Universidad de Valladolid

Facultad de Ciencias

TRABAJO FIN DE GRADO

Grado en Estadística

Computación optimizada de métodos de Análisis Cluster robusto

Autor:

D. Javier Crespo Guerrero

Tutor:

D. Luis Ángel García Escudero

2023

Agradecimientos

A mis profesores del Grado en Estadística, por formarme todos estos años y transmitirme sus conocimientos y pasión por la estadística. En especial a Luis Ángel, por su gran dedicación en este trabajo, su invaluable ayuda y cuidado a la hora de trabajar, de la que he podido aprender mucho.

A mi familia, en especial a mis padres y mi hermano, por su apoyo, comprensión y paciencia. Gracias por enseñarme a afrontar los retos y no tirar nunca la toalla.

A Paula, por estar a mi lado siempre en los buenos y malos momentos. Gracias por todos estos años.

Por último, a todos los amigos y compañeros que me han acompañado durante estos últimos años, gracias por todo.

Resumen

El análisis Cluster, fundamental en la Ciencia de Datos moderna, aborda problemas de clasificación en los que no se tiene una etiqueta explícita para las observaciones, sino que se busca agrupar las observaciones atendiendo a los valores que tomen los individuos en ciertas variables. Este problema es especialmente complicado cuando tratamos datos de alta dimensionalidad, agravándose especialmente con la presencia de observaciones "atípicas" que distorsionan los resultados.
En este trabajo se ha desarrollado un paquete de software que implementa métodos de clustering robusto basados en recortes imparciales, incluyendo los métodos TCLUST y RLG. Ese tipo de recorte excluye una proporción de datos potencialmente más atípicos a través de la estructura de los propios datos. Los métodos TCLUST y RLG se han mejorado a nivel computacional para resolver con eficiencia y comodidad problemas complejos de alta dimensionalidad, cada vez más habituales en la práctica actual de la Estadística. Se han considerado novedosas técnicas de inicialización y su mejorada eficiencia computacional ha sido probada en diferentes escenarios de simulación, así como en su aplicación a datos reales en alta dimensionalidad.

Abstract

Cluster analysis, a fundamental technique in modern Data Science, addresses classification problems where explicit labels for observations are not available, and instead aims to group the observations attending to the individuals' values in some variables. This problem becomes particularly challenging when dealing with high-dimensional data, and is further complicated by the presence of "outlier" observations that distort the results.
In this work, a software package has been developed that implements robust clustering methods based on impartial trimming, including TCLUST and RLG methods. This type of trimming excludes a proportion of potentially more atypical data based on the structure of the data itself. The TCLUST and RLG methods have been computationally enhanced to efficiently and conveniently solve complex high-dimensional problems, which are becoming increasingly common in current statistical practice. Novel initialization techniques have been considered, and the improved computational efficiency has been tested in various simulation scenarios, as well as its application to real high-dimensional data.

Índice general

Agradecimientos **I**

Resumen **II**

1. Introducción **1**
1.1. Contexto y objetivos 1
1.2. Estructura de la memoria 4
1.3. Herramientas 5

2. Metodología e implementaciones disponibles **6**
2.1. Metodología y algoritmos 6
2.1.1. TCLUST 6
2.1.2. RLG 10
2.2. Implementaciones computacionales de TCLUST y RLG disponibles . . . 12
2.2.1. Paquete **tclust** en CRAN 12
2.2.2. Función `tclust()` en R 13
2.2.3. Función `rlg()` en C++ 14

3. Paquete "robClus" **15**
3.1. Paquete **robClus** 15
3.1.1. Código e instalación 16
3.1.2. Función `tclust.new()` 17
3.1.3. Función `rlg()` 20
3.1.4. Funciones `simula.tclust()` y `simula.rlg()` 21
3.1.5. Datasets 22
3.1.6. Documentación de las funciones 23
3.2. Simulaciones en la aplicación de TCLUST 23

4. Inicialización por agregación **33**
4.1. Metodología propuesta 33
4.2. Ejemplo ilustrativo 35
4.3. Simulación con TCLUST con agregación 37
4.4. Simulación con RLG con agregación 39

5. Ejemplo de aplicación a datos reales **43**
5.1. MNIST 43
5.2. Aplicación de los algoritmos en **robClus** 44

5.3. Análisis de los resultados 45
5.3.1. Resultados TCLUST 45
5.3.2. Resultados RLG 46

6. Conclusiones y líneas futuras **49**

A. Código fuente de `robClus` **54**
A.1. Función `tclust.new` 54
A.1.1. Función de alto nivel en R 54
A.1.2. Funciones de bajo nivel en C++ 57
A.2. Función `rlg()` 63
A.3. Funciones de simulación de datos 65

Capítulo 1

Introducción

1.1. Contexto y objetivos

Las técnicas de clasificación son herramientas básicas en muchas disciplinas científicas y constituyen la base de numerosos procedimientos englobados bajo el epígrafe general de Ciencia de Datos. En particular, las técnicas de clasificación no supervisada o Análisis Cluster utilizan la información en conjuntos de datos para crear grupos o "clusters" de individuos con características similares para individuos dentro del mismo cluster y bien diferenciadas para individuos en otro grupos (ver, por ejemplo, Everitt et al. (2011)). El Análisis Cluster es utilizado para facilitar la interpretación, comprensión y manejo de los conjuntos de datos.

Las técnicas más típicamente aplicadas en Análisis Cluster han debido adaptarse a dificultades propias de la creciente dimensionalidad en los conjuntos de datos tratados y ser también capaces de resistir bien la presencia de observaciones atípicas.

La alta dimensionalidad es un requerimiento cada vez más habitual en el tratamiento de datos. Esta alta dimensionalidad surge de forma natural en la Ciencia de Datos actual por la variada tipología de formatos complejos de datos que se deben analizar, donde muchos de estos formatos nos llevan directamente a problemas de moderada o alta dimensionalidad.

La presencia de observaciones atípicas, que se alejan del modelo asumido, es bien sabido que puede afectar significativamente al ajuste y calidad de muchos métodos estadísticos. Se denominan "métodos robustos" a aquellos métodos estadísticos que son capaces de lidiar razonablemente bien con estas desviaciones (Rousseeuw and Leroy, 2005; Maronna et al., 2019).

Es también bien conocido que la modificación de una pequeña proporción de observaciones anómalas puede afectar de forma muy negativa al resultado de muchos métodos de Análisis Cluster típicamente aplicados (García-Escudero and Gordaliza, 1999). Se podría pensar que este problema puede superarse incrementando el número de clusters K a detectar y agrupar las observaciones atípicas en nuevos clusters. Sin embargo, esta idea no siempre es la más adecuada cuando el usuario fija K sin ser consciente de la presencia de contaminación. Además esta aproximación al problema puede derivar en un número bastante elevado de clusters muy difícil de interpretar. Nótese también que la detección de observaciones atípicas puede ser un objetivo en sí mismo ya que estas observaciones atípicas pueden tener un alto interés científico-económico en muchas ocasiones.

Diversas técnicas de Anáisis Cluster robusto han sido introducidas para superar la falta de robustez de las técnicas de Analisis Clusters. Algunas de estas propuestas pueden ser revisadas en Ritter (2014), García-Escudero et al. (2010) y García-Escudero et al. (2016).

En este Trabajo Fin de Grado (en adelante TFG) nos centraremos en técnicas específicas en las que investigadores del Departamento de Estadística e Investigación Operativa de la Universidad de Valladolid han jugado un papel muy importante en su introducción y desarrollo. Estas técnicas están basadas en los denominados "recorte imparciales" (Gordaliza, 1991; Cuesta-Albertos et al., 1997; García-Escudero et al., 2008, 2010).

El uso de recortes es una forma bastante habitual de proporcionar robustez a metodologías estadísticas. La idea de los recortes es eliminar una fracción α de observaciones, en las que idealmente estén incluidas las observaciones más atípicas. Posteriormente, sin esa fracción α de observaciones que son recortadas, se aplica la técnica estadística deseada y así poder neutralizar el efecto de estas observaciones atípicas. Por ejemplo, en problemas univariantes, la bien conocida media recortada elimina las fracción $\alpha/2$ de observaciones menores y la fracción $\alpha/2$ de observaciones mayores previamente a considerar la media de las observaciones no recortadas. Desgraciadamente, en problemas con datos en $\mathbb{R}^p$ para $p > 1$ no hay un orden natural para decidir cuál es la fracción $\alpha/2$ y $\alpha/2$ de observaciones "menores" y "mayores". Incluso, en Análisis Cluster, no está claro que las observaciones atípicas deban estar situadas en bordes "exteriores" del conjunto de datos y podrían ser observaciones "puente" (bridge points) situadas entre los clusters.

Para resolver esta problemática en la determinación de observaciones a recortar en Análisis Cluster fueron introducido los recortes imparciales. El término "imparcial" significa que es el propio conjunto de datos disponible el que determina cuáles son las observaciones a recortar y no decisiones subjetivas por parte del usuario. En concreto, nos centraremos en el método "TCLUST" (García-Escudero et al., 2008) y el método de "Robust Linear Grouping" (García-Escudero et al., 2009). El TCLUST permite la búsqueda de clusters con forma elípticas y dispersiones bastante generales. Su predecesor, las k-medias recortadas introducidas en Cuesta-Albertos et al. (1997), solo permitían detectar clusters esféricos con la misma dispersión. El Robust Linear Grouping, que a partir de ahora denotaremos por RLG, proporciona una extensión de las k-medias recortadas pero permitiendo agrupaciones entorno a subespacios afines en dimensiones menores a p.

Como sucede para la mayoría de los algoritmos aplicados en el uso de técnicas robustas en problemas multivariantes, los algoritmos necesarios tienen una mayor complejidad computacional que sus análogos no robustos. Esto sucederá también para el TCLUST y el RLG que requieren de la aplicación de los denominados "pasos de concentración". Estos pasos de concentración son extensiones de los pasos de concentración (C-steps) introducidos en Rousseeuw and van Driessen (1999) y utilizados en la implementación de la regresión Least Trimmed Square (LTS) y el estimador Minimum Covariance Determinant (MCD) (Rousseeuw and Leroy, 2005). Estas técnicas, LTS y MCD, fueron las primeras en ser capaces de resistir altas fracciones de observaciones contaminantes en problemas de localización y escala multivariantes.

Como ocurre con el LTS y MCD, los pasos de concentración solo permiten a las técnicas de Analisis Cluster robusto encontrar extremos (mínimos/máximos) locales de las funciones objetivo que definen el problema y es imprescindible inicializar dichos pasos desde un número no necesariamente pequeño de inicializaciones aleatorias para buscar el

mínimo/máximo global de la función objetivo.

El TCLUST está actualmente implementado en el paquete **tclust** en R (Fritz et al., 2012) en CRAN y la toolbox **FSDA** en MATLAB (Riani et al., 2012). Estas implementaciones funcionan de forma bastante satisfactoria en dimensiones no demasiado elevadas y cuando el número de grupos a buscar K no es demasiado grande. Desgraciadamente, su comportamiento no es tan satisfactorio al abandonar estas condiciones por un incremento notable en su complejidad computacional. La causa claramente constatada de esa mayor dificultad computacional es atribuible a la dificultad en encontrar inicializaciones aleatorias adecuadas.

Otra dificultad de la implementación del TCLUST mediante el paquete **tclust** en CRAN es que esa implemtación resulta bastante opaca y cerrada y no permite fácilmente modificaciones como, por ejemplo, el poder tratar problemas de verosimilitudes tipo "mixturas recortadas". Esta opacidad ha implicado la reescritura total del algoritmo para TCLUST. Esta reescritura se ha realizado combinando R y C++ para conseguir una buena eficiencia computacional y el resultado ha sido incorporado a un nuevo paquete denominado `robClus`. Se han cuidado aspectos como la documentación y el crear una estructura y notación común que permita la futura uniformización de diversos algoritmos de Análisis Cluster robusto basados en recortes imparciales en un mismo paquete.

El algoritmo disponible para el RLG eran una serie de funciones resultado del TFG Fernández Iglesias (2021). Estas funciones, suministradas por el tutor del TFG, para aplicar el RLG aprovechan una eficiente implementación en C++ pero no tenían un acople directo con las funciones en el paquete **tclust** en CRAN. Estas funciones forman parte ahora del nuevo paquete **robClus**.

Otro de los objetivos principales del TFG es desarrollar esquemas de inicialización mejorados que permitan aplicar estas técnicas de Análisis Cluster robusto basado en recortes a problemas de mayor dimensionalidad o a mayores números de grupos. Así, una de las lineas maestras de las funciones desarrolladas es permitir una nueva estructura de programas donde solo se considera la iteración completa de las inicialiazaciones que resultan inicialmente más "prometedoras" para el TCLUST y el RLG. Así se trata de aprovechar al máximo el esfuerzo computacional y no perder el tiempo aplicando pasos de concentración a soluciones iniciales con valores de la función objetivo poco prometedores. Así, se retienen solo unas pocas soluciones iniciales que, tras pocos pasos de concentración iniciales, muestren los mejores valores en la función objetivo. Solamente estas inicializaciones se iterarán por completo.

Otra de las contribuciones del TFG es que el paquete **robClus** se ha programado para permitir la iteración de las inicializaciones aleatorias de forma paralelizada y se consiguen así claras mejoras computacionales al aprovechar mucho más la capacidad computacional de los computadores modernos.

Se abordará también una nueva posible solución a esta problemática en la inicialización aleatoria mediante la "combinaciones de inicializaciones" (ensemble initialization). Estas ideas han sido exitosamente utilizadas en Lipor et al. (2021) en problemas de Análisis Cluster no robusto. Esta filosofía ha tenido que ser adapatada en este TFG al caso de los recortes imparciales. Las simulaciones realizadas muestran que esta idea parece mostrar resultados muy esperanzadores y, por tanto, la idea podría ser de gran utilidad para reducir tiempos computacionales y abordar problemas más complejos en Análisis Cluster robusto. La idea subyacente este enfoque de "agregación" de inicializaciones es tratar de

aprovechar una combinación adecuada de toda la información "parcialmente correcta" resultante de las iteraciones por pasos de concentración correspondientes a todas las inicializaciones aleatorias probadas. Se ha visto que esta combinación de información parece ser notablemente más informativa que el resultado individual de la "mejor" inicialización. Esta novedosa inicialización podría ser objeto de publicación científica.

En todo momento se está trabajando con valores de K y α que se suponen conocidos. La determinación de K es uno de los problemas más complejos a resolver en Análisis Cluster. Su determinación no puede ser considerado como un problema siempre "bien definido" ya que dicho K puede ser en muchas ocasiones dependiente del fin último del Análisis Cluster para el investigador. Este dificultad es evidentemente mayor en Análisis Cluster robusto dado que la determinación del parámetro K esta interrelacionada con la del parámetro α. Por tanto, aunque no es muy realista suponer que K y α son conocidos, esta será una suposición que realizaremos a lo largo de este TFG. Hay que tener en cuenta que los métodos que actualmente se disponen para determinar estos parámetros como, por ejemplo, los descritos en García-Escudero et al. (2011); Cerioli et al. (2018) requieren resolver estos métodos para muchas combinaciones de K y α. Por tanto, ser capaces de aplicar estas metodologías de forma eficiente y rápida es un primer paso importante para poder proponer valores razonables para K y α.

1.2. Estructura de la memoria

Las metodologías y los algoritmos disponibles para el TCLUST y RLG serán revisados en la Sección 2, junto a las implementaciones que se han usado como punto de partida de este TFG. Tras esta revisión, la Sección 3 presentará el nuevo paquete `robClus` que pretende proporcionar un primer e importante paso en el desarrollo de un entorno general para aplicar las técnicas de Análisis Cluster robusto basadas en recortes imparciales. Este paquete `robClus` se ha implementado de una forma computacionalmente eficiente, incorporando nuevo código en C++, para poder abordar problemas de dimensionalidades p alta y/o mayores números de clusters K. La Sección 3 presentará las nuevas funciones en dicho paquete `robClus` y sus parámetros principales. Estas funciones en `robClus` están directamente diseñadas para obtener e iterar solamente las inicializaciones más prometedoras y tratar de conseguir ventajas computacionales. Esta ventaja computacional será corroborada en un estudio de simulación también en la Sección 3.

Posteriormente, la Sección 4 presenta y explora la nueva inicialización mejorada basadas en agregación de inicializaciones (*ensemble initialization*) en problemas de Análisis Cluster no robusto basado en recortes. Las simulaciones que se muestran en esta sección muestran resultados muy prometedores y, por tanto, la idea podría ser de gran utilidad para reducir tiempos computacionales y abordar problemas más complejos.

La Sección 5 tratará la aplicación de las funcionalidades del paquete `robClus` en un conjunto de datos reales (MNIST) y, en particular, se usará para mostrar la eficacia de la paralelización y los esquemas de inicialización mejorados. Aunque estos datos reales son comúnmente utilizados como ejemplo en problemas de aprendizaje supervisados, en este TFG se tratarán como un problema de aprendizaje no supervisado, claramente más complejo. Los datos tienen de forma natural una dimensión p bastante más alta que los problemas previamente tratados mediante TCLUST y RLG.

Finalmente, la Sección 6 expondrá las conclusiones finales extraídas de la realización de este TFG y cuáles son las lineas futuras de las implementaciones que se pueden abordar. Estas lineas futuras pensamos que son realmente bastante prometedoras y pueden aprovechar los avances alcanzados en este TFG.

1.3. Herramientas

A continuación se muestran las herramientas utilizadas para la realización del TFG.

- ***R***: Lenguaje de programación de código abierto muy popular en la ciencia de datos. Utilizado para el desarrollo de software estadístico, ejecución de simulaciones y análisis de resultados.
- ***RStudio***: Entorno de Desarrollo Integrado (IDE) para el lenguaje R. Utilizado para el desarrollo, ejecución y depuración del código R.
- ***C++***: Lenguaje de programación compilado y orientado a objetos que destaca por su eficiencia. Utilizado para el desarrollo de los pasos básicos de los algoritmos que necesitan ser computacionalmente más eficientes.
- ***Visual Studio Code***: Entorno de Desarrollo Integrado (IDE) de propósito general. Utilizado para el desarrollo y depuración del código en C++ debido a sus mayores comodidades para el desarrollo en este lenguaje.
- ***Overleaf***: plataforma en línea para la edición y escritura de documentos LaTeX. Utilizada para la escritura de la presente memoria.

Capítulo 2

Metodología e implementaciones disponibles

2.1. Metodología y algoritmos

A continuación se presentará las dos metodologías, el TCLUST y RLG, de Análisis Cluster robusto que serán consideradas en este TFG. Estas técnicas, como ya fue indicado, están basadas en recortes imparciales. Se presentarán los algoritmos básicos que pueden ser usados para la implementación práctica de estos recortes imparciales.

2.1.1. TCLUST

Los métodos de recorte en Estadística Robusta buscan alcanzar la robustez excluyendo una proporción prefijada α de observaciones de la muestra previamente a aplicar la técnica estadística tradicional. Entre las observaciones recortadas se espera que estén incluidas todas las observaciones atípicas y así evitar su efecto negativo.

Previamente a presentar la metodología TCLUST, presentaremos la técnica pionera de la aplicación de recortes en el campo del Análisis Cluster. Esta primera metodología fueron las k-medias recortadas introducidas en Cuesta-Albertos et al. (1997). Dados un conjunto de observaciones $\{x_1, ..., x_n\}$ en $\mathbb{R}^p$ y fijados una proporción $\alpha \in [0, 1)$ de observaciones a recortar y un número K de clusters, se buscan K centroides $\{m_1...m_K\} \subset \mathbb{R}^p$ y una partición $\{R_0, R_1, ..., R_K\}$ del conjunto de índices $\{1, ..., n\}$ con $\#R_0 = [n\alpha]$ que minimice

$$\sum_{k=1}^{K} \sum_{i \in R_k} \|x_i - m_k\|^2. \tag{2.1}$$

La función objetivo es muy similar a la función objetivo tradicional en el problema de k-medias pero el sumatorio en (2.1) no considera las observaciones con índices en R_0, que serán las observaciones recortadas. Como ya comentamos, decimos que utiliza un recorte "imparcial" porque no se preestablecen direcciones privilegiadas de recorte en $\mathbb{R}^p$, sino que éstas observaciones recortadas se determinan a partir de los datos disponibles. Nótese también que si $\alpha = 0$ estamos ante el problema clásico de las k-medias.

El algoritmo básico para la implementación computacional de las k-medias recortadas, introducido en García-Escudero et al. (2003), se basa en una adaptación del algoritmo

clásico de k-medias (Lloyd, 1982) incorporando recortes. El algoritmo se basa en una filosofía de "pasos de concentración" introducida en el algoritmo fast-MCD (Rousseeuw and van Driessen, 1999) para computar el estimador de Minimo Determinante de la Covarianza (MCD). La idea es que en cada paso de concentración nos quedamos con una proporción $1-\alpha$ de observaciones cada vez más "fiables". Estas observaciones más fiables se obtienen en las zonas donde las observaciones estén más "concentradas" y serán las que se usen exclusivamente para actualizar secuencialmente los parámetros. En el caso de las k-medias recortadas, estas zonas donde esperamos que las observaciones estén más concentradas corresponden a las observaciones más próximas a los K centroides en el paso anterior. La Figura 2.1 muestra un ejemplo de tres pasos de concentración para tres centros iniciales (marcados con "$\times$") cuando se considera $K = 3$ y $\alpha = 0.05$. En tres pasos de concentración tenemos los centroides "$\times$" situados en las zonas más concentradas de los clusters.

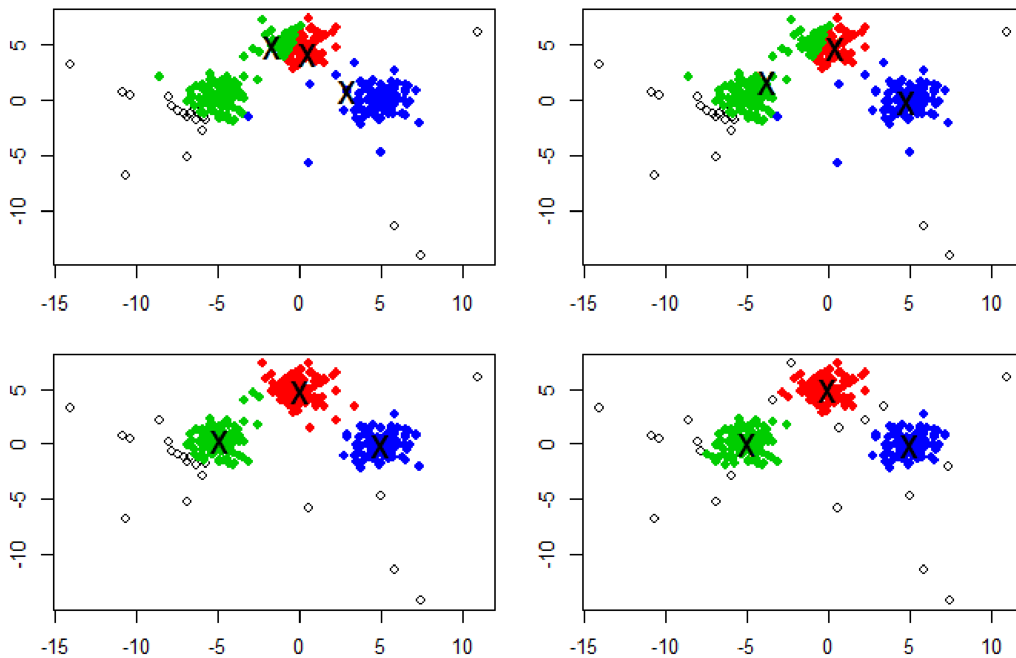


Figura 2.1: Ejemplo de la aplicación de tres pasos de concentración para centroides iniciales marcados con "$\times$" al aplicar el algoritmo para las k-medias recortadas con $K = 3$ y $\alpha = 0.05$.

La técnica TCLUST fue introducida en García-Escudero et al. (2008) como generalización de las k-medias recortadas. En TCLUST se asume un modelo probabilístico subyacente de verosimilitud tipo "clasificación" recortada que asume que las componentes están generadas por distribuciones normales p-variantes. Este enfoque, respecto a las k-medias recortadas que restringen los clusters a componentes esféricos y con similar dispersión, aporta una mayor flexibilidad como puede apreciarse en la Figura 2.2.

Así, el TCLUST busca K centroides $\{m_1...m_K\} \subset \mathbb{R}^p$, K matrices de dispersión $S_1, ..., S_K$ (matrices simetricas y definidas positivas), K pesos $p_1, ..., p_K$ con $\sum_{k=1}^{K} p_k = 1$ y una partición $\{R_0, R_1, ..., R_K\}$ con $\#R_0 = [n\alpha]$ que maximice la verosimilitud de

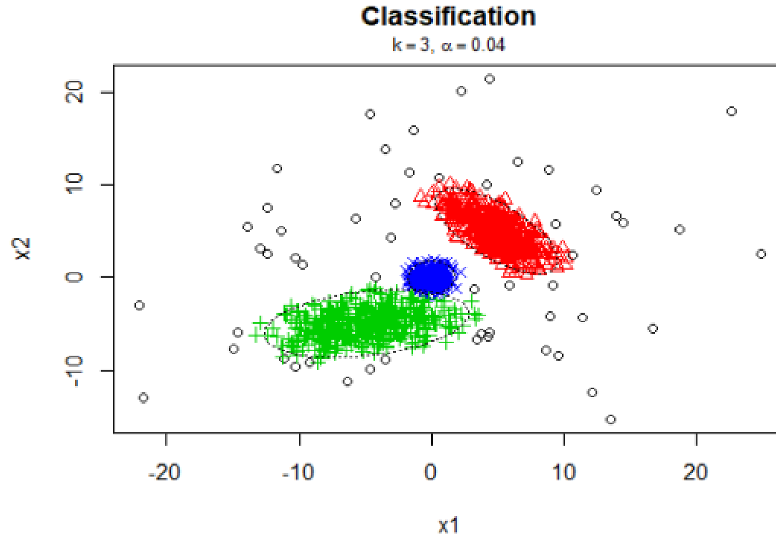


Figura 2.2: Clustering con TCLUST con $K = 3$ y $\alpha = 0.05$.

clasificación recortada

$$\sum_{k=1}^{K} \sum_{i \in R_k} \log(p_k \phi(x_i; m_k, S_k)). \qquad (2.2)$$

donde $\phi(\cdot; \mu; \Sigma)$ denota la función de densidad de una normal p-variante con media μ y matriz de varianzas-covarianzas Σ.

Si se desea, los pesos $p_1, ..., p_K$ pueden eliminarse de la función objetivo (2.2) si se quiere favorecer clusters de tamaños similares.

A mayores, el TCLUST aplica restricciones sobre las matrices $S_1, ..., S_K$ de dispersión, de forma que éste evite la detección de componentes espurios cuyas matrices de dispersión tengan determinantes $|S_k|$ que se acerquen a 0, como puede apreciarse con una de las componentes ajustadas en la Figura 2.3, cuando no se consideran restricciones.

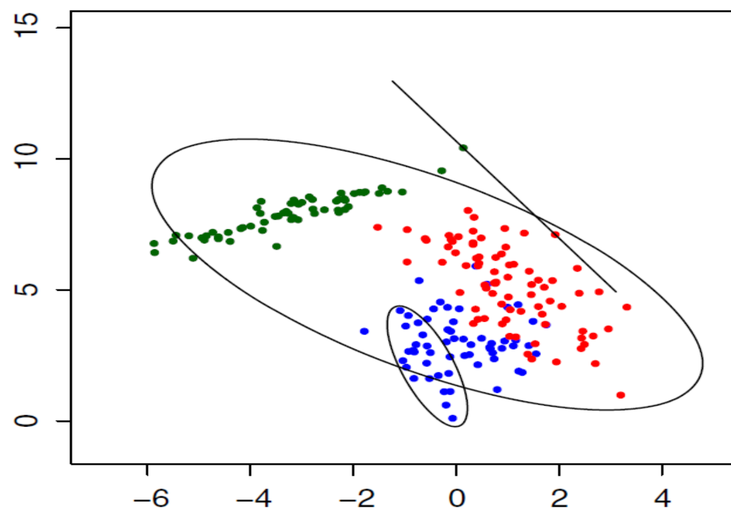


Figura 2.3: Ejemplo de componente espuria detectada.

Si $\{\lambda_j(S_k)\}_{j=1}^p$ son los autovalores de la k-ésima matriz de dispersión, fijado una constante $c \geq 1$, la restricción sobre los autovalores exige que

$$\frac{\max_{k=1,...,K;j=1,...,p} \lambda_j(S_k)}{\min_{k=1,...,K;j=1,...,p} \lambda_j(S_k)} \leq c. \tag{2.3}$$

Del mismo modo, la función objetivo en (2.2) se puede modificar para trabajar con verosimilitudes de "mixtura" recortadas (Neykov et al., 2007; García-Escudero et al., 2014). En este caso se propone maximizar la una función objetivo del tipo

$$\sum_{i=1}^{n} \log \left(\sum_{k=1}^{K} p_k \phi(x_i; m_k, S_k) \right). \tag{2.4}$$

En la maximización de (2.4) se imponen también las restricciones (2.3) sobre los autovalores.

Algoritmo para el TCLUST

La maximización de la función objetivo (2.2) con la restricción (2.3) no es un problema computacionalmente sencillo, ya que teóricamente requiere trabajar con todas las particiones posibles de los índices $\{1, .2, ..., n\}$ en $K+1$ clases (una de ellas incluyendo exactamente $[n\alpha]$ indices.

A continuación se describe el algoritmo originalmente propuesto en García-Escudero et al. (2008) y, posteriormente, optimizado en Fritz et al. (2013):

1. Se consideran `nstart` inicializaciones

   1.1. Se escogen

   $$K(p+1)$$

   observaciones al azar que definen K centroides iniciales $m_1^0, ..., m_K^0$ y K matrices de dispersión iniciales $S_1^0, ..., S_K^0$. También se definen $p_1^0, ..., p_K^0$ pesos aleatorios que sumen 1.

   1.2. Se realiza un número `niter` de pasos de concentración.

   1.2.1. Se mantiene la proporción $1-\alpha$ de observaciones con mayor contribución a la verosimilitud, calculada a partir de los parámetros de la iteración previa. De este modo, en la iteración l, dados los centroides $\{m_k^{l-1}\}_{k=1}^K$ y las matrices de dispersión $\{S_k^{l-1}\}_{k=1}^K$ junto a los pesos $\{p_k^{l-1}\}_{k=1}^K$, se calculan

   $$d_i = \max_{k=1,...,K} p_k^{l-1} f(x_i; m_k^{l-1}, S_k^{l-1})$$

   Mediante su ordenación en $d_{(1)} \leq d_{(2)} \leq ... \leq d_{(n)}$ definimos

   $$H = \{i : d_i \geq d_{([n\alpha]+1)}\}$$

   Este subconjunto H de índices se divide en $H = H_1 \cup ... \cup H_K$ con

   $$H_k = \{i \in H \text{ tal que } p_k^{l-1} f(x_i; m_k^{l-1}, S_k^{l-1}) = \max_{\underline{k}=1,...,K} p_{\underline{k}}^{l-1} f(x_i; m_{\underline{k}}^{l-1}, S_{\underline{k}}^{l-1})\}$$

1.2.2. Se usan $p_k^l = \frac{n_k}{[n(1-\alpha)]}$ para actualizar los pesos y las medias muestrales de las observaciones con índices en cada unos de los H_k para actualizar los centroides m_k^l. Sus correspondientes matrices de varianzas-covarianzas muestrales son usadas para actualizar las matrices de dispersión S_k^l. Sobre estas matrices S_k^l se imponen las restricciones en (2.3) siguiendo el procedimiento de truncamiento óptimo de autovalores que fue descrito en Fritz et al. (2013).

2. Se devuelve los parámetros y la asignación que haya conseguido un mayor valor de la función objetivo (2.2).

El algoritmo para maximizar la clasificación recortada de mixturas en (2.4) es similar a este algoritmo pero hay asignaciones parciales de cada observación a los grupos que crean unos pesos. Estos pesos son usados para actualizar los pesos p_g^l y los parámetros m_g^l son actualizados mediante medias ponderadas y las matrices de dispersión m_g^l son también actualizados mediante matrices de varianzas-covarianzas muestrales ponderadas. Las restricciones sobre los autovalores son también aplicadas en estas actualizaciones.

2.1.2. RLG

El RLG, introducido en García-Escudero et al. (2009), explota el hecho de que en dimensiones altas los datos puedan agruparse entorno a subespacios afines de dimensiones menores y se sigue aplicando la idea de los recortes imparciales para alcanzar robustez. En vez de buscar K centroides, se supone que hay relaciones entre las variables que pueden resultar en distinto número de dimensiones intrínsecas en los subespacios afines aproximantes para cada cluster. Estas dimensiones intrinsecas serán denotadas por q_k (con $0 \leq q_k < p$) para $k = 1, ..., K$. En la Figura 2.4 se puede ver un ejemplo donde los clusters se agrupan entorno a una línea $q_1 = 1$ y dos planos $q_2 = q_3 = 2$.

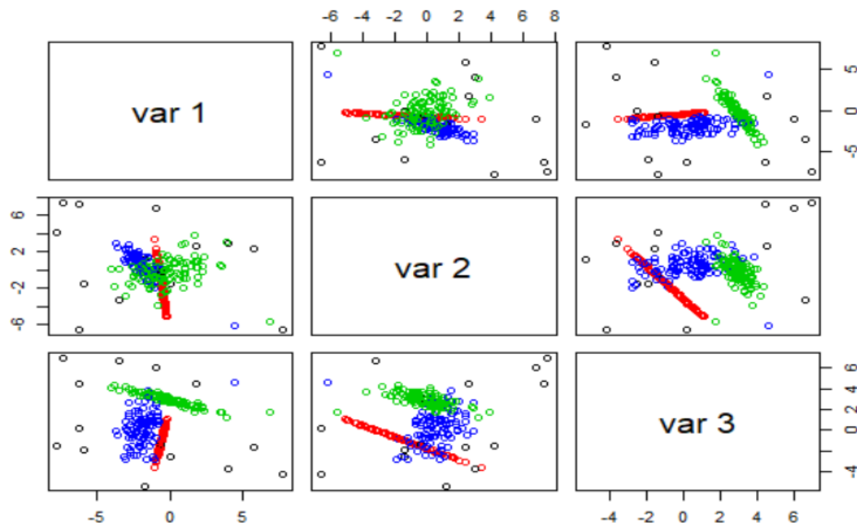


Figura 2.4: Clustering tridimensional con RLG basado en subespacios afines de dimensiones 1, 2 y 2 para un tamaño de recorte $\alpha = 0.05$.

Dado un conjunto $\{x_1, ..., x_n\}$ de observaciones en $\mathbb{R}^p$ y un número K de clusters, se buscan K subespacios afines $h_1, ..., h_K$ con dimensiones intrínsecas $q_1, ..., q_K$ y una partición $\{R_0, R_1, ..., R_K\}$ de índices $\{1, ..., n\}$ con $\#R_0 = [n\alpha]$ que minimice

$$\sum_{k=1}^{K} \sum_{i \in R_k} \|x_i - \mathrm{Pr}_{h_k}(x_i)\|^2, \tag{2.5}$$

denotando $\mathrm{Pr}_h(x)$ la proyección ortogonal de $x \in \mathbb{R}^p$ en el subespacio afín h.

Nótese que cuando $q_1 = ... = q_K = 0$ estamos ante el problema de las k-medias recortadas y cuando $K = 1$ estamos ante una robustificación del método de Análisis de Componentes Principales por recortes imparciales LTS-PCA que fue propuesto en Maronna (2005).

Algoritmo para el RLG

A continuación se describe el algoritmo propuesto en García-Escudero et al. (2009). Para ser más precisos, habría que decir que el algoritmo originalmente propuesto en García-Escudero et al. (2009) solo contemplaba el caso $q_1 = ... = q_K = p - 1$ y que correspondería a una robustificación de un algoritmo clustering combinado con regresión ortogonal.

1. Se consideran `nstart` inicializaciones aleatorias:
   - 1.1. Se escogen
     $$\sum_{k=1}^{K} (q_j + 1)$$
     observaciones al azar que definan K subespacios iniciales $h_1^0, ..., h_K^0$ y con dimensiones intrínsecas $q_1^0, ..., q_K^0$.
   - 1.2. Se realiza un número `niter` de pasos de concentración:
     - 1.2.1. Se mantiene la proporción $1 - \alpha$ de observaciones más próximas a los subespacios afines de la iteración previa. De este modo, en la iteración l, se consideran las distancias ortogonales a los subespacios afines
       $$d_i = \min_{k=1,...,K} \|x_i - \mathrm{Pr}_{h_k^{l-1}}(x_i)\|^2$$
       Mediante $d_{(1)} \leq d_{(2)} \leq ... \leq d_{(n)}$ definimos
       $$H = \{i : d_i \geq d_{([n\alpha]+1)}\}$$
       Este subconjunto H de índices se divide en $H = H_1 \cup ... \cup H_K$ con
       $$H_k = \{i \in H \text{ tal que } \|x_i - \mathrm{Pr}_{h_k^{l-1}}(x_i)\|^2 = \min_{\underline{k}=1,...,K} \|x_i - \mathrm{Pr}_{h_{\underline{k}}^{l-1}}(x_i)\|^2\}$$
     - 1.2.2. Se actualizan los subespacios afines $\{h_k^l\}_{k=1}^K$ resolviendo K problemas de Análisis en Componentes Principales clásicos que encuentran cada subespacio de dimensión q_k que mejor aproxime a las observaciones con indices en H_k.

2. Se devuelve los parámetros y la asignación que haya conseguido un menor valor de la función objetivo (2.5).

2.2. Implementaciones computacionales de TCLUST y RLG disponibles

A continuación se describen las implementaciones computacionales de las que se han partido en la elaboración de este TFG y se describen las limitaciones que han sido encontradas a estas implementaciones.

2.2.1. Paquete tclust en CRAN

Se disponía de la librería **tclust** y que es públicamente accesible desde el repositorio CRAN (Fritz et al., 2012) en `https://cran.r-project.org/web/packages/tclust/index.html`.

Esta paquete **tclust** es, sin duda, una de las implementaciones más populares de las técnicas de Análisis Cluster robusto. La librería está implementada en R y C++, lo que hace sea computacionalmente muy eficiente. Esta alta eficiencia computacional, junto a una justificación teórica adecuada, explica su éxito y popularidad en Análisis Cluster robusto. Sin embargo, el código en C++ resulta bastante opaco y no resulta fácil realizar la más mínima modificación del código, ni el poder añadir nuevas funcionalidades. Esto ha sido, por ejemplo, el caso de la extensión del algoritmo TCLUST para tratar verosimilitud de mixtura recortadas (Neykov et al., 2007; García-Escudero et al., 2014) dada en (2.4).

Esta opacidad ha hecho que sea también difícil el mantenimiento del paquete, para lo que los miembros del equipo de investigación de la Universidad de Valladolid han tenido que pedir colaboración a Valentin Teodorov en Viena, Austria.

Por tanto, el tutor del TFG estaba interesado en recuperar cierto control sobre las técnicas implementadas en el paquete **tclust**, usando un código menos opaco y más fácilmente modificable. Esto ha implicado "reprogramar" completamente la función `tclust()` desde cero, manteniendo el uso de lenguaje C++ para no perder eficiencia computacional, pero tratando de proporcionar un código más manejable que incluye modificaciones y favorezca la mantenibilidad, permitiendo extensiones futuras.

Como ya fue comentado, el paquete **tclust** en CRAN funciona muy satisfactoriamente cuando p y K son pequeños, pero necesita de un número `nstart` muy grande de inicializaciones cuando p o K no lo son. Puesto que se necesita escoger $K \times (p+1)$ observaciones al azar en cada inicialización, cuando p y/o K sean grandes, no es fácil que esas $K \times (p+1)$ observaciones garanticen una buena inicialización en el Paso 1 del algoritmo presentado para el TCLUST. Nótese que no es suficiente con que ninguna de esas $K \times (p+1)$ observaciones de la inicialización sean outliers bastante extremos. Los centroides m_k^0 y las matrices de dispersión S_k^0 iniciales se obtienen (siguiendo la propuesta en García-Escudero et al. (2008) y Fritz et al. (2013)) mediante medias y matrices de varianzas-covarianzas muestrales de K subgrupos, cada uno con $p+1$ observaciones, desde las $K \times (p+1)$ observaciones aleatorias seleccionadas para la inicialización. Por tanto, estas particiones dependen del orden en que éstas han sido aleatoriamente seleccionadas esas observaciones. De hecho, la situación ideal para el buen funcionamiento de los pasos de concentración sería una arreglo de las $K \times (p+1)$ observaciones del tipo:

$$
G \times (p+1) \quad \Rightarrow \quad \begin{array}{ll} \text{Primeras } p+1 \text{ observaciones} & \hookrightarrow \text{Cluster } 1 \\ \text{Segundas } p+1 \text{ observaciones} & \hookrightarrow \text{Cluster } 2 \\ \vdots & \\ K\text{-ésimas } p+1 \text{ observaciones} & \hookrightarrow \text{Cluster } G \end{array}
$$

La Figura 2.5 resume esta situación.

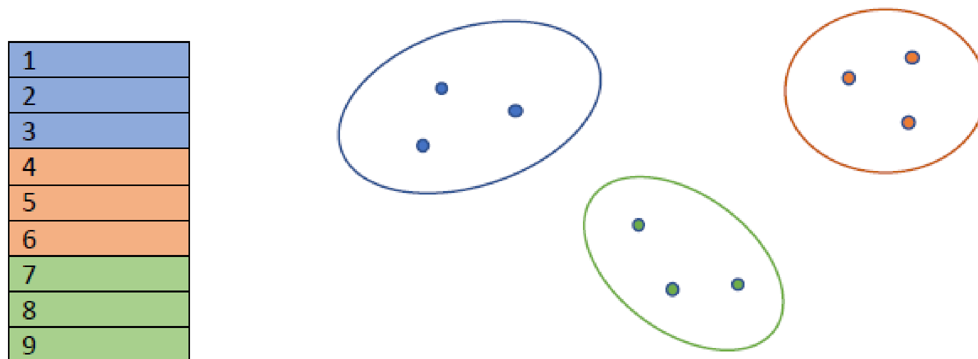


Figura 2.5: Situación ideal en la elección de las $K \times (p+1)$ observaciones aleatorias para inicializar parámetros en el algoritmos del TCLUST.

Aunque el proceso iterativo pueda resolver una mala inicalización en ciertas ocasiones, en la mayoría de los casos no se consigue arreglar. Resumiendo, esta selección aleatoria de $K \times (p+1)$ observaciones con una ordenación del tipo a la mostrada en la Figura 2.5 es extremadamente improbable cuando se manejen problemas en dimensiones p altas o, incluso, moderadamente altas. Este problema, aún con p pequeño, también va a surgir cuando el número de grupos K sea grande. Éste es el origen de la necesidad de pensar en esquemas de inicialización optimizada.

2.2.2. Función `tclust()` en R

También se disponía de una implementación del algoritmo TCLUST puramente en R y que era frecuentemente utilizada por el grupo de investigación en el que participa el tutor del TFG para aplicar técnicas de recortes imparciales no disponibles en el paquete **tclust** en CRAN. El código de esta función, proporcionada por el tutor, es accesible en el script `tclust_simplif.R` en el repositorio `https://github.com/javcres/Robclus-resources`. Se ha partido de una versión simplificada de este código pero que, a diferencia del paquete **tclust** en CRAN, sí que permite trabajar con verosimilitudes recortadas tipo mixturas.

Al estar programada puramente en R, esta implementación es muy poco eficiente a nivel computacional y requiere tiempos de computación bastante elevados incluso para problemas aparentemente muy sencillos para la aplicación de TCLUST. Adicionalmente, esta función `tclust()` en R tiene exactamente los mismos problemas para valores de K y/o p elevados del mismo tipo a los señalados en la Sección 2.2.1.

Esa función además tenía problemas con la no compatible notación (respecto al paquete **tclust** en CRAN) en los parámetros de entrada y en las salidas proporcionadas. Esta falta de coherencia notacional dificultaba su adaptabilidad con programas previos de que dispone el equipo de investigación.

2.2.3. Función `rlg()` en C++

Por último, se disponía de una implementación muy eficiente en C++ del algoritmo RLG, resultado de un TFG previo dirigido por el tutor de este TFG en Fernández Iglesias (2021). Esta implementación contaba también con una notación muy diferente respecto a las implementaciones anteriormente comentadas. Por poner un ejemplo, se usa el valor `1000` para representar las observaciones recortadas y las asignaciones de las observaciones a los K clusters se representan con valores `0`,..., `K-1`, mientras que el paquete **tclust** en CRAN utiliza el valor `0` para las observaciones recortadas y `1`,..., `K` para las asignaciones a clusters. Esta implementación tampoco contaba con la opción de paralelizar las inicializaciones aleatorias.

Resulta también muy interesante poder cargar las funciones para aplicar TCLUST y RLG de forma rápida y cómoda desde un único paquete y no ejecutando funciones aisladas totalmente desconectadas en distintos archivos de código, disponiendo además de manuales de ayuda y ejemplos para facilitar el aprendizaje y uso de las funciones, así como una interfaz unificada de formato de parámetros y resultados.

Capítulo 3

Paquete “robClus”

Una de las contribuciones principales de este TFG ha sido el desarrollo del paquete **robClus** que será presentado en este Capítulo. Las dificultades descritas en la Sección 2.2 en las implementaciones disponibles constituyen la motivación al desarrollo de este paquete.

3.1. Paquete robClus

Se ha programado un paquete en R que incorpora nuevas implementaciones de los algoritmos para el TCLUST y el RLG de forma compacta y uniforme. Este paquete resulta computacionalmente eficiente, puesto que las funciones base están programadas en C++, pero también se evita la opacidad total en la implementación dado que las funciones de alto nivel están programadas en R.

La parte central de la nueva implementación realizada en el paquete **robClus** se basa en una filosofía que pretende ahorrar tiempo computacional solo iterando completamente unas pocas soluciones más “prometedoras”. Así, se han diseñado algoritmos de inicialización `tclust_c1()` y `rlg_c1()` donde se realiza un número muy pequeño `niter1` de pasos de concentración en las `nstart` inicializaciones aleatorias Solo se escoge un subconjunto `nkeep` de soluciones no muy iteradas pero sí “prometedoras” para ser completamente iteradas hasta la convergencia o llegar a un máximo de `niter2` pasos de concentración con las funciones `tclust_c2()` y `rlg_c2()`. El número `nkeep` de soluciones a iterar completamente es bastante menor que `nstart`, obteniéndose una apreciable disminución en tiempos computacionales. Esta reducción en tiempos computacionales puede ser utilizada para considerar, por ejemplo, valores de `nstart` mayores. La filosofía es aplicable de forma análoga al TCLUST y al RLG, solo cambiando la forma en que entendemos que una solución es prometedora, por tener altos valores de (2.2) y (2.4) para el TCLUST y por tener valores pequeños de (2.5) para el RLG.

Estas ideas son de especial utilidad en problemas de alta dimensionalidad, donde la gran mayoría de las inicializaciones aleatorias acaban “estancándose” en un extremo local sin llegar nunca al extremo absoluto que requiere la solución correcta. Estos problemas de alta dimensionalidad requieren un gran número de inicializaciones aleatorias, donde no todas pueden ser iteradas completamente. Con este nuevo método de filtrado de inicializaciones se pretende llegar a un compromiso entre tiempo de cómputo y calidad de la solución final.

A continuación se describe el pseudocódigo de la filosofía básica en los algoritmos de TCLUST y RLG que se aplica con este nuevo paquete **robClus**.

Algorithm 1 Algoritmo tclust.new

```
Require: nstart, niter1, niter2, nkeep
  j ← 1
  iter_list ← <lista vacía>
  for j in 1:nstart do
     iter ← <inicialización aleatoria>
     iter ← <niter1 pasos de concentración>
     <añade iter a iter_list>
  end for
  best_iter ← <mejores nkeep elementos de iter_list>
  for iter in best_iter do
     iter ← <niter2 pasos de concentración>
  end for
  return <mejor iteración de best_iter>
```

3.1.1. Código e instalación

Se puede encontrar el código del paquete en el repositorio público de *Github* `https://github.com/javcres/Robclus-resources`, mediante el archivo comprimido de nombre `robClus_1.0.tar.gz`. Este paquete requiere de la librería **Rcpp** y la librería **RcppArmadillo** para la compilación de las funciones escritas en C++, del paquete **doParallel** para las funciones de procesamiento en paralelo y del paquete **Rtools** para instalar el paquete, pues no está subido al repositorio CRAN y hay que "construir el paquete" manualmente. Se puede instalar con la siguiente línea de código:

```
install.packages(
    "C:/<ruta_al_directorio>/robClus_1.0.tar.gz",
    repos = NULL, type="source"
)
```

Si se quiere modificar el código, se debe descomprimir `robClus_1.0.tar.gz`, modificar el código y volver a comprimir el paquete empezando por el siguiente código en R, con el directorio de trabajo situado dentro de la carpeta raíz del paquete:

```
Rcpp::compileAttributes()
# Paso recomendado para generar la documentacion (no obligatorio)
roxygen2::roxygenise()
tools::package_native_routine_registration_skeleton(
    dir = "C:/ruta_al_directorio_padre/robClus/",
    character_only = FALSE
)
```

A continuación se copia la salida (código en C++) de la última función a un archivo `/src/init.c`. Por último, hay que ejecutar el siguiente código por la terminal del PC (no por la consola de R):

```
R CMD build C:/ruta_al_directorio_padre/robClus/
```

Con todo ello se obtiene un archivo comprimido listo para ser instalado.

3.1.2. Función `tclust.new()`

Se ha tenido que reescribir la función `tclust()` en CRAN en la nueva función `tclust.new()` con una nueva aproximación, que opta por un algoritmo en 3 fases (ver el efecto del parámetro `trace=TRUE`) y siguiendo el esquema de pasar por la iteración completa de solo unas pocas `nkeep` soluciones más prometedoras.

Este es el mensaje que obtenemos al cargar el paquete **robClus**:

```
> library(robClus)
 ________  ________  ________  ________  ___       ___  ___  ________
|\   __  \|\   __  \|\   __  \|\   ____\|\  \     |\  \|\  \|\   ____\
\ \  \|\  \ \  \|\  \ \  \|\ /\ \  \___|\ \  \    \ \  \\\  \ \  \___|_
 \ \   _  _\ \  \\\  \ \   __  \ \  \    \ \  \    \ \  \\\  \ \_____  \
  \ \  \\  \\ \  \\\  \ \  \|\  \ \  \____\ \  \____\ \  \\\  \|____|\  \
   \ \__\\ _\\ \_______\ \_______\ \_______\ \_______\ \_______\____\_\  \
    \|__|\|__|\|_______|\|_______|\|_______|\|_______|\|_______|\_________\
                                                               \|_________|

Universidad de Valladolid (UVa) - https://www.uva.es
```

Como ha sido comentado, la base de la nueva implementación son las nuevas funciones `tclust_c1()` y `tclust_c2()`. Estas dos funciones están implementadas en C++ para una mayor eficiencia computacional. La función `tclust_c1()` genera inicializaciones aleatorias a las que aplica pocos `niter1` pasos de concentración. Por otro lado, la función `tclust_c2()` realiza pasos de concentración hasta la convergencia o alcanzar un número máximo de iteraciones `niter2` (se toma siempre `niter1` $\ll$ `niter2`). Las funciones `tclust_c1()` y `tclust_c2()` van a jugar también un papel muy importante en la implementación por agregación de inicializaciones que será presentada en la Sección 4.

En primer lugar, se realizan `nstart` aplicaciones de `tclust_c1()`, con sus correspondientes inicializaciones aleatorias y sus `niter1` pasos de concentración. Posteriormente, se escogen las `nkeep` mejores iteraciones y se realizan `niter2` pasos de concentración con la función `tclust_c2()`. Finalmente se devuelve la solución que maximice la función objetivo en (2.2) o (2.4). Este método nos permite reducir tiempos computacionales pero manteniendo la funcionalidad de `tclust.new()` ajustando el parámetro `nkeep`.

El parámetro `restr.fact` permite fijar la constante $c \geq 1$ que usamos en la restricción del ratio máximo de autovalores (ecuación 2.3). Valores de `restr.fact` próximos a 1 nos llevan a clusters más próximos a la esfericidad y con dispersiones más similares mientras que valores grandes de `restr.fact` lleva a clusters elípticos con mayores diferencias permitidas en dispersiones.

Se proporcionan, como es lógico, la mejor solución al problema de TCLUST alcanzada con `$cluster` con la mejor asignación (`0` las observaciones recortadas) y valor objetivo de la función dado por `$obj`, `$centers` para los vectores de localización y `$cov` para las

matrices de covarianzas, `$size` y `$weights` para el tamaño y pesos de los clusters. En `$posterior` se proporcionan las probabilidades de pertenencia a los distintos clusters y que comentaremos con más detalle a continuación.

También `tclut.new()` va a devolver el objeto `$cluster.ini` que es una matriz `nstart`$\times n$ que contiene en cada fila la asignación a clusters (0 para observaciones recortadas) en las `nstart` inicializaciones tras los `niter1` pasos de concentración. También devuelve `$obj.ini`, siendo éste un vector de longitud `nstart` con los valores alcanzadas en cada inicialización para la función objetivo tras ese número relucido de inicializaciones aleatorias.

Esta salida en `$cluster.ini` es también el punto de partida para la inicialización por agregación que se tratará en la Sección 4.

Como último método de mejora computacional de los algoritmos, se ha optado por la paralelización de las primeras fases del algoritmo (el cálculo e iteraciones iniciales de las `nstart` primeras inicializaciones).Para ello se ha utilizado la librería **doParallel** de R, incorporando a las funciones `tclust.new()` y `rlg()` dos argumentos `parallel` y `n.cores`. `parallel` es un valor booleano que indica si se quiere paralelizar o no la ejecución, `n.cores` permite elegir el número de núcleos a utilizar en la ejecución (`-1` para utilizar todos núcleos los disponibles y `-2` para utilizar todos salvo uno). Este método supone una gran mejora computacional, especialmente en problemas de alta dimensionalidad.

Cabe destacar que las comparaciones en las secciones 3.2 y 4 se han realizado sin hacer uso de la paralelización. Esto es debido a que se pretende comparar los algoritmos y no las implementaciones, para obtener resultados más generalizables. Sin embargo, dentro de las simulaciones ya se paralelizan las distintas ejecuciones, por lo que aunque no se utilicen los algoritmos en su versión paralela se está "aprovechando al máximo" la potencia computacional disponible.

Se ha incluido un parámetro `scale` que permite escalar el conjunto de datos por columnas de forma robusta, utilizando la mediana en vez de la media y el MAD (Median Absolute Deviation) en vez de la desviación típica muestral.

Además, se ha incluido un parámetro `trace` que permite seguir la ejecución de la función cuando `trace=TRUE`. En esta salida se pueden apreciar claramente las tres fases en el proceso iterativo.

```
Phase 1: obtaining 50 solutions.
  |==========================================================| 100%

Phase 2: obtaining 5 best solutions out of the intial 50 solutions.

Phase 3: applying 20 concentration steps to each of the 5 best solutions.
  |=============================-----------------------------| 50%
```

También se ha añadido la opción de trabajar con verosimilitudes recortadas tipo mixtura. Para ello se dispone de un parámetro `opt` que puede tomar los valores `"HARD"` (función objetivo (2.2)) o `"MIXT"` (función objetivo (2.4)). En este segundo caso se usa el algoritmo propuesto en García-Escudero et al. (2014). En el modo `opt="HARD"` se devuelve una matriz de probabilidades de asignación 'a posteriori' que son "duras" o

“crisp”, en otras palabras, binarias 0-1 en `$posterior`. En este caso `$posterior` toma el valor 1 en la celda (i,k) (para $i = 1, ...n$ y $k = 1, ..., K$) si la observación x_i es asignada al cluster k y 0 en el caso contrario. Por otro lado, la matriz `$posterior` cuando `opt="MIXT"` contiene en la celda (i,k) las probabilidades de pertenencia de la observación x_i al cluster k (sumando 1) aplicando la regla de Bayes. Las filas de la matriz `$posterior` son todas iguales a 0 cuando la observación x_i es recortada en ambos casos, `opt="HARD"` y `opt="MIXT"`.

Para ser más precisos, en el caso `opt="MIXT"`, el algoritmo iterativo mostrado en la Sección 2.1.1 se modifica incluyendo unas probabilidades de pertenencia

$$\tau_{k,i}^l = \frac{p_k^{l-1} f(x_i; m_k^{l-1}, S_k^{l-1})}{\sum_{\underline{k}=1}^K p_{\underline{k}}^{l-1} f(x_i; m_{\underline{k}}^{l-1}, S_{\underline{k}}^{l-1})},$$

pero que se modifican a $\tau_{k,i}^l = 0$, para todo $k = 1, ..., K$, si la observación x_i es recortada por estar entre la proporción α de observaciones que menos contribuyen a la verosimilitud de mixturas recortadas en (2.4). Estas probabilidades de pertenencia $\tau_{k,i}^l$, como sucede en los algoritmos EM clásicos, son finalmente usadas para actualizar los pesos mediante

$$p_k^l = \frac{\sum_{i=1}^n \tau_{k,i}^l}{[n(1-\alpha)]},$$

las localizaciones por

$$m_k^l = \frac{\sum_{i=1}^n \tau_{k,i}^l x_i}{\sum_{i=1}^n \tau_{k,i}^l},$$

y las matrices de dispersión por

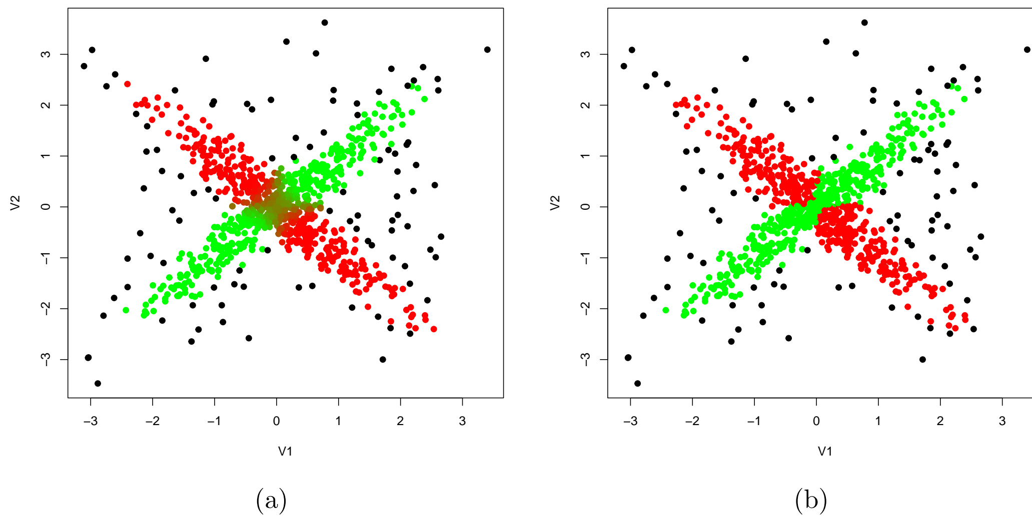
$$S_k^l = \frac{\sum_{i=1}^n \tau_{k,i}^l (x_i - m_k^l)(x_i - m_k^l)'}{\sum_{i=1}^n \tau_{k,i}^l}.$$

Otro parámetro importante es `equal.weights` que toma valores booleanos `TRUE` o `FALSE`. Cuando usamos `equal.weights=FALSE` estaremos incluyendo los pesos p_k en las funciones objetivos (2.2)) y (2.4). Por otro lado, si fijamos `equal.weights=TRUE` se impone $p_1 = ... = p_K = 1/K$, en las maximizaciones de las funciones objetivos. Esto es equivalente a elimnar estos pesos de las maximizaciones de las verosimilitudes recortadas. Esta selección `equal.weights=TRUE` favorece que los clusters detectados tengan tamaños lo más similares posibles en términos de no tener números muy diferentes de observaciones asignadas en `$cluster`.

El usar `equal.weights=FALSE` hace que algún cluster pueda desaparezca completamente, con un $p_k \approx 0$. Aunque hayamos pedido un determinado número de clusters K puede ser que `tclust.new()` nos devuelva un número de clusters menor que K cuando el parámetro `k` sea mayor que el número razonable de clusters para ese conjunto de datos. Esa es justo la idea que el trabajo García-Escudero et al. (2011) propone para determinar los parámetros K y α a elegir para un determinado conjunto de datos.

Otra utilidad de `equal.weights=TRUE` es aplicar la metodología de k-medias recortadas, con función objetivo (2.1), considerando los parámetros `restr.fact=1` y `opt="HARD"`.

En el caso `opt="MIXT"`, la Figura 3.1 muestra un caso con $K = 2$ y $\alpha = 1/9$ donde los dos clusters se cruzan con los colores RGB ajustados según la probabilidad de pertenencia a los $K = 2$ clusters para la asginación por mixturas, en comparación con la asignación “hard”. Se puede ver que en las zonas que se entrecruzan los clusters los colores se mezclan, puesto que las dos probabilidades de pertenencia a cada cluster son similares, mientras que en la asignación “hard” no hay ninguna variación en el color.



(a) (b)

Figura 3.1: Ejemplo de una asignación (a) por mixturas (b) “hard”.

Por último, se ha cambiado el método de determinación de centros en la inicialización aleatoria. Los algoritmos en García-Escudero et al. (2008); Fritz et al. (2013) proponen determinar los K centros iniciales como la media de K subgrupos con $p+1$ observaciones obtenidas de las $K(p+1)$ observaciones aleatorias en la inicialización. Se proponer ahora elegir simplemente K de las observaciones, de esas $K(p+1)$ observaciones, como centroides m_k^0 iniciales. El objetivo es ganar robustez, ya que en dimensión p elevada puede darse frecuentemente el caso de escoger por azar observaciones muy separadas entre sí, obteniendo centroides iniciales m_k^0 que no reflejen la localización real de ninguno de los clusters subyacentes.

3.1.3. Función `rlg()`

Se ha incorporado la función `rlg()` en C++ desarrollada en Fernández Iglesias (2021) al paquete **robClus**, junto a las funciones `rlg_c1()` y `rlg_c2()` también programadas en C++. Estas funciones siguen una mecánica en 3 fases análoga, basadas en la iteración completa de solo las soluciones más prometoras. Las funciones `rlg_c1()` y `rlg_c2()` son los análogos a las funciones `tclust_c1()` y `tclust_c2()`.

Para la incorporación al paquete de las funciones desarrolladas en Fernández Iglesias (2021), se ha cambiado el nombre de los parámetros de entrada y la función devuelve como una salida con notación y resultados uniformes respecto a la salida de la función `tclust.new()`. De éstos cabe destacar que, además de vectores de localización en

`$center` (por estar tratándose de subespacios afines) ahora se devuelve `$U` como una lista con K matrices que almacenan por columnas los vectores ortonormales (vectores unitarios y ortogonales) necesarios para generar cada uno de los subespacios afines. Así se cuenta también con salidas `$obj.ini` y `$cluster.ini`, que serán necesarias para la inicialización por agregación a presentar en la Sección 4. También se ha añadido un argumento `trace` que permite al usuario seguir la ejecución de las iteraciones en las tres fases. Por último, se permite la paralelización, mediante los argumentos `parallel` y `n.cores`, en la búsqueda de la inicialización más prometedora de las `nstart` inicializaciones aleatorias probadas, de un modo análogo a lo realizado para la función `tclust.new()`

3.1.4. Funciones **simula.tclust()** y **simula.rlg()**

Se han creado funciones `simula.tclust()` y `simula.rlg()` que permiten simular conjuntos de datos para poder explorar el funcionamiento de las técnicas TCLUST y RLG, respectivamente. Estas funciones permiten personalizar el numero de observaciones, el número de clusters, la dimensión de los datos y las características de los clusters.

Estas funciones van a ser también usadas en los estudios de simulación planteados en este TFG.

Función para simular en TCLUST

Esta función `simula.tclust()` genera `n` observaciones aleatorias siguiendo dos tipos diferentes de escenarios e incluyendo una fracción del 10 % de observaciones contaminantes. En el escenario `type=1` se consideraran `k` clusters (`k` puede ser 3 o 6) esféricos y con una misma dispersión pero con localizaciones bastante próximas. Por otro lado, el escenario `type=2` corresponde a `k` clusters elípticos con diferentes orientaciones y con cierto solapamiento en algunos caso. Los dos escenarios han sido generados con componentes normales cuyas matrices de varianzas-covarianzas satisfacen la restricción de los autovalores en (2.3) para $c = 1$ en el caso `type=1` y con $c = 9^2 = 81$ para `type=2`. La función `simula.tclust()` genera conjuntos de datos en dimensión p ($\mathtt{p} \geq 2$), donde la información que permite distinguir los `k` clusters está en las primeras dos variables y las $\mathtt{p}-2$ variables restantes son variables "ruido" que no proporcionan información sobre la estructura de grupos y tienen cada una de ellas distribuciones normales estándar independientes. El parámetro `balanced = 1` genera clusters con los mismos tamaños (todos con el 30 % de las observaciones para `k=3` y del 15 % para `k=6`). El parámetro `balanced = 2` crea grupos de distinto tamaño (grupos de tamaños 25 %, 30 % y 35 % para `k=3` y dos grupos con el 12.5 %, dos con 15 % y dos con 17.5 % para `k=6`).

Se llama a la función mediante la orden `simula.tclust(n,p,k,type,balanced)`. La función tiene especificados valores por defecto de esos parámetros si se desean especificar menos parámetros.

La Figura 3.2 muestra ejemplos de esos conjuntos de datos cuando `n=400` en los dos escenarios y distintos números de grupos, usando colores distintos para los clusters y el color negro para las observaciones contaminantes.

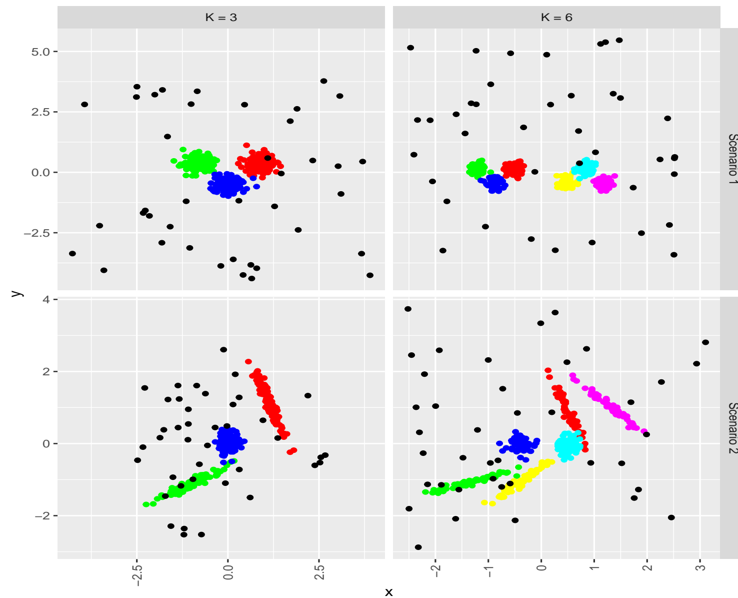


Figura 3.2: Los dos escenarios dados por el parámetro `type` (en filas) y el parámetro `k` (en columnas) al aplicar la función `simula.tclust()`. Los datos han sido previamente estandarizados para que sus escalas sean comparables.

Función para simular en RLG

La función `simula.rlg()` genera ahora `n` datos incluyendo una fracción `alpha` de observaciones contaminantes que pueden ser usadas para ser testar el funcionamiento del método RLG. Estas observaciones son generadas en dimensión `p` con `p` $\geq$ 2. Los datos son generados de tal forma que los observaciones no-contaminadas se agrupan entorno a $K = 3$ subespacios afines en dimensiones $q_1 = q_2 = q_3 =$ `q` para $0 \leq$ `q` $<$ `p`. Se pueden fijar también un parámetros `var` que controla la "dispersión" en los errores ortogonales y el parámetro `sep.means` que controla la separación entre los vectores de media de los subespacios afines. La función se llama de la forma `simula.rlg(q,p,n,var,sep.means,alpha)`, donde se dispone también de valores por defecto para estos parámetros. En la figura 3.3 se puede ver un ejemplo de datos simulados mediante la función `simula.rlg()` utilizando los parámetros `q=2, p=5, n=400, var=0.01,sep.means=0.00`

3.1.5. Datasets

Se han incorporado archivos de datos `M5data` y `LG5data` que son utilizados en los ficheros de ayuda de las funciones `tclust.new()` y `rlg()`, respectivamente. Estos datos se cargan directamente en el espacio e trabajo del R desde el paquete **robClus** con la orden `data()`. En la figura 3.4 se pueden ver representaciones gráficas de éstos.

El archivo de datos `M5data` es el mismo que ya aparecía en el paquete **tclust** en CRAN y ha sido utilizado como ejemplo de "benchmark" en bastantes artículos dedicados al Análisis Cluster robusto. Por otro lado, los datos `LG5data` son el resultado de una aplicación de `simula.rlg (q = 2, p = 10, n = 200, var = 0.001, sep.means = 0.1, alpha = 0.1)`. Es decir, son datos en $\mathbb{R}^{10}$ agrupados entorno

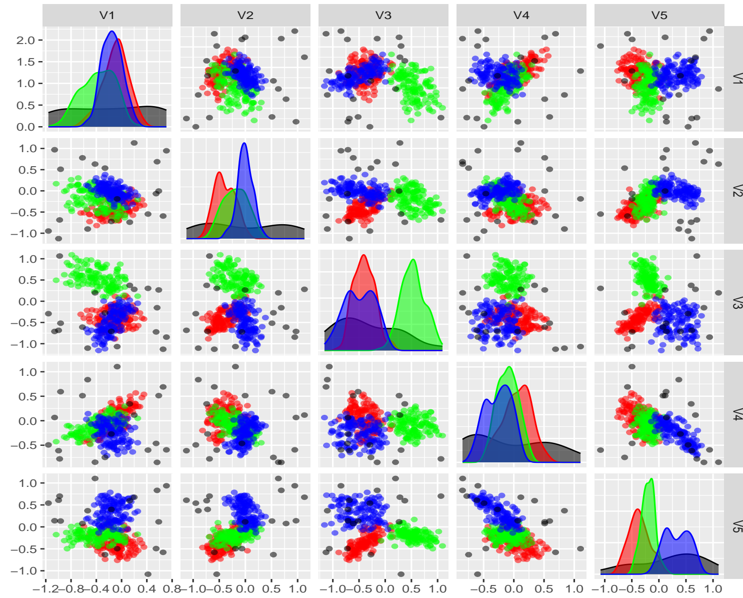


Figura 3.3: Gráfico de pares correspondiente a datos simulados mediante la función simula.rlg().

a 3 subepacios afines de dimensiones $q_1 = q_2 = q_3 = 2$ y con un 10 % de contaminación. En ambos conjuntos de datos se devuelven las etiquetas "reales" de los clusters en la última columna (con 0 para las observaciones contaminadas) en la última columna de esos datos para poder evaluar la precisión de los distintos métodos aplicados.

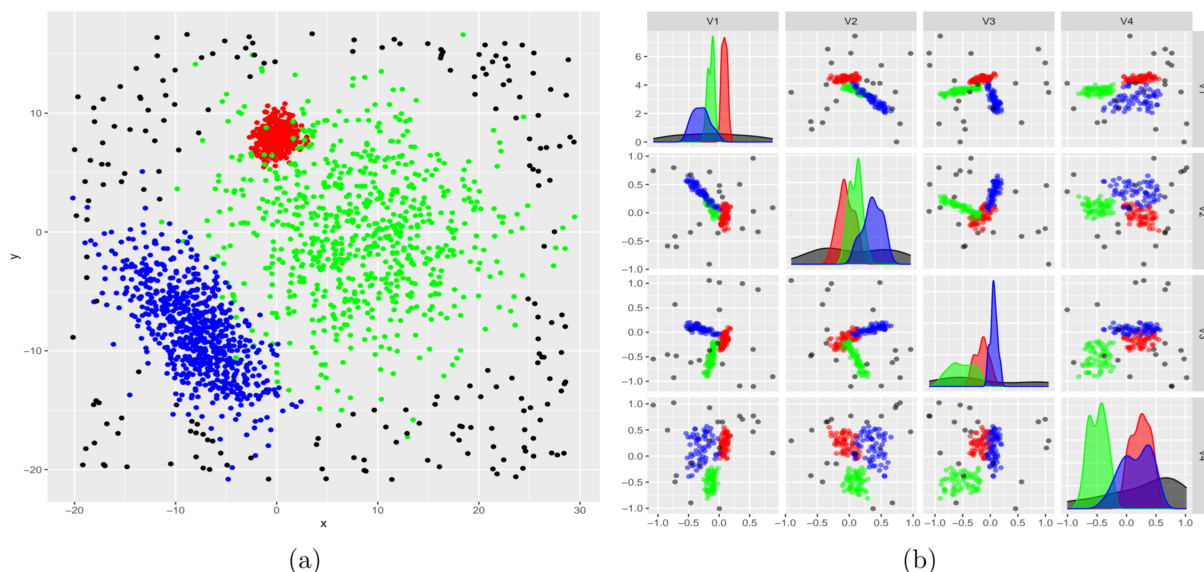
3.1.6. Documentación de las funciones

El paquete **robClus** y todas sus funciones han sido debidamente documentadas. Esta ayuda es accesible con la orden help(<funcion>). A modo de ejemplo, en la Figura 3.5 se muestran capturas de pantalla con la ayuda parcial del paquete y la función tclust.new().

3.2. Simulaciones en la aplicación de TCLUST

En esta sección se van a comparar las distintas implementaciones disponibles (que fueran comentadas en la Sección 2.2) y las obtenidas por la aplicación del paquete **robClus**. Se consideran aspectos como el valor alcanzado en las funciones objetivos y se prestará especial atención a los tiempos de ejecución. Para hacer esta comparación de tiempos, todas las simulaciones de este TFG se han realizado con una máquina con procesador Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz con 10 procesadores principales y 20 lógicos, con sistema operativo Windows 10, 32GB de memoria RAM y versión 4.1.0 de R. Todas las simulaciones se realizan en paralelo utilizando la librería **parallel** en CRAN. El código completo de estas simulaciones puede verse en el archivo simulaciones_tclust.Rmd del repositorio https://github.com/javcres/Robclus-resources.

Los métodos utilizados se van a representar como *Métodos 1-4* (esta numeración será la usada en los gráficos que van a mostrar los resultados obtenidos en la simulación). En



(a) (b)

Figura 3.4: Gráficos de los datasets (a) `M5data` y (b) las cuatro primeras variables de `LG5data`.

los 4 métodos se utilizarán el mismo número `nstart` de inicializaciones aleatorias.

Método 1 la función `tclust()` implementada exclusivamente en R (con muchas funcionalidades pero sin tener partes implementadas en C++).

Método 2 la función `tclust()` en el paquete **tclust** disponible en CRAN con `nstep=20` pasos de concentración.

Método 3 la función `tclust.new()` en el paquete **robClus** tratando de emular al *Método 2* para lo que se eligen `niter1 = 3` y `niter2 = 20 - 3 = 17` (así teniendo también un mismo número máximo de pasos de concentración igual a 20) y `nkeep=nstart` de tal forma que todas las inicializaciones son iteradas.

Método 4 la función `tclust.new()` en el paquete **robClus** tratando de aprovechar la posibilidad de iterar solo las soluciones más "prometedoras". Para ello se elige `niter1 = 3` y `niter2 = 20 - 3 = 17` (mismo número máximo de pasos de concentración) pero ahora `nkeep=10` ($<$ `nstart`).

Como vamos a usar datos generados con la función `simula.tclust()`, en todos los casos, elegiremos un valor del parámetro `restr.fact` mayor que el c necesario en la restricción (2.3) y, así, tomaremos `restr.fact=2` para el ecenario `type=1` y `restr.fact=100` para el escenario `type=2`.

Vamos a realizar una primera simulación sencilla comparando entre los 4 métodos anteriormente comentados cuando obtenemos con `simula.tclust()` 100 muestras aleatorias obtenidas cada una con parámetros `n=800` y `p=2`, los dos valores de los parámetros `k` (3 y 6) y `balance=1`. Se aplican los *Métodos 1-4* con las opciones `opt="HARD"` (el único caso que se puede manejar con el paquete **tclust** en CRAN) y `equal.weights=FALSE` y tomando para todos `nstart=100` y un número máximo

robClus-package {robClus} R Documentation

Robust Clustering Based on Trimming

Description

Provides functions for robust trimmed clustering. The methods are described in Garcia-Escudero (2008) <doi:10.1214/07-AOS515>, Fritz et al. (2012) <doi:10.18637/jss.v047.i12>, Garcia-Escudero et al. (2009) <doi:10.1111/j.1467-9868.2008.00682.x> and others.

Details

The DESCRIPTION file:

Package:	robClus
Type:	Package
Encoding:	UTF-8
Title:	Robust Clustering Based on Trimming
Version:	1.0
Date:	2023-03-22
Author:	Javier Crespo Guerrero, Luis Angel García Escudero, Agustín Mayo Iscar
Maintainer:	Javier Crespo Guerrero <javier.crespo.guerrero@estudiantes.uva.es>
Description:	Provides functions for robust trimmed clustering. The methods are described in Garcia-Escudero (2008) <doi:10.1214/07-AOS515>, Fritz et al. (2012) <doi:10.18637/jss.v047.i12>, Garcia-Escudero et al. (2009) <doi:10.1111/j.1467-9868.2008.00682.x> and others.
License:	To be decided...
Imports:	Rcpp (>= 1.0.7), MASS, doParallel
LinkingTo:	Rcpp, RcppArmadillo
References:	Fritz, H.; Garcia-Escudero, L.A.; Mayo-Iscar, A. (2012), "tclust: An R Package for a Trimming Approach to Cluster Analysis". Journal of Statistical Software, 47(12), 1-26. Garcia-Escudero, L.A.; Gordaliza, A.; Matran, C. and Mayo-Iscar, A. (2008), "A General Trimming Approach to Robust Cluster Analysis". Annals of Statistics, Vol.36, 1324-1345. García-Escudero, L. A., Gordaliza, A., San Martin, R., Van Aelst, S., & Zamar, R. (2009). Robust linear clustering. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 71, 301-318. García-Escudero, L. A., Gordaliza, A., & Mayo-Íscar, A. (2014). A constrained robust proposal for mixture modeling avoiding spurious solutions. Advances in Data Analysis and Classification, 27-43.
RoxygenNote:	7.2.3

Index of help topics:

```
LG5data                 LG5data dataset
M5data                  M5data dataset
rlg                     Robust Linear Grouping
rlg_c1                  Internal function for concentration steps
                        (initializations) in rlg
rlg_c2                  Internal function for concentration steps
                        (refinement) in rlg
robClus-package         Robust Clustering Based on Trimming
simula.rlg              Simulate contaminated data set for applying rlg
simula.tclust           Simulate contaminated data set for applying
                        TCLUST
tclust.new              TCLUST method for robust clustering
tclust_c1               Internal function for concentration steps
                        (initializations) in tclust.new
tclust_c2               Internal function for concentration steps
                        (refinement) in tclust.new
```

(a)

tclust.new {robClus} R Documentation

TCLUST method for robust clustering

Description

tclust.new searches for k (or less) clusters with different covariance structures in a data matrix x. Relative cluster scatter can be restricted by a constant value restr.fact. For robustifying the estimation, a proportion alpha of observations is trimmed. In particular, the trimmed k-means method is represented by the tclust.new method, by setting parameters restr.fact = 1, opt = "HARD" and equal.weights = TRUE.

Usage

```
tclust.new(
  x,
  k,
  alpha = 0.05,
  nstart = 50,
  niter1 = 3,
  niter2 = 20,
  nkeep = 5,
  equal.weights = FALSE,
  restr.fact = 5,
  opt = "HARD",
  scale = FALSE,
  parallel = FALSE,
  n.cores = -1,
  zero_tol = 1e-16,
  trace = 0
)
```

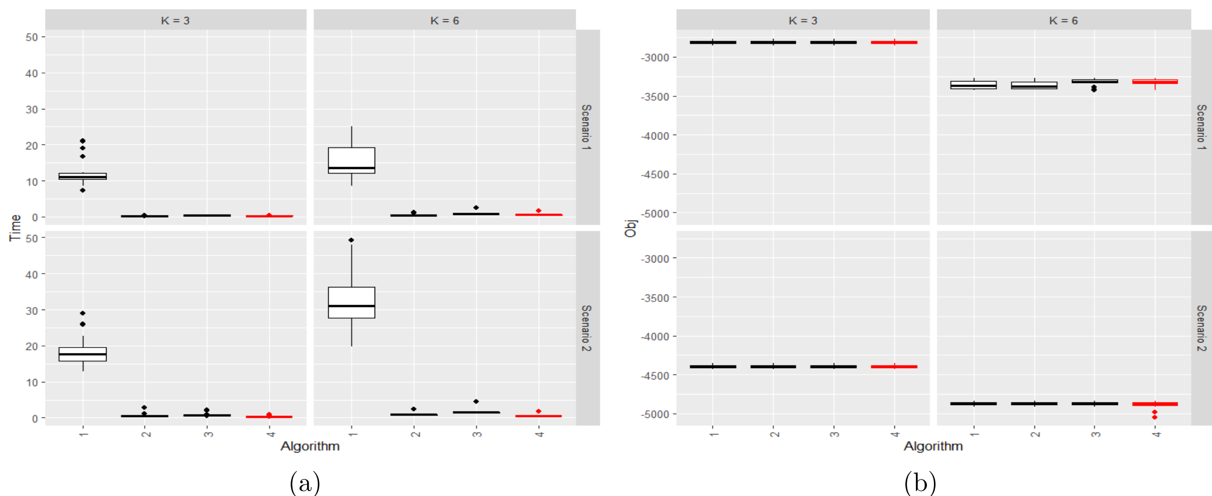
Arguments

`x`	A matrix or data.frame of dimension n x p, containing the observations (row-wise).
`k`	The number of clusters initially searched for.
`alpha`	The proportion of observations to be trimmed.
`nstart`	The number of random initializations to be performed.
`niter1`	The number of concentration steps to be performed for the nstart initializations.
`niter2`	The maximum number of concentration steps to be performed for the nkeep solutions kept for further iteration. The concentration steps are stopped, whenever two consecutive steps lead to the same data partition.
`nkeep`	The number of iterated initializations (after niter1 concentration steps) with the best values in the target function that are kept for further iterations
`equal.weights`	A logical value, specifying whether equal cluster weights (TRUE) or not (FALSE) shall be considered in the concentration and assignment steps.
`restr.fact`	The constant restr.fact >= 1 constrains the allowed differences among group scatters in terms of eigenvalues ratio. Larger values imply larger differences of group scatters, a value of 1 specifies the strongest restriction.
`opt`	Define the target function to be optimized. A classification likelihood target function is considered if opt="HARD" and a mixture classification likelihood if opt="MIXT".
`scale`	A robust centering and scaling (using the median and MAD) is done if TRUE.

(b)

Figura 3.5: Ejemplo de la documentación general disponible para el paquete **robClus** en (a) y la especifica para la función `tclust.new()` en (b).

de pasos de concentración igual a 20. La Figura 3.6 se muestran los resultados de la simulación.



(a) (b)

Figura 3.6: Resultados de la primera simulación incluyendo los 4 métodos *Métodos 1-4*: (a) tiempo computacional y (b) valores de la función objetivo.

Podemos observar que la Figura 3.6 que valores en la función objetivo son similares en los 4 métodos (quizás los *Métodos 3-4* correspondientes a `tclust.new()` tienen un valor ligeramente mejor). Sin embargo, los tiempos del *Método 1*, con la implementación puramente en R, son muy superiores a los del resto, lo que dificulta la comparación y ralentizaba bastante las simulaciones. El *Método 1* funciona bien pero no es nada eficiente en comparación con el resto, incluso en este caso bastante simple, por lo que vamos a *excluirlo* definitivamente del resto de simulaciones más pesadas computacionalmente.

Con el conjunto reducido de métodos (*Métodos 2-4*), vamos a hacer una segunda simulación más pesada. Con este fin se usa la función `simula.tclust()` para obtener 80 muestras aleatorias con diversos parámetros `n` ($n = 200, 1000$) y `p` ($p = 4, 10$), los dos valores de los parámetros `k` (3 y 6) y `balance=1` (se han corrido también el caso `balance=2` pero, por brevedad se omiten estos resultados, pues las conclusiones que obtenemos son similares al caso balanceado aunque se pueden consultar en *Github* `https://github.com/javcres/Robclus-resources` los resultados obtenidos en la simulación). Para estos conjuntos simulados de datos, se aplican los *Métodos 2-4* con las ópciones `opt="HARD"` y `equal.weights=FALSE`. Se han probado distintos números de inicializaciones aleatorias (nstart=400 y nstart=1000), un mismo número máximo de pasos de concentración igual a 20.

La Figura 3.7 muestran los resultados para `n=200` y `nstart=400`. En estas gráficas representamos el tiempo (`Time`) en el eje de ordenadas y también el valor alcanzado en la función objetivo (`Obj`).

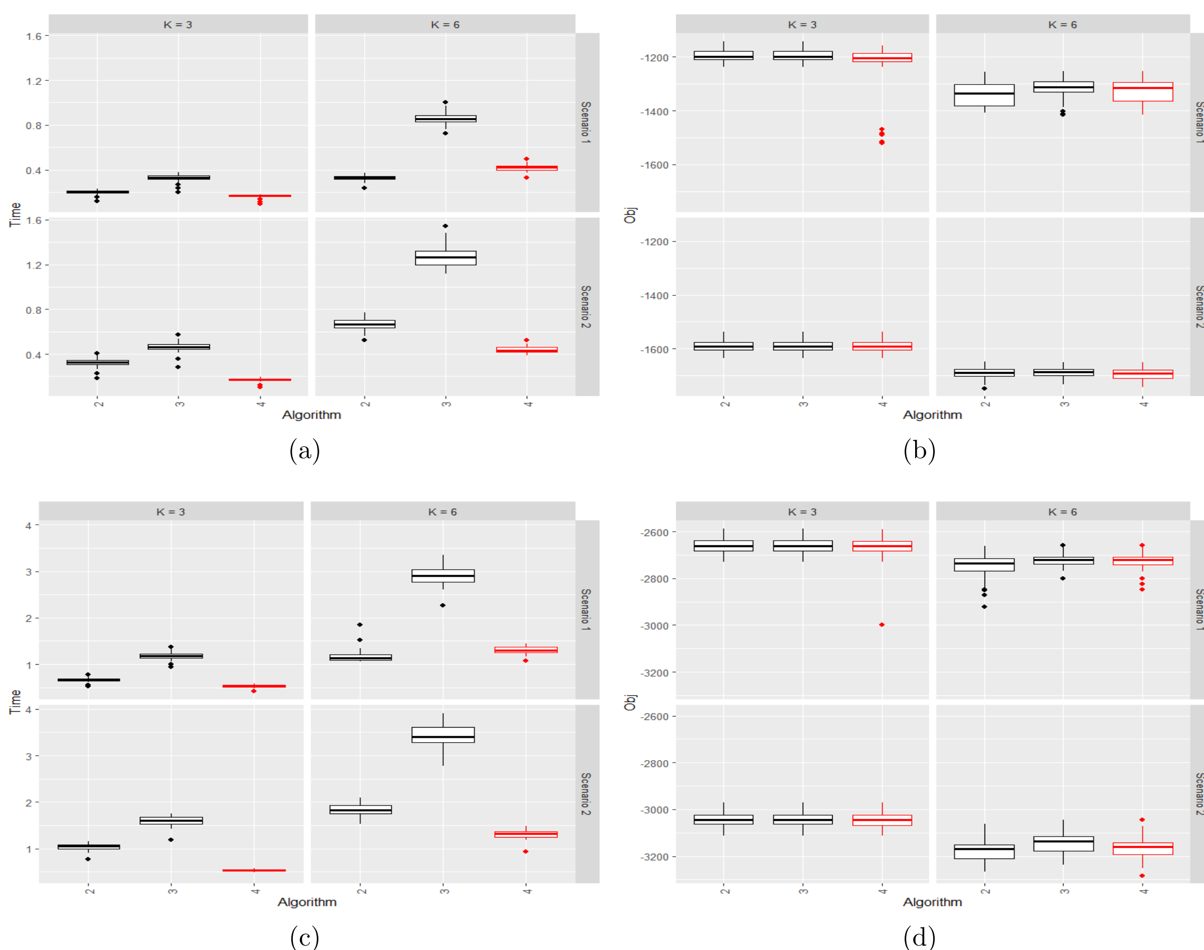


Figura 3.7: Resultados de tiempos computacionales y valores alcanzados en la función objetivo cuando `n=200` y `nstart=400` para `p=4` en (a) y (b) y para `p=10` en (c) y (d). Los arreglos de gráficos 2×2 consideran los dos posibles `type` (en filas) y los dos posibles `k` (en columnas).

Se observa que la función `tclust()` del CRAN (*Método 2*) y `tclust.new()` (*Métodos 3-4*) sacan valores de la función objetivo muy similares, siendo `tclust.new()` ligeramente mejor en algunos casos. Cabe recordar que en `tclust.new()` se cambia la forma de determinar los centros inciales, lo que puede estar influyendo positivamente en los resultados. El *Método 4*, con solo iteración completa de las inicializaciones más prometedoras, permite obtener valores de la función objetivo muy similares y con una cierta ganancia de tiempo, aunque en este caso no sea excesiva esa diferencia. El *Método 3* donde `tclust.new()` simula al `tclust()` en CRAN obtiene tiempos de ejecución algo mayores pero en todo caso del mismo orden que los del paquete en CRAN. Esto es debido a que `tclust.new()` está diseñado para seleccionar las `nkeep` mejores, utilizando estructuras de datos y pasos intermedios para almacenarlas que no tienen mucho sentido considerar cuando `nkeep = nstart`. Se puede ver también que el *Método 2* en el caso `p=4` y `k=6` tiene un menor tiempo computacional. Esto es debido a que ese problema es muy sencillo y se puede resolver en pocos pasos de concentración. La función `tclust_c1()` que utiliza `tclust.new()` no dispone de criterio de parada al realizar las `niter1` primeras inicializaciones, por lo que a pesar de que converja pronto éste sigue haciendo más etapas mientras que `tclust()` en CRAN sí que implementa ese criterio de parada desde los primeros pasos de concentración. En cualquier caso, la diferencia no es demasiado notable y no será apreciable para casos más complejos.

La Figura 3.8 muestran los resultados para `n=200` y `nstart=1000` con conclusiones bastante parecidas a las anteriores, pero se observa que hay mayores diferencias en los tiempo de ejecución.

La Figura 3.9 muestra los resultados para `n=100` y `nstart=400`. Al ser éste un problema más complicado, se puede apreciar en la Figura 3.9 mucho mejor que el *Método 4* resulta en mucho mejor tiempo de ejecución sin sacrificar demasiada precisión. Los *Métodos 2-3* tienen tiempos de ejecución comparables, aunque los del *Método 3* son algo peores, pero siempre dentro del mismo orden de complejidad. También se observa que el *Método 2* tiene mayor variabilidad en en el valor de la función objetivo que se alcanza.

También, se muestran en la Figura 3.10 los resultados obtenidos en la simulación para `n=1000` y `nstart=1000`. Este es el problema más complejo, por el menor tamaño muestral y el menor número de incializaciones, y es en el que mejor se puede apreciar la disminución del tiempo de ejecución, sin sacrificar resultados, que se obtiene usando el `Método 4` solamente iterando las inicializaciones más prometedoras.

Las Figuras 3.11 y 3.12 nos muestran los errores de mal-clasificación al aplicar los *Métodos 2-4* en el estudio de simulación realizado cuando estamos en dimensión $p = 4$ (Figura 3.11) y en dimensión $p = 10$ (Figura 3.12). Vemos que los *Métodos 2 y 4* parecen más inestables en el error que consiguen, mientras que el *Método 3* por lo general tiene menos varianza. El *Método 2* está resultando en mayores errores, esto podría estar influido por la selección inicial de los centros, que en el *Método 2* se está haciendo mediante la media de subgrupos con $p+1$ observaciones y en el resto tomando como centro inicial una única observación en cada subgrupo. En general los *Métodos 3 y 4* generan los mismos errores, pero vemos que hay casos como el gráfico inferior derecho de la Figura 3.11 (a) donde peores valores de la función objetivo resultan en un mejor error de clasificación, por lo que el *Método 4* sale beneficiado. Se mantienen las conclusiones anteriores, aunque en este caso parece que hay, además, mucha menos variabilidad en los errores.

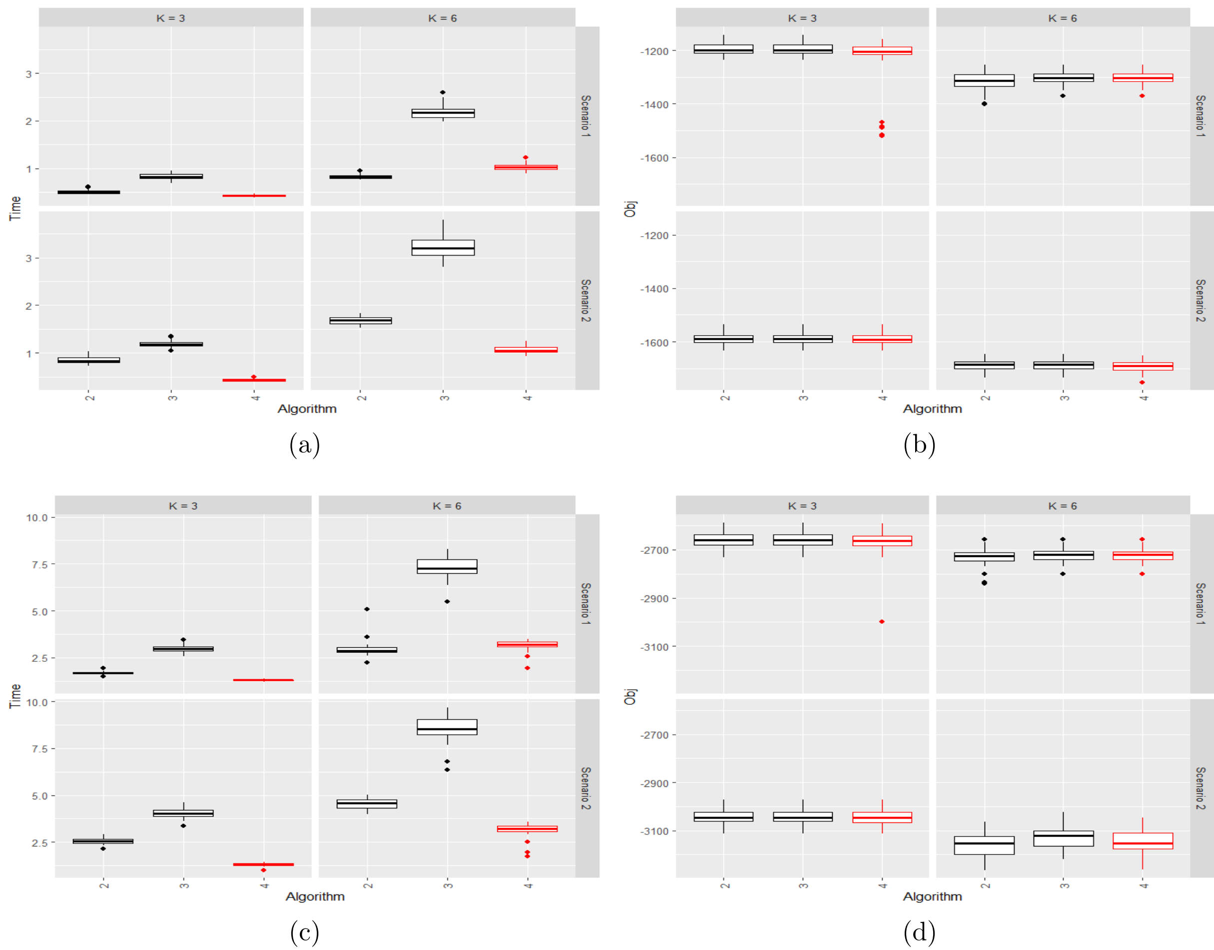


Figura 3.8: Resultados de tiempos computacionales y valores de la función objetivo cuando `n=200` y `nstart=1000` para `p=4` en (a) y (b) y para `p=10` en (c) y (d).

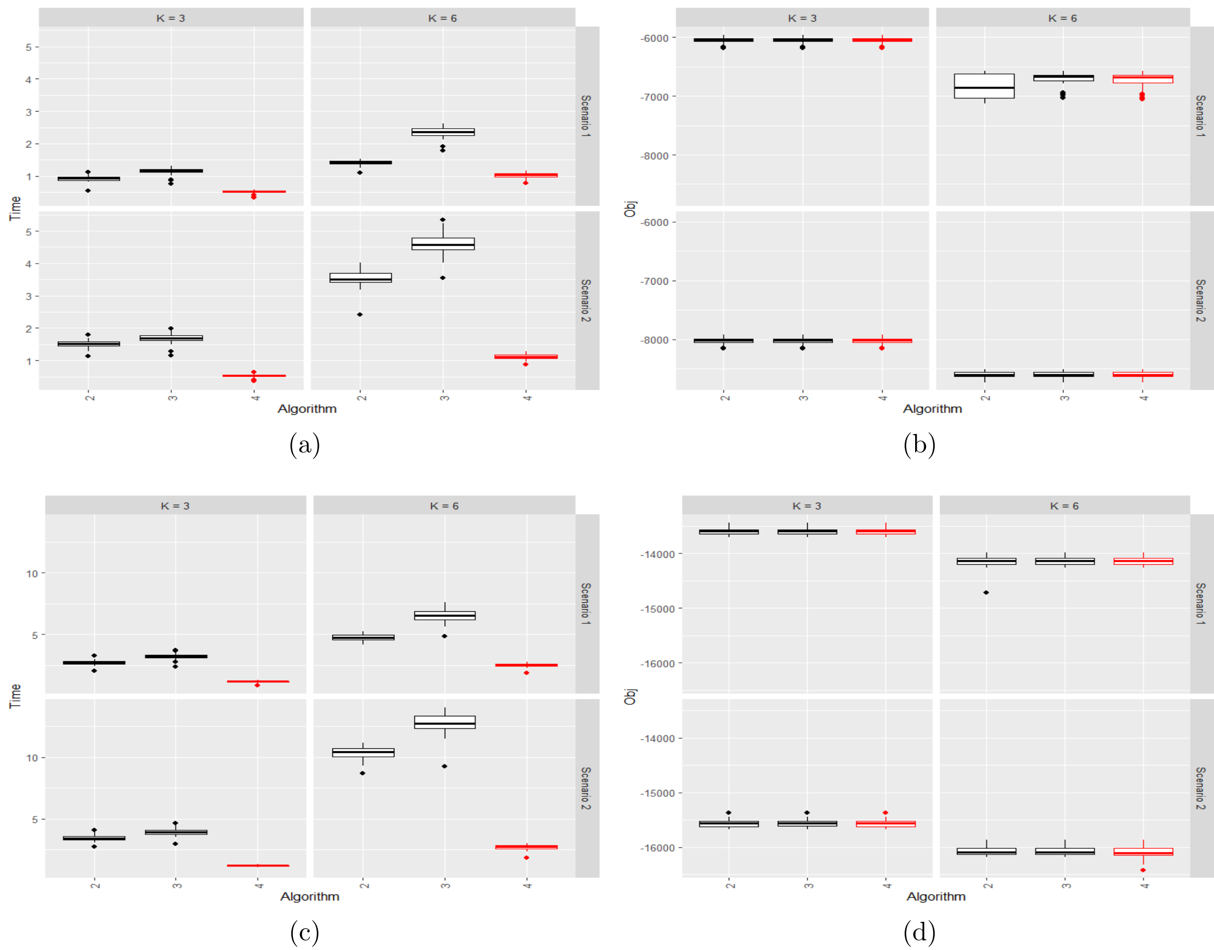


Figura 3.9: Resultados de tiempos computacionales y valores de la función objetivo cuando n=1000 y nstart=400 para p=4 en (a) y (b) y para p=10 en (c) y (d)

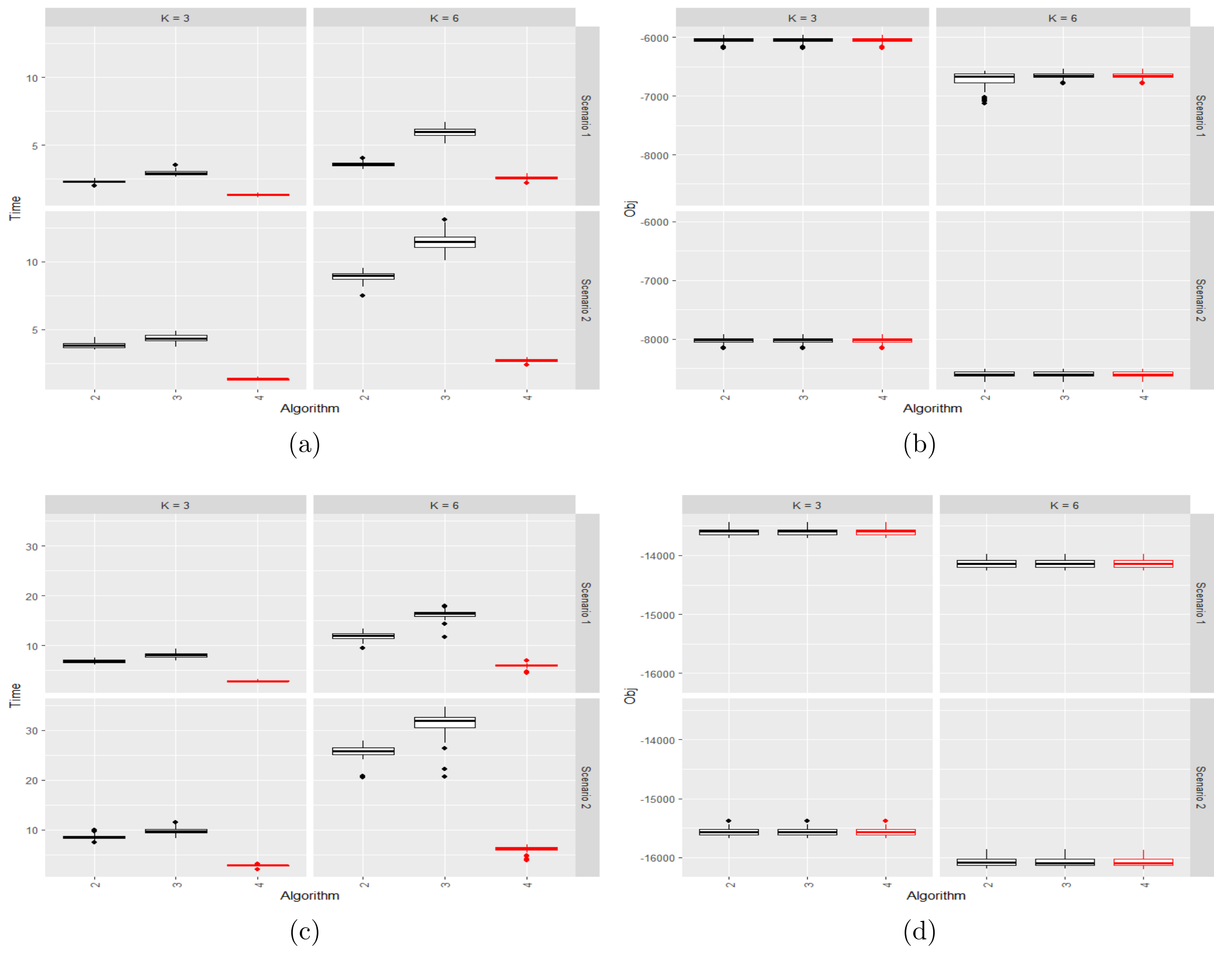


Figura 3.10: Resultados de tiempos computacionales y valores de la función objetivo cuando `n=1000` y `nstart=1000` para `p=4` en (a) y (b) y para `p=10` en (c) y (d)

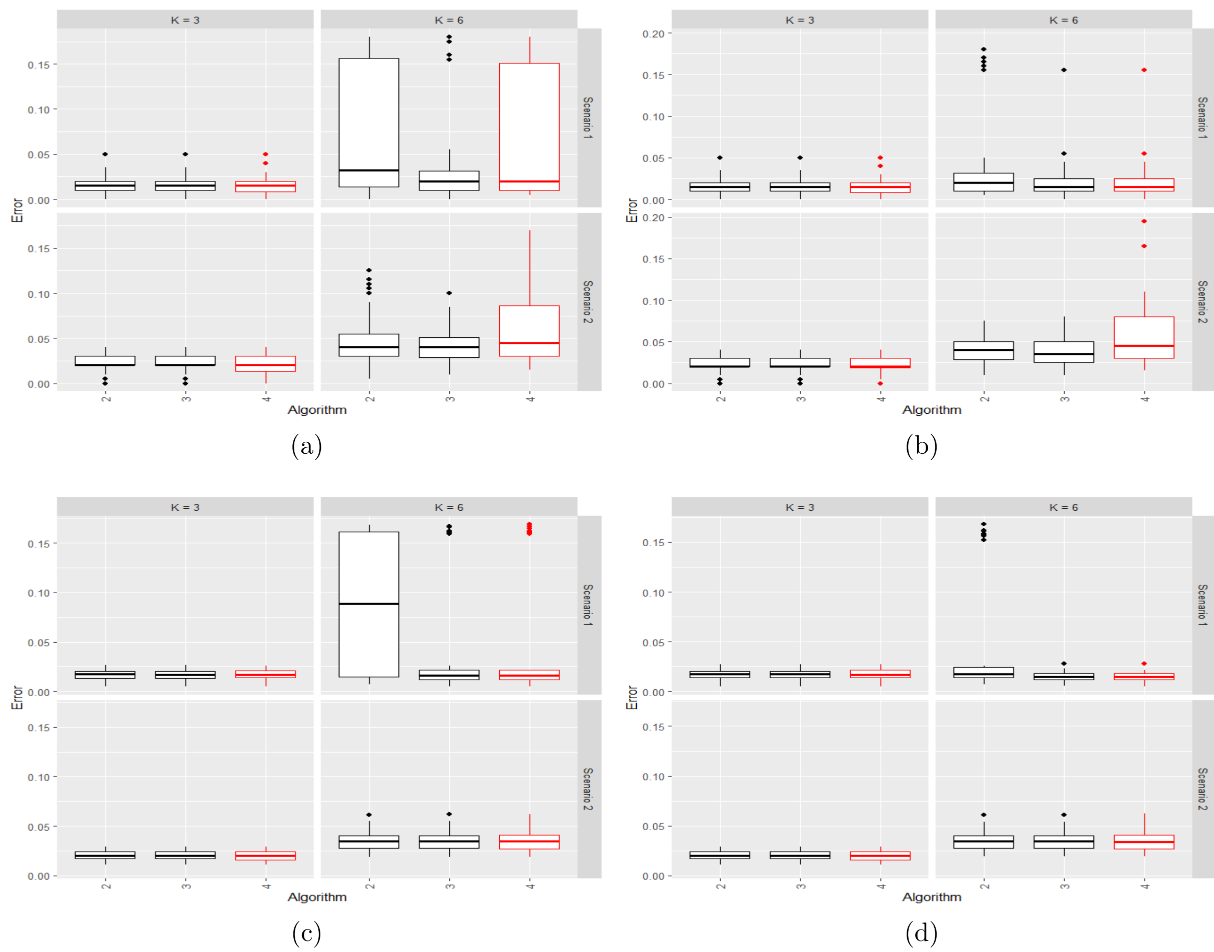


Figura 3.11: Errores en el estudio de simulación en dimension p=4 cuando n=200 y nstart=400 en (a), n=200 y nstart=1000 en (b), n=1000 y nstart=400 en (c) y n=1000 y nstart=1000 en (d).

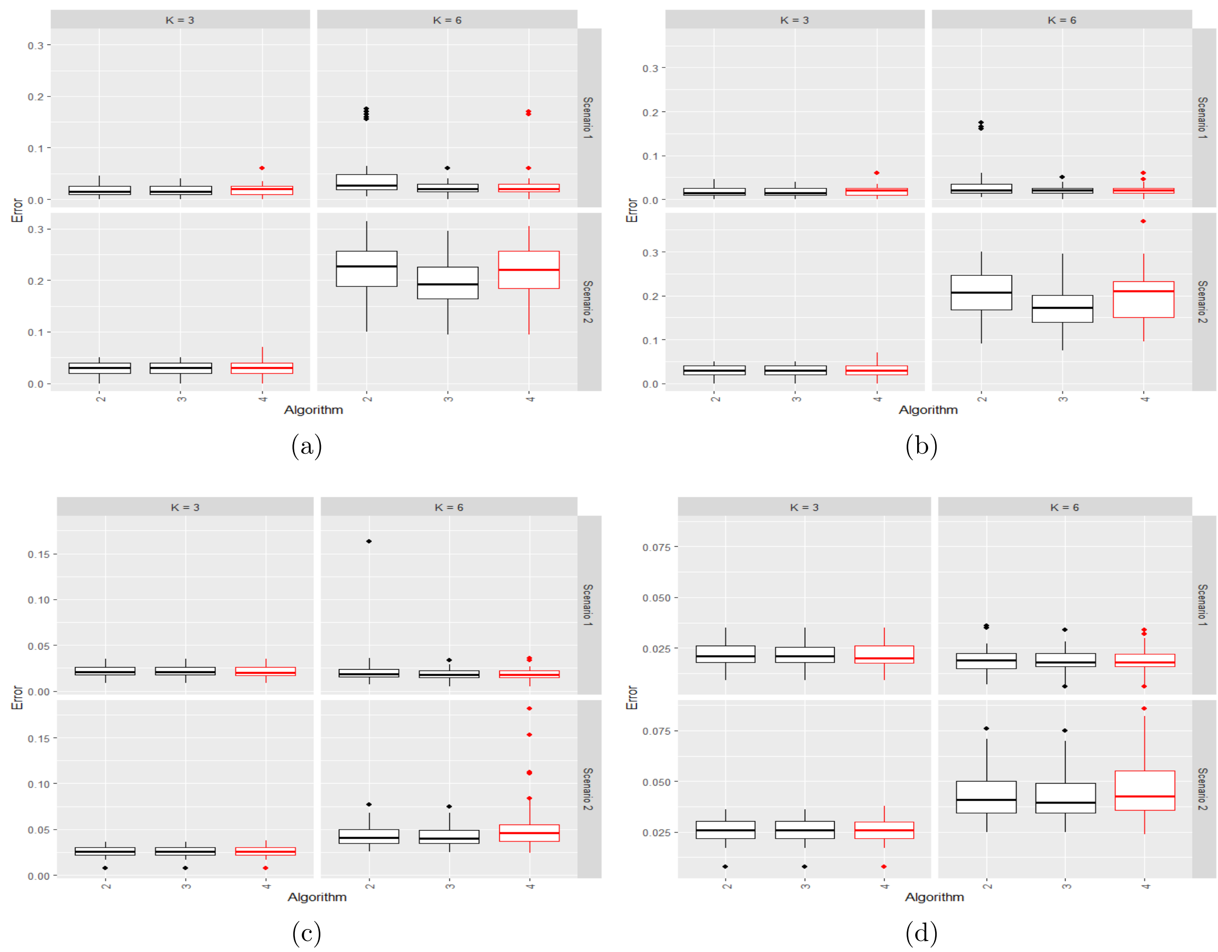


Figura 3.12: Errores en dimensión p=10 cuando n=200 y nstart=400 en (a), n=200 y nstart=1000 en (b), n=1000 y nstart=400 en (c) y n=1000 y nstart=1000 en (d).

Capítulo 4

Inicialización por agregación

4.1. Metodología propuesta

El caso ideal para los algoritmos presentados en la Sección 3 sería recurrir a un gran número `nstart` de inicializaciones aleatorias, aplicar pasos de concentración y quedarse con la “mejor” solución iterada, de modo que esta mejor solución iterada sea el resultado final del algoritmo. Sin embargo, este enfoque implica que toda la información “parcialmente correcta” de la aplicación de los pasos de concentración en el resto de inicializaciones es olvidada por completo.

Lipor et al. (2021) introdujo recientemente un procedimiento de “agregación” de inicializaciones (ensemble initializations) en Análisis Cluster por subespacios que propone combinar la información útil resultante de todas las inicializaciones aleatorias (escasamente) iteradas. Este trabajo explica cómo esa agregación de información es frecuentemente más informativa que el resultado obtenido por la única mejor inicialización e incluye algunos resultados teóricos para respaldar esta afirmación.

Este TFG partía de la idea de que esta filosofía de agregación de inicialización podría ser bastante útil en un Análisis Cluster robusto al, también, aprovechar la información parcialmente correcta que resulta después de los pasos de concentración de todas las `nstart` inicializaciones aleatorias. Uno de los objetivos del TFG era comprobar la validez de esta intuición con programas adecuados.

Para ser más precisos, la metodología propuesta en Análisis Cluster robusto basado en recortes con agregación de inicializaciones de es la siguiente:

1. Consideramos `nstart` inicializaciones aleatorias mediante la función `tclust_c1()` o `rlg_c1()` con `niter1` pasos de concentración (`niter1` no es necesario que sea demasiado grande) para obtener $\{\mathcal{C}_b\}_{b=1}^{\texttt{nstart}}$ particiones de los índices $\{1, 2, ..., n\}$, con
$$\mathcal{C}_b = \{R_0^b, R_1^b, ..., R_G^b\},$$
y donde R_0^b son los índices de las observaciones recortadas en la b-ésima inicialización para $b = 1, ..., \texttt{nstart}$.

2. Obtenemos la matriz de “afinidades” A $(n \times n)$ con términos
$$A_{ii'} \leftarrow \frac{1}{\texttt{nstart}} \#\big\{b : x_i \text{ y } x_{i'} \text{ están co-agrupados (y no recortados) en } \mathcal{C}_b\big\}, \quad (4.1)$$

es decir,

$$A_{ii'} \leftarrow \frac{1}{\texttt{nstart}} \#\big\{b : \{i,i'\} \subset R_k^b \text{ para algún } R_k^b \text{ con } k \neq 0\big\}.$$

3. Calcular

$$A_i = \sum_{i'=1}^{n} A_{ii'} \text{ para } i = 1,...,n, \tag{4.2}$$

y sus valores ordenados en orden creciente en $A_{(1)} \leq A_{(2)} \leq ... \leq A_{(n)}$. Consideraremos como observaciones inicialmente "no recortadas" tras la agregación de las `nstart` inicializaciones a aquellas observaciones x_i con $i \in H$ para

$$H = \{i : A_{(i)} \geq A_{([\alpha n])}\},$$

y las observaciones inicialmente "recortadas" serán las observaciones con índices en $H_0 = \{1,2,...,n\} \setminus H$.

4. Se aplica una técnica de Análisis Cluster jerárquico sobre las observaciones $\{x_i : i \in H\}$. En este TFG sólo se ha usado el "criterio de Ward" como método de Análisis Cluster jerárquico pero otras técnicas pueden ser consideradas análogamente o se pueden usar técnicas de "spectral clustering" (ya que se parte de una matriz de afinidades). Para aplicar este Análisis Cluster jerárquico para las $[n(1-\alpha)]$ observaciones en $\{x_i : i \in H\}$ se usa la disimilaridad

$$\widetilde{d}(x_i, x_{i'}) = 1 - \frac{A_{ii'}}{\text{máx}_{i,i'=1,...,n} A_{ii'}}. \tag{4.3}$$

Este Análisis Cluster jerárquico nos proporcionará una partición de los índices en H en $H = H_1 \cup ... \cup H_K$.

5. Finalmente, se usan las funciones `tclust_c2()` o `rlg_c2()` con un numero de iteraciones `niter2` más elevado (o hasta que se alcance convergencia en las funciones objetivo) considerando como partición inicial de estas funciones a la partición de los índices $\{1,2,...,n\}$ dada por $H_0 \cup H_1 \cup ... \cup H_K$. Las funciones introducidas en el paquete **robClus** permiten este tipo de inicialización como fue explicado en la Sección 3. El resultado de estas funciones `tclust_c2()` o `rlg_c2()` será el resultado final que devuelve este procedimiento de agregación de inicializaciones.

Nótese que al definir los $A_{ii'}$ en la matriz A, con las observaciones que aparecen "coagrupadas" en las `nstart` inicializaciones aleatorias, estamos consiguiendo afinidades $A_{ii'}$ altas para aquellos pares de observaciones x_i y $x_{i'}$ que en más inicializaciones iteradas acaben en un mismo cluster (aunque no siempre acaben en el mismo).

Por otro lado, las observaciones más atípicas acaban menos veces co-agrupadas con otras observaciones, ya que son recortadas en muchas de las inicializaciones (no se coagrupa con ninguna observación cuando una observación es recortada). Eso hace que estas observaciones atípicas estén asociadas con valores pequeños en los $A_{ii'}$ en la matriz de afinidad y, posteriormente, tendrán menores valores consecuentemente en sus A_i en (4.1).

Al obtener los términos $A_{ii'}$ de la matriz A, podemos dar más importancia a los co-agrupaciones en particiones correspondientes a los mejores valores de la función objetivo encontrados en las `nstart` inicializaciones (es decir, pesos mayores para valores grandes en (2.2) o (2.4) y pesos mayores para valores pequeños de (2.5)). Se podría también reemplazar $\widetilde{d}(x_i, x_{i'})$ en (4.3) por $d^{\dagger}(x_i, x_{i'}) = 1 - A_{ii'}/\texttt{nstart}$.

Otra posibilidad a explorar, que no ha sido objeto de estudio en este TFG, sería considerar un nivel de recorte inicial α preventivo (seguramente más alto de lo necesario) y posteriormente aplicar alguna técnica de Análisis Cluster "jerárquico robusto" basado en la matriz A. Algunos métodos robustos en esa dirección son Li et al. (2007) y Balcan et al. (2014). Esta idea puede hacer que solo las observaciones con menor A_i sean las finalmente recortadas. Nótese que la información en un gráfico de $\{(i, A_{(i)})\}_{i=1}^n$ puede ser útil para "redefinir" el tamaño de recorte y sugerirnos utilizar un de recorte final diferente al tamaño de recorte inicial α.

Sólo se ha usado en este TFG el criterio de Ward como técnica para el Análisis Cluster jerárquico pero también se pueden considerar el resultado de varios métodos jerárquicos y inicializar el Paso 5 desde diversas inicializaciones y quedarnos finalmente con aquella solución iterada con el mejor valor encontrado en la función objetivo.

Este método puede presentar problemas cuando n sea grande porque la matriz A tiene n^2 términos y también por la complejidad extra que tiene aplicar Analisís Cluster jerárquico para $[n(1-\alpha)]$ observaciones cuando n es grande. Pensamos que esta dificultad puede superarse considerando submuestreo (subsampling) del conjunto de datos en la búsqueda de inicialiaciones aleatorias. El Paso 5 puede inicializarse con parámetros (pesos, medias, matrices de dispersión o subespacios afines) de las componentes encontradas en la fase de agregación asociada a la submuestra de datos. Esta posibilidad no ha sido explorada en este TFG pero también parece una idea bastante razonable y que podría producir también una notable reducción de tiempos computacionales al poder trabajar reduciendo el n en la fase inicialización agregada.

4.2. Ejemplo ilustrativo

Para ilustrar la metodología se han generado $n = 400$ observaciones en dimensión $p = 10$ agrupadas entorno a 6 clusters y un 10 % de contaminación. Con este fin se ha realizado la ejecución de la función de simulación de datos para el TCLUST que fue presentada en la Sección 3, de la forma `simula.tclust(n=400,type=1,k=6,p=10,balance=1)`. La Figura 4.1 muestra un gráfico por pares de las primeras cuatro variables de ese conjunto de datos (10-dimensional).

Se aplica primero el algoritmo del TCLUST en la Sección 2.1.1 con valores correctos de los parámetros (`k=6`, `alpha=0.1` y `restr.fact=81`) y aplicando 20 pasos de concentración (sin criterio de parada) para `nstart=100` inicializaciones aleatorias. La Figura 4.2(a) representa los valores de la función objetivo (2.2) después de los pasos de concentración para esas `nstart=100` inicializaciones aleatorias. Se ha marcando en color rojo el mayor valor de la función objetivo encontrado en esas inicializaciones, igual a -6743.988. La partición asociada a esa "mejor" inicialización encontrada se representa en la Figura 4.2(b), mostrando solo las dos primeras variables. Podemos ver que esa solución es muy deficiente, con una tasa de mala-clasificación del 20.75 % puesto que se

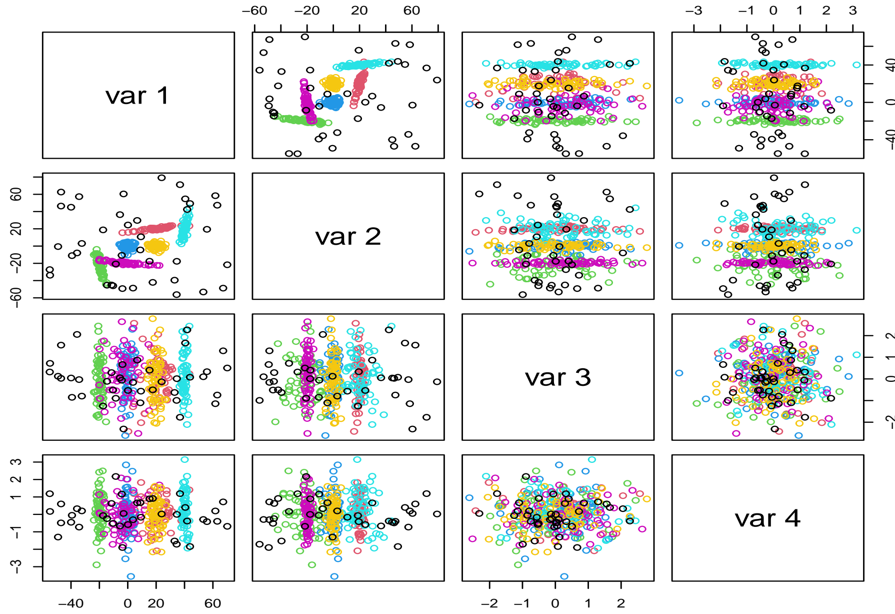


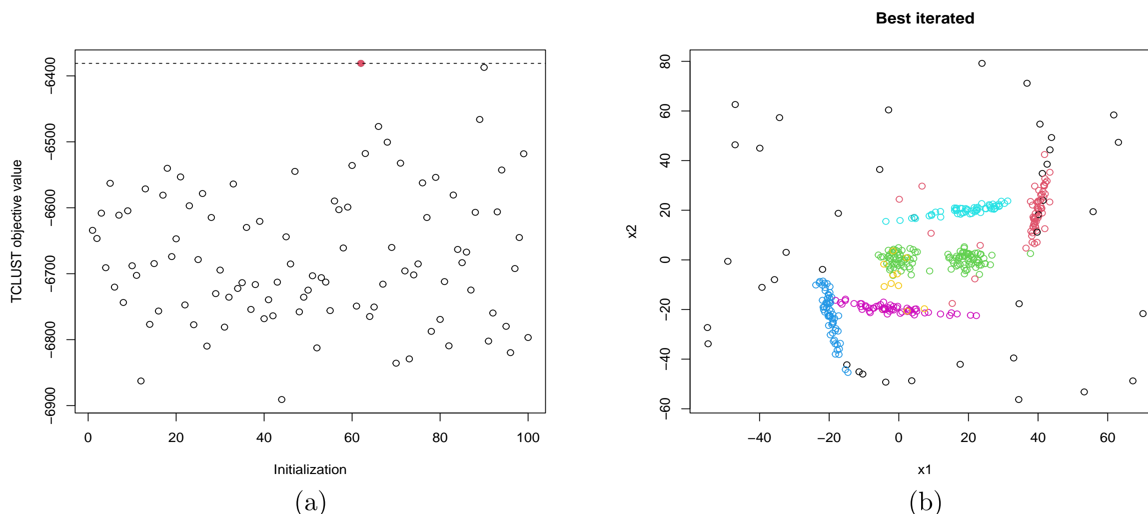
Figura 4.1: Datos para ilustrar el procedimiento de agregación de inicializaciones. Se representan las 4 primeras variables (de 10 variables) usando colores distintos para los 6 clusters y color negro para las observaciones contaminantes.

detectan clusters claros muy mezclados. Quizás `nstart=100` no fue un número suficiente de inicializaciones para la complejidad (p y K) del problema. El tiempo computacional empleado en realizar estas inicializaciones y los pasos de concentración fue de 2.848035 segundos.

La Figura 4.3 muestra el resultado de los 20 pasos de concentración en las 4 primeras inicializaciones aleatorias (de las `nstart=100` realizadas). Como cabía esperar, ninguna de ellas es suficientemente buena por la complejidad del problema pero sí vemos que en todos ellas sí tenemos algún éxito "parcial" al detectar bien alguno de los grupos y gran parte de las observaciones contaminantes. Esta información "parcialmente" correcta es a que queremos explotar con nuestra metodología de agregación.

El uso adecuado de la información parcialmente correcta derivada de las inicializaciones aleatorias se realiza mediante la matriz de afinidad A, que es representada mediante un "heatmap" en la Figura 4.4(a). En este heatmap pueden apreciarse, cerca de la diagonal, una estructura de 6 cajas asociadas a afinidades más altas y que se corresponden, sin duda, a los $K = 6$ clusters principales. La Figura 4.4(b) muestra un gráfico las afinidades sumadas por filas (4.2) ordenadas. Es decir, en ese gráfico se representa $\{(i, A_{(i)})\}_{i=1}^{400}$. Podemos que ver que hay una proporción inicial de observaciones con A_i muy pequeños correspondientes al 10 % de observaciones contaminantes. Esas observaciones, con todas sus afinidades bastante bajas, se pueden visualizar con colores claros en los extremos superior y derecho de dicho heatmap, porque no son demasiado co-agrupadas con otras observaciones.

La Figura 4.5 muestra el resultado final de la inicialización por agregación en este ejemplo para esas `nstart=100` inicializaciones (representando solo las dos primeras



(a) (b)

Figura 4.2: Valores de la función objetivo en las `nstart=100` inicializaciones aleatorias marcando en color rojo el mayor valor encontrado en (a). Partición asociada a esa mejor inicialización en las dos primeras variables.

variables). Podemos ver que el resultado obtenido es muy satisfactorio, con una tasa de error de mala-clasificación bastante mejorada hasta 5.5 % y el valor de la función objetivo ha subido a -6352.729 (mayor que -6743.988 correspondiente a la "mejor" inicialización). El coste del tiempo extra en alcanzar esta solución mejorada (calcular la matriz A y el Análisis Cluster jerárquico) ha sido de 0.844341 segundos extra respecto a los 2.848035 segundos iniciales.

Por otro lado, si hubiéramos ejecutado el algoritmo de TCLUST con `nstart=100` $\times 5$ inicializaciones aleatorias (es decir, `nstart=500`) hubiéramos empleado un tiempo total de 14.07603 segundos (notablemente mayor que 2.848035+0.844341) y obtendríamos una tasa de mala-clasificación de 6.5 % y un valor de la función objetivo peor igual a -6599.572.

El buen comportamiento en este ejemplo ilustrativo veremos que queda confirmado en los estudios de simulación que se mostrarán a continuación, y lo mismo veremos que sucede al aplicar esta misma idea al RLG.

4.3. Simulación con TCLUST con agregación

Para poner a prueba la nueva propuesta de inicialización se ha realizado un estudio de simulación partiendo de los datos generados por la función `simula.tclust()`.

En esta simulación consideraremos 3 métodos que denotaremos por *Métodos 1-3.*

Método 1 El uso del paquete **tclust** en CRAN de partida con `nstart` inicializaciones aleatorias y 20 pasos de concentración.

Método 2 La propuesta de inicialización con agregación con `niter1` = `niter2` = 20 y usando el criterio de Ward como método de Análisis Cluster jerárquico.

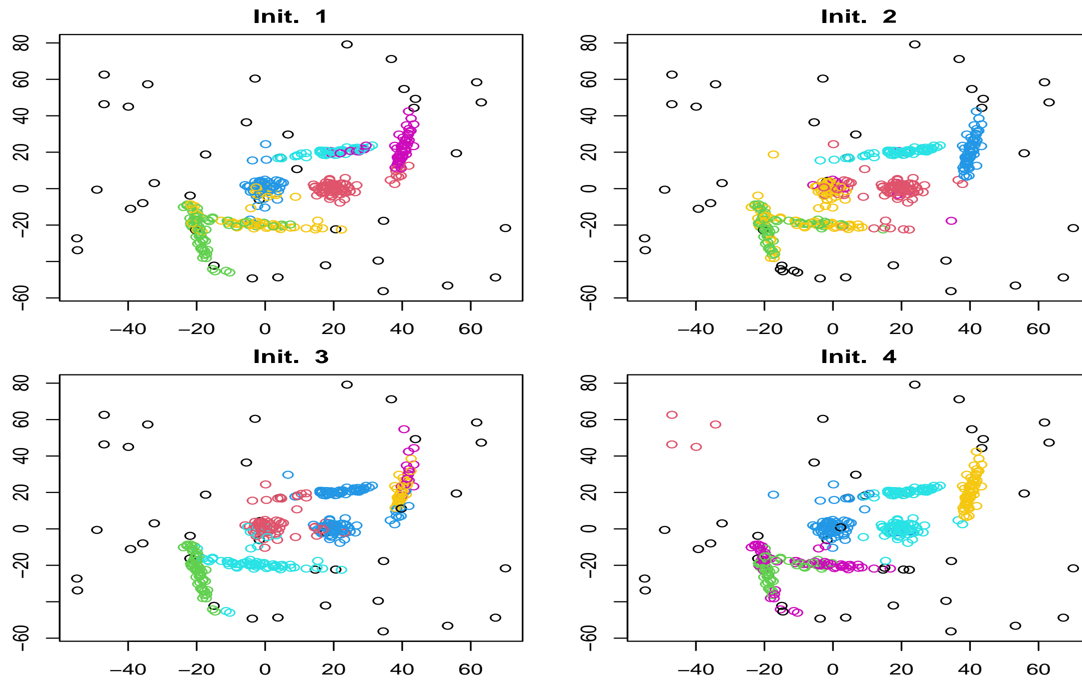


Figura 4.3: Resultado de los 20 pasos de concentración en 4 inicializaciones aleatorias.

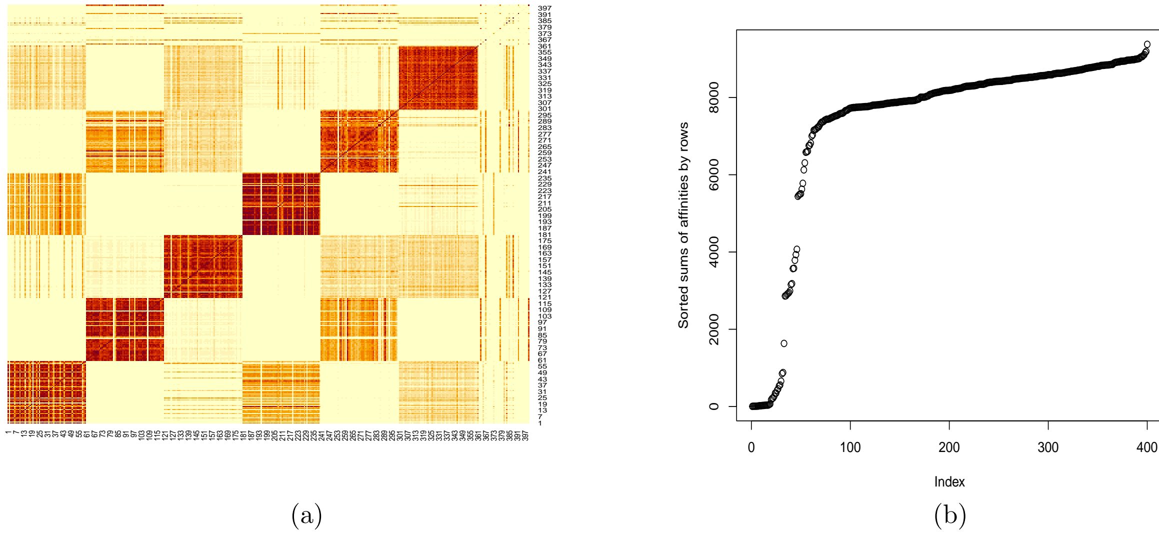
Método 3 El uso del paquete **tclust** de partida pero considerando un número mayor de inicializaciones aleatorias igual a 5×`nstart` inicializaciones aleatorias y 20 pasos de concentración, donde `nstart` es el número de inicializaciones aleatorias usadas en el *Método 1*.

Estos *Métodos 1-3* son aplicados con valores `nstart` iguales a 100 y 400 y parámetros correctos de k, α y c en la maximización de (2.3). Para generar los datos, se ha usado la función `simula.tclust(n,p,k,type,balanced)` con `n=200, 500, 1000`; `type = 1, 2`; `p=4, 10` y `balanced = 1, 2`, generando 20 conjuntos de datos para cada combinación de todos estos parámetros, sobre los que se aplican los *Métodos 1-3*.

Hay demasiadas combinaciones de parámetros como para mostrar los resultados de todas las combinaciones, por lo que vamos a mostrar 3 simulaciones en orden creciente de complejidad del problema correspondiente. Todos los resultados mostrados corresponden al caso de clusters equilibrados en tamaños, es decir, `balanced=1`, pero similares resultados se han obtenido para el caso `balanced=2`. Se pueden ver los resultados completos de todas las simulaciones en `https://github.com/javcres/Robclus-resources`.

Se muestran en la Figura 4.6 los resultados para `n=200`, `nstart=100` y `p=4`, lo que correspondería al problema más sencillo de todos. A pesar de ser el caso más sencillo, ya se puede empezar a ver que el método de inicialización por agregación en el *Método 2* supone una mejora en función objetivo con respecto al *Método 1* con un muy pequeño incremento en el tiempo computacional. El método de agregación parece ser preferible a utilizar un número mayor `nstart` de inicializaciones aleatorias, que también mejora los resultados pero en menor medida y con un mayor coste computacional.

La Figura 4.7 muestra los resultados para `n=500`, `nstart=100` y `p=10`, un problema de complejidad media de entre todos los simulados. Se observa que se obtiene mayor ganancia en términos de reducir el error de mala-clasificación y aumentar la función



(a) (b)

Figura 4.4: "Heatmap" de la matriz de afinidad A basado en el resultado de las `nstart=100` inicializaciones aleatorias en (a). Gráfico de los valores ordenados $A_{(i)}$ para $i = 1, ..., 400$ en (b).

objetivo con el *Método 2* que aumentando el número de inicializaciones aleatorias como se hace en el *Método 3*, además de que el tiempo de computación del *Método 2* es mucho menor.

Por último, se muestra en la Figura 4.8 los resultados para `n=1000`, `nstart=400` y `p=10`, lo que corresponde al problema más complejo de todos. En este problema se acentúan las diferencias entre métodos. Se puede ver claramente como la inicialización por agregación en el *Método 2* mejora considerablemente los resultados, con un incremento muy pequeño del tiempo de ejecución.

4.4. Simulación con RLG con agregación

Vamos a realizar simulaciones análogas a las que se han realizado con TCLUST ahora con el método RLG. Los tres métodos probados son los mismos que en la Sección 4.3 y se usará la misma notación, *Métodos 1-3*, y el *Método 2* corresponde al nuevo tipo de inicialización usando agregación basada en las `nstart` inicializaciones. En la simulación se han considerado valores de `nstart` iguales a 150 y 400. Ahora los datos se simulan con la función `simula.rlg(q,p,n,var,sep.means,alpha)` y los parámetros `n = 150,400`; `p = 200`; `q = 5`; `var =0.01`; `sep.means = 0` y `alpha=0.1`, considerando 20 conjuntos aleatorios por cada combinación de parámetros. Nótese que no se ha considerando un caso precisamente de dimensionalidad baja ya que $p = 200$.

Se mostrarán solo algunos resultados de la simulación a modo de ejemplo en el caso `n=400`, pero los resultados completos de todas las simulaciones pueden verse en `https://github.com/javcres/Robclus-resources`.

Las conclusiones son las mismas que al aplicar TCLUST. Utilizando la inicialización mediante agregación (*Método 2*) conseguimos una notable mejora en resultados con un

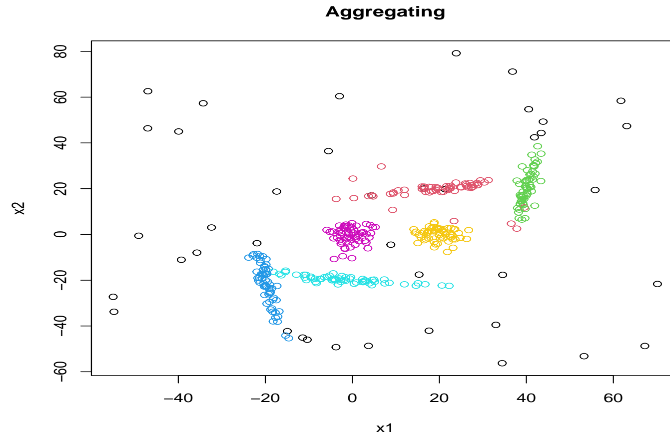
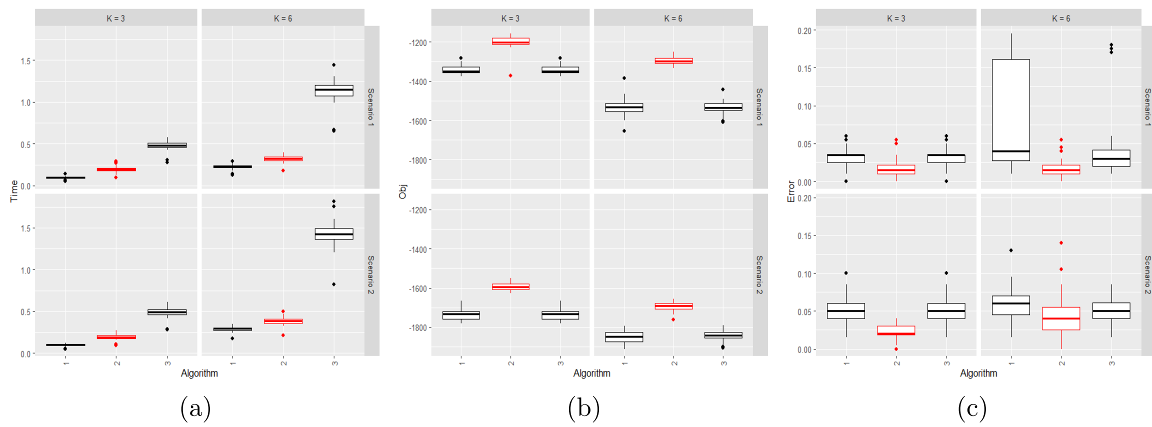


Figura 4.5: Partición resultante de aplicar la inicialización mediante agregación (solo se muestran las dos primeras variables).



(a) (b) (c)

Figura 4.6: Resultados TCLUST para `n=200`, `nstart=100` y `p=4`: (a) tiempo computacional, (b) función objetivo y (c) error de mala-clasificación.

pequeño tiempo extra de ejecución. Esta estrategia parece funcionar mucho mejor que utilizar un número mayor de inicializaciones.

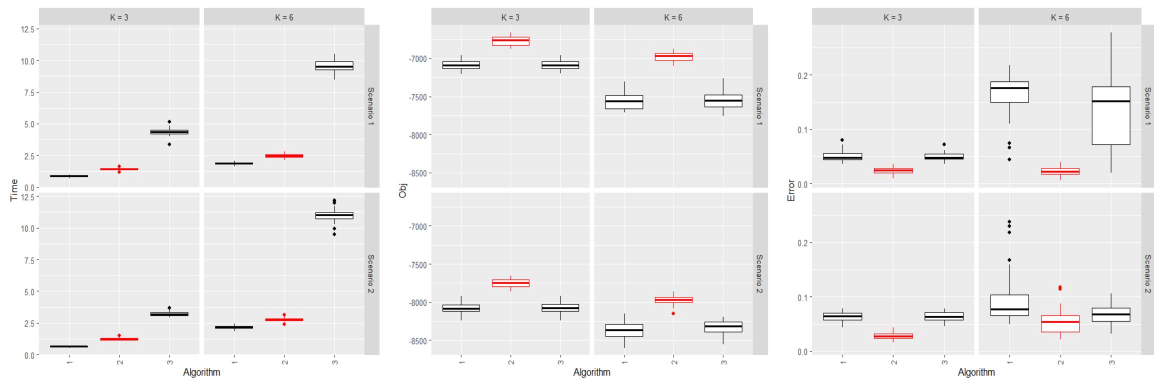


Figura 4.7: Resultados TCLUST para n=500, nstart=100 y p=10.

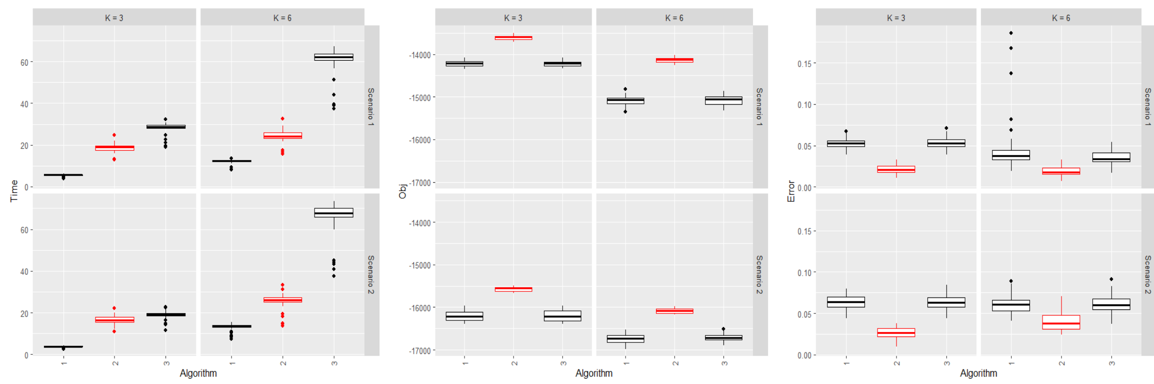


Figura 4.8: Resultados TCLUST para n=1000, nstart=400 y p=10.

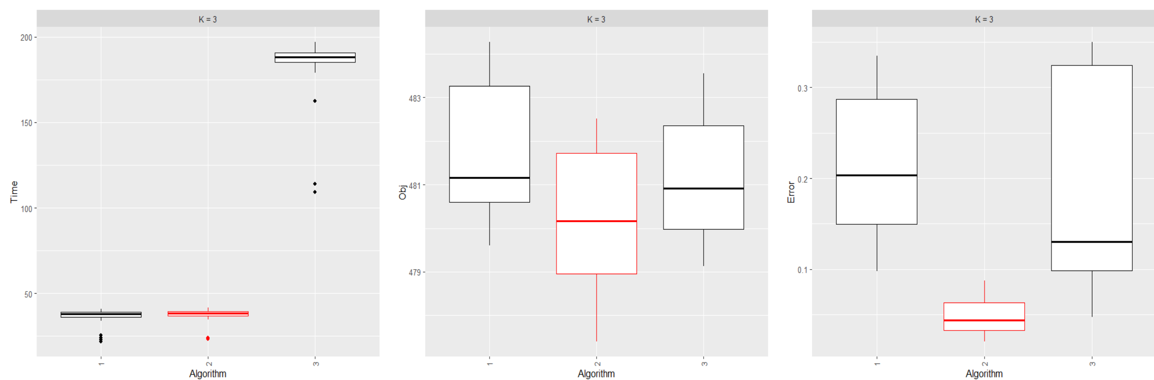


Figura 4.9: Resultados RLG para n=400 y nstart=100.

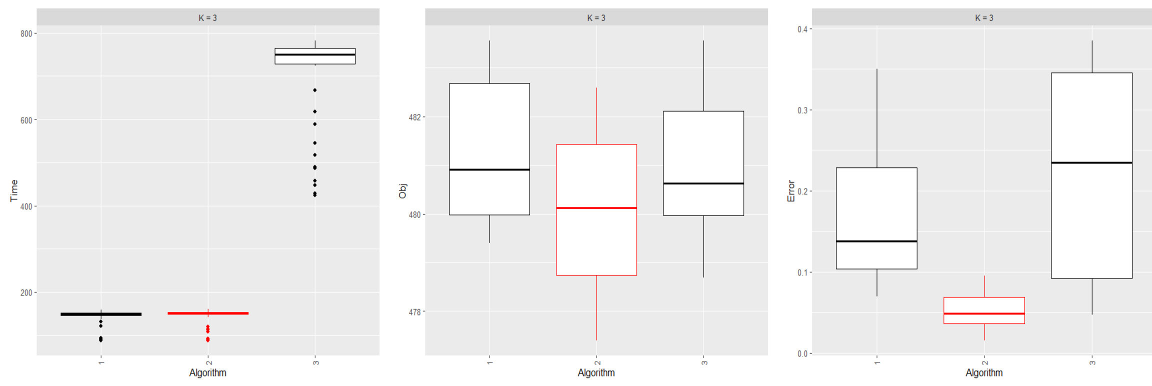


Figura 4.10: Resultados RLG para `n=400` y `nstart=400`.

Capítulo 5

Ejemplo de aplicación a datos reales

A continuación se muestra un ejemplo de aplicación de las implementaciones desarrolladas a un caso real, donde $p = 256$ es relativamente elevado.

5.1. MNIST

El conjunto de datos MNIST es un conjunto de datos ampliamente utilizado en la comunidad de aprendizaje automático. Contiene imágenes de dígitos escritos a mano, con el propósito de ser utilizados en tareas de clasificación de imágenes. Ha sido utilizado para la evaluación y comparación de numerosos modelos de aprendizaje automático supervisado.

En este TFG se va a utilizar una versión reducida del dataset `usps358` (Bouveyron et al., 2019). Se tienen 1756 imágenes a resolución 16×16 píxeles ($p = 256$) de dígitos escritos a mano correspondientes a los dígitos 3, 5 y 8 ($K = 3$), los más complicados de clasificar. En la Figura 5.1 se pueden ver algunos ejemplos de esos dígitos escritos a mano.

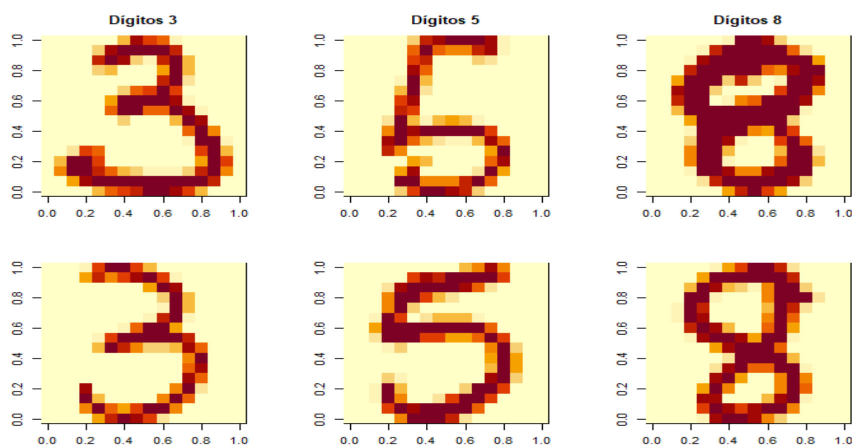


Figura 5.1: Ejemplo de 6 dígitos incluidos en el conjunto de datos *usps358*.

Al usar Análisis Cluster (clasificación no-supervisada) estamos incrementando la dificultad del problema. Más aún, tiene sentido abordarlo como un problema de clasificación

con ruido, puesto que hay números escritos de forma extremadamente ambigua, como puede apreciarse en la Figura 5.2, y que la metodología de recortes podría permitir su identificación y eliminación del conjunto de datos.

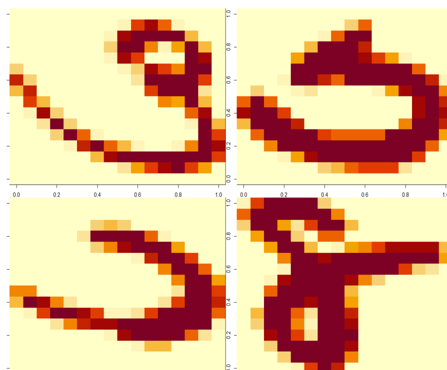


Figura 5.2: Cuatro dígitos "atípicos" en `usps358`, correspondientes a dígitos 3, 5, 5 y 8, respectivamente.

5.2. Aplicación de los algoritmos en robClus

A continuación se comparan los distintos algoritmos disponibles en el paquete **robClus** para aplicar TCLUST y RLG. Al ser éste un problema de alta dimensionalidad, se han comparado los métodos en su versión paralelizada y sin paralelizar. Denotaremos los 5 métodos como *Métodos 1-5*. Los parámetros a usar para `alpha`, `restr.fact` y `d` se han fijado en `alpha = 0.05`, `restr.fact = 4` y `d = c(4,4,4)` (se puede consultar el código usado para realizar esta selección en el archivo `caso_real.Rmd` del repositorio `https://github.com/javcres/Robclus-resources`).

Método 1 La función `tclust()` implementada en la librería **tclust** en CRAN considerando `nstart = 1000` inicializaciones e `iter.max = 20` etapas de concentración.

Método 2 La función `tclust.new()` de **robClus** sin paralelizar, considerando `niter1 = 3`,`niter2 = 20`, `nkeep = 5`, `restr.fact = 4` y un `nstart` de `1000`.

Método 3 La función `tclust.new()` de **robClus** con los mismos argumentos que el *Método 2* pero en su versión paralelizada.

Método 4 La función `rlg()` de **robClus** sin paralelizar, considerando `d = c(4,4,4)` y el resto de parámetros iguales a los del *Método 2*.

Método 5 La función `rlg()` de **robClus** con los mismos argumentos que el *Método 4* pero en su versión paralelizada.

En la Tabla 5.1 se muestran los resultados de la aplicación de esos *Métodos 1-5*. Podemos ver que hay unas diferencias muy grandes de tiempo entre los algoritmos en sus versiones optimizadas y paralelizadas pasando, por ejemplo, a tiempos de escasamente 2 minutos con respecto a tiempos (prácticamente prohibitivos) de casi dos horas en

la implementación actual del paquete **tclust** en CRAN. Aprovechar toda la potencia computacional disponible es crucial ante problemas complejos de alta dimensionalidad.

Podemos ver que mejores valores de la función objetivo no siempre resultan en un menores errores de clasificación, pero esos errores son comparables entre el mismo método, TCLUST o RLG, aplicado. Las tasas de error que arrojan los resultados son razonablemente buenas teniendo en cuenta que estamos tratando un problema de clasificación supervisada mediante métodos de clasificación no supervisados.

En este estudio no se han mostrado los resultados de los métodos basados en agregación de inicializaciones por coincidir prácticamente con los no agregados en este caso (aunque tampoco implicaban un incremento notable en tiempos computacionales). Quizás esta solución agregada sí fuera interesante en problemas de clustering con más de $K = 3$ dígitos, pero esa posibilidad no ha sido explorada en este TFG.

Método	**Tiempo (minutos)**	**Función objetivo**	**Error de clasificación**
Método 1: **tclust** en CRAN	114.19	-195129.33	0.125
Método 2: `tclust.new()` base	22.01	-194973.67	0.147
Método 3: `tclust.new()` paralelizado	1.98	-195004.26	0.126
Método 4: `rlg()` base	25.71	12248.54	0.150
Método 5: `rlg()` paralelizado	2.84	12248.47	0.159

Cuadro 5.1: Resultados de la simulación para datos `usps358`.

5.3. Análisis de los resultados

A continuación se proporcionan más detalles sobre las características de los clusters encontrados por los algoritmos.

5.3.1. Resultados TCLUST

Utilizando los *Métodos 2-3* somos capaces de obtener una clasificación más que razonable al abordar este problema de clasificación no supervisado. El 0 en la fila de la siguiente tabla corresponde a las observaciones y las columnas son los dígitos reales:

```
      Dígitos:
         '3'   '5'   '8'
0        21    41    26
1        46    21   470
2       576    31    39
3        15   463     7
```

La Figura 5.3 muestra algunos ejemplos de datos bien clasificados. En la figura 5.4 se pueden ver algunos dígitos etiquetados como “ruido” (recortados), y se puede observar que se tratan de dígitos que en algunos casos son incluso difícilmente reconocibles para un humano, por lo que los resultados que proporciona este método parecen tener sentido.

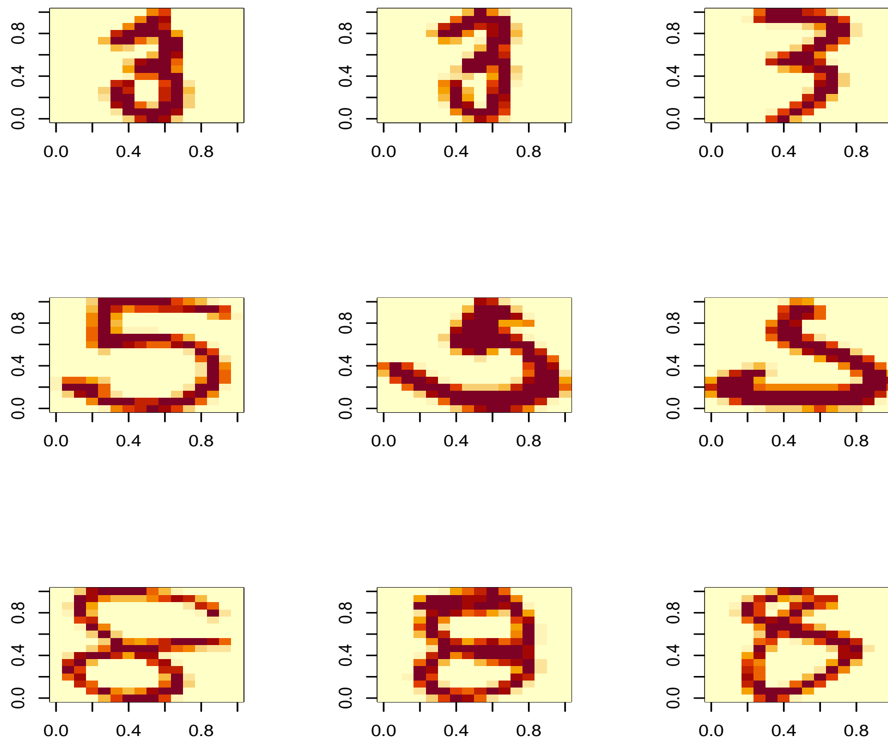


Figura 5.3: Ejemplos de 9 observaciones bien clasificadas con el algoritmo `tclust.new()`.

5.3.2. Resultados RLG

Vemos en el Cuadro 5.1 que este método es capaz de proporcionar soluciones razonablemente buenas al problema. En este caso, nos vamos a centrar en simplemente ilustrar las características de los clusters obtenidos por el RLG, para el grupo de observaciones que “detectan” el dígito 3.

La Figura 5.5 muestra las características en la escritura del dígito 3 que recogen las 4 primeras componentes principales representando los pesos o “loadings”, usando colores rojizos y azulados codificando sus signos y blanco cuando el peso sea próximo a 0. La primera componente parece estar relacionada con dígitos más anchos por la izquierda, la segunda componente parece tener que ver con dígitos en los que el arco superior del 3 baja más y el 3 se ve más simétrico la parte superior e inferior.

El RLG se basa, al igual que el PCA, en buscar “reconstrucciones” de los datos en subespacios de dimensiones menores a la dimensión original de nuestros datos. La diferencia del RLG respecto al PCA clásico es que estas reconstrucciones son ahora dependientes del cluster al que se asigna la observación y eso ayuda a que necesitemos un número menor de componentes q_k para obtener buenos resultados. De esta forma, podemos expresar las observaciones como la suma del dígito “promedio” (centroide del cluster) y la suma de los “scores” del dígito (dependientes del cluster al que se ha asignado el dígito) multiplicando a los “loadings” (del tipo a los mostrados en la Figura 5.5). Por

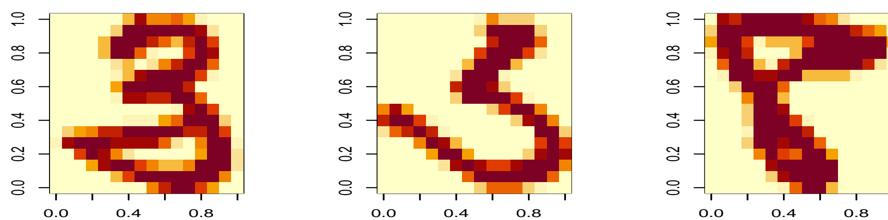


Figura 5.4: Ejemplos de 3 observaciones clasificadas como "ruido" (recortadas) por el método `tclust.new()`, correspondientes a dígitos 3, 5 y 8.

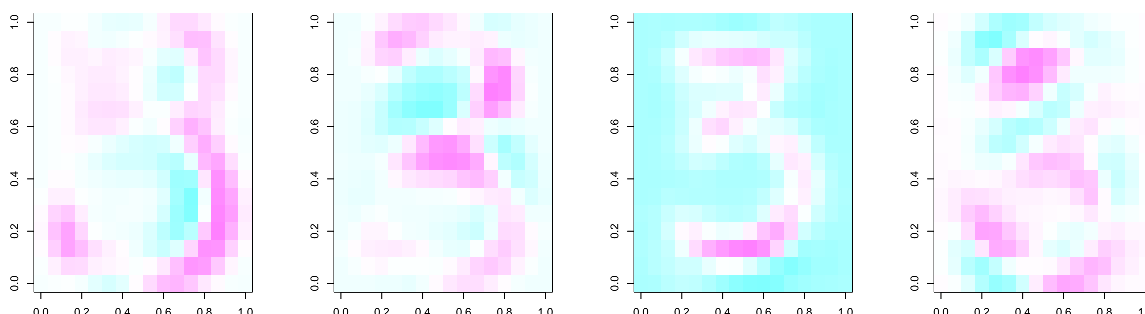


Figura 5.5: Representación de los loadings de las 4 componentes relativas al cluster correspondiente a los dígitos 3.

ejemplo, si nos quedamos con la observación 71 (asignada al cluster correspondiente a los dígitos 3), la Figura 5.6 muestra la reconstrucciones obtenidas al tener en cuenta distintos números de componentes.

Podemos ver que según se van añadiendo componentes se van aproximando mejor los contornos de este dígito al dígito real observado en la Figura 5.6(a).

Procediendo de la misma forma con todas las observaciones asignadas al cluster con los dígitos 3, podemos representar los "scores" en las dos primeras componentes de las observaciones. La Figura 5.7 muestra dichos "scores" y sobre este gráfico se han mostrado las imágenes de dígitos reconstruidos con 2 componentes utilizando valores extremos para los "scores" en las distintas zonas del gráfico de "scores". De esta forma, podemos confirmar que la primera componente recoge la anchura en el trazo del dígito, con valores negativos asociados a dígitos más estrechos y valores positivos con dígitos más anchos. La segunda componente parece estar relacionada con la curvatura superior del trazo, donde valores de "score" positivos en el segundo eje están asociados a trazos con una curvatura del arco superior más amplia, resultando en dígitos más simétricos, y valores negativos con dígitos más asimétricos, con trazo en la curvatura superior más cerrado respecto al arco inferior.

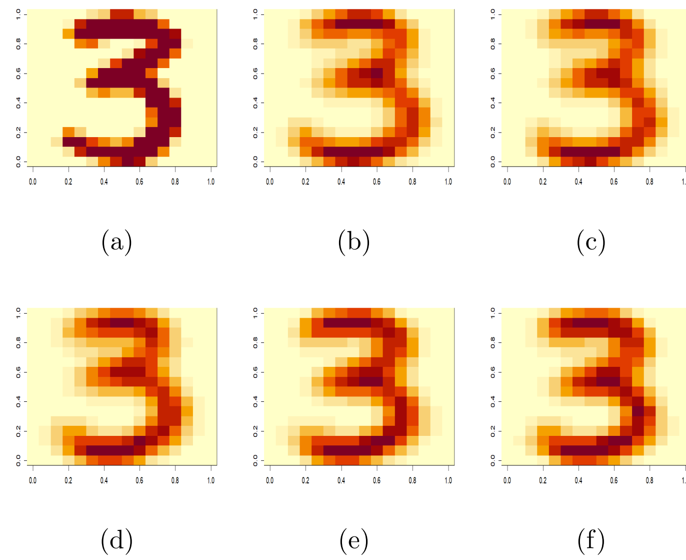


Figura 5.6: Dígito número 71 del conjunto `usps358` en (a) y su reconstrucción usando solo la media (b), la primera componente en (c), las dos primeras en (d), las tres primeras en (e) y las 4 componentes en (f).

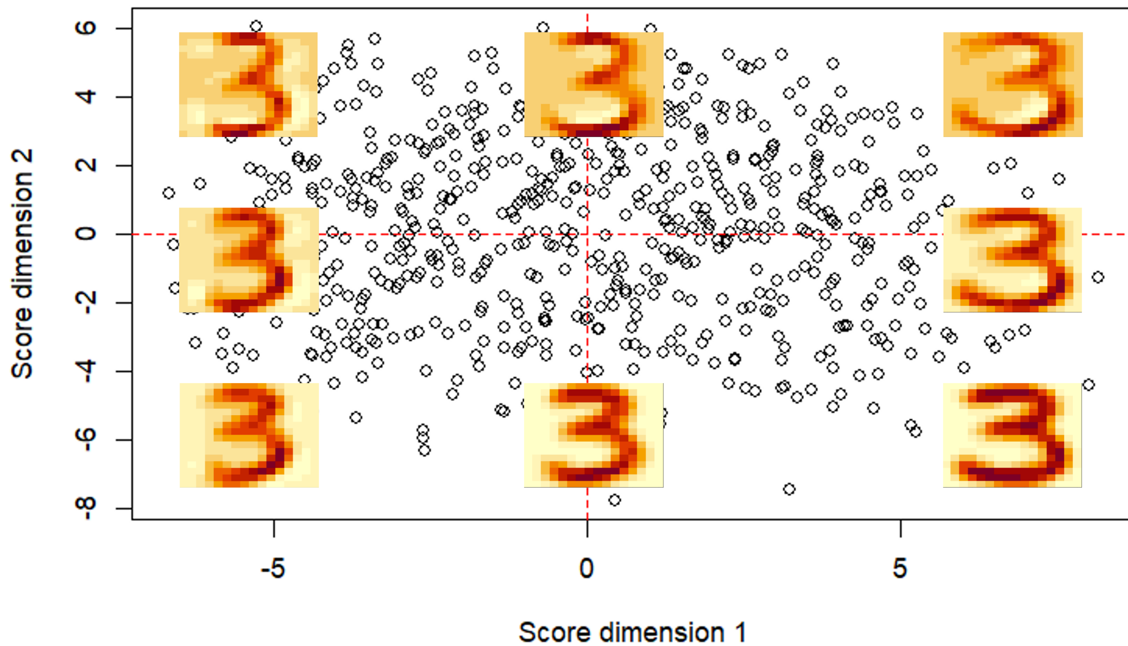


Figura 5.7: Representación los "scores" en las dos primeras componentes de cada dígito asignado al cluster correspondiente al dígito 3. Se han mostrado reconstrucciones usando la media y dos componentes considerando "scores" fijados en -6, 0 y 6.

Capítulo 6

Conclusiones y líneas futuras

En este trabajo se ha realizado una revisión de distintos métodos disponibles de clustering robusto basado en recortes imparciales y se ha desarrollado el paquete **robClus** en R, que desarrolla los algoritmos TCLUST y RLG. De este modo se proporciona una interfaz unificada y uniforme en cuanto a formatos de entrada y salida, que permite abordar de una forma cómoda y eficiente problemas de Análisis Cluster, incluso en situaciones de dimensionalidad moderadamente elevada. El paquete está completamente documentado y ha sido probado a través de simulaciones y un conjunto de datos reales.

La función `tclust.new()` es una implementación del algoritmo TCLUST en R y C++ (para una mayor eficiencia computacional) al que se han incorporado distintas mejoras a nivel algorítmico. Así se considera un nuevo método de inicialización considerando solamente las inicializaciones "más prometedoras", como las únicas inicializaciones a iterar completamente. Esta parte del algoritmo es fácilmente paralelizable, lo que produce también una mejora computacional notable. Se aportan facilidades a la hora de trabajar con este método como el poder seguir en tiempo real el progreso de éste o el poder estandarizar (de forma robusta) directamente los datos.

El método RLG, originalmente programado en Fernández Iglesias (2021), ha sido parcialmente modificado en la función `rlg()` para uniformar los formatos de entrada y salida y se le ha incorporado un sistema de seguimiento del progreso en tiempo real y la posibilidad de paralelizar las inicializaciones.

Dentro del paquete **robClus** también se incorporan dos conjuntos de datos de prueba y dos funciones, `simula.tclust()` y `simula.rlg()`, que permiten crear datos de una forma cómoda que se amolden a las suposiciones de estos algoritmos y nos permitan simular diferentes escenarios de complejidad.

Una vez programado este paquete, se han realizado simulaciones para comparar la mejora de la inicialización considerando solo las inicializaciones más prometedoras respecto a las implementaciones de TCLUST disponibles (una puramente en R y otra en R y C++). El nuevo método de inicialización muestra una mejora muy notable en tiempos de computación, sin sacrificar su eficiencia en términos de la función objetivo. Una de estas implementaciones corresponde al paquete **tclust** en CRAN, que es una implementación muy utilizada y claramente de referencia en la aplicación de Análisis Cluster robusto.

Otra de las contribuciones más importantes de este TFG es la propuesta y la implementación computacional de una metodología nueva de inicialización basada en la agregación de inicializaciones. Este nuevo método se ha mostrado altamente eficaz en

problemas donde no es trivial contar con una inicialización aislada lo suficientemente buena.

Se han aplicado estas implementaciones al caso de la clasificación no-supervisada de dígitos escritos a mano, como ejemplo de datos de dimensión elevada. Estos algoritmos proporcionan una clasificación más que razonable, conjuntamente a una razonable detección de dígitos atípicos. Se aprecia también una ventaja computacional considerable entre las versiones paralelizadas y con inicialización optimizada respecto a las implementaciones previas. En este conjunto de datos se ha ilustrado el interés de la aplicación del RLG en términos de interpretabilidad.

Este TFG abre las puertas a numerosas extensiones y posibilidades computacionales. Primeramente, el resultado de estas funciones pueden organizarse en "clases" en R, lo que permite aplicar funciones en R, ya prácticamente disponibles, para representar gráficamente de forma inmediata el resultado de las funciones `tclust.new()` y `rlg()`. El objetivo final sería poder subir a CRAN el paquete **robClus** en sustitución del actual paquete **tclust**, por los inconvenientes de opacidad en el código comentados en esta memoria.

El contar con implementaciones computacionalmente eficientes abre también la posibilidad a desarrollar técnicas rápidas para la selección automatizada de los parámetros. Para poder elegir parámetros como K, α o c en TCLUST o las dimensiones intrínsecas q_k en RLG, se necesita ser capaz de implementar rápidamente TCLUST y RLG para distintas combinaciones de todos estos parámetros como se hace, por ejemplo, en García-Escudero et al. (2011). Así, no es difícil pensar en implementar funciones como `ctlcurves()` dentro de **robClus**.

Otras lineas de trabajo futuro tienen que ver con aprovechar estas inicializaciones mejoradas para tratar problemas en los que es aconsejable considerar recortes por celdas. Los recortes por celdas (cellwise trimming) son una idea interesante para alcanzar robustez en problemas de dimensión p elevada porque, especialmente, en esos casos no tiene sentido recortar una observación $x_i = (x_{i1}, x_{i2}, ..., x_{ip})' \in \mathbb{R}^p$ entera porque contenga una o muy pocas celdas x_{ij} atípicas. En ese caso, es preferible recortar exclusivamente las celdas x_{ij} atípicas. Esta posibilidad es claramente factible partiendo del trabajo García-Escudero et al. (2021) y las implementaciones en R ya realizadas en Población-Criado (2022). Otra posibilidad a explorar es combinar estas técnicas de recorte con métodos para clustering en altas dimensiones, como el método HDDC en Bouveyron et al. (2007).

Finalmente, una de las líneas más inmediatas de culminar este TFG es concretar el material sobre inicialización por agregación, descrito en el Capítulo 4, en una publicación en alguna revista científica. Este material es novedoso a nivel metodológico y el resultado de las simulaciones mostradas en esta memoria hacen evidente que se trata de una importante mejora en el entorno del clustering robusto. Ese trabajo podría ser el punto de partida a nuevos refinamientos analizando, por ejemplo, cómo poder aprovechar mejor la información en la matriz de afinidad A o cómo tratar conjuntos más grandes de datos (con n mayores) mediante la combinación de "submuestreo" y agregación de inicializaciones.

Bibliografía

Balcan, M., Liang, Y., and Gupta, P. (2014). Robust hierarchical clustering. *The Journal of Machine Learning Research*, 15:3831–3871.

Bouveyron, C., Celeux, G., Murphy, T. B., and Raftery, A. E. (2019). *MBCbook: Companion Package for the Book "Model-Based Clustering and Classification for Data Science"by Bouveyron et al. (2019, ISBN:9781108644181).* R package version 0.1.1.

Bouveyron, C., Girard, S., and Schmid, C. (2007). High-dimensional data clustering. *Computational statistics & data analysis*, 52:502–519.

Cerioli, A., García-Escudero, L., Mayo-Iscar, A., and Riani, M. (2018). Finding the number of normal groups in model-based clustering via constrained likelihoods. *Journal of Computational and Graphical Statistics*, 27:404–416.

Cuesta-Albertos, J., Gordaliza, A., and Matrán, C. (1997). Trimmed k-means: an attempt to robustify quantizers. *Annals of Statistics*, 25:553–576.

Everitt, B. S., Landau, S., Leese, M., and Stahl, D. (2011). *Cluster analysis*. John Wiley & Sons.

Fernández Iglesias, J. (2021). Trabajo fin de grado: Implementación de técnicas para Análisis Cluster robusto en torno a subespacios afines.

Fritz, H., García-Escudero, L., and Mayo-Iscar, A. (2012). tclust: An r package for a trimming approach to cluster analysis. *Journal of Statistical Software*, 47(12).

Fritz, H., García-Escudero, L., and Mayo-Iscar, A. (2013). A fast algorithm for robust constrained clustering. *Computational Statistics & Data Analysis*, 61:124–136.

García-Escudero, L., Gordaliza, A., , and Mayo-Iscar, A. (2014). A review of robust clustering methods. *Advances in Data Analysis and Classiffication*, 8:27–43.

García-Escudero, L. and Gordaliza, A. (1999). Robustness properties of k-means and trimmed k-means. *Journal of the American Statistical Association*, 94:956–969.

García-Escudero, L., Gordaliza, A., and Matrán, C. (2003). Trimming tools in exploratory data analysis. *Journal of Computational and Graphical Statistics*, 12:434–449.

García-Escudero, L., Gordaliza, A., Matrán, C., and Mayo-Iscar, A. (2008). A general trimming approach to robust cluster analysis. *Annals of Statistics*, 36:1324–1345.

García-Escudero, L., Gordaliza, A., Matrán, C., and Mayo-Iscar, A. (2010). A review of robust clustering methods. *Advances in Data Analysis and Classiffication*, 4:89–109.

García-Escudero, L., Gordaliza, A., Matrán, C., and Mayo-Iscar, A. (2011). Exploring the number of groups in robust model-based clustering. *Statistics and Computing*, 21:585–599.

García-Escudero, L., Gordaliza, A., Matrán, C., Mayo-Iscar, A., and Hennig, C. (2016). Robustness and outliers. In C. Hennig, M. Meila, F. and R. Rocci, editors, *Handbook of Cluster Analysis*, pages 653 – 678. Serie Chapman & Hall/CRC Handbooks of Modern Statistical Methods.

García-Escudero, L., Gordaliza, A., San Martín, R., Van Aelst, S., and Zamar, R. (2009). Robust linear clustering. *Journal of the Royal Statistical Series B-Statistical Methodology*, 71:301 – 318.

García-Escudero, L. A., Rivera-García, D., Mayo-Íscar, A., and Ortega, J. (2021). Cluster analysis with cellwise trimming and applications for the robust clustering of curves. *Information Sciences*, 573:100–124.

Gordaliza, A. (1991). Best approximations to random variables based on trimming procedures. *Journal of Approximation Theory*, 64:162–180.

Li, Z., Liu, J., Chen, S., and Tang, X. (2007). Noise robust spectral clustering. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8.

Lipor, J., Hong, D., Tan, Y., and Balzano, L. (2021). Subspace clustering using ensembles of k-subspaces. *Information and Inference: A Journal of the IMA*, 10:73–107.

Lloyd, S. (1982). Least squares quantization in PCM. *IEEE transactions on information theory*, 28:129–137.

Maronna, R. (2005). Principal components and orthogonal regression based on robust scales. *Biometrics*, 47:264–273.

Maronna, R. A., Martin, R. D., Yohai, V. J., and Salibián-Barrera, M. (2019). *Robust statistics: theory and methods (with R)*. John Wiley & Sons.

Neykov, N., Filzmoser, P., Dimova, R., and Neytchev, P. (2007). Robust fitting of mixtures using the trimmed likelihood estimator. *Computational Statistics & Data Analysis*, 4:299–308.

Población-Criado, D. (2022). Trabajo fin de grado: Análisis Cluster con contaminación por celdas.

Riani, M., Perrotta, D., and Torti, F. (2012). FSDA: A MATLAB toolbox for robust analysis and interactive data exploration. *Chemometrics and Intelligent Laboratory Systems*, 116:17–32.

Ritter, G. (2014). *Cluster Analysis and Variable Selection.* CRC Press, Boca Raton.

Rousseeuw, P. and van Driessen, K. (1999). A fast algorithm for the minimum covariance determinant estimator. *Technometrics*, 41:212–223.

Rousseeuw, P. J. and Leroy, A. M. (2005). *Robust regression and outlier detection*. John wiley & sons.

Apéndice A

Código fuente de `robClus`

A continuación se muestra el código fuente de las funciones que componen el paquete `robClus`. Por brevedad, se omiten las cabeceras con la documentación.

A.1. Función `tclust.new`

A.1.1. Función de alto nivel en R

A continuación se detalla el código R de la función `tclust.new()`.

```
#' TCLUST method for robust clustering
#'
#' @description tclust.new searches for k (or less) clusters with different covariance structures in a data
#' matrix x. Relative cluster scatter can be restricted by a constant value restr.fact.
#' For robustifying the estimation, a proportion alpha of observations is trimmed.
#' In particular, the trimmed k-means method is represented by the tclust.new method,
#' by setting parameters restr.fact = 1, opt = "HARD" and equal.weights = TRUE.
#'
#' @param x A matrix or data.frame of dimension n x p, containing the observations (row-wise).
#' @param k The number of clusters initially searched for.
#' @param alpha The proportion of observations to be trimmed.
#' @param nstart The number of random initializations to be performed.
#' @param niter1 The number of concentration steps to be performed for the nstart initializations.
#' @param niter2 The maximum number of concentration steps to be performed for the nkeep solutions kept for further iteration.
#' The concentration steps are stopped, whenever two consecutive steps lead to the same data partition.
#' @param nkeep The number of iterated initializations (after niter1 concentration steps) with the best values in the target
#' function that are kept for further iterations
#' @param equal.weights A logical value, specifying whether equal cluster weights (TRUE) or not (FALSE) shall be considered
#' in the concentration and assignment steps.
#' @param restr.fact The constant restr.fact >= 1 constrains the allowed differences among group scatters in terms of eigenvalues ratio.
#' Larger values imply larger differences of group scatters, a value of 1 specifies the strongest restriction.
#' @param zero_tol The zero tolerance used. By default set to 1e-16.
#' @param scale A robust centering and scaling (using the median and MAD) is done if TRUE.
#' @param parallel A logical value, specifying whether the nstart initializations should be done in parallel.
#' @param n.cores The number of cores to use when paralellizing, only taken into account if parallel=T.
#' @param opt Define the target function to be optimized.
#' A classification likelihood target function is considered if opt=HARD and a mixture classification likelihood if opt=MIXT.
#' @param trace Defines the tracing level, which is set to 0 by default.
#' Tracing level 1 gives additional information on the stage of the iterative process.
#'
#' @return The function returns the following values:
#' \itemize{
#' \item cluster - A numerical vector of size n containing the cluster assignment for each observation.
#' Cluster names are integer numbers from 1 to k, 0 indicates trimmed observations.
#' Note that it could be empty clusters with no observations when equal.weights=FALSE.
#' \item obj - The value of the objective function of the best (returned) solution.
#' \item size - An integer vector of size k, returning the number of observations contained by each cluster.
#' \item weights - Vector of Cluster weights
#' \item centers - A matrix of size p x k containing the centers (column-wise) of each cluster.
#' \item cov -  An array of size p x p x k containing the covariance matrices of each cluster.
#' \item code - A numerical value indicating if the concentration steps have converged for the returned solution (2).
#' \item posterior - A matrix with k columns that contains the posterior probabilities of membership of each observation
#' (row-wise) to the k clusters.
#' This posterior probabilities are 0-1 values in the opt=HARD case.
#' Trimmed observations have 0 membership probabilities to all clusters.
#' \item cluster.ini - A matrix with nstart rows and number of columns equal to the number of observations and where each row shows
#' the final clustering assignments (0 for trimmed observations) obtained after the niter1 iteration
#' of the nstart random initializations.
#' \item obj.ini - A numerical vector of length nstart containing the values of the target function
#'obtained after the niter1 iteration of the nstart random initializations.
```

```r
#' \item x - The input data set.
#' \item k - The input number of clusters.
#' \item alpha -The input trimming level.
#' }
#'
#' @details The procedure allows to deal with robust clustering with an alpha proportion of trimming level and searching for k clusters.
#' We are considering classification trimmed likelihood when using opt=HARD so that HARD clustering assignments are done.
#' On the other hand, mixture trimmed likelihood are applied when using opt=MIXT
#' so providing a kind of clusters posterior probabilities for the observations.
#'
#' This iterative algorithm performs "concentration steps" to improve the current cluster assignments.
#' For approximately obtaining the global optimum, the procedure is random initialized nstart times
#' and niter1 concentration steps are performed for them.
#' The nkeep most promising iterations, i.e. the nkeep iterated solutions with the initial best values for the target function,
#' are then iterated until convergence or until niter2 concentration steps are done.
#'
#' The parameter restr.fact defines the cluster scatter matrices restrictions,
#' which are applied on all clusters during each concentration step.
#' The parameter restr.fact restricts the ratio between the maximum and minimum eigenvalue of all cluster's
#' covariance structures to that parameter.
#' Setting restr.fact to 1, yields the strongest restriction, forcing all clusters to be spherical and equally scattered.
#'
#' Cluster components with similar sizes are favouring when considering equal.weights=TRUE while equal.weights=FALSE admits
#' possible different prior probabilities for the components and it can easily return empty clusters when the number of clusters
#' is greater than apparently needed.
#'
#' @author Javier Crespo Guerrero, Luis Angel Garcia Escudero, Agustin Mayo Iscar.
#'
#' @references
#'
#' Fritz, H.; Garcia-Escudero, L.A.; Mayo-Iscar, A. (2012), "tclust: An R Package for a Trimming Approach to Cluster Analysis".
#' Journal of Statistical Software, 47(12), 1-26.
#' URL http://www.jstatsoft.org/v47/i12/
#'
#' Garcia-Escudero, L.A.; Gordaliza, A.; Matran, C. and Mayo-Iscar, A. (2008),
#' "A General Trimming Approach to Robust Cluster Analysis".
#' Annals of Statistics, Vol.36, 1324-1345.
#'
#' Garcia-Escudero, L. A., Gordaliza, A., & Mayo-Iscar, A. (2014).
#' A constrained robust proposal for mixture modeling avoiding spurious solutions.
#' Advances in Data Analysis and Classification, 27-43.
#'
#' @export
#'
#' @examples
#'
#' #----------------- EXAMPLE 1 -----------------
#' data (M5data)
#' x <- M5data[, 1:2]
#' clus <- tclust.new(x, k = 3, alpha=0.1, restr.fact = 50, trace = T)
#' plot (x, col=clus$cluster+1)
#'
tclust.new <- function(x, k, alpha = 0.05, nstart = 50, niter1 = 3, niter2 = 20, nkeep = 5,
                       equal.weights = FALSE, restr.fact=5, opt="HARD",
                       scale = FALSE, parallel = FALSE, n.cores = -1, zero_tol = 1e-16,
                       trace = 0) {

  # Initial checks
  if( !is.matrix (x))
    x <- as.matrix(x)
  if(any(is.na(x)))
    stop ("x cannot contain NA")
  if (!is.numeric (x))
    stop ("Parameter x: numeric matrix/vector expected")
  if (!k >= 1)
    stop ("Parameter k: must be >= 1")
  if (alpha < 0 || alpha > 1)
    stop ("Parameter alpha: must be in [0,1]")
  if (niter1 < 0 || as.integer(niter1) != niter1)
    stop ("Parameter niter1: must be an integer >= 0")
  if (niter2 < 0 || as.integer(niter2) != niter2)
    stop ("Parameter niter2: must be an integer >= 0")
  if (nkeep < 0 || as.integer(nkeep) != nkeep)
    stop ("Parameter nkeep: must be an integer >= 0")
  if (nkeep > nstart)
    stop ("Parameter nkeep: must be <= nstart")
  if(!is.logical(equal.weights))
    stop ("Parameter equal.weights: must be a logical TRUE or FALSE")
  if(opt != "HARD" && opt != "MIXT")
    stop ("Parameter opt: must be \"HARD\" or \"MIXT\"")
  if(!is.logical(parallel))
    stop ("Parameter parallel: must be a logical TRUE or FALSE")
  if (n.cores < -2 || as.integer(n.cores) != n.cores)
    stop ("Parameter n.cores: must be an integer >= 0, -1 or -2")
  if(zero_tol < 0)
    stop ("Parameter zero_tol: must be >= 0")
  if(trace != 0 & trace !=1)
    stop ("Parameter trace: must be 0 or 1")

  # Scale the data using the median for robustness
  if(scale){
    standard <- function(x){(x-median(x,na.rm=TRUE))/mad(x,na.rm=TRUE)}
    x <- apply(x,2,standard)
  }
```

```
###
# FIRST STEP: get nstart solutions with niter1 concentration steps
###
if(trace){
  cat(paste("\nPhase 1: obtaining ", nstart, " solutions.\n", sep = ""))
  if(parallel) cat(paste("\n Parallelizing initializations using ", n.cores, " cores.
    Progress bar will not display accurate information.\n", sep = ""))
  pb <- txtProgressBar(min = 0, max = nstart, style = 3)
}

if(!parallel){
  cluster.ini <- vector("list", nstart) ## for containing the best values for the parameters after several random starts
  obj.ini <- rep(0, nstart) ## for containing best objective values

  for(j in 1:nstart) {
    assig_obj <- tclust_c1(x, k, alpha, restr.fact, niter1, opt, equal.weights, zero_tol) # niter1 steps!
    cluster.ini[[j]] <- assig_obj$cluster
    obj.ini[j] <- assig_obj$obj

    if(trace){
      setTxtProgressBar(pb, j)
    }
  }
} else {

  # Setup parallel cluster
  if(n.cores == -1){
    n.cores <- detectCores()
  } else if (n.cores == -2){
    n.cores <- detectCores() - 1
  }
  parclus <- makeCluster(n.cores)
  registerDoParallel(parclus)

  count <- 0
  comb <- function(...) { # Custom combination function for the foreach loop
    count <<- count + length(list(...)) - 1
    setTxtProgressBar(pb, count)
    flush.console()
    c(...) # this can feed into .combine option of foreach
  }

  init.results <- foreach(j = 1:nstart,
                          .packages = "robClus",
                          .combine = ifelse(trace, "comb", "c"),
                          .multicombine = T,
                          .inorder = F) %dopar% {
    assig_obj <- tclust_c1(x, k, alpha, restr.fact, niter1, opt, equal.weights, zero_tol) # niter1 steps!
    list(assig_obj$cluster, assig_obj$obj)
  }

  stopCluster(parclus)

  cluster.ini <- init.results[0:(nstart-1) * 2 + 1] # Impair positions of cluster.ini
  obj.ini <- unlist(init.results[1:nstart * 2]) # Pair positions of cluster.ini
}


###
# SECOND STEP: get nkeep best solutions so far
###
if(trace){
  cat(paste("\n\nPhase 2: obtaining ", nkeep, " best solutions out of the intial ", nstart," solutions.\n", sep = ""))
}
best_index <- order(obj.ini, decreasing = T)[1:nkeep]
best_assig_list <- cluster.ini[best_index]

###
# THIRD STEP: apply niter2 concentration steps to nkeep best solutions, return the best solution
###
if(trace){
  cat(paste("\nPhase 3: applying ", niter2, " concentration steps to each of the ", nkeep," best solutions.\n", sep = ""))
  pb2 <- txtProgressBar(min = 0, max = nkeep, style = 3)
}

best_iter <- NULL
best_iter_obj <- -Inf

for(j in 1:nkeep){
  iter <- tclust_c2(x, k, best_assig_list[[j]], alpha, restr.fact, niter2, opt, equal.weights, zero_tol=1e-16)

  if(iter$obj > best_iter_obj){
    best_iter <- iter
    best_iter_obj <- iter$obj
  }

  if(trace){
    setTxtProgressBar(pb2, j)
  }
}
if(trace){
  cat("\n\n")
}
```

```
  best_iter <- c(best_iter, list(
    cluster.ini = matrix(unlist(cluster.ini), byrow = T, nrow = length(cluster.ini)),
    obj.ini = obj.ini,
    x = x,
    k = k,
    alpha = alpha
    )
  )
  return (best_iter)
}
```

A.1.2. Funciones de bajo nivel en C++

A continuación se detalla el código C++ de las funciones `tclust_c1()` y `tclust_c2()`.

```
// [[Rcpp::depends(RcppArmadillo)]]

#include "robClus_types.h"

#include <Rcpp.h>

using namespace Rcpp;

namespace Rcpp {
  template <> SEXP wrap(const iteration& iter) {
    return Rcpp::List::create(
       _["obj"] = iter.obj,
       _["cluster"] = iter.cluster,
       _["size"] = iter.size,
       _["weights"] = iter.weights,
       _["centers"] = iter.centers,
       _["cov"] = iter.cov,
       _["code"] = iter.code,
       _["posterior"] = iter.posterior);
  }

  template <> SEXP wrap(const params& pa) {
    return Rcpp::List::create(
       _["n"] = pa.n,
       _["p"] = pa.p,
       _["alpha"] = pa.alpha,
       _["trimm"] = pa.trimm,
       _["no_trim"] = pa.no_trim,
       _["k"] = pa.k,
       _["equal_weights"] = pa.equal_weights,
       _["zero_tol"] = pa.zero_tol);
       _["restr_fact"] = pa.restr_fact;
       _["opt"] = pa.opt;
  }
}

/**
 * Convert an iteration object to Rcpp List
 *
 * @param iter: iteration object.
 *
 * @returns Rcpp List with same fields as the iteration object.
 */
Rcpp::List iter_to_list(iteration &iter)
{
  return Rcpp::List::create(
       _["obj"] = iter.obj,
       _["cluster"] = iter.cluster,
       _["size"] = iter.size,
       _["weights"] = iter.weights,
       _["centers"] = iter.centers,
       _["cov"] = iter.cov,
       _["code"] = iter.code,
       _["posterior"] = iter.posterior);
}

/**
 * Apply restrictions to eigenvalues.
 *
 * @param autovalues: p x k matrix containin eigenvalues.
 * @param ni_ini: current sample size of the clusters.
 * @param factor_e: the level of the constraints.
 * @param zero_tol: tolerance level.
 *
 * @returns matrix with constrained eigenvalues.
 */
arma::mat restr2Eigenv(arma::mat autovalues, arma::vec ni_ini, double factor_e, double zero_tol)
{

  // Inicializations
  double c = factor_e;
  arma::mat d = autovalues.t();
  int p = autovalues.n_rows;
  int k = autovalues.n_cols;
  int n = arma::accu(ni_ini);
```

```
  arma::mat nis(k, p);
  nis.each_col() = ni_ini;

  // d_ is the ordered set of values in which the restriction objective function change the definition
  // points in d_ correspond to the frontiers for the intervals in which this objective function has the same definition
  // ed is a set with the middle points of these intervals
  arma::vec d_ = arma::sort(arma::vectorise(arma::join_cols(d, d / c)));

  int dim = d_.n_elem;

  arma::vec d_1 = arma::join_cols(d_, arma::vec({d_.back() * 2}));

  arma::vec d_2 = arma::join_cols(arma::vec({0}), d_);

  arma::vec ed = (d_1 + d_2) / 2;

  dim++;

  // the only relevant eigenvalues are those belong to a clusters with sample size greater than 0.
  // eigenvalues corresponding to a clusters whit 0 individuals has no influence in the objective function
  // if all the eigenvalues are 0 during the smart initialization we assign to all the eigenvalues the value 1
  if (arma::max(d.elem(arma::find(nis > 0))) <= zero_tol)
  {
    return arma::mat(p, k, arma::fill::zeros); // solution corresponds to 0 matrix
  }

  // we check if the eigenvalues verify the restrictions
  if (std::abs(arma::max(d.elem(arma::find(nis > 0))) / arma::min(d.elem(arma::find(nis > 0)))) <= c)
  {
    d.elem(arma::find(nis == 0)).fill(arma::mean(d.elem(arma::find(nis > 0))));
    return d.t(); // the solution corresponds to the input because it verifies the constraints
  }

  arma::mat t(k, dim);
  arma::mat s(k, dim);
  arma::mat r(k, dim);

  arma::vec sol(dim);
  arma::vec sal(dim);

  for (int mp_ = 0; mp_ < dim; mp_++)
  {
    for (int i = 0; i < k; i++)
    {
      r(i, mp_) = arma::accu(d.row(i) < ed(mp_)) + arma::accu(d.row(i) > ed(mp_) * c);
      s(i, mp_) = arma::accu(d.row(i) % (d.row(i) < ed(mp_)));
      t(i, mp_) = arma::accu(d.row(i) % (d.row(i) > ed(mp_) * c));
    }

    sol(mp_) = arma::accu(ni_ini / n % (s.col(mp_) + t.col(mp_) / c)) / (arma::accu(ni_ini / n % (r.col(mp_))));

    arma::mat sol_mp_matrix(k, p, arma::fill::value(sol(mp_)));
    arma::mat c_sol_mp_matrix(k, p, arma::fill::value(c * sol(mp_)));

    arma::mat e = sol(mp_) * arma::conv_to<arma::mat>::from(d < sol(mp_)) +
                  d % arma::conv_to<arma::mat>::from((d >= sol(mp_)) % (d <= c * sol(mp_))) +
                  (c * sol(mp_)) * arma::conv_to<arma::mat>::from(d > c * sol(mp_));

    arma::mat o = -1.0 / 2.0 * nis / n % (arma::log(e) + d / e);

    sal(mp_) = arma::accu(o);
  }

  // m is the optimum value for the eigenvalues procedure
  int eo = sal.index_max();
  double m = sol(eo);

  // based on the m value we get the restricted eigenvalues
  return (
             m * arma::conv_to<arma::mat>::from(d < m) +
             d % arma::conv_to<arma::mat>::from(d >= m) % arma::conv_to<arma::mat>::from(d <= c * m) +
             (c * m) * arma::conv_to<arma::mat>::from(d > c * m))
      .t();
}

/**
 * Manages constraints of a clustering stage.
 *
 * @param iter: a reference to the cluster information. Its values are modified.
 * @param pa: a reference to the procedure parameters.
 */
void fRestr(iteration &iter, params &pa)
{
  arma::cube u(pa.p, pa.p, pa.k); // Eigenvectors
  arma::mat d(pa.p, pa.k); // Eigenvalues

  for (int ki = 0; ki < pa.k; ki++)
  {

    arma::vec eigval;
    arma::mat eigvec;

    arma::eig_sym(eigval, eigvec, iter.cov.slice(ki));
```

```
    u.slice(ki) = eigvec;
    d.col(ki) = eigval;
  }

  d.elem(find(d < 0)).fill(0); // all eigenvalue < 0 are assigned to 0, this issue appears for numerical errors
  d = restr2Eigenv(d, iter.size, pa.restr_fact, pa.zero_tol);

  // checking for singularity in all clusters.
  int code = d.max() > pa.zero_tol;
  // iter("code") = code;
  iter.code = code;

  if (code)
  {
    for (int ki = 0; ki < pa.k; ki++)
    { // reconstructing the cov matrices
      iter.cov.slice(ki) =
          u.slice(ki) *
          arma::diagmat(d.col(ki)) *
          u.slice(ki).t();
    }
  }
}

/**
 * Calculates the initial cluster assignment and initial values for the parameters.
 *
 * @param x: matrix with observations and features.
 * @param iter: a reference to the cluster information. Its values are modified.
 * @param pa: a reference to the procedure parameters.
 */
void initClusters(arma::mat x, iteration &iter, params &pa)
{
  arma::mat iter_center = arma::mat(pa.p, pa.k); // Column ki stores the centers of cluster p
  arma::cube iter_sigma = arma::cube(pa.p, pa.p, pa.k); // Covariance matrix of each cluster
  arma::vec size; // Cluster sizes
  arma::vec weights; // Cluster weigths

  arma::ivec idx = arma::randi(pa.k * (pa.p + 1), arma::distr_param(0, pa.n - 1));

  for (int ki = 0; ki < pa.k; ki++)
  {
    // Select the p+1 next elements of idx
    arma::uvec rows_to_select = arma::conv_to<arma::uvec>::from(idx.rows(ki * (pa.p + 1), ki * (pa.p + 1) + pa.p));

    // Set cluster centers to random observation
    iter_center.col(ki) = x.row(rows_to_select(0)).t();

    arma::mat X_ini = x.rows(rows_to_select);

    // Calculate cov matrix of cluster k
    iter_sigma.slice(ki) = pa.p / (float)(pa.p + 1) * arma::cov(X_ini);
  }

  iter.centers = iter_center.t();
  iter.cov = iter_sigma;

  if (pa.equal_weights)
  {
    size = arma::vec(pa.k, arma::fill::value(pa.no_trim / pa.k));
    weights = arma::vec(pa.k, arma::fill::value(1 / (double)pa.k));
  }
  else
  {
    weights = runif(pa.k);
    weights /= arma::accu(weights);
    size = arma::round(pa.n * weights);
  }

  iter.size = size;
  iter.weights = weights;
}

/**
 * FUNCTIONS TO CALCULATE THE DENSITY OF A MULTIVARIATE NORMAL
 *
 * Taken from: Nino Hardt, Dicko Ahmadou, Benjamin Christoffersen.
 * Faster Multivariate Normal densities with RcppArmadillo and OpenMP
 * https://gallery.rcpp.org/articles/dmvnorm_arma/
 */

static double const log2pi = std::log(2.0 * M_PI);

/* C++ version of the dtrmv BLAS function */
void inplace_tri_mat_mult(arma::rowvec &x, arma::mat const &trimat)
{
  arma::uword const n = trimat.n_cols;

  for (unsigned j = n; j-- > 0;)
  {
    double tmp(0.);
    for (unsigned i = 0; i <= j; ++i)
      tmp += trimat.at(i, j) * x[i];
    x[j] = tmp;
```

```
  }
}

arma::vec dmvnrm_arma_fast(arma::mat const &x,
                           arma::rowvec const &mean,
                           arma::mat const &cov,
                           bool const logd = false)
{
  using arma::uword;
  uword const n = x.n_rows,
              xdim = x.n_cols;
  arma::vec out(n);
  arma::mat const rooti = arma::inv(trimatu(arma::chol(cov)));
  double const rootisum = arma::sum(log(rooti.diag())),
               constants = -(double)xdim / 2.0 * log2pi,
               other_terms = rootisum + constants;

  arma::rowvec z;
  for (uword i = 0; i < n; i++)
  {
    z = (x.row(i) - mean);
    inplace_tri_mat_mult(z, rooti);
    out(i) = other_terms - 0.5 * arma::dot(z, z);
  }

  if (logd)
    return out;
  return exp(out);
}

/**
 * Calculate the objective function value.
 *
 * @param x: matrix with observations and features.
 * @param iter: a reference to the cluster information. Its values are modified.
 * @param pa: a reference to the procedure parameters.
 */
void calcObj(arma::mat x, iteration &iter, params &pa)
{
  int n = pa.n;
  int k = pa.k;
  Rcpp::String opt = pa.opt;

  arma::vec ww(n);
  arma::vec w;

  if (opt == "HARD")
  {
    for (int ki = 0; ki < k; ki++)
    {
      w = iter.weights(ki) *
          dmvnrm_arma_fast(x, iter.centers.row(ki), iter.cov.slice(ki)) %
          arma::conv_to<arma::mat>::from(iter.cluster == ki + 1);
      ww = w % (w >= 0) + ww; // calculates each individual contribution for the obj funct hard
    }
  }
  else
  {
    for (int ki = 0; ki < k; ki++)
    {
      w = iter.weights(ki) * dmvnrm_arma_fast(x, iter.centers.row(ki), iter.cov.slice(ki));
      ww = w % (w >= 0) + ww; // calculates each individual contribution for the obj funct mixture
    }
  }
  iter.obj = arma::accu(arma::log(ww.elem(arma::find(iter.cluster > 0))));
}

/**
 * Estimate the model parameters.
 *
 * @param x: matrix with observations and features.
 * @param iter: a reference to the cluster information. Its values are modified.
 * @param pa: a reference to the procedure parameters.
 */
void estimClustPar(arma::mat x, iteration &iter, params &pa)
{
  for (int ki = 0; ki < pa.k; ki++)
  {
    if (iter.size(ki) > pa.zero_tol)
    {
      iter.centers.row(ki) = (iter.posterior.col(ki).t() * x) / iter.size(ki);
      // x centered
      arma::mat X_c = x;
      X_c.each_row() -= iter.centers.row(ki);
      X_c.each_col() %= iter.posterior.col(ki);
      iter.cov.slice(ki) = (X_c.t() * X_c) / iter.size(ki);
    }
    else
    {
      iter.centers.row(ki) = arma::mat(1, pa.p);
      iter.cov.slice(ki) = arma::mat(pa.p, pa.p, arma::fill::eye);
    }
  }
}
```

```
/**
 * Find cluster assignment and trimming.
 *
 * @param x: matrix with observations and features.
 * @param iter: a reference to the cluster information. Its values are modified.
 * @param pa: a reference to the procedure parameters.
 */
void findClustAssig(arma::mat x, iteration &iter, params &pa)
{
  int k = pa.k;
  int n = pa.n;
  int no_trim = pa.no_trim;
  bool equal_weights = pa.equal_weights;
  Rcpp::String opt = pa.opt;

  arma::mat ll(n, k);
  for (int ki = 0; ki < k; ki++)
  {
    ll.col(ki) = iter.weights(ki) * dmvnrm_arma_fast(x, iter.centers.row(ki), iter.cov.slice(ki));
  }

  arma::uvec old_assig = iter.cluster;

  arma::vec pre_z;
  arma::uvec tc_set;

  if (opt == "HARD")
  {
    pre_z = arma::max(ll, 1);
  }
  else
  {
    pre_z = arma::sum(ll, 1);
  }

  // Determine elements to trim
  arma::uvec sorted_index = arma::sort_index(pre_z, "descending");
  arma::uvec last_indexes = arma::linspace<arma::uvec>(no_trim, n - 1, n - no_trim);
  arma::uvec obs_to_trim = sorted_index.elem(last_indexes);

  pre_z.elem(obs_to_trim) = arma::zeros<arma::vec>(n - no_trim);

  tc_set = pre_z > 0;

  // Cluster assignment with trimming
  iter.cluster = (arma::index_max(ll, 1) + 1) % tc_set;

  // Find assignation matrix posterior
  if (opt == "MIXT")
  {
    iter.posterior = ll;
    iter.posterior.each_col() /= (pre_z + (pre_z == 0));
  }
  else
  {

    arma::uvec one_to_n = arma::linspace<arma::uvec>(0, n - 1, n);
    arma::uvec aux_assig = iter.cluster - 1 + (iter.cluster == 0);

    // 2xn matrix containing (observation, cluster) pairs in each column
    arma::umat subscripts = (arma::join_rows(one_to_n, aux_assig)).t();

    iter.posterior = arma::mat(n, k);
    iter.posterior.elem(arma::sub2ind(arma::size(iter.posterior), subscripts)) = arma::ones<arma::vec>(n);
  }

  iter.posterior.each_col() %= arma::conv_to<arma::vec>::from(tc_set); // Set to 0 all trimmed rows

  if (opt == "HARD" && arma::all(old_assig == iter.cluster))
  {
    iter.code = 2;
  }

  // Obtain the clusters size
  iter.size = (arma::sum(iter.posterior, 0)).t();

  if (!equal_weights)
  {
    iter.weights = iter.size / no_trim;
  }
}

/**
 * Apply niter concentration steps to initial solution given by iter and pa.
 *
 * @param niter: number of concentration steps.
 * @param x: matrix with observations and features.
 * @param iter: a reference to the cluster information. Its values are modified.
 * @param pa: a reference to the procedure parameters.
 */
void concentration_steps(int niter, arma::mat x, iteration &iter, params &pa)
{
```

```
  for (int i1 = 0; i1 < niter; i1++)
  {
    fRestr(iter, pa); // restricting the clusters' scatter structure (Changes the iter object)

    if (iter.code == 0)
    {
      if (i1 > 0)
      {
        calcObj(x, iter, pa);
        Rcpp::warning("Data in no general position");
      }
      else
      {
        arma::cube cov = arma::cube(pa.p, pa.p, pa.k);
        cov.each_slice() = arma::eye(pa.p, pa.p);
        iter.cov = cov;
      }
    }

    findClustAssig(x, iter, pa); // estimates the cluster's assigment and TRIMMING (mixture models and HARD )
    if ((int)iter.code == 2 || (i1 == niter - 1))
    {
      break;
    }

    estimClustPar(x, iter, pa); // estimates the cluster's parameters
  }
  calcObj(x, iter, pa); // calculates the objetive function value
}

//' Internal function for concentration steps (initializations) in tclust.new
//' @export
// [[Rcpp::export]]
Rcpp::List tclust_c1(arma::mat x, int k, double alpha = 0.05,
                     double restr_fact = 12, int niter1 = 3, Rcpp::String opt = "HARD",
                     bool equal_weights = false, double zero_tol = 1e-16)
{

  int n = x.n_rows;
  int p = x.n_cols;
  int no_trim = std::floor(n * (1 - alpha));

  params pa;
  pa.n = n;
  pa.p = p;
  pa.alpha = alpha;
  pa.no_trim = no_trim;
  pa.trimm = n - no_trim;
  pa.k = k;
  pa.equal_weights = equal_weights;
  pa.zero_tol = zero_tol;
  pa.restr_fact = restr_fact;
  pa.opt = opt;

  iteration iter;
  iter.obj = 0.0,
  iter.cluster = arma::uvec(n);
  iter.size = arma::vec(k);
  iter.weights = arma::vec(k);
  iter.centers = arma::mat(k, p);
  iter.cov = arma::cube(p, p, k);
  iter.code = 0;
  iter.posterior = arma::mat(n, p);

  initClusters(x, iter, pa); // Cluster random initialization
  concentration_steps(niter1, x, iter, pa); // Apply niter1 concentration steps

  return Rcpp::List::create(
      _["obj"] = iter.obj,
      _["cluster"] = iter.cluster);
}

//' Internal function for concentration steps (refinement) in tclust.new
//' @export
// [[Rcpp::export]]
iteration tclust_c2(arma::mat x, int k, arma::uvec cluster, double alpha = 0.05,
                     double restr_fact = 12, int niter2 = 20, Rcpp::String opt = "HARD",
                     bool equal_weights = false, double zero_tol = 1e-16)
{

  int n = x.n_rows;
  int p = x.n_cols;
  int no_trim = std::floor(n * (1 - alpha));

  // Build the posterior matrix
  arma::uvec one_to_n = arma::linspace<arma::uvec>(0, n - 1, n);
  arma::uvec aux_assig = cluster - 1 + (cluster == 0);

  // 2xn matrix containing (observation, cluster) pairs in each column
  arma::umat subscripts = (arma::join_rows(one_to_n, aux_assig)).t();

  arma::mat posterior(n, k);
  posterior.elem(arma::sub2ind(arma::size(posterior), subscripts)) = arma::ones<arma::vec>(n);
```

```
  arma::vec size = (arma::sum(posterior, 0)).t();

  params pa;
  pa.n = n;
  pa.p = p;
  pa.alpha = alpha;
  pa.no_trim = no_trim;
  pa.trimm = n - no_trim;
  pa.k = k;
  pa.equal_weights = equal_weights;
  pa.zero_tol = zero_tol;
  pa.restr_fact = restr_fact;
  pa.opt = opt;

  iteration iter;
  iter.obj = 0.0,
  iter.cluster = cluster;
  iter.size = size;
  iter.weights = size / no_trim;
  iter.centers = arma::mat(k, p);
  iter.cov = arma::cube(p, p, k);
  iter.code = 0;
  iter.posterior = posterior;

  estimClustPar(x, iter, pa);

  concentration_steps(niter2, x, iter, pa);
  return iter;
}
```

A.2. Función `rlg()`

En este caso ya se partía de una implementación sólida, por lo que sólo se muestra la función de alto nivel en R `rlg()`, que es a la que se ha realizado modificaciones y añadido la documentación correspondiente.

```
#' Robust Linear Grouping
#'
#' @description rlg searches for clusters around affine subspaces of dimensions given by vector d
#' (the length of that vector is the number of clusters). For instance d=c(1,2) means that we are clustering around a line and a plana.
#' For robustifying the estimation, a proportion alpha of observations is trimmed. In particular,
#' the trimmed k-means method is represented by the rlg method, if d=c(0,0,..0) (a vector of length k with zeroes).
#'
#' @param x A matrix or data.frame of dimension n x p, containing the observations (row-wise).
#' @param d A numeric vector of length equal to the number of clusters to be detected.
#' Each component of vector d indicates the intrinsic dimension of the affine subspace
#' where observations on that cluster are going to be clustered.
#' All the elements of vector d should be smaller than the problem dimension minus 1.
#' @param alpha The proportion of observations to be trimmed.
#' @param nstart The number of random initializations to be performed.
#' @param niter1 The number of concentration steps to be performed for the nstart initializations.
#' @param nkeep The number of iterated initializations (after niter1 concentration steps)
#' with the best values in the target function that are kept for further iterations
#' @param niter2 The maximum number of concentration steps to be performed for the nkeep solutions kept for further iteration.
#' The concentration steps are stopped, whenever two consecutive steps lead to the same data partition.
#' @param scale A robust centering and scaling (using the median and MAD) is done if TRUE.
#' @param parallel A logical value, specifying whether the nstart initializations should be done in parallel.
#' @param n.cores The number of cores to use when paralellizing, only taken into account if parallel=T.
#' @param trace Defines the tracing level, which is set to 0 by default.
#' Tracing level 1 gives additional information on the stage of the iterative process.
#' @param zero.tol The zero tolerance used. By default set to 1e-16.
#'
#' @return The function returns the following values:
#' \itemize{
#' \item centers - A matrix of size p x k containing the location vectors (column-wise) of each cluster.
#' \item U - A list with k elements where each element is p x d_j matrix whose d_j columns are unitary and orthogonal vectors
#' generating the affine subspace (after subtracting the corresponding clusters' location parameter in centers).
#' d_j is the intrinsic dimension of the affine subspace approximation in the j-th cluster, i.e., the elements of vector d.
#' \item cluster - A numerical vector of size n containing the cluster assignment for each observation.
#' Cluster names are integer numbers from 1 to k, 0 indicates trimmed observations.
#' \item obj - The value of the objective function of the best (returned) solution.
#' \item cluster.ini - A matrix with nstart rows and number of columns equal to the number of observations and
#' where each row shows the final clustering assignments (0 for trimmed observations) obtained after the niter1
#' iterations of the nstart random initializations.
#' \item obj.ini -A numerical vector of length nstart containing the values of the target function
#' obtained after the niter1 iteration of the nstart random initializations.
#' \item x - The input data set.
#' \item dimensions - The input d vector with the intrinsic dimensions. The number of clusters is the length of that vector.
#' \item alpha - The input trimming level.
#' }
#'
#' @details The procedure allows to deal with robust clustering around affine subspaces with an alpha proportion
#' of trimming level by minimizing
#' the trimmed sums of squared orthogonal residuals.
#' Each component of vector d indicates the intrinsic dimension of the affine subspace
#' where observations on that cluster are going to be clustered.
```

```r
#' Therefore a component equal to 0 on that vector implies clustering around centres, equal to 1 around lines,
#' equal to 2 around planes and so on.
#' The procedure so allows simultaneous clustering and dimensionality reduction.
#'
#' This iterative algorithm performs "concentration steps" to improve the current cluster assignments.
#' For approximately obtaining the global optimum,
#' the procedure is random initialized nstart times and niter1 concentration steps are performed for them.
#' The nkeep most promising iterations, i.e. the nkeep iterated solutions with the initial best values for the target function,
#' are then iterated until convergence or until niter2 concentration steps are done.
#'
#' @author Javier Crespo Guerrero, Jesus Fernandez Iglesias, Luis Angel Garcia Escudero, Agustin Mayo Iscar.
#'
#' @references
#'
#' Garcia Escudero, L. A., Gordaliza, A., San Martin, R., Van Aelst, S., & Zamar, R. (2009).
#' Robust linear clustering. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 71, 301-318.
#'
#' @export
#'
#' @examples
#' data (LG5data)
#' x <- LG5data[, 1:10]
#' clus <- rlg(x, d = c(2,2,2), alpha=0.1, trace = T)
#' plot(x, col=clus$cluster+1)
#'
rlg <-function(x, d, alpha = 0.05, nstart = 50, niter1 = 3, niter2 = 20, nkeep = 5,
    scale = FALSE, parallel = FALSE, n.cores = -1, trace = FALSE){
  x <- as.matrix(x)
  if(is.null(nstart)){nstart <- sum(d)*40}

  if(scale){
    standard <- function(x){(x-median(x,na.rm=TRUE))/mad(x,na.rm=TRUE)}
    x <- apply(x,2,standard)
  }

  clusters<-matrix(nrow = nstart, ncol = dim(x)[1])
  ecms<-rep(0,nstart)

  if(trace){
    cat(paste("\nPhase 1: obtaining ", nstart, " solutions.\n", sep = ""))
    if(parallel) cat(paste("\n Parallelizing initializations using ", n.cores,
        " cores. Progress bar will not display accurate information.\n", sep = ""))
    pb <- txtProgressBar(min = 0, max = nstart, style = 3)
  }

  if(!parallel){
    for(inicializacion in 1:nstart){
      fit<-rlg_c1(x,d,alpha,niter1=niter1)
      clusters[inicializacion,]<-fit$cluster
      ecms[inicializacion]<-fit$obj

      if(trace){
        setTxtProgressBar(pb, inicializacion)
      }
    }
  } else {
    # Setup parallel cluster
    if(n.cores == -1){
      n.cores <- detectCores()
    } else if (n.cores == -2){
      n.cores <- detectCores() - 1
    }
    parclus <- makeCluster(n.cores)
    registerDoParallel(parclus)

    count <- 0
    comb <- function(...) { # Custom combination function for the foreach loop
      count <<- count + length(list(...)) - 1
      setTxtProgressBar(pb, count)
      flush.console()
      c(...) # this can feed into .combine option of foreach
    }

    init.results <- foreach(j = 1:nstart,
                            .packages = "robClus",
                            .combine = ifelse(trace, "comb", "c"),
                            .multicombine = T,
                            .inorder = F) %dopar% {
                              fit <- rlg_c1(x,d,alpha,niter1=niter1)
                              list(fit$cluster, fit$obj)
                            }
    stopCluster(parclus)

    clusters <- init.results[0:(nstart-1) * 2 + 1] # Impair positions of cluster.ini
    clusters <- do.call(rbind, clusters) # Turn list into matrix
    ecms <- unlist(init.results[1:nstart * 2]) # Pair positions of cluster.ini
  }

  if(trace){
    cat(paste("\n\nPhase 2: obtaining ", nkeep, " best solutions out of the intial ", nstart," solutions.\n", sep = ""))
  }
  num<-min(nkeep,length(na.omit(ecms)))
  ecms_ord<-sort(ecms,decreasing = FALSE)
  candidatos<-ecms_ord[1:num]
```

```
  grupos_optimos<-matrix(nrow = 0, ncol = dim(x)[1])
  minecms<-c()

  for(i in 1:num){
    for(j in 1:length(ecms)){
      if(!is.na(ecms[j])){
        if(candidatos[i] == ecms[j]){
          if(sum(unique(clusters[j,]) != 0) == length(d)){
            grupos_optimos<-rbind(grupos_optimos,clusters[j,])
            minecms<-c(minecms,ecms[j])
          }
          break
        }
      }
    }
  }

  n_optimos<-length(minecms)
  resultado_ecm<-c()
  resultado_grupo<-matrix(nrow = 0, ncol = dim(x)[1])

  if(trace){
    cat(paste("\nPhase 3: applying ", niter2, " concentration steps to each of the ", n_optimos," best solutions.\n", sep = ""))
    pb2 <- txtProgressBar(min = 0, max = nkeep, style = 3)
  }

  if(n_optimos == 1){
    fit<-rlg_c2(x,d,grupos_optimos,alpha,niter2=niter2)

    if(trace){
      setTxtProgressBar(pb2, n_optimos)
    }

    grupos <- fit$cluster
    optimo <- fit$obj
    centers <- matrix(NA,nrow=dim(x)[2],ncol=length(d))
    u <- vector("list", length(d))

    for (k in 1:length(d)){
      nk <- dim(x[grupos==k,])[1]
      centers[,k] <- apply(x[grupos==k,],2,mean)
      u[[k]] <- svd(t( x[grupos==k,]-rep(1,nk) %*% t(centers[,k])),nu=d[k])$u
    }

    ret <- list(obj = optimo, cluster = grupos , cluster.ini = clusters, obj.ini = ecms ,
        dimensions = d , centers=centers , U = u, x=x, alpha=alpha)


    return(ret)
  }else{
    for(omega in 1:n_optimos){
      fit<-rlg_c2(x,d,grupos_optimos[omega,],alpha,niter2=niter2)
      resultado_grupo<-rbind(resultado_grupo,fit$cluster)
      resultado_ecm<-c(resultado_ecm,fit$obj)

      if(trace){
        setTxtProgressBar(pb2, omega)
      }
    }
  }
  if(trace){
    cat("\n\n")
  }
  optimo<-min(resultado_ecm)
  flag<-which.min(resultado_ecm)
  grupos<-resultado_grupo[flag,]

  centers <- matrix(NA,nrow=dim(x)[2],ncol=length(d))
  u <- vector("list", length(d))
  for (k in 1:length(d)){
    nk <- dim(x[grupos==k,])[1]
    centers[,k] <- apply(x[grupos==k,],2,mean)
    #u[[k]] <- princomp(x[grupos==k,])$loadings[,1:d[k],drop=F]
    u[[k]] <- svd(t( x[grupos==k,]-rep(1,nk) %*% t(centers[,k])),nu=d[k])$u
  }
  ret <- list(obj = optimo, cluster = grupos , cluster.ini = clusters, obj.ini = ecms ,
    dimensions = d , centers=centers , U = u, x=x, alpha=alpha)
  # class(ret) <- "rlg"

  return(ret)
}
```

A.3. Funciones de simulación de datos

A continuación se muestra el código en R de las funciones `simula.tclust` y `simula.rlg`.

```
simula.rlg <- function(q=2, p=10, n=200, var=0.01,sep.means=0.0,alpha=0.05){

  if (q>=p) stop("Invalid_intrinsic_dimension")

  par_n = max(2, ceiling(n*(1-alpha)/3))
  par_sd = sqrt(var)

  rot1 = qr.Q(qr(matrix(rnorm(p*p),ncol=p)))
  rot2 = qr.Q(qr(matrix(rnorm(p*p),ncol=p)))

  U1 = qr.Q(qr(matrix(rnorm(p*q),ncol=q)))
  U2 = rot1 %*% U1
  U3 = rot2 %*% U2

  Q1 = matrix(runif(q*par_n), ncol = par_n)
  Q2 = matrix(runif(q*par_n), ncol = par_n)
  Q3 = matrix(runif(q*par_n), ncol = par_n)

  X1 = rnorm(p,0,sep.means) %*% t(rep(1,par_n)) + U1 %*% Q1 + matrix(rnorm(p*par_n, sd=par_sd), ncol=par_n)
  X2 = rnorm(p,0,sep.means) %*% t(rep(1,par_n)) + U2 %*% Q2 + matrix(rnorm(p*par_n, sd=par_sd), ncol=par_n)
  X3 = rnorm(p,0,sep.means) %*% t(rep(1,par_n)) + U3 %*% Q3 + matrix(rnorm(p*par_n, sd=par_sd), ncol=par_n)

  X <- t(cbind(X1, X2, X3))

  # contaminating
  n.cont <- n-3*par_n
  rr <- max(1,abs(X))
  X.cont <- matrix(runif(n.cont*p,-rr,rr),ncol=p)

  X <- rbind(X,X.cont)

  # true
  true <- c(rep(1:3, each=par_n),rep(0,n.cont))

  # return
  return(list(x=X,true=true))
}

simula.tclust <- function(n,p=4,k=3,type=2,balanced=1){

  if (k!=3 & k!=6) stop("Invalid_k")
  if (p<2) stop("Invalid_p")

  if(k==6){nn <- n/2}else{nn <- n}

  if (balanced==1){ nn1=nn2=nn3=ceiling(nn*0.3)}else{
                  nn1=ceiling(nn*0.25)
                  nn2=ceiling(nn*0.3)
                  nn3=ceiling(nn*0.35)}

  nh <- nn1+nn2+nn3

  Y <- matrix(NA,ncol=p,nrow=n)

  if (type==1){
    ### type=1
    if(k==3){
        Y[1:nn1,] <- MASS::mvrnorm(nn1,c(4,4),matrix(c(1,0,0,1),nrow=2))
        Y[(nn1+1):(nn1+nn2),] <- MASS::mvrnorm(nn2,c(-4,4),matrix(c(1,0,0,1),nrow=2))
        Y[(nn1+nn2+1):(nn1+nn2+nn3),] <- MASS::mvrnorm(nn3,c(0,0),matrix(c(1,0,0,1),nrow=2))
        Y[(nh+1):n,] <- matrix(runif(p*(n-nh),-20,20),ncol=p)
        if(p>2){Y[1:n,3:p] <- MASS::mvrnorm(n,rep(0,p-2),diag(p-2))}
      }
      else
      {
      Y[1:nn1,] <- MASS::mvrnorm(nn1,c(4,4),matrix(c(1,0,0,1),nrow=2))
      Y[(nn1+1):(nn1+nn2),] <- MASS::mvrnorm(nn2,c(-4,4),matrix(c(1,0,0,1),nrow=2))
      Y[(nn1+nn2+1):(nn1+nn2+nn3),] <- MASS::mvrnorm(nn3,c(0,0),matrix(c(1,0,0,1),nrow=2))
      Y[(nh+1):(nh+nn1),] <- MASS::mvrnorm(nn1,c(24,0),matrix(c(1,0,0,1),nrow=2))
      Y[(nh+nn1+1):(nh+nn1+nn2),] <- MASS::mvrnorm(nn2,c(16,0),matrix(c(1,0,0,1),nrow=2))
      Y[(nh+nn1+nn2+1):(nh+nn1+nn2+nn3),] <- MASS::mvrnorm(nn3,c(20,4),matrix(c(1,0,0,1),nrow=2))
      Y[(2*nh+1):n,] <- matrix(runif((n-2*nh)*p,-20,40),ncol=p)
      if(p>2){Y[1:n,3:p] <- MASS::mvrnorm(n,rep(0,p-2),diag(p-2))}
    }
  }

  if (type==2){
    ### type=2
    alp1 <- runif(1,0,2*pi)
    u1 <- matrix(c(cos(alp1),sin(alp1),-sin(alp1),cos(alp1)),ncol=2)
    alp2 <- runif(1,0,2*pi)
    u2 <- matrix(c(cos(alp2),sin(alp2),-sin(alp2),cos(alp2)),ncol=2)

    if(k==3){
        Y[1:nn1,] <- MASS::mvrnorm(nn1,c(20,20),t(u1) %*% diag(c(1,9^2)) %*% u1)
        Y[(nn1+1):(nn1+nn2),] <- MASS::mvrnorm(nn2,c(-20,-20),t(u1) %*% diag(c(9^2,1)) %*% u1)
        Y[(nn1+nn2+1):(nn1+nn2+nn3),] <- MASS::mvrnorm(nn3,c(0,0),t(u1) %*% diag(c(3^2,3^2)) %*% u1)
        Y[(nn1+nn2+nn3+1):n,] <- matrix(runif(p*(n-nh),-50,50),ncol=p)
        if(p>2){Y[1:n,3:p] <- MASS::mvrnorm(n,rep(0,p-2),diag(p-2))}
      }
      else
      {
        Y[1:nn1,] <- MASS::mvrnorm(nn1,c(20,20),t(u1) %*% diag(c(1,9^2)) %*% u1)
        Y[(nn1+1):(nn1+nn2),] <- MASS::mvrnorm(nn2,c(-20,-20),t(u1) %*% diag(c(9^2,1)) %*% u1)
```

```
        Y[(nn1+nn2+1):(nn1+nn2+nn3),] <- MASS::mvrnorm(nn3,c(0,0),t(u1) %*%diag(c(3^2,3^2)) %*%u1)
        Y[(nh+1):(nh+nn1),] <- MASS::mvrnorm(nn1,c(40,20),t(u2) %*%diag(c(1,9^2)) %*%u2)
        Y[(nh+nn1+1):(nh+nn1+nn2),] <- MASS::mvrnorm(nn2,c(0,-20),t(u2) %*%diag(c(9^2,1)) %*%u2)
        Y[(nh+nn1+nn2+1):(nh+nn1+nn2+nn3),] <- MASS::mvrnorm(nn3,c(20,0),t(u2) %*%diag(c(3^2,3^2)) %*%u2)
        Y[(2*nh+1):n,] <- matrix(runif((n-2*nh)*p,-60,80),ncol=p)
        if(p>2){Y[1:n,3:p] <- MASS::mvrnorm(n,rep(0,p-2),diag(p-2))}
      }
    }

  # true
  true <- rep(0,n)
  if(k==3){true[1:nh] <- rep(1:3,c(nn1,nn2,nn3))}else{true[1:(2*nh)] <- rep(1:6,c(nn1,nn2,nn3,nn1,nn2,nn3))}

  # Return
  return(list(x=Y,true=true))
}
```