates <- cbind(adep_lnglat,ades_lnglat)
  incProgress(0.4)
})
```

Lo que generará una barra de progreso como la de la figura 6.11. Estas barras de progreso se han implementado para diferentes fases de la ejecución de la aplicación.

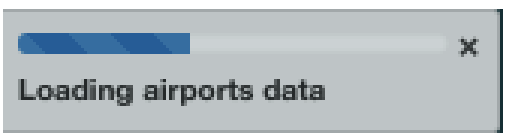


Figura 6.11: Barra de progreso para la carga de datos de los aeropuertos

Al igual que se cogía el valor de los aeropuertos seleccionados, el script `"server.R"` comprobará qué checkbox o radioButtons, y en base a estos cambiará la visualización ya sea el tipo de mapa, o qué gráfica se realiza sobre el mapa seleccionado.

Para obtener los mapas se ha utilizado el paquete de R `ggmap` [23]. Este paquete se conecta con la API de Google Maps [13], de manera que se puede pasar por parámetro el nombre de un aeropuerto y te devuelve sus coordenadas, y viceversa, si se le pasa unas coordenadas te da la calle, ciudad o con lo que se correspondan las coordenadas. A su vez, este permite graficar los mapas y la principal razón por la que se ha elegido este paquete: se integra con el paquete `ggplot2`, que es el que utiliza el paquete `traclus` para realizar las gráficas.

Finalmente, combinando la parte de la interfaz de usuario se obtendrá la aplicación final, como se puede observar en las figuras 6.12, 6.13 y 6.14, para diferentes combinaciones de parámetros establecidos.

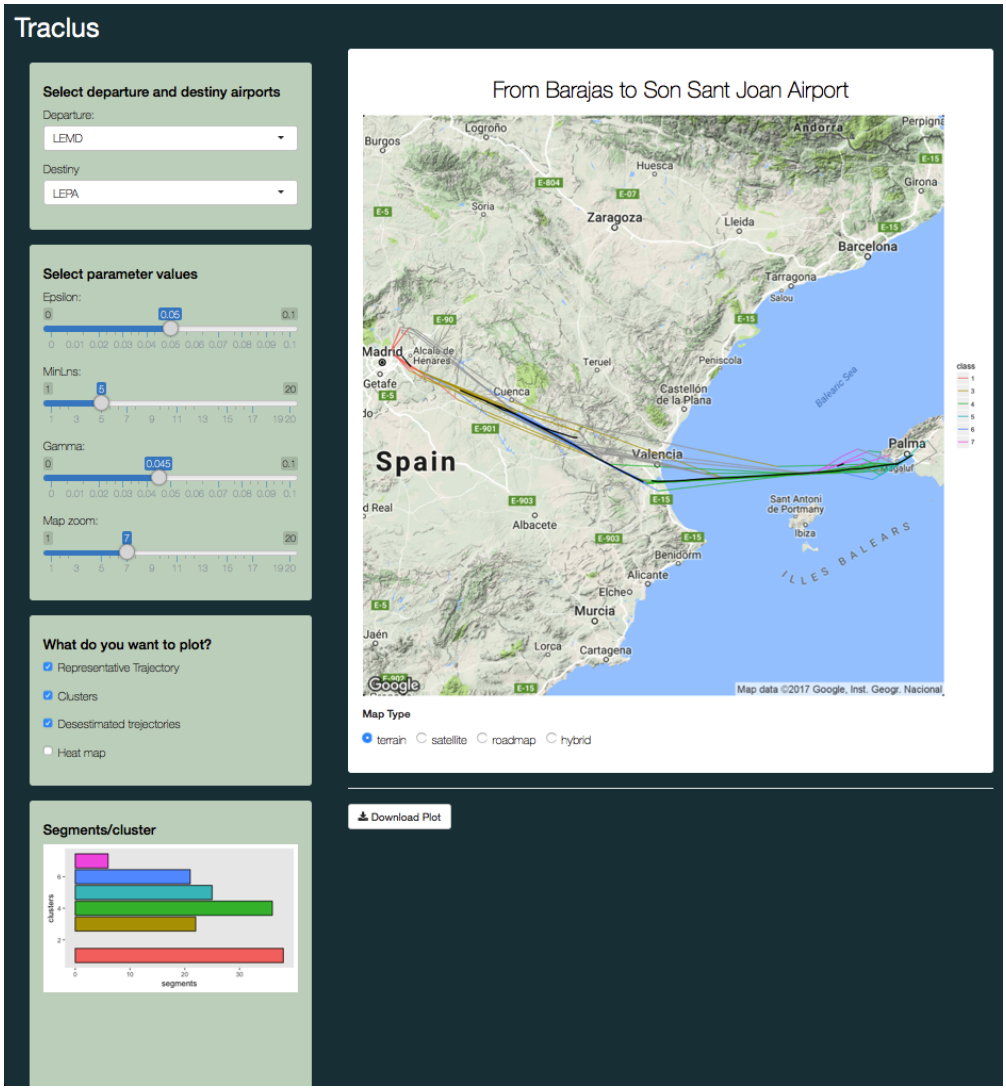


Figura 6.12: Ejemplo de ejecución de la aplicación

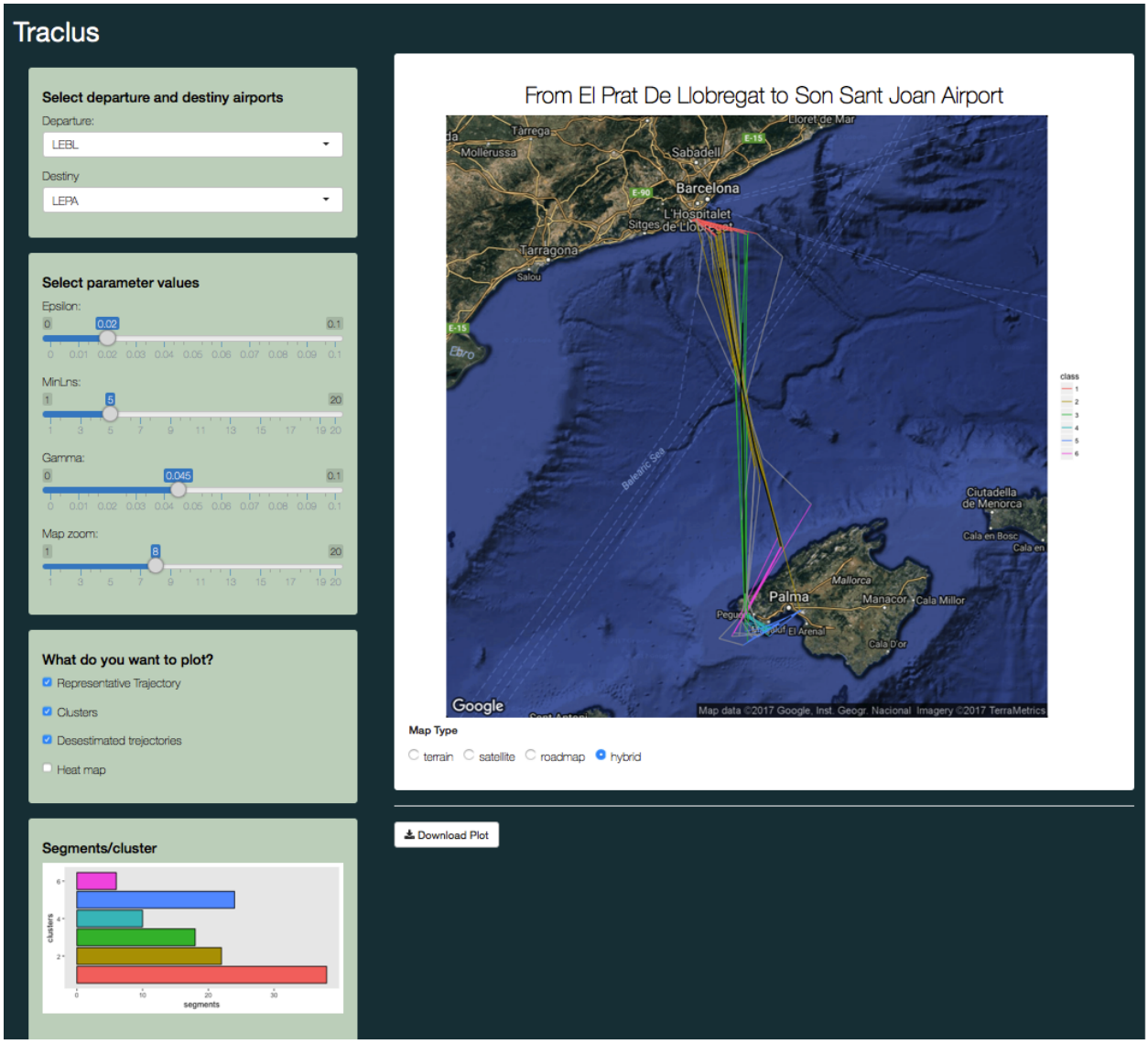


Figura 6.13: Ejemplo de ejecución de la aplicación

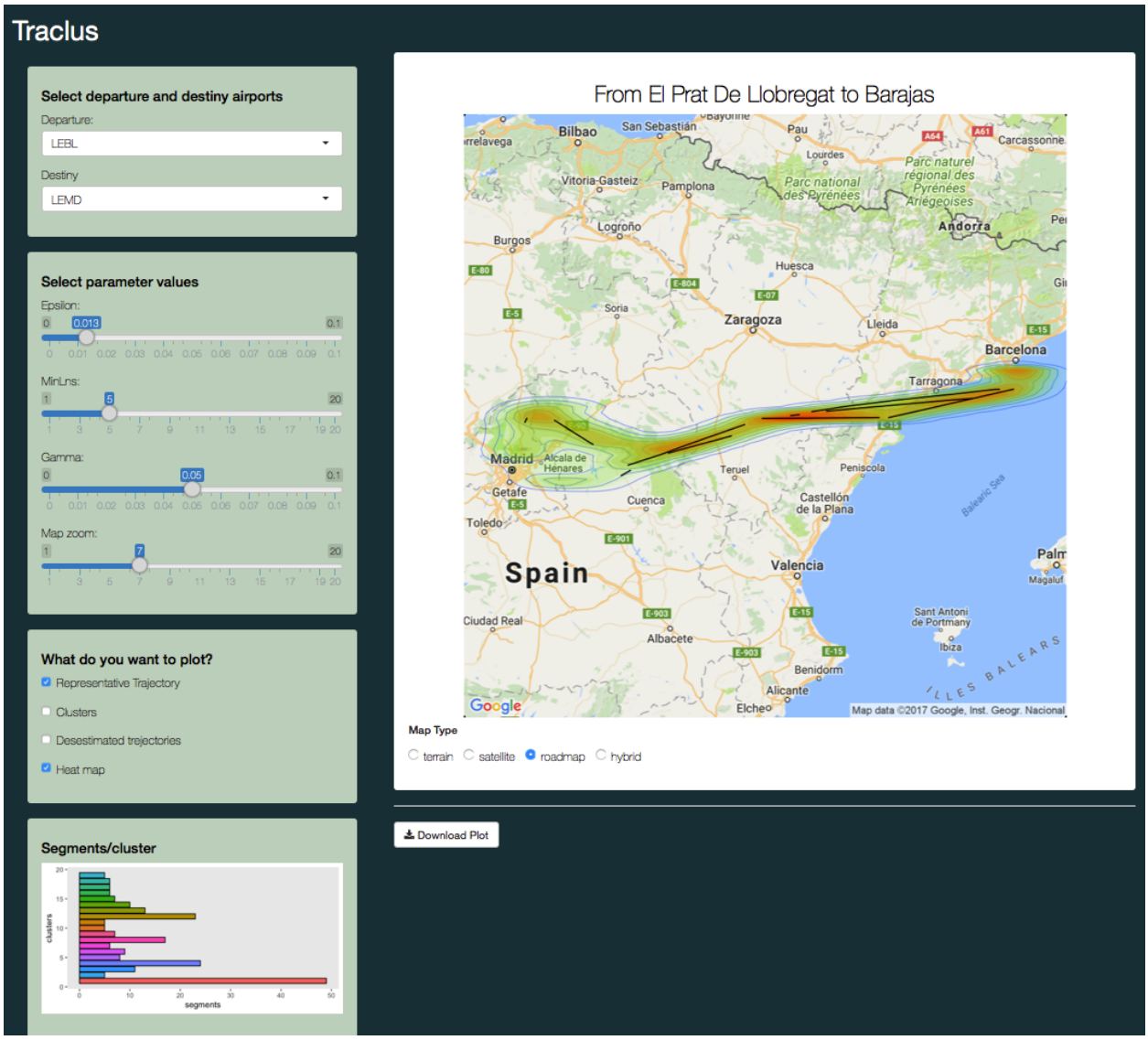


Figura 6.14: Ejemplo de ejecución de la aplicación

Capítulo 7

Benchmarking

La principal razón que motiva la realización de este apartado es la enorme diferencia entre los tiempos de ejecución que existe al implementar *TRACLUS* en un script de R y en un paquete para R (escrito en C). La causa por la que existe esta gran diferencia es que R es un lenguaje de programación de tipo interpretado, es decir, es una lenguaje de alto nivel. Estos lenguajes son más próximos al lenguaje humano y tienen una capa de abstracción para evitar a la persona que los utilice tener que entrar en detalles específicos. En un lenguaje de alto nivel basta con escribir algo como `a = 5` para asignar a la variable a el valor 5.

Puesto que el procesador solo entiende las instrucciones en código máquina [6], los programas escritos en lenguajes de alto nivel deberán ser traducidos para ejecutarlos (esto es lo que pasa con R y su alta latencia). Un programa compilado es similar a un libro traducido, aunque el libro esté originalmente escrito en otro idioma, una vez que se ha traducido al castellano se podrá leer sin ayuda el libro (en este caso, el lenguaje C). Sin embargo, un programa interpretado es equivalente a no traducir un libro, y siempre que se quiera leer este se tenga que contar con los servicios de un interprete que vaya traduciendo frase por frase (en este caso, lenguaje R). Por esta razón, R será siempre un lenguaje más lento que C.

Puesto lo que se busca es un equilibrio entre velocidad de cómputo para transformación de los datos y sencillez de manejo, se ha creado el paquete para R, donde se reúnen scripts de R y código en C.

A continuación, se van a comparar los resultados y los tiempos de ejecución de utilizar y no utilizar código C para la manipulación y transformación de los datos.

Se va a tomar como ejemplo para todo el capítulo los vuelos entre *El Prat de Llobregat - LEBL* y *Madrid - LEMD* del día 6 de mayo de 2016. Para estos vuelos, se han establecido los parámetros: $\epsilon = 0{,}0195; MinLns = 5; w\_dist = [1/3, 1/3, 1/3]; gamma = 0{,}05$.

7.1. Resultados obtenidos

En primer lugar se van a comparar los resultados que se obtienen al ejecutar las implementaciones de la sección 5.

En este análisis se debe tener presente el apartado 2.3.3, donde se hablaba de las propiedades ideales de la partición. Se decía que para que una partición fuese óptima tenía que tener dos características principales: *precisión*, en lo que refiere a que cuanto mayor sea el número de particiones

de una trayectoria más precisa será la partición puesto que será más similar a la trayectoria real, y *concisión*, en lo que refiere a que una partición será óptima cuando el número de particiones sea el mínimo.

Para comenzar, se van a establecer unos parámetros en los que basarse para realizar el benchmarking de resultados para cada uno de los tipos de ejecución. Estos serán:

- **Número de segmentos**, es decir, el número de segmentos obtenidos al particionar.
- **Número de clusters**
- **% de segmentos que pertenecen a un cluster**
- **% de segmentos que son considerados ruido**
- **% de segmentos que son desestimados**
- **% de segmentos no pertenecientes a ningún cluster**, que es la suma de los considerados ruido y los desestimados.

7.1.1. Datos escalados

En primera instancia, estos parámetros serán calculados para la implementación en R de *TRACLUS* escalando los datos [4], tomando estos los valores:

Nº de segmentos	Nº de clusters	% pertenecientes	% ruido	% desestimados	% no pertenecientes
465	16	51.1828 %	41.72043 %	7.096774 %	48.8172 %

Cuadro 7.1: Valores de los parámetros para la implementación en R escalando los datos

La gráfica obtenida de los segmentos con su trayectoria representativa para el caso de la implementación en R escalando los datos se puede ver en la figura 7.1.

A continuación se va a ejecutar la implementación en C con los mismos parámetros, es decir, ejecutando las funciones del paquete, que quedarían como se puede ver en la tabla 7.2.

Nº de segmentos	Nº de clusters	% pertenecientes	% ruido	% desestimados	% no pertenecientes
376	17	43.08511 %	55.58511 %	1.329787 %	56.91489 %

Cuadro 7.2: Valores de los parámetros para la implementación en C escalando los datos

Quedando la gráfica que representa los datos obtenidos mediante el paquete de *TRACLUS* para datos escalados como se puede ver en la figura 7.2

7.1.2. Datos sin escalar

Se va a continuar siguiendo la misma dinámica, pero ahora para los datos sin escalar.

En la tabla 7.3, se pueden visualizar los valores que toman los parámetros para la implementación del algoritmo en R sin escalar los datos.Para los cuales se obtiene la gráfica de la figura 7.1.2.

Nº de segmentos	Nº de clusters	% pertenecientes	% ruido	% desestimados	% no pertenecientes
381	11	85.5643 %	14.4357 %	0 %	14.4357 %

Cuadro 7.3: Valores de los parámetros para la implementación en R sin escalar los datos

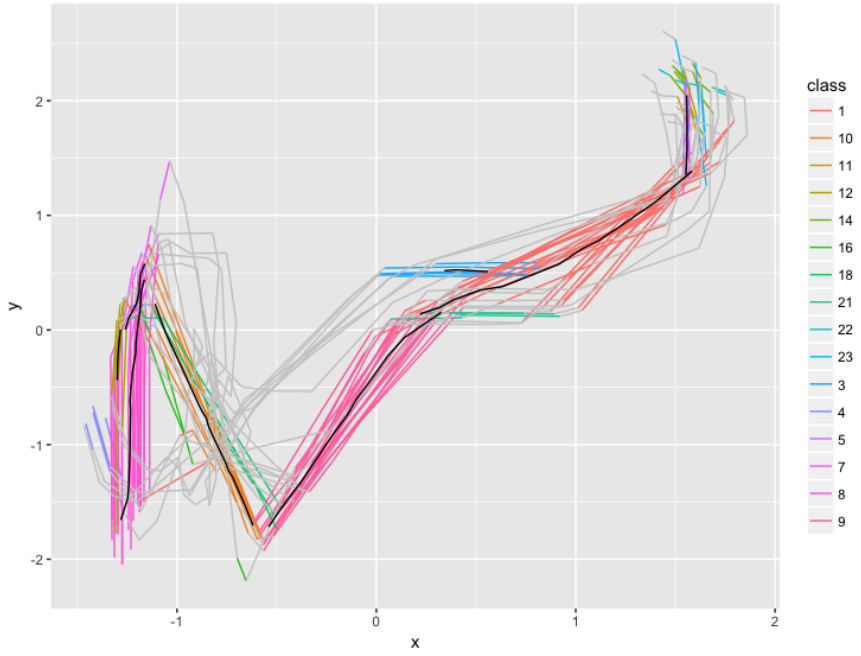


Figura 7.1: Clusters y subtrayectorias para la implementación de R escalando los datos

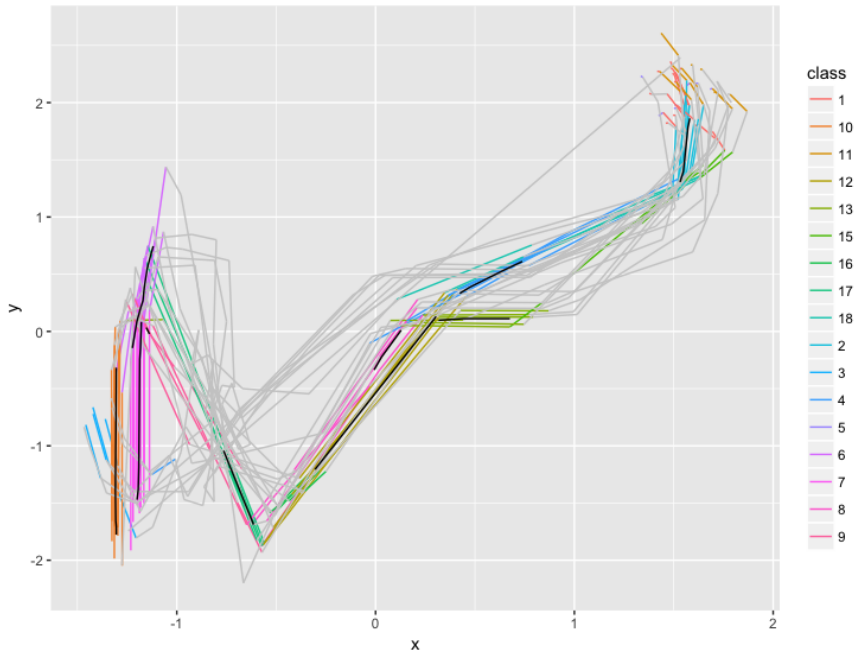


Figura 7.2: Clusters y subtrayectorias para la implementación de C escalando los datos

Finalmente, se van a mostrar los datos obtenidos para la implementación del paquete para R con C de los datos sin escalar (véase la tabla 7.4), para los cuales se tiene la gráfica correspondiente en la figura 7.4.

A modo de resumen, en la tabla 7.5, según la implementación se pueden ver los datos que se han recogido y como difieren entre ellos para el mismo conjunto de datos.

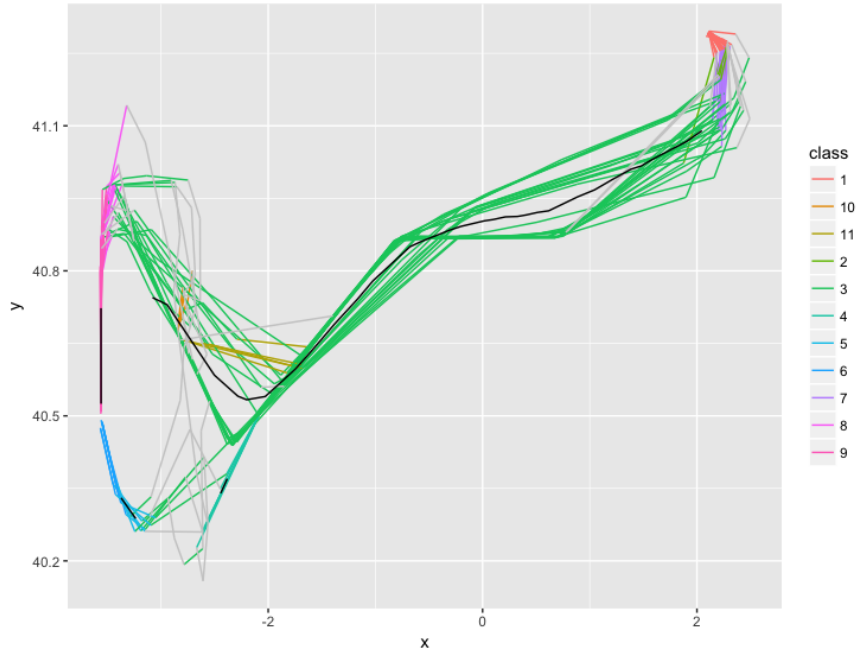


Figura 7.3: Clusters y subtrayectorias para la implementación de R sin escalar los datos

Nº de segmentos	Nº de clusters	% pertenecientes	% ruido	% desestimados	% no pertenecientes
340	21	80.58824 %	19.41176 %	0 %	19.41176 %

Cuadro 7.4: Valores de los parámetros para la implementación en R sin escalar los datos

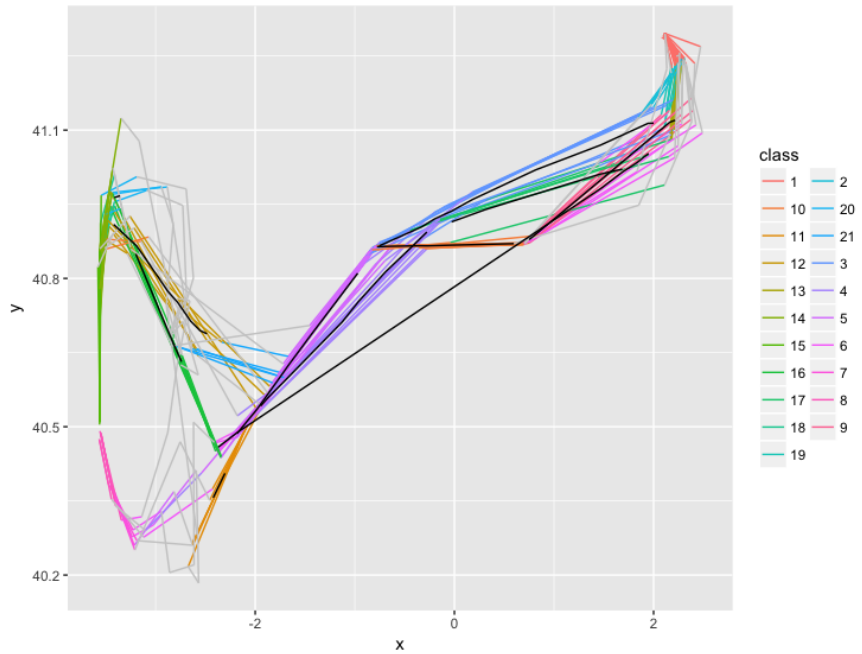


Figura 7.4: Clusters y subtrayectorias para la implementación de C sin escalar los datos

Visualizando estos datos, se pueden obtener conclusiones en lo que respecta a las diferentes implementaciones y a los resultados obtenidos de ellas.

Implementación	Nº de segmentos	Nº de clusters	% pertenecientes	% ruido	% desestimados	% no pertenecientes
R con escalado	465	16	51.1828 %	41.72043 %	7.096774 %	48.8172 %
C con escalado	376	17	43.08511 %	55.58511 %	1.329787 %	56.91489 %
R sin escalar	381	11	85.5643 %	14.4357 %	0 %	14.4357 %
C sin escalar	340	21	80.58824 %	19.41176 %	0 %	19.41176 %

Cuadro 7.5: Resumen de los valores de los parámetros para cada implementación

En primer lugar, y respecto al número de segmentos obtenidos, se puede decir que, tanto para los datos escalados como sin escalar, se obtiene un mayor número de segmentos en la implementación de R que en la de C. Esto se traduce en que la implementación de C cumplirá en mayor parte la *propiedad ideal* de la concisión, ya que el número de segmentos resultantes de particionar ha de ser mínimo para considerar la partición óptima, a diferencia de la implementación en R, que cumplirá en mayor parte la *propiedad ideal* de la precisión, que dice que cuanto mayor sea el número de particiones de las trayectorias mayor será la precisión de la partición.

En cuanto al número de cluster, se puede observar que la implementación de R sin escalar los datos es la que funciona de manera menos precisa, aportando para 381 segmentos únicamente 11 clusters. En cambio, la implementación de C sin escalar aporta 21 clusters para 340 segmentos particionados. En la figura se puede visualizar un cluster central coloreado en verde, del cual la implementación en C obtiene múltiples cluster con sus diferentes subtrayectorias comunes (ver figura 7.4). Lo mismo pasa cuando se escalan los datos: en la figura 7.1 se puede ver un cluster de color rosa en el centro, del cual la implementación en C obtiene dos clusters (ver figura 7.2).

A su vez, se puede decir que las implementaciones sitúan en diferentes clusters a un mayor número de segmentos cuando los datos no se escalan que cuando si lo hacen. Así como no se desestiman segmentos pertenecientes a cluster, de manera que los segmentos son o ruido o pertenecientes a algún cluster.

En cuanto a la razón de las diferencias obtenidas de los resultados, no se ha localizado un motivo aparente que justifique este hecho, siendo una posible razón la integración entre los lenguajes y el mapeo de las variables entre ellos. [26]

7.2. Tiempos de ejecución

Una vez se han comparado resultados obtenidos, tiene lugar hablar de los tiempos de ejecución de cada una de las implementaciones. Como se ha mencionado en la introducción del presente capítulo, la ejecución de R respecto a la de C siempre será más lenta.

Para comprobar esto inicialmente, se implementó la función que calculaba la distancia paralela entre dos trayectorias (véase el apartado 2.3.2) en R y en C, pero utilizando en C las variables auxiliares de tipo SEXP [20], por lo que no se estaba utilizando C puro. En la figura 7.5 se puede ver como en la mayoría de las veces salvo en tres casos, el tiempo de ejecución de C (no puro) es menor que el de R.

Respecto a las cuatro implementaciones de la sección anterior, se ha medido el tiempo que ha llevado su ejecución con la función de R `System.time()`. Esta función devuelve un objeto de tipo `proc.time()` [16], del cual nos interesa el atributo `elapsed`, que nos dirá el tiempo final de ejecución de la función.

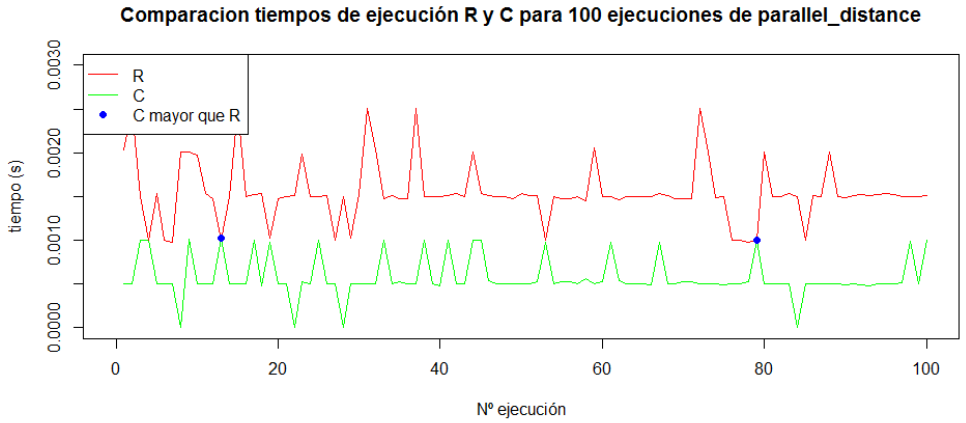


Figura 7.5: Ejemplo de comparación de tiempos para C y R

En la tabla 7.6, se puede observar los tiempos que ha tardado cada una de las implementaciones que se veían en la tabla 7.5.

Implementación	Particionado (s)	Clustering (s)	Total (s)	Total (min)
R con escalado	1512.065	537.004	2049.069	34,15115
C con escalado	1.865	0.028	1.893	0,03155
R sin escalar	622.533	345.557	1524,533	25,40888
C sin escalar	6.318	0.036	6,354	0,104233

Cuadro 7.6: Resumen de tiempos de las ejecuciones

Como conclusión final, y tal y como se había previsto, la implementación del algoritmo en código C es mucho más rápida que la implementación en R. Cabe destacar la diferencia en la implementación de C escalando los datos y sin escalarlos, ya que por alguna razón, al hacerlo sin escalar la duración es de aproximadamente 5 segundos mayor que si se escalan, pese que al escalado requiere la ejecución de un proceso a mayores.

Por lo tanto, para el escalado, se tiene que la implementación de C respecto a R es del orden de **810,75 veces más rápida**, y para los datos sin escalar es **98,53 veces más rápida**.

Capítulo 8

Conclusiones y Trabajo Futuro

Por último, se van a presentar las conclusiones que se pueden extraer del trabajo realizado, así como los principales logros conseguidos respecto a los objetivos planteados inicialmente. Además, se plantearán posibles ampliaciones y mejoras de las implementaciones realizadas.

La primera y mayor dificultad encontrada fue llevar a cabo la parte de investigación. Debido al desconocimiento de este ámbito, se tuvo que iniciar por un nivel bajo relacionado con el ámbito de los algoritmos de clustering, conociendo desde los más básicos hasta otros más complejos, y estudiando la posibilidad de aplicarlos a trayectorias, para finalmente dar lugar a *TRACLUS*, que además de ser aplicable únicamente a trayectorias, este proporciona características que le diferencian del resto.

Otra dificultad fue el aprendizaje del lenguaje R, el cual fue un lenguaje totalmente desconocido inicialmente. La curva de aprendizaje no fue excesivamente pronunciada, ya que este lenguaje está creado por estadísticos con el fin de facilitar su trabajo, y, al ser un lenguaje interpretado, el tipo de sentencias que se crean, la lógica que sigue y el tipo de ficheros sobre los que se trabaja hacen de R un lenguaje de aprendizaje sencillo.

Por otra parte, el trabajo, manipulación y transformación de grandes cantidades de datos, como se da el caso en el proyecto, era una situación totalmente nueva que hubo que afrontar, teniendo archivos de millones de filas que había que utilizar con los diferentes métodos creados.

Como se ha podido observar, R experimenta una gran latencia al trabajar con grandes conjuntos de datos, ya que se tienen que traducir sus sentencias de código al ser un lenguaje interpretado. Sin embargo, este permite integrar otro tipo de lenguajes, como en este caso es C, mediante el cual se consiguió optimizar y agilizar el trabajo para la consecución de los objetivos del proyecto.

Además, el trabajo diario de los datos ha permitido conocer de manera breve como aportar valor a estos y poder mostrar esto de alguna manera, en este caso, obteniendo las similitudes entre ellos, de los que se han obtenido los clusters, y el patrón que han seguido, siendo este la subtrayectoria común.

Puesto que R está creado para, entre otras cosas el análisis y la visualización de los datos, este proyecto ha permitido conocer de mejor manera como se deben visualizar los datos según la lógica de los mismos, observarlos y en base a eso poder tomar decisiones de qué se debe mostrar y cómo se debe mostrar.

8.1. Conclusiones

La conclusión principal que se obtiene de este proyecto es que actualmente, existen muchos métodos de recogida de datos, pero en numerosas ocasiones no se explotan y no se conoce el valor que de verdad poseen. Para mostrar y generar este valor que de verdad tienen los datos, existen diversas herramientas, en este caso, la herramienta ha sido *TRACLUS*. Con la utilización de este algoritmo se aportará valor a los datos de los vuelos, pudiendo así ver patrones comunes y viendo extraños que sufren las trayectorias respecto a las planeadas inicialmente y respecto al resto de trayectorias. Con esto, se podría ayudar a una compañía aérea que quisiese ver como están actuando sus aviones, que tipo de aviones son los que no forman parte del cluster que forma la mayoría de los trayectorias, y entonces, ayudar en la toma de decisiones respecto a inversiones en dichos aviones, generando así valor a la recolección de las grandes cantidades de datos durante los diversos y numerosos vuelos.

Otra conclusión obtenida del TFG es que, de la solución inicial propuesta para la consecución final de los objetivos, hasta la solución final, varía mucho según van progresando las diferentes fases del proyecto, pero, sobre todo, en la fase de implementación.

8.2. Trabajo futuro

Aunque la implementación realizada es totalmente funciona, existen varias vias de expansión del proyecto:

- La **publicación del paquete de R en CRAN**, modificando este de manera que cumpla las políticas exigidas por *CRAN*. Esto sería un punto no muy lejano y beneficioso para el desarrollador del paquete, ya que R tiene detraás una comunidad enorme, por lo que la detección de fallos y de oportunidades de ampliación sería mucho más ágil y sencilla, gracias a la ayuda y a la información aportada por los usuarios del paquete.
- La **ampliación de *TRACLUS* a 3 dimensiones**, pudiendo realizar clustering con la altura de las trayectorias de los aviones y así poder ver, por ejemplo, en que momento de la trayectoria los aviones comienzan a descender para aterrizar, si lo hacen simultaneamente, localizar las causas de que un conjunto de trayectorias no ha comenzado a descender en el momento preciso, etc.
- **Integración de streaming a la aplicación web**, de manera que se pueda visualizar como se van realizando los clusters de las trayectorias según van mandando los aviones los datos a las torres de control y ver como una trayectoria se ha desviado de la subtrayectoria común a la mayoría de trayectorias.
- **Investigar las diferencias de resultados**, de manera que se localice el motivo de los diferentes resultados entre R y C y se unifique la salida que generan ambos. [11]

8.3. Aprendizajes

Una vez finalizada la implementación del TFG, se han adquirido diversos conocimientos.

En primer lugar, se ha aprendido a programar en lenguaje R y cómo manejar grandes conjuntos de datos con él. Previamente a este proyecto, el conocimiento de este lenguaje era nulo, por lo que la evolución ha sido absolutamente positiva.

En segundo lugar, me ha permitido aprender como, a partir del planteamiento de un algoritmo, se transforma esto en algo útil y se pueden ver resultados más allá de los incluidos la descripción del algoritmo.

También me ha permitido conocer, dentro del inmenso mundo de los lenguajes de programación, las grandes diferencias entre un lenguaje interpretado y uno compilado, de manera que al final se ha tenido que utilizar un lenguaje compilado de bajo nivel (C) para que el interpretado pueda funcionar de manera óptima (R).

Además, este proyecto me ha enseñado una metodología de trabajo totalmente independiente. Se está acostumbrado a trabajar con la consecución de ciertos hitos en ciertas fechas, y en este caso la ausencia de estos ha llevado a la implantación de los mismos de manera independiente.

Además, en numerosas ocasiones se han tenido que aplicar conocimientos adquiridos en otras asignaturas:

- Para la implementación del código se ha tenido que recurrir a los conocimientos adquiridos en **Fundamentos de programación**, **Programación orientada a objetos** y **Programación y Estructura de Datos**.
- Para la parte de gestión del proyecto se han aplicado conocimientos adquiridos en asignaturas como **Organización y Gestión de Empresas**, **Gestión de Proyectos Basados de las Tecnologías de la Información** y **Modelado de Software**.

Bibliografía

[1] *1.3 Empezando - Fundamentos de Git*. (visitado el 12-07-2017). URL: https://git-scm.com/book/es/v1/Empezando-Fundamentos-de-Git.

[2] Mihael Ankers y col. *OPTICS: Ordering Points To Identify the Clustering Structure*. URL: http://www.dbs.ifi.lmu.de/Publikationen/Papers/OPTICS.pdf.

[3] *Application layout guide*. (visitado el 11-07-2017). URL: https://shiny.rstudio.com/articles/layout-guide.html.

[4] *base*. (visitado el 11-07-2017). URL: https://www.rdocumentation.org/packages/base/versions/3.4.1/topics/scale.

[5] *Clustering - Fuzzy C-means*. (visitado el 11-07-2017). URL: https://home.deib.polimi.it/matteucc/Clustering/tutorial_html/cmeans.html.

[6] *Código Máquina*. (visitado el 12-07-2017). URL: https://www.mastermagazine.info/termino/4330.php.

[7] *Data Hub for all the World's Airports*. (visitado el 11-07-2017). URL: https://www.world-airport-codes.com/.

[8] Martin Ester y col. *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise*. URL: https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf.

[9] *Excelacom*. (visitado el 11-07-2017). URL: http://www.excelacom.com/resources/blog/2016-update-what-happens-in-one-internet-minute.

[10] Matthew Finnegan. *Boeing 787s to create half a terabyte of data per flight, says Virgin Atlantic*. (visitado el 11-07-2017). Mar. de 2013. URL: http://www.computerworlduk.com/data/boeing-787s-create-half-terabyte-of-data-per-flight-says-virgin-atlantic-3433595/.

[11] *Foreign Function Interface*. URL: https://stat.ethz.ch/R-manual/R-devel/library/base/html/Foreign.html.

[12] Scott Gaffney y Padharic Smyth. *Trajectory Clustering with Mixtures of Regression Models*. URL: http://www.datalab.uci.edu/papers/trtraj.pdf.

[13] *Google Maps API*. (visitado el 12-07-2017). URL: https://developers.google.com/maps/web-services/?hl=es-419.

[14] Jiawei Han Jae-Gil Lee y Kyu-Young Whang. *Trajectory Clustering: A Partition-and-Group Framework*. URL: http://hanj.cs.illinois.edu/pdf/sigmod07_jglee.pdf.

[15] J. MacQueen y Nathan S. Netanyahu. *Some methods for classification and analysis of multivariate observations*. (visitado el 11-07-2017). URL: http://projecteuclid.org/download/pdf_1/euclid.bsmsp/1200512992.

[16] *proc.time object*. (visitado el 12-07-2017). URL: https://stat.ethz.ch/R-manual/R-devel/library/base/html/proc.time.html.

[17] *R Markdown*. (visitado el 11-07-2017). URL: http://rmarkdown.rstudio.com/.

[18] *R Workspace File*. (visitado el 11-07-2017). URL: https://fileinfo.com/extension/rdata.

[19] *S (programming language)*. Jul. de 2017. URL: https://en.wikipedia.org/wiki/S_(programming_language).

[20] *SEXPs variables*. (visitado el 12-07-2017). URL: https://cran.r-project.org/doc/manuals/r-release/R-ints.html#SEXPs.

[21] *Shiny Widget Gallery*. (visitado el 12-07-2017). URL: https://shiny.rstudio.com/gallery/widget-gallery.html.

[22] Aarti Singh. *Minimum Description Length*. URL: http://www.cs.cmu.edu/~aarti/Class/10704/lec13-MDL.pdf.

[23] *Spatial Visualization with ggplot2 [R package ggmap version 2.6.1]*. (visitado el 12-07-2017). URL: https://cran.r-project.org/web/packages/ggmap/index.html.

[24] *The Comprehensive R Archive Network*. (visitado el 11-07-2017). URL: https://cran.r-project.org/.

[25] *The R Project for Statistical Computing*. (visitado el 12-07-2017). URL: https://www.r-project.org/.

[26] *Writing R Extensions*. URL: https://cran.r-project.org/doc/manuals/r-release/R-exts.html#System-and-foreign-language-interfaces.

[27] Tian Zhang, Raghu Ramakrishnan y Miron Livny". *BIRCH: An Efficient Data Clustering Method for Very Large Databases*. URL: https://www.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf.