Trabajo Fin de Máster: Big Data Clustering

Trabajo Fin de Máster

Daniel Tizón Galisteo

Trabajo de investigación para el Máster en Inteligencia Artificial Avanzada

Dirigido por el

Prof. Dr. D. Luis Manuel Sarro Baro

Julio 2017

"El verdadero genio consiste en la capacidad para evaluar
información incierta y contradictoria"

Winston Churchill

Agradecimientos

A lo largo del último año, he dedicado una gran parte de mi tiempo a sacar adelante este proyecto, lo cual hubiera sido aún más complicado, sino fuera porque trata de algunos de los temas más apasionantes para mí a día de hoy como son el Big Data, la astronomía, y la Inteligencia Artificial. Quiero agradecer en primer lugar al tutor de mi TFM Luis Manuel Sarro, el haberme permitido llevar a cabo este proyecto bajo su tutela, ya que a pesar de tener una agenda muy completa debido a los múltiples proyectos en los cuales participa, accedió a dirigir mi TFM, y cuando lo he necesitado ha sabido sacar algo de tiempo para guiarme en el desarrollo de este proyecto.

No puedo tampoco olvidarme de mi actual pareja, Sandra Castillejo, la cual siempre ha sido muy comprensiva, amable, y generosa, a lo largo de todo este tiempo, poniendo todo de su parte para hacerme las cosas más fáciles. Con Sandra a mi lado, siento que no hay reto imposible de superar, porque ella siempre está ahí para apoyarme, y es capaz de sacarme una sonrisa incluso en los momentos más duros.

Por último, quiero mencionar en los agradecimientos a mis amigos y a mis compañeros de trabajo, que siempre han estado ahí para escucharme, animarme, echarme una mano, o darme feedback sobre mi proyecto. Entre otros, tengo que citar a María Sánchez, Sonia Santos, Ángel Pérez, Guillermo Colmena, Ismael Montealegre, Jesús Diez, José Cabrera, y Juan Mena.

Resumen

En este trabajo he realizado una investigación sobre algoritmos de clusterización que tienen órdenes de complejidad lineales o logarítmicos respecto al tiempo de ejecución, y que pueden ser paralelizables, y por tanto nos permitan trabajar con grandes cantidades de datos. Además, hay que tener en cuenta que puesto que utilizaré un cluster de Spark, los algoritmos que podremos utilizar estarán limitados por aquellos que se encuentran implementados en la librería MLlib de Apache Spark. También he llevado a cabo un estudio de distintos índices de validación interna y externa que podemos emplear para evaluar la calidad de los grupos o clusters creados por dichos algoritmos. Como caso de uso, he utilizado los datos astrométricos procedentes de millones de estrellas de nuestra galaxia proporcionados por la misión Gaia de la Agencia Espacial Europea para realizar una clusterización de dichas estrellas, con el objetivo de tratar de encontrar cúmulos estelares nuevos o recabar más información sobre los ya existentes. Para llevar a cabo el caso de estudio, dada la gran cantidad de datos a tratar, he utilizado la infraestructura facilitada por la DPAC (Data Processing and Analysis Consortium), consistente en un cluster de Apache Spark formado por 6 nodos con 16 cores y 64Gb de RAM cada uno.

Abstract

In this work I have done an investigation about clustering algorithms with linear or logarithmic orders of complexity in execution time, and can work in a distributed way, so we can work with a lot of data. Furthermore, due to that I will use a cluster of Apache Spark, the choice of the algorithms will be limited by the clustering algorithms implemented in the machine learning library of Spark (MLlib). I have also carried out a study of some internal and external validation indexes used to evaluate the quality of the groups or clusters created by these algorithms. As a use case, I have used the astrometric data from millions of stars in our galaxy provided by the Gaia mission of the European Space Agency (ESA) to perform a clustering of these stars, the objective will be to find new star clusters or gather new information about existing ones. In order to carry out the case study, given the large amount of data to be processed, I had to make use of the infrastructure provided by the Data Processing and Analysis Consortium (DPAC), which consisted of 6 nodes with 16 cores and 64Gb of RAM each, which featured the distributed computing framework Apache Spark.

Índice

1. Introducción 8

2. Estado del arte 12

2.1 El mundo del Big Data: Hadoop vs Spark 13
2.1.1. Hadoop Distributed File System (HDFS) 14
2.1.2. Hadoop Map Reduce 16
2.1.3. Apache Spark 19

2.2. Minería de datos con Big Data 22
2.2.1. Integración y recopilación de información 22
2.2.2. Selección, limpieza, y transformación de información 24
2.2.3. Técnicas de Minería de datos 30
2.2.4. Evaluación e interpretación 47

3. Caso de estudio 56

4. Resultados 59

4.1. Escalabilidad de los algoritmos de clusterización 59

4.2. Escalabilidad de los índices de validación 62

4.3 Experimentos 65
4.3.1. Introducción a los experimentos realizados 65
4.3.2. Fase 1: Primeras pruebas 66
4.3.3. Fase 2: mezclas gaussianas, aumento de la precisión de "k", probabilidad de pertenencia al grupo alta, y uso de sistema de scoring 71
4.3.4. Fase 3: error relativo paralaje máximo 20%, nuevo sistema de scoring, más iteraciones, y uso del índice de Rand 75
4.3.5. Fase 4: eliminación de grupos muy dispersos, y re-clusterización individual de los grupos más compactos 86

5. Conclusiones 87

6. Bibliografía 89

Índice de figuras

Figura 1: Mapa Vía Láctea generado con datos de Gaia
Figura 2: Arquitectura HDFS
Figura 3: Modelo de ejecución de MapReduce
Figura 4: Ejecución de una tarea en Spark
Figura 5: Clusters generados con datos en bruto
Figura 6: Clusters generados tras estandarización
Figura 7: Comparación MLlib vs Mahout con algoritmo ALS
Figura 8: Objeto no convexo en verde y objeto convexo en azul
Figura 9: Grupos no convexos
Figura 10: Clusterizacion de grupos no convexos por k-means
Figura 11: Dataset de 3 grupos de distintos tamaños
Figura 12: Agrupación realizada por k-means
Figura 13: Dataset de 3 grupos de diferente densidad
Figura 14: Clusterizacion con k-means
Figura 15: Distribución gaussiana con dos dimensiones
Figura 16: Puntos cercanos a la frontera
Figura 17: Grupos compactos
Figura 18: Grupos con alta conectividad
Figura 19: Gráfica comparativa de tiempos de ejecución de algoritmos de clusterización
Figura 20: Gráfica del índice de Ball para dataset de Iris
Figura 21: Gráfica con tiempos de ejecución para cada índice según incrementamos tamaño dataset
Figura 22: Gráfica con tiempos de ejecución para cada índice según incrementamos el valor de “k”
Figura 23: Representación en 3d de una clusterización de 45 grupos con GMM
Figura 24: Representación en 3d de una clusterización de 8 grupos con GMM
Figura 25: Representación en 3d de una clusterización de 8 grupos con GMM (quitando Grupo5)
Figura 26: Representación en 3d de una clusterización de 8 grupos con GMM (quitando Grupo5 y Grupo4)
Figura 27: Representación en 3d de una clusterización de 6 grupos con GMM del dataset formado por Grupo5
Figura 28: Representación en 3d de una clusterización de 7 grupos con GMM del dataset formado por Grupo2
Figura 29: Representación en 3d de una clusterización de 3 grupos con GMM del dataset formado por Grupo4

Capitulo 1: Introducción

La minería de datos constituye un pilar fundamental de la investigación actual en astronomía. Tal como refleja el artículo *Borne, K. D. (2009)*, la minería de datos se ha utilizado frecuentemente en astronomía para llevar a cabo múltiples tareas, como clasificación, clusterización, o descubrimiento de objetos anómalos.

En este trabajo vamos a tratar de resolver una tarea que sería inabordable para un ser humano, agrupar miles de millones de objetos celestes en función de sus características. En concreto, vamos a utilizar datos procedentes de la sonda Gaia, que obtendrá diferentes medidas sobre unas mil millones de estrellas de nuestra galaxia, la Vía Láctea (lo cual supone aproximadamente un 1% de todas las estrellas que se cree componen nuestra galaxia) en una misión que se prevé dure 5 años, concretamente desde el año 2013 que se lanzo la sonda al espacio, hasta el año 2018, el cual es el año estipulado para poner fin a la misión de la sonda Gaia.

Gaia recogerá medidas astrométricas, fotométricas, y de velocidad radial sobre cada una de estas mil millones de estrellas, y además realizará en promedio unas 70 observaciones de cada una de ellas. Para este trabajo, en un primer momento, utilizaremos sólo los datos relativos a la astrometría para llevar a cabo la tarea de agrupación. Aunque, quizás en posteriores trabajos podría resultar interesante utilizar también otras variables y observar qué diferencias existe entre las agrupaciones que obtenemos usando unas variables y otras.

El artículo *Borne, K. D. (2009)* recoge como la clusterización es una de las tareas de minería de datos más utilizada en astronomía, y en concreto, la clusterización de objetos celestes (estrellas, galaxias, etc.) en base a sus medidas astrométricas. Sin embargo, a día de hoy no existen tantos artículos que traten sobre la clusterización en astronomía con datasets de un tamaño tan enorme como el que vamos a tratar, y que obliga al uso de herramientas Big Data. El objetivo de este trabajo es preparar el terreno para llevar a cabo una clusterización de un dataset que contendrá unos dos mil millones de objetos. Veremos pruebas de rendimiento y escalado con distintos algoritmos de minería de datos y evaluación que escalan bien según aumenta el tamaño del dataset, y las distintos experimentos que llevaremos a cabo será con un dataset de unos 2 millones de objetos (TGAS dataset), y haremos unas pruebas finales para ver cómo se comportan dichos algoritmos con un dataset de dos mil millones de elementos (GAIA dataset).

Para realizar esta tarea vamos a necesitar utilizar una librería de Machine Learning que tenga implementados algoritmos de clusterización escalables que permitan trabajar con Big Data. Lo que necesitamos básicamente es tener algoritmos de clusterización distribuidos cuyo orden de complejidad tendrá que ser del orden de O(n) o inferiores, y sobre todo que no tengan ordenes de complejidad tan elevados como $O(n^2)$, puesto que si inicialmente empezaremos trabajando con "sólo" 2 millones de elementos, el objetivo final es clusterizar cerca de 2.000 millones de elementos, y con ese objetivo en mente llevaré a cabo este trabajo.

En la figura 1 podemos ver un mapa de nuestra galaxia, la Vía Láctea, generado gracias a los datos aportados por la sonda Gaia hasta el momento (2016), es decir, que se trata del mapa más completo de nuestra galaxia, basado en datos reales, que se ha realizado hasta la fecha, y se espera que la calidad de dicho mapa pueda verse incrementada en los próximos años, según se vayan liberando más datos procedentes de la sonda espacial.

La vía láctea es el nombre que recibe la galaxia en la cual se encuentra el sistema solar, y por tanto, el planeta en el que habitamos, el planeta Tierra.

Nuestra galaxia, la vía láctea, tiene forma de espiral, y se calcula que contiene entre 100.000 y 400.000 millones de estrellas.

El mapa de la Vía Láctea que vemos a la derecha se ha realizado usando información procedente de Gaia.

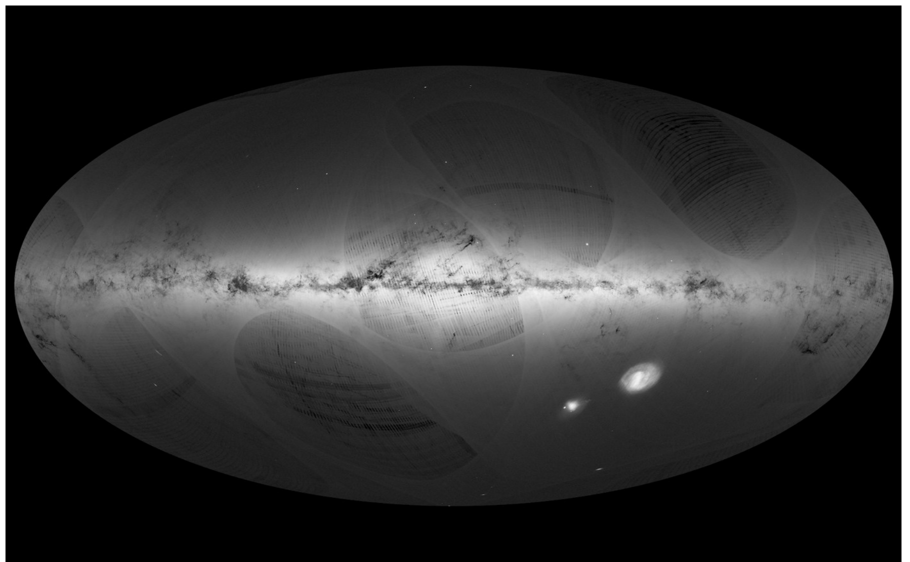


Figura 1: Mapa Vía Láctea generado con datos de Gaia

Una tarea de “clustering” es una tarea de aprendizaje no supervisado, en la cual sólo disponemos de un conjunto de variables sobre cada elemento de nuestro dataset, y debemos agrupar dichos elementos utilizando única y exclusivamente la información que nos proporcionan dichas variables. Uno de los problemas más difíciles de resolver en este tipo de tareas es decidir cuál es el número de grupos en el que se divide nuestro dataset. Para ayudarnos en dicha tarea emplearé distintos índices de evaluación interna y externa que nos ayudarán a decidir cuál es la mejor agrupación para nuestro dataset. Vamos a intentar conseguir que los individuos de un mismo grupo resulten muy similares entre sí, y a su vez que se diferencien lo máximo posible de los individuos pertenecientes a otros grupos. También intentaremos que los grupos que estamos obteniendo sean compactos, y se encuentren dispersos.

Es importante definir formalmente a qué nos referimos cuando decimos que dos instancias o ejemplos son “similares”. Para ello, vamos a usar el concepto matemático de “distancia”, y tendremos que tener en cuenta que existen varias métricas de distancia, y la elección de una u otra va a tener una influencia considerable en la formación de los distintos grupos. Por ejemplo, tenemos la distancia euclídea (distancia clásica que viene dada por la longitud de la recta que une dos puntos en el espacio euclídeo), la distancia de Manhattan (distancia por cuadras o city block), la distancia de Chebychev (es simplemente la discrepancia más grande en alguna de las dimensiones), o la distancia de Mahalanobis (esta distancia no asume que los atributos sean independientes).

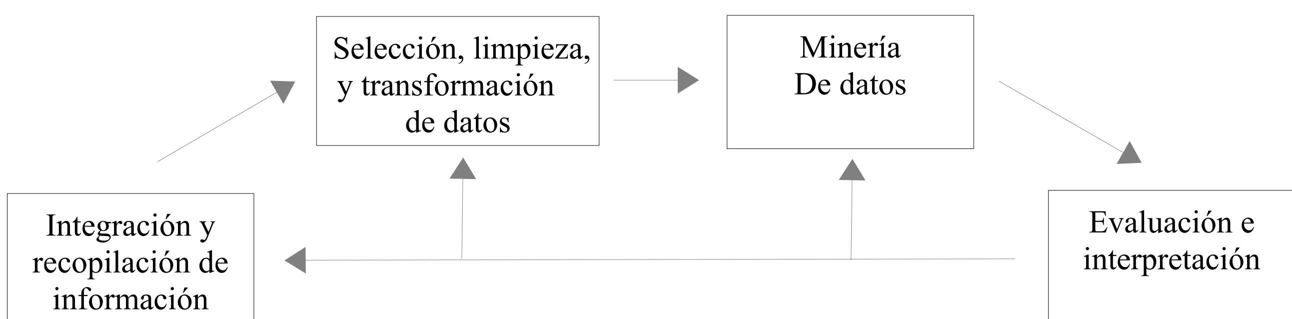
Tal como aparece en *Hernandez Orallo J., Ramirez Quintana M., Ferri Ramirez C. (2005)*, en una tarea de minería de datos se suelen llevar a cabo los siguientes pasos para conseguir extraer conocimiento a partir de los datos:

1. Integración y recopilación de información: El primer paso es obtener los datos que vamos a utilizar, los cuales pueden provenir de distintas fuentes, y por tanto habrá que integrarlos en un almacén de datos que permita su posterior tratamiento.

2. Selección, limpieza, y transformación de datos: Una vez que hemos obtenido todos los datos necesarios, el siguiente paso es seleccionar las variables relevantes, eliminar las variables irrelevantes o redundantes, eliminar ejemplos con información errónea o incompleta, generar nuevas variables mediante una transformación de las variables originales, decidir si daremos algún tratamiento especializado a los outliers o valores atípicos (valores demasiado altos o demasiado bajos respecto al resto), etc.

3. Minería de datos: En esta etapa debemos aplicar un algoritmo adecuado en función de la tarea que queramos llevar a cabo, teniendo en cuenta si es una tarea de clasificación, regresión, agrupamiento (clustering), búsqueda de correlaciones, reglas de asociación, etc. Además de elegir el algoritmo más conveniente para la tarea que queremos llevar a cabo, también tendremos que ocuparnos de los hiperparámetros del algoritmo elegido.

4. Evaluación e interpretación: Por último, una vez que hemos generado un modelo, debemos evaluar su comportamiento, y en caso de no ser bueno, habría que volver a las fases anteriores para ver si podemos mejorar su rendimiento. También puede ser útil interpretar nuestro modelo para sacar conclusiones sobre por qué funciona como funciona. No obstante, hay algoritmos que generan modelos difícilmente interpretables, como por ejemplo una red neuronal artificial que termina generando una matriz de pesos. Mientras que otros modelos, como por ejemplo un árbol de decisión, se pueden comprender e interpretar mejor.



Este proceso de 4 pasos es un proceso iterativo, y tras llevar al paso 4 y evaluar nuestro modelo, si no obtenemos un resultado satisfactorio, puede ser necesario volver de nuevo a alguno de los pasos anteriores: intentar recopilar más información, tratar de eliminar alguna variable irrelevante que este generando mucho ruido, optimizar los parámetros de nuestro algoritmo, o incluso elegir otro algoritmo diferente.

En el caso que nos ocupa, una tarea de agrupamiento o clusterización, el paso 4, la “evaluación e interpretación” es uno de los puntos más complicados de realizar, ya que ¿cómo podemos evaluar los resultados obtenidos si realmente no sabemos cual es la respuesta correcta? Para tratar de resolver este problema, podemos usar índices de validación interna (por ejemplo, índice de Ball y Hall, Silhouette, índice de Hartigan, índice de Ratkowsky, etc.), o índices de validación externa (si sabemos que ciertos ejemplos deben ir en el mismo grupo o en grupos separados).

Los índices de validación interna siempre están a nuestra disposición, puesto que para evaluar la calidad de los grupos generados se basan sólo en los propios datos usados para generar esos grupos, como la consistencia interna de los grupos, y la separación de los miembros de un grupo respecto a los de otros grupos. Por otro lado, los índices de validación externa no siempre pueden usarse, ya que para poder usarlos necesitamos tener información sobre elementos que sabemos que deben ir en el mismo grupo, o sobre elementos que sabemos que no deben ir en el mismo grupo, y esta información no siempre está disponible.

Otro problema respecto a la evaluación de grupos, es que existen múltiples implementaciones de índices de evaluación interna y externa en varios lenguajes de programación (R, Java, Python, etc.). Sin embargo, no existen muchas versiones escalables y paralelizables de dichos índices, y en Spark no tenemos ningún índice implementado, por lo que una de nuestras principales tareas será implementar dichos índices en Spark para ser capaces de utilizarlos para evaluar la calidad de nuestros grupos, y de esta forma poder elegir qué modelo y qué número de grupos representan mejor a nuestro dataset.

Capitulo 2: Estado del arte

Esta sección va a quedar dividida en cinco puntos, un primer punto dedicado al estado del arte en el mundo del Big Data y a llevar a cabo una toma de contacto y una pequeña comparativa sobre los dos frameworks más potentes que existen actualmente para ello (Apache Hadoop y Apache Spark). Y los cuatro puntos siguientes estarán dedicados a las cuatro fases en que se divide una tarea de minería de datos, pero centrándonos en cómo realizar dichas tareas hoy en día desde el mundo del Big Data:

1. Integración y recopilación de información
2. Selección, limpieza, y transformación de datos
3. Minería de datos
4. Evaluación e interpretación

Dado que la minería de datos con Big Data es campo muy extenso y que actualmente se encuentra en constante ebullición, sería imposible abarcar el estado del arte en detalle para cada uno de estos cuatro puntos. Por tanto, en este trabajo voy a hacer un breve repaso a los dos primeros puntos, y cómo podríamos abarcarlos teniendo en cuenta siempre el punto de vista de que queremos trabajar con Big Data, y voy centrarme principalmente en cómo resolver los dos últimos puntos.

En concreto, voy a ver cómo nos podemos enfrentar desde el estado del arte a los dos últimos puntos cuando tenemos que llevar a cabo una tarea de clusterización para un dataset con millones de registros. Haré una breve mención a los algoritmos de clustering espectral, ya que forman parte del estado del arte en clustering, sobre todo para datasets que contienen grupos no compactos sino más bien guiados por su conectividad. Aunque, los algoritmos de clustering espectral no escalan demasiado bien, ya que tienen un orden de complejidad de $O(n^3)$, y por lo tanto, es difícil justificar su uso cuando tenemos en mente tratar un dataset de miles de millones de registros, puede resultar interesante mencionarlos, ya que para datasets de cientos de miles de objetos funcionan sin problemas, y por otra parte, en la librería MLlib de Spark, se encuentra implementado un algoritmo de clustering espectral, concretamente, el algoritmo Power Iteration Clustering (PIC), por lo que no sería demasiado difícil hacer uso de dicho algoritmo, si la situación lo requiere. Por ejemplo, podríamos utilizar un algoritmo de clustering espectral en nuestro caso de estudio para llevar a cabo subclusterizaciones sobre partes más pequeñas del dataset que pertenezcan a una región espacial específica.

Por tanto, voy a centrar este apartado en ver cuál es el estado del arte de los algoritmos de clusterización escalables capaces de trabajar con Big Data, y también cual es el estado del arte de las técnicas de evaluación para medir la calidad de los grupos generados por un algoritmo de clusterización, teniendo también aquí presente, que dichas medidas de evaluación sean escalables y puedan ser aplicables para datasets de miles de millones de registros.

2.1. El mundo del Big Data: Hadoop vs Spark

Hasta hace poco tiempo, el framework de referencia para trabajar con Big Data era Hadoop y todo el ecosistema montado en torno a él. Básicamente, lo que Hadoop ofrece es un sistema de archivos distribuido (Hadoop Distributed File System), y un motor de ejecución conocido como MapReduce que permite manipular grandes cantidades de datos. El origen de Hadoop está en unos artículos publicados por Google a principios del siglo XXI que sentarían las bases sobre el futuro del Big Data: "*Ghemawat, S., Gobioff, H., Leung, S. (2003). The Google File System",* y "*Dean, J., Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters"*.

Hadoop se presenta como un sistema de archivos distribuidos y un framework de procesamiento distribuido, los cuales proveen un modelo sencillo de programación distribuida, usando para ello una arquitectura maestro/esclavo, que es escalable, tolerante a fallos, y es capaz de correr sobre hardware no especializado. Hadoop está desarrollado en Java, por lo que cualquier máquina capaz de correr Java, puede formar parte de un cluster de Hadoop.

Hoy en día se continua usando Hadoop, o al menos en parte, ya que si bien su sistema de archivos distribuido (HDFS) es de uso habitual en la mayoría de proyectos relacionados con Big Data, lo cierto es que el sistema de ejecución de Hadoop basado en Map Reduce ha sido en gran parte reemplazado por Apache Spark. Más adelante veremos las características del motor de ejecución de Spark, y las ventajas que ofrece frente al motor de ejecución original de Hadoop, pero ahora centrémonos en el sistema de archivos distribuido de Hadoop (HDFS) y su arquitectura.

Debido a que Hadoop es una tecnología en constante desarrollo, sus componentes principales se pueden ir viendo modificados con el tiempo, así que la arquitectura de Hadoop que veremos a continuación estará basada en la versión 2.7.3 de Hadoop, pero teniendo en mente que ya existe una versión 3.0 alpha que traerá novedades al HDFS, como la codificación de borrado, que se espera pueda llegar a reducir el coste de almacenamiento hasta un 50%.

Primero veremos cómo funciona el HDFS, y a continuación pasaremos a ver cómo es la arquitectura Map Reduce de Hadoop, y veremos un ejemplo de cómo ejecutar una tarea usando dicho paradigma.

2.1.1. Hadoop Distributed File System (HDFS)

El HDFS tiene una arquitectura de maestro/esclavo, donde contamos con un “NameNode” que hace de maestro, y varios “DataNode” que hacen de esclavos.

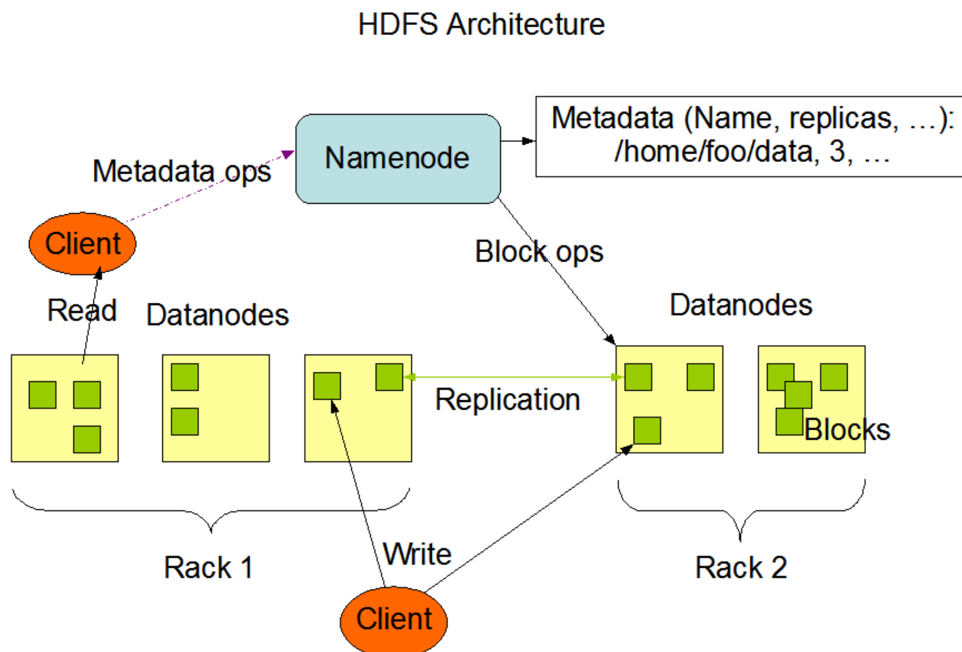


Figura 2: Arquitectura HDFS

Por lo general, a la hora de montar un cluster de Hadoop, lo normal es tener una máquina en la que se ejecute el NameNode, y un conjunto de máquinas que ejecutan un DataNode cada una.

El NameNode es el nodo maestro, y es el responsable de guardar todos los metadatos sobre la información almacenada en los DataNodes. Sin embargo, no guarda los datos en sí.

Los DataNodes también tienen varias responsabilidades como responder a las peticiones de lectura y escritura de los clientes, y llevar a cabo la replicación de bloques. Y es en los DataNodes donde se almacena toda la información que guardemos en nuestro HDFS.

Cuando queremos escribir en el HDFS, lo primero que tenemos que hacer es preguntarle al NameNode en qué DataNodes podemos escribir, y este nos dará una respuesta en función de la carga media de cada DataNode, del factor de replica que tengamos, y de la cantidad de datos que queramos escribir, y entonces ya podremos escribir directamente en los DataNodes, sin que la información pase por el NameNode, lo cual haría la escritura mucho más lenta, ya que crearíamos un cuello de botella en el NameNode. Lo mismo es aplicable para el proceso de lectura de información contenida en el HDFS, primero debemos realizar una petición al NameNode para que nos indique en qué DataNodes se encuentra la información que buscamos, y posteriormente ya podríamos leer directamente de los DataNodes.

La información almacenada en el HDFS se encuentra replicada por un factor de replica que por defecto es de 3, pero que podemos modificar si lo vemos conveniente.

Tener la información replicada ofrece varias ventajas, la primera y más evidente es que el HDFS va a seguir funcionando perfectamente aunque se caiga un DataNode (tolerancia a fallos), puesto que la información que contiene dicho DataNode va a estar almacenada también en otros DataNodes.

Otra ventaja importante, es que al tener la misma información almacenada en distintas máquinas, cuando ejecutemos un programa que trabaje con datos del HDFS, vamos a poder aprovechar mejor la localidad del dato (si un proceso no necesita traerse datos de otros nodos va a ejecutarse más rápidamente porque no va a necesitar intercambiar datos por la red), ya que más procesos van a tener disponible los datos en la misma máquina en la cual se están ejecutando.

El NameNode toma todas las decisiones sobre la replicación de bloques, y cada cierto tiempo, recibe “Hearbeats” o actualizaciones del estado de cada uno de los DataNodes, con información sobre el funcionamiento actual del DataNode, y también sobre la lista de los bloques que contiene actualmente cada DataNode. La replicación de datos es crítica para el HDFS, y es una de sus principales características, que le permite ofrecer cierta tranquilidad y seguridad a los usuarios sobre los datos que almacenan en el HDFS, ya que aunque perdamos un nodo, la información almacenada en dicho nodo se encontrará al menos en otra máquina (siempre que hayamos usado para configurar el HDFS un factor de réplica de al menos 2), y además permite reducir el tráfico de red al poder aprovechar la localidad del dato.

La localidad del dato hace referencia a que los programas que ejecutemos (tanto en Hadoop como en el caso de Spark) funcionarán de una forma mucho más eficiente si se ejecutan cerca del dato, y no es necesario transferir datos a través de la red lo cual suele ser bastante costoso y más cuando hablamos de Big Data. Para aprovechar la localidad del dato, el HDFS en el caso de Hadoop o cualquier otro almacén de datos que vayamos a utilizar (por ejemplo Cassandra que es una base de datos muy empleada como almacén de datos en el mundo Big Data), debería estar alojado en las mismas máquinas en las cuales tengamos nuestros procesos de MapReduce, para de esta forma aprovechar la localidad del dato.

En el HDFS la información se almacena en bloques, que tienen un tamaño predeterminado que podemos establecer nosotros mismos o dejar el que viene por defecto, que son 128Mb por bloque en la versión 2.7.3. Este tamaño es un factor muy importante en el rendimiento del HDFS, ya que si el tamaño por bloque es muy pequeño, por ejemplo 4Kb por bloque, y quieres leer un archivo de 1Gb, se van a producir unas 256.000 peticiones del NameNode a los DataNodes para obtener dicho archivo. Pero por otra parte, si tienes un tamaño de bloque de digamos 256Mb, con sólo 4 peticiones ya tendrías el archivo leído, aunque ahora el problema que tendríamos es que si necesitáramos almacenar un archivo de tan sólo 10Mb, ocuparía 256Mb, ya que este es el tamaño de bloque que hemos establecido, y estaríamos desperdiciando gran cantidad de espacio en el HDFS. Por tanto, es importante evaluar qué tamaño de bloque es más conveniente para nosotros en función del uso que le vayamos a dar al HDFS.

Por último, conviene destacar que los datos nunca pasan por el NameNode. El programa cliente solicita al NameNode los metadatos necesarios para llevar a cabo su operación, pero luego, el cliente leerá o guardará los datos directamente de los DataNodes. De lo contrario, se formaría un cuello de botella en el NameNode, y no serían eficientes los procesos de lectura o escritura.

2.1.2. Hadoop Map Reduce

A continuación vamos a ver algunos de los puntos más relevantes del motor de ejecución MapReduce, para de este modo tratar de entender cuál es la filosofía de trabajo que usan la mayoría de motores de ejecución Big Data de hoy en día para procesar cantidades enormes de datos.

En *Dean, J., Ghemawat, S. (2004)*, los autores comentan como los ingenieros de Google llevaban varios años realizando algoritmos capaces de funcionar en distribuido con cientos de nodos para llevar a cabo múltiples tareas que requerían el tratamiento de enormes cantidades de datos. El desarrollo de dichos algoritmos es complejo, porque requiere tener en cuenta varios aspectos importantes, como la forma de distribuir los datos y su procesamiento a lo largo de varias máquinas, qué hacer cuando una máquina falla al procesar su parte, etc. Como una reacción a estos problemas, surgió el motor de ejecución MapReduce, el cual se basa en las primitivas “map” y “reduce”, que están presentes en Lisp y en otros lenguajes funcionales. Los ingenieros de Google se dieron cuenta que la mayoría de operaciones que tenían que realizar involucraban una operación de “map” (en la cual se aplica una transformación a los datos registro a registro) para obtener un conjunto de tuplas de tipo clave/valor, que luego serán mezcladas en una operación “reduce” que se encargará de llevar a cabo una operación determinada sobre todas las tuplas que comparten la misma clave. El uso de un modelo funcional junto con la realización de operaciones map y reduce permite distribuir el procesamiento a lo largo de varios nodos sin muchos problemas, y además permitiéndonos reponernos si falla algún nodo al realizar su parte.

En la siguiente figura podemos ver cómo funciona el motor de ejecución MapReduce:

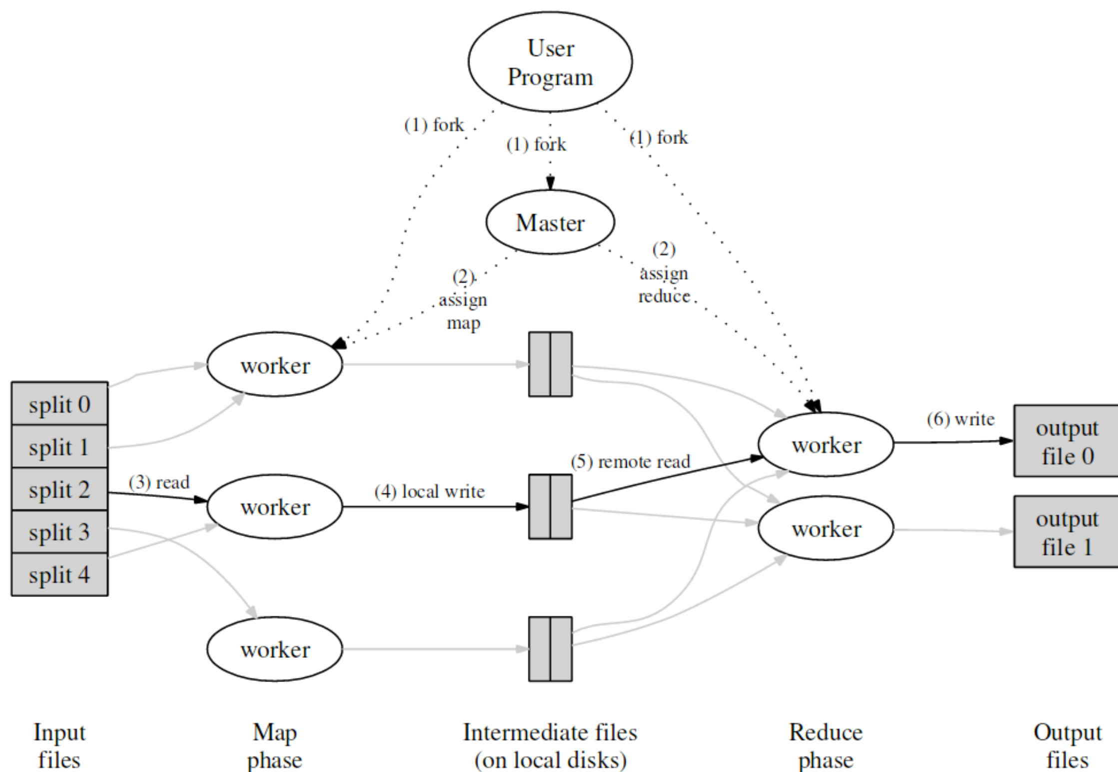


Figura 3: Modelo de ejecución de MapReduce

Siguiendo el esquema visto en la figura anterior, el modelo de ejecución de MapReduce funciona de la siguiente manera:

1. Se dividen los datos de entrada en distintas partes, y se distribuye el programa por todos los nodos del cluster.

2. Uno de los nodos del cluster será el maestro, y será el encargado de dividir el trabajo a través del resto de nodos que actúan como "workers". El master selecciona "workers" que estén inactivos y les asigna una tarea que puede ser de tipo "map" o de tipo "reduce".

3. Un worker al cual se le asigna una tarea de tipo "map" leerá los contenidos de una de las partes en las cuales dividimos los datos de entrada, llevará a cabo la tarea de tipo "map" definida por el usuario, y guardará en memoria las tuplas clave/valor que genere.

4. Cada cierto tiempo las tuplas almacenadas en memoria en cada uno de los "workers" son escritas a disco, y su ubicación se le envía al master que será responsable de asignar dicha información a los procesos de tipo "reduce".

5. Cuando un "worker" es notificado por el master para llevar a cabo una tarea de tipo "reduce", entonces dicho worker usa llamadas de procedimiento remoto para obtener las tuplas clave/valor generadas por varios procesos de tipo "map", y agrupa todas las tuplas que tienen la misma clave. El "particionador" que usa Hadoop por defecto garantiza que siempre lleguen todas las tuplas clave/valor correspondientes a una determinada clave al mismo proceso de tipo "reduce".

6. La salida generada por un proceso de tipo "reduce" será un conjunto de claves, con un conjunto de valores asociado a cada una de dichas claves. Y dicha salida será añadida a un fichero de salida final asociado a la partición que ha sido procesada.

7. Cuando todas las tareas de tipo "map" y de tipo "reduce" han finalizado, el programa se puede dar por completado, y se notifica al usuario de que sus datos de salida ya se encuentran disponibles en una carpeta, con un número de ficheros que será igual al número de particiones generadas, que dependerá a su vez del número de particiones que haya configurado el usuario en el programa de MapReduce.

A grandes rasgos este sería el funcionamiento general de un programa de tipo MapReduce.

Uno de los problemas que tiene el modelo de ejecución MapReduce es que obliga al usuario a definir unas funciones de tipo map y reduce para cualquier tipo de programa, y muchas veces también es necesario definir otras funciones intermedias de tipo combine. Esto a veces se adapta bien a nuestra tarea, pero otras veces hace que el desarrollo de cualquier aplicación, hasta la más trivial, resulte un poco tedioso. Por ejemplo, si queremos contar el número de palabras que aparecen en un documento usando para ello MapReduce, el código que deberíamos utilizar para ello sería así:

```
public class WordCount {

  public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws Exception {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws Exception {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Código 1: Código MapReduce para contar palabras

Mientras que si queremos llevar a cabo la misma operación, pero utilizando ahora Apache Spark, el código se reduce bastante y además resulta muy intuitivo.

```
val texto = spark.sparkContext.textFile(args[0])
val res = texto.flatMap(x => x.split(" ").map((_, 1))).countByKey
res.write.csv(args[1])
```

Código 2: Código Spark para contar palabras

2.1.3. Apache Spark

Siempre se podría pensar que la potencia que ofrece Apache Spark para escribir código de forma tan sencilla y en tan pocas líneas de código (como vemos en el código marcado como Código 2 en la página anterior), podría ser a costa de que dicho código fuera mucho menos eficiente que su homólogo en MapReduce. Sin embargo, ocurre justo lo contrario, Spark no sólo permite escribir programas de forma más cómoda y sencilla, sino que además es más rápido que MapReduce en la mayoría de operaciones que lleva a cabo, y a veces no sólo es un poco más rápido, sino hasta 100 veces más rápido como aparece en *Meng X., Bradley J., Yavuz B., Sparks E., Venkataraman S., Liu D., Freeman J., Zaharia M. and cols (2016)*.

Si a esto le añadimos, que Apache Spark es un proyecto de código abierto que tiene un gran apoyo de la comunidad (en el año 2016 ha sido el proyecto Big Data de código abierto con mayor número de commiters), no es difícil entender porque en los últimos años Apache Spark ha ido poco a poco reemplazando a MapReduce como el motor de ejecución Big Data por antonomasia. Así que a continuación vamos a ver como es la arquitectura de este motor de ejecución de carácter general y distribuido, y qué le hace tan potente respecto a su principal competidor, Apache Hadoop.

Apache Spark empezó en el año 2009 como parte de un programa de investigación de la Universidad de California iniciado por Matei Zaharia, y en el año 2013 el proyecto fue donado a la fundación Apache. Una de las primeras cosas que llama la atención es que mientras que Hadoop fue escrito en Java, Spark ha sido desarrollado en Scala, un lenguaje de programación más funcional y conciso que Java.

Para empezar, vamos a comentar brevemente los distintos componentes de los que está compuesto Apache Spark:

- Spark Core: este es el componente principal de Spark, ya que contiene las funcionalidades básicas necesarias para ejecutar procesos en Spark, por lo que es usado por el resto de componentes de Spark. La principal abstracción de Apache Spark son los RDD (Resilient Distributed Dataset), que son colecciones de elementos de un determinado tipo (por ejemplo podemos tener RDD’s de String, Longs, Rows, etc.), que son inmutables (de solo lectura), resistentes a fallos, y distribuidos a lo largo de varios nodos del cluster. El API de Spark contiene varios métodos para transformar distintos tipos de datos de entrada (por ejemplo CSV, parquet, ElasticSearch, Cassandra, Hbase, etc.) en un RDD. Además el Spark Core contiene toda la lógica necesaria para acceder a varios sistemas de archivos, como HDFS, GlusterFS, Amazon S3, etc.

- Spark SQL: este componente es el encargado de permitirnos trabajar con datos estructurados, y además ser capaces de lanzar consultas SQL sobre dichos datos como si de una base de datos tradicional se tratara. La principal abstracción de este componente es el DataFrame (RDD de Rows), que en Spark 1.6 pasa a considerarse un Dataset de Rows. Durante la ejecución de un job de Spark las operaciones que realizamos con SparkSQL son optimizadas usando un componente conocido como “Catalyst”, y finalmente el trabajo realizado tanto con dataframes como con datasets acaba siendo traducido a operaciones con RDDs.

- Spark Streaming: este componente es el encargado del tratamiento de datos en tiempo real. Aunque, a decir verdad, lo que realmente hace este componente es crear mini-batchs compuestos por un RDD que contiene los datos que han llegado en la ventana de tiempo que hayamos definido (por ejemplo cada 5 segundos), y que son almacenados en Dstreams (Discretized Streams). No obstante, en la versión 2.0 de Spark han añadido un nuevo API para streaming estructurado que hace que los programas de streaming acaben siendo muy similares a aquellos de tipo batch.

- Spark MLlib: este componente es el que contiene todos los algoritmos de aprendizaje automático desarrollados utilizando Spark. Aquí podemos encontrar algoritmos de clasificación (como por ejemplo, arboles de decisión), regresión (como por ejemplo, regresión lineal), y clusterización (como por ejemplo mezclas gaussianas). Aquí nos encontramos con dos API's, uno más antiguo, basado en RDD's (almacenado en el paquete spark.mllib), y que a partir de la versión 2.0 de Spark ha entrado en modo mantenimiento (con vistas a ser eliminado en la versión 3.0 de Spark), y un nuevo API basado en DataFrames (almacenado en el paquete spark.ml) que es el foco actual de desarrollo. Esto nos permite usar el Spark Core para leer datos en un DataFrame, y directamente poder empezar a generar un modelo usando el nuevo API basado en DataFrames.

- Spark GraphX: este componente de Spark contiene varios algoritmos importantes para el trabajo con grafos, como por ejemplo "page rank", SVD++, "shortest paths", etc. La principal abstracción de este componente son los RDD's de vértices (VertexRDD) y los RDD's de lineas que conectan esos vértices (EdgeRDD) que dan lugar a los gráfos.

A continuación vamos a ver como es la arquitectura de Apache Spark. Una aplicación en Spark está compuesta por un programa llamado "Driver" que es el encargado de arrancar varias operaciones en paralelo sobre un cluster, y un conjunto de nodos que serán los encargados de llevar a cabo las distintas tareas y que son conocidos como "workers".

El "Driver" contiene la aplicación que hemos codificado en Spark para realizar distintas operaciones sobre datasets, y dos subcomponentes, el "scheduler" y el "SparkContext", que son los encargados de solicitar recursos al cluster (memoria y cores), dividir la lógica de la aplicación en distintas fases y tareas, enviar las tareas a los executors que se encuentran en los nodos "workers", y recolectar todos los resultados cuando queremos que los datos de un RDD dejen de estar distribuidos, y pasen a estar en una sóla máquina (en este caso sería en el Driver), lo cual puede ser hecho mediante el comando "collect". El objeto "SparkContext" se usa para construir RDD's a partir de los datos de entrada, y nos permite llevar a cabo operaciones en paralelo, como por ejemplo contar el número de veces que aparece la palabra "Exception" en un fichero de log de 500 Gb que ha sido previamente leído y transformado en un RDD de Strings. Para llevar a cabo este tipo de operaciones en paralelo, el Driver solicita a los nodos de nuestro cluster, conocidos como workers, que tengan algún espacio libre que ejecuten tareas en algunos de sus executors.

Un nodo de tipo worker puede tener varios executors, cada uno de ellos con un número de cores y de memoria RAM determinada (que puede configurarse previamente a la ejecución de un programa de Spark), y a su vez cada executor puede tener espacio para ejecutar varias tareas. Por otra parte, un cluster de Spark puede ejecutar distintos jobs que no tengan nada que ver uno con el otro (por ejemplo un Job aparece en color verde, y otro en color amarillo), y a cada job se le asignaran un conjunto de recursos (principalmente cores y memoria), o se mantendrá en espera hasta que se liberen suficientes recursos del cluster de Spark para poder ejecutarse. Todo esto queda reflejado en la siguiente figura:

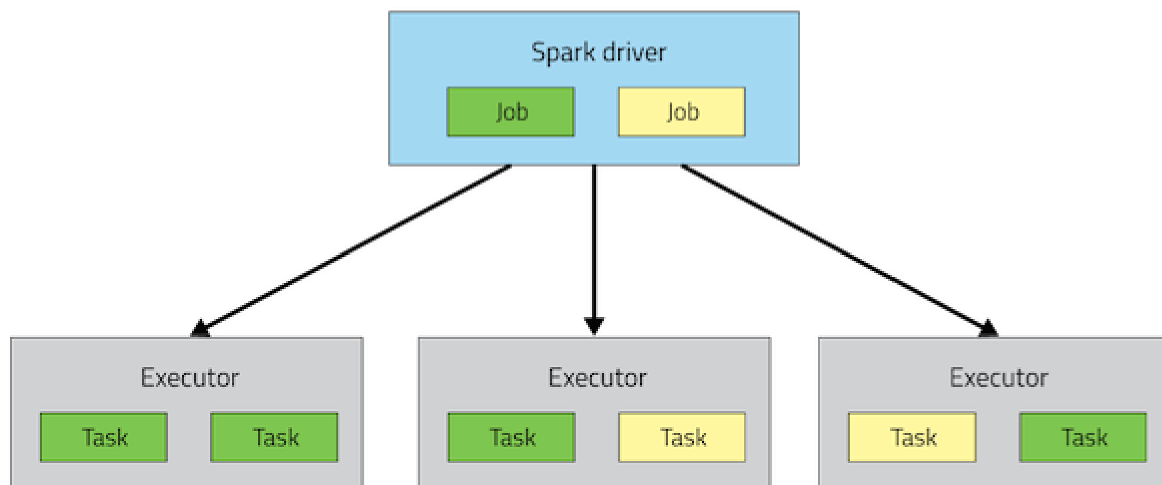


Figura 4: Ejecución de una tarea en Spark

Existen varios artículos que muestran la rapidez de Spark frente al motor de ejecución de Hadoop, conocido como MapReduce. Por ejemplo, en *Meng X., Bradley J., Yavuz B., Sparks E., Venkataraman S., Liu D., Freeman J., Zaharia M. and cols (2016)* aparece información que pone de manifiesto como para algunos procesos de aprendizaje automático, la diferencia de velocidad entre ambos frameworks puede estar en el orden de 100 veces mayor velocidad de ejecución para Spark. Dos de las características más relevantes para entender el por qué Spark tiene un rendimiento tan bueno son las siguientes:

- <u>DAG (Directed Acyclic Graph):</u> Spark siempre espera el mejor momento para empezar a trabajar, y no se pone a procesar tareas tan pronto como le llegan, sino que espera hasta que realmente es necesario ejecutar una acción porque deba devolver algún tipo de información. Mientras que las tareas que simplemente transforman un RDD en otro RDD simplemente se añaden al grafo dirigido acíclico (DAG) y no producen el lanzamiento de una tarea. Antes de llevar a cabo una acción, Spark optimiza todas las tareas que debe hacer para de esta forma determinar cuál es el modo óptimo de llevarlas a cabo. Además, esta forma de trabajar permite a Spark que si tiene que realizar varias tareas distintas, pueda aprovechar la salida de una tarea que se encuentra en memoria, para directamente pasarle dicha referencia a la siguiente tarea, y de esta forma evitar tener que escribir los datos de salida a disco, con el gasto en tiempo que genera la escritura y posterior lectura de dichos datos.

- Procesamiento de datos en memoria: Spark intenta trabajar con los datos en memoria, siempre que los recursos de los workers lo permiten, de esta forma, el rendimiento mejora ostensiblemente con respecto a Hadoop, donde cada operación Map-Reduce acaba volcando su salida al HDFS. Esta diferencia puede no tener un impacto demasiado grande para determinadas tareas donde solo es necesario pasar una vez a través de todos los datos, pero en determinadas tareas que requieren pasar múltiples veces por los mismos datos, como por ejemplo al usar algoritmos de aprendizaje automático, el procesamiento en memoria RAM que hace Apache Spark hace que el rendimiento de éste sea muy elevado respecto al modelo de ejecución en disco de Map-Reduce.

A modo de resumen de este apartado, podríamos decir que las principales características y diferencias entre Apache Hadoop y Apache Spark son las siguientes:

Apache Hadoop	**Apache Spark**
Provee un modelo sencillo de programación distribuida	Provee un modelo sencillo de programación distribuida, pero más flexible e intuitivo que Hadoop
Cada fase de un job de MapReduce necesita leer/escribir de disco antes de pasar a la siguiente fase	Se realiza procesamiento en memoria RAM siempre que es posible, y sólo se escribe a disco cuando es necesario
La programación se realiza a bajo nivel, utilizando simplemente operaciones englobadas como de tipo Map o de tipo Reduce	La programación se realiza a alto nivel, con un mayor nivel de abstracción, y un optimizador que se encarga de preparar el plan de ejecución a bajo nivel
Arquitectura maestro/esclavo	Arquitectura maestro/esclavo
Escalable	Escalable
Tolerante a fallos	Tolerante a fallos
Corre sobre hardware no especializado	Corre sobre hardware no especializado
Desarrollado en Java, que además es el lenguaje de programación utilizado para programar en este framework	Desarrollado en Scala, el cual es el lenguaje recomendado de programación en Spark, pero también tiene APIs para programar en Java, python, y R

2.2. Minería de datos con Big Data

2.2.1. Integración y recopilación de información

El primer paso que debemos dar antes de llevar a cabo cualquier tarea de extracción de conocimiento es obtener la materia prima que usaremos para generar dicho conocimiento, y dicha tarea no es trivial porque hoy en día existe conocimiento almacenado en todo tipo de fuentes. Por tanto, debemos tener en cuenta que una parte importante del trabajo es la integración y recopilación de información. Para el caso de estudio que trata este trabajo, ya contaremos con dicha información recopilada en un directorio del HDFS en formato parquet (formato columnar compatible con Hadoop), por lo que no nos supondrá ningún desafío esta primera etapa del proceso de extracción de conocimiento.

Hoy en día, existen múltiples herramientas que nos permiten llevar a cabo el proceso de integración y recopilación de información, y dado la diversidad de fuentes que existen, no podemos citar todas y cada una de ellas, ya que el uso de una u otra herramienta dependerá de la fuente de donde provengan nuestros datos. Por ejemplo, si tuviéramos nuestros datos en una base de datos relacional como Oracle, podríamos utilizar Apache Sqoop, una herramienta que usa procesos Map-Reduce para obtener los datos de las tablas que necesitemos, y los exporta directamente al HDFS en el formato que le especifiquemos de entre los que soporta (CSV, parquet, Hbase, etc.). Al usar Hadoop, para su funcionamiento, Sqoop nos permite extraer grandes cantidades de datos de bases de datos relacionales en un tiempo razonable, aunque eso sí, tendremos que tener en cuenta el impacto que puede llegar a tener en el rendimiento de dicha base de datos, ya que Sqoop cuando está llevando a cabo una extracción puede abrir un número elevado de conexiones y hacer que el rendimiento de la base de datos se resienta.

Una de las fuentes de información más grande del planeta es sin lugar a dudas Internet, por lo que no es de extrañar, que desde que aparecieron los primeros “web crawlers” a principios de los 90, estos hayan evolucionado hasta convertirse en auténticos devoradores de la información que aparece en las miles de millones de páginas webs que existen hoy en día (exactamente 1.130.500.653 a día 4 de enero de 2017 según http://www.internetlivestats.com/total-number-of-websites/ y este número sería aún mayor, puesto que estas estadísticas no tienen en cuenta el inmenso número de páginas que forman parte de la deep web). Google usa “web crawlers” para recopilar información de las distintas páginas web que visitan, e indexar las páginas web (uno de sus crawlers más conocido es Googlebot), y luego usa dicha información para mejorar su buscador, de forma que cuando busquemos en Google la palabra “perros”, no sólo obtengamos una lista de páginas que contienen la palabra “perro” repetida muchas veces, sino las páginas que hablan sobre “perros” y que además son más relevantes según el sistema de rating de páginas de Google.

Si queremos obtener información más específica, como por ejemplo los precios de una tienda de comercio electrónico, podemos recurrir al “web scraping”, que mediante procesos de parseo de metadatos de las páginas web (CSS, DOM, etc.), o de búsqueda de patrones, es capaz, de extraer información de una página web, y además podría hacerlo de forma periódica en caso de que fuera necesario (por ejemplo para obtener las fluctuaciones en los precios de artículos que nos interesan) .

A día de hoy, uno de los mayores problemas a los que se enfrentan estos robots que recorren la World Wide Web recopilando todo tipo de información, no es un problema tecnológico, sino un problema legal, y es que los propietarios de muchas páginas web pueden no estar de acuerdo en que la información que tienen en sus servidores sea recopilada de forma periódica por un robot y usada para sus propios fines. Por tanto, antes de iniciar un proceso de "web scraping" deberíamos tener en cuenta la legalidad vigente en el país de origen, las condiciones legales del sitio web, derechos de autor, de propiedad intelectual, o de marca registrada. O incluso tendremos que informarnos sobre si el uso de los datos obtenidos mediante nuestro robot podría considerarse como competencia desleal.

Por último, tras recopilar toda la información que necesitemos de múltiples fuentes (bases de datos relacionales, archivos en cualquier formato como CSV, XML, parquet, etc., o directamente de páginas web de Internet), el siguiente paso sería usar una herramienta que nos permita integrar toda esa información en un mismo sitio y a ser posible en un mismo formato que sea fácilmente digerible por nuestros algoritmos de minería de datos. Además, seguramente, necesitaremos unir los datos de distintas fuentes pero que se refieren a una misma instancia. Por ejemplo, podríamos haber recopilado de una base de datos el título de un disco de música, y el ID del usuario que ha comprado dicho disco, y por otra parte, podríamos haber usado un robot para obtener información adicional sobre cientos de miles de discos de música, y nos puede interesar unir dicha información por ejemplo, para utilizar posteriormente un algoritmo recomendador, que nos recomiende discos de música en función de los discos que hemos comprado anteriormente. En la siguiente página vamos a ver un ejemplo más en detalle, y cómo se realizaría su implementación en Apache Spark.

Imaginemos que hemos reunido información relevante sobre personas, y por un lado tenemos en parquet el contenido de una extracción que hemos realizado de una tabla de una base de datos, que contiene el id de usuario, el email, y el teléfono, y por otro lado, tenemos un CSV con cientos de miles de emails e información de todo tipo sobre dichos emails que hemos obtenido usando un proceso de "web scraping". Ahora, deberíamos usar alguna herramienta que nos permita trabajar con ambos formatos (CSV y parquet), y que sea capaz de unir los datos por algún campo específico (en nuestro caso por el email), y a su vez, estaría bien, que dicha herramienta fuera capaz de procesar grandes cantidades de datos en un tiempo razonable.

Actualmente, si queremos trabajar con una cantidad considerable de datos, una de las herramientas más potentes que existen para ello, es Spark, el cual nos permite leer archivos de distintos formatos y distintas fuentes (Cassandra, ElasticSearch, CSV, JSON, parquet, etc.), y transformarlos en RDD's para poder trabajar con ellos en memoria sin necesidad de guardarlos en disco para llevar a cabo distintas operaciones y transformaciones. Posteriormente podemos guardar dichos datos ya transformados e integrados en donde deseemos, puesto que Spark, también permite escribir dichos datos en varios formatos y almacenes. En Spark podríamos hacer esto con tan sólo 4 líneas, e incluso directamente desde su intérprete o shell:

```
val users = spark.read.parquet("hdfs://server:9000/home/app/users")
val info = spark.read.csv(hdfs://server:9000/home/app/info")
val resJoin = users.join(info, users("email") === info("email"))
resJoin.write.parquet("hdfs://server:9000/home/app/result")
```

2.2.2. Selección, limpieza, y transformación de información

En cuanto al segundo paso del proceso de extracción de conocimiento, la selección, limpieza, y transformación de información, cabe decir que la selección de variables relevantes es foco de gran investigación en la actualidad. Y en el caso que nos ocupa, una tarea de clusterización, es un punto clave, ya que si usamos demasiadas variables irrelevantes entonces dichas variables van a generar tanto ruido, que van a ser las que lleven todo el peso de las agrupaciones, y entonces los grupos creados serán también poco relevantes. No obstante, para este trabajo, tenemos claro cuáles son las variables relevantes, puesto que han sido seleccionadas y consensuadas con un experto en el dominio (astrofísica), y su número no es muy elevado (cinco variables). Además, si tenemos en cuenta que contaremos con un gran número de ejemplos (miles de millones), no parece que vaya a ser necesario utilizar ningún método de selección de variables en nuestro caso.

Básicamente, existen dos tipos de métodos de selección de variables:

- Métodos de filtro: tienen en cuenta principalmente medidas estadísticas para aplicar una puntuación a cada variable, y poder seleccionar así las variables más relevantes (coeficientes de correlación, test Chi-cuadrado, ganancia de información, etc.)

- Métodos basados en modelo (métodos de envolvente o wrapper): usan diferentes subconjuntos de los atributos para generar modelos y evaluar su calidad. Normalmente, se obtienen mejores resultados usando este tipo de métodos, sin embargo son métodos muy costosos computacionalmente, puesto que hay que generar muchos modelos (uno por cada subconjunto de variables que queramos probar), lo cual puede ser un problema si tenemos un número muy elevado de variables, o si tenemos un dataset con muchos ejemplos.

Hay que tener en cuenta que si queremos usar un método de selección de variables basado en modelo para llevar a cabo una tarea de clusterización, no tenemos una medida como la precisión obtenida por el modelo generado para saber qué puntuación deberíamos otorgar a el subconjunto de atributos usado, por lo que deberíamos recurrir a índices de validación interna o externa, los cuales en muchas ocasiones no ofrecen una puntuación tan clara para un subconjunto de atributos como la ofrecida por la precisión.

Otra opción sería utilizar una selección paso a paso de atributos, de forma que empezamos realizando una clusterización con cada uno de los atributos por separado, escogemos aquel atributo que forma grupos de mayor calidad de acuerdo a algún índice de validación interna o externa, y le añadimos a la lista de atributos seleccionados. Después haríamos lo mismo con el resto de atributos individuales y con la lista de atributos seleccionados, y continuaríamos seleccionando los que mejor resultado dan, hasta obtener el número de atributos que deseamos.

Por otra parte, tenemos la opción de usar métodos de transformación de variables, como el análisis de componentes principales (PCA), que nos permiten también reducir el número de variables de entrada a nuestro modelo, pero a costa de transformar las variables originales en otras nuevas, y por tanto perdiendo poder de comprensibilidad de los modelos que generemos posteriormente con ellas.

Actualmente, gracias al auge de herramientas big data, ya no es tan necesario llevar a cabo una "selección de datos", es decir, un muestreo del dataset para quedarnos con un número más reducido de datos en beneficio de un menor tiempo de generación del modelo que queremos obtener. No obstante, puede ser interesante realizar un muestreo de datos para generar varios prototipos, que nos ayuden a decidir qué modelo aplicar, aunque siempre correremos el riesgo de haber dejado fuera de la muestra algunos ejemplos que fueran muy relevantes por sus características y que pudieran impactar de forma importante en el modelo generado.

Respecto a la transformación de datos, en muchos casos, es conveniente recurrir a un experto en el dominio que puede sugerirnos algunas transformaciones interesantes que pueden ayudarnos a la hora de crear un modelo con mayor poder predictivo. Por ejemplo, si estamos interesados en predecir el precio por metro cuadrado de la vivienda en distintos barrios, y tenemos varios atributos, entre ellos, el número de habitantes de un barrio, y la superficie que ocupa dicho barrio, podría ser interesante obtener un nuevo atributo, que nos de la densidad de habitantes de ese barrio, dividiendo el número de habitantes entre la superficie ocupada por el barrio (en kilómetros cuadrados por ejemplo).

También es importante llevar a cabo la limpieza de los datos recopilados, por ejemplo detectando ejemplos que parecen incorrectos, como por ejemplo, ejemplos donde aparece que una persona tiene una edad negativa o mayor de 200 años, o imaginemos que la superficie ocupada por un barrio fuera mayor que el tamaño del país en el cual se encuentra dicho barrio. Para detectar estos casos, podemos recurrir a una sumarización de los datos, obteniendo estadísticos básicos como media, desviación típica, varianza, mínimo, máximo, etc. y también podemos recurrir a histogramas, o cualquier otra herramienta estadística que nos ayude a detectar estos casos.

Igualmente importante es detectar atributos que son nulos o incorrectos para la mayoría de ejemplos que forman parte de nuestro dataset, y para esos casos hay que pensar si merece la pena emplear tiempo en "intentar" arreglar esos atributos, o si es mejor directamente descartarlos y tratar de buscar otra fuente de información más fiable para obtener información relacionada con dichos atributos.

Uno de los mayores problemas que nos podemos encontrar cuando tenemos muchas variables y no realizamos correctamente la selección de variables relevantes es que caigamos en la denominada "maldición de la dimensionalidad". Esta expresión fue acuñada por Bellman en 1961 para referirse al problema que observaba en muchos algoritmos, los cuales funcionaban bastante bien cuando trabajaban con pocas dimensiones, pero no ofrecían buenos resultados cuando usaban muchas dimensiones. El problema es que cuantas más variables tenemos, más difícil es para un algoritmo generalizar a partir de los datos usados para el entrenamiento, puesto que el espacio cubierto por el dataset es una una parte muy pequeña del espacio formado por tantas variables. Y esto hace que en espacios con muchas dimensiones, todos los ejemplos se parezcan mucho.

Otro tema tema importante que debemos tratar antes de aplicar un algoritmo de mineria de datos, es la normalización de los datos. Este tema es de gran importancia en los algoritmos de clusterización, ya que si tenemos una dimensión cuya escala es demasiado grande en comparación con el resto, dicha dimensión va a ser la que finalmente determine a qué grupo pertenecerá cada punto de nuestro dataset. Por ejemplo, si tenemos el dataset de la siguiente figura, donde nos encontramos con dos dimensiones X1 y X2, y usamos k-means para realizar una clusterización, obtendríamos los dos grupos que vemos en rojo y azul en la figura 5.

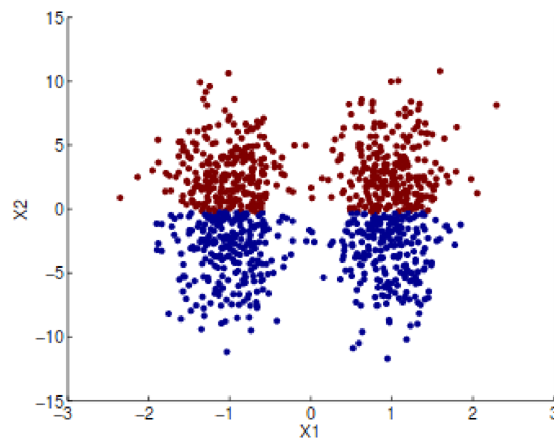


Figura 5: Clusters generados con datos en bruto

Esta agrupación se habría producido de esta forma debido a que la dimensión X2 como vemos tiene una escala mucho mayor que la dimensión X1, y esto hace que la dimensión X2 sea dominante, y sea la que se imponga a la hora de crear los grupos. Sin embargo, si antes de usar el algoritmo de clustering, normalizamos ambas dimensiones, por ejemplo mediante un proceso de estandarización (restando a cada punto la media y dividiendo entre la desviación típica), entonces ahora la agrupación que obtendremos se parecería más a la siguiente:

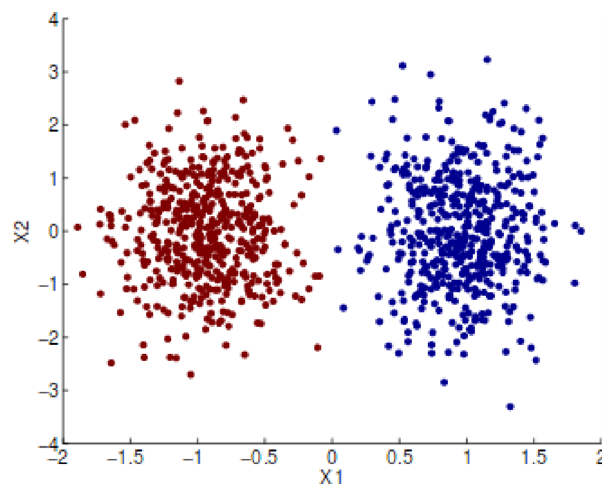


Figura 6: Clusters generados tras estandarización

Como podemos ver, la normalización previa de nuestro dataset antes de aplicar el algoritmo de clustering, puede tener un impacto importante en los grupos que obtenemos. Por tanto, siempre es recomendable valorar si es conveniente o no realizar una normalización de nuestros datos, puesto que no siempre es recomendable realizar una normalización.

El objetivo de normalizar nuestro dataset suele ser intentar dar la misma importancia a todas las variables, logrando así algo de objetividad, lo cual puede ser una buena idea, sobre todo si no tenemos conocimiento sobre el dominio en el que estamos trabajando, pero si tenemos conocimiento relevante de dicho dominio, o podemos contar con algún experto en la materia, puede ser interesante utilizar dicho conocimiento para escalar las variables, asignar un peso determinado a cada variable en función de su importancia relativa, o llevar a cabo una determinada normalización que nos ayude a obtener los grupos que buscamos.

Cuando usamos un método de clusterización, cada variable va a tener una importancia relativa a su escala, y como hemos visto, esto puede remediarse recurriendo a algún tipo de normalización. Pero además, nosotros podemos influir en la importancia dada a una variable determinada, por ejemplo multiplicándola por un determinado peso, o repitiendo dicha variable. Al final, las agrupaciones son procesos con una importante carga subjetiva, puesto que la agrupación que hagamos siempre va a depender de las variables en las cuales pongamos más énfasis, ya sea dándolas un mayor peso, repitiéndolas, o incluso modificando sus unidades de medida.

Por último, puede ser importante realizar una detección de valores anómalos o atípicos (conocidos como "outliers" en la literatura inglesa), que son objetos del dataset que presentan valores muy diferentes respecto al resto de los objetos del dataset, ya sea debido a errores en los datos, o a valores reales pero muy extremos. Existen tareas de minería de datos, como la detección de fraude en operaciones financieras, o la detección de intrusiones no deseadas en sistemas conectados a una red, cuyo principal objetivo es la detección de outliers.

Existen varias técnicas para la detección de outliers:

- Técnicas basadas en k-NN: Podemos utilizar el método k-NN (k vecinos más próximos), el cual se basa en calcular la distancia máxima d entre cada punto y sus k vecinos más próximos, y luego ordenar los objetos en orden descendente por dicha distancia máxima d, de forma que los primeros objetos de la lista serían outliers. La generación del modelo de k-NN no tiene ningún coste computacional a priori, ya que dicho modelo simplemente memoriza todos los ejemplos del dataset y es en el momento de llevar a cabo una predicción cuando se realizan los cálculos pertinentes. Para la detección de outliers, podemos guardar también las distancias máximas de los l elementos que consideremos como outliers, y posteriormente, con cada nuevo ejemplo comparamos la distancia máxima a sus k vecinos más próximos con la distancia máxima de los l outliers que hemos seleccionado anteriormente.

- Técnicas estadísticas: este tipo de técnicas simplemente buscan objetos del dataset que se encuentren muy alejados del resto. Por ejemplo, podríamos estandarizar los datos (centrando en la media y dividiendo por la desviación típica), y luego detectar aquellos puntos del dataset que tienen en alguna de sus dimensiones un valor superior o inferior a un determinado valor (si nuestras variables siguen una distribución cercana a la normal, podríamos elegir 5, por ejemplo, ya que prácticamente ningún ejemplo debería desviarse tanto de la media). Este método de detección de outliers es muy simple, y no tiene en cuenta por ejemplo que pueda haber distintos grupos de elementos que tienen sus propias características diferenciadoras del resto y que no deberían considerarse outliers. Otro de los problemas que presenta este método es que es univariado, y en cuanto una variable se desvía de la media, aunque el resto de variables tengan valores normales para esa instancia, se va a considerar como un outlier.

- Técnicas basadas en clustering: las técnicas de detección de outliers basadas en clustering, en general, se basan en considerar como outliers a aquellos elementos que tras ser agrupados por un proceso de clusterización (por ejemplo usando el algoritmo k-means) se encuentran más alejados de su centroide. Por ejemplo, en el artículo *Chawla, S., Gionis, A. (2013)* aparece un método basado en k-means para detectar outliers, a la vez que clusterizamos un dataset.

  Para utilizar esta técnica de detección de outliers sugerida por Chawla y Gionis hay que tener en cuenta que debemos conocer el número de grupos k en los que se divide el dataset, y el número de outliers l que contiene dicho dataset. Podemos utilizar la ayuda de un experto en el dominio para obtener buenas aproximaciones de k y l, o proceder de forma experimental. Esta técnica que nos permite clusterizar un dataset a la vez que vamos obteniendo outliers de una forma unificada, funcionaría, a grandes rasgos, de la siguiente manera:

  1. Seleccionamos k puntos al azar del dataset, que serán considerados centroides.

  2. Calculamos la distancia de cada punto del dataset original a su centroide más cercano.

  3. A continuación, los puntos son ordenados en forma decreciente, en función de la distancia a su centroide más cercano, y los l primeros puntos son considerados outliers.

  4. Se crea un nuevo dataset usando todos los puntos del dataset original menos los l puntos seleccionados en el paso 3.

  5. Calculamos los nuevos centroides a partir del dataset generado en el punto 4, y volvemos a repetir todo el proceso desde el punto 2 hasta que logremos la convergencia o alcancemos un número máximo de iteraciones.

  Este algoritmo puede ser una buena alternativa cuando conocemos el número de grupos en los que se divide nuestro dataset, y además tenemos una buena estimación del número de outliers que nos podemos encontrar. Por otra parte, si necesitamos obtener los outliers de datasets de gran tamaño, este método puede resultar una buena elección, ya que resulta un algoritmo totalmente escalable y con un orden de complejidad lineal.

- Técnicas basadas en redes neuronales: en el artículo *Hawkins, S., He, H., Williams, G., Baxter, R. (2002)*, aparece una técnica de detección de outliers que está basada en la construcción de una red neuronal replicante (Replicator Neural Network), es decir, una red neuronal en la cual tanto la entrada como la salida de la red va a ser la misma, el vector de datos definido por cada instancia. Pero entre la capa de entrada y la capa de salida se sitúan varias capas ocultas (Hawkings y cols. emplean tres capas ocultas exactamente, aunque en otros artículos como *Toth, L., Gosztolya, G. (2004)* han demostrado que no es necesario usar tres capas ocultas, y se pueden obtener resultados similares usando un número diferente de capas ocultas). Veamos a continuación cómo funciona la técnica de detección de outliers propuesta por Hawkins:

  - En primer lugar, tenemos que construir una red neuronal con una capa de entrada de "n" neuronas y una capa de salida con el mismo número "n" neuronas, donde "n" es el número de variables del dataset. La red neuronal también contará con tres capas ocultas, cuyo tamaño será elegido experimentalmente con el objetivo de minimizar el error de reconstrucción de todas las instancias del dataset. La función de activación utilizada en la primera y la última capa oculta será la función tangente sigmoidal (que va de -1 a 1), mientras que en la capa oculta del medio se usa una función en escalera que transforma los puntos distribuidos de forma continua a la entrada en un número discreto de valores, de forma que conseguimos así comprimir los datos (podríamos decir que esta capa intermedia realiza una especie de clusterización).

  - A continuación, una vez hemos construido nuestra red neuronal, pasamos a entrenarla, con el objetivo de que sea capaz de devolvernos a la salida una reconstrucción lo más cercana posible al propio vector de entrada.

  - Una vez que hemos entrenado nuestra red, para detectar outliers ya sólo nos queda introducir datos a nuestra red neuronal, y calcular el error cometido por la red a la hora de tratar de reconstruir el vector de entrada, y aquellos ejemplos en los cuales el error cometido sea mayor, serían considerados outliers.

El método basado en redes neuronales replicantes propuesto por Hawkins puede resultar costoso computacionalmente, ya que requiere entrenar una red neuronal a partir de nuestros datos, aunque se puede paralelizar en parte si para entrenar la red neuronal usamos pequeños batchs de ejemplos, de forma que los pesos de la red sólo se actualizan tras haber pasado todos los ejemplos del batch, y posteriormente calculamos el error total cometido por los ejemplos del batch que estamos procesando actualmente, para a partir de dicho error actualizar la matriz de pesos de la red. En cuanto al último paso del método propuesto por Hawkins, que consiste en obtener el error cometido por cada ejemplo, al no tener que llevar a cabo ya ninguna modificación de los pesos de la red, esta parte puede paralelizarse completamente sin problemas.

2.2.3. Técnicas de Minería de datos

Respecto al tercer paso, la tarea de minería de datos, debemos tener en cuenta que para llevar a cabo una clusterización de varios miles de millones de objetos, no podemos utilizar cualquier librería de aprendizaje automático existente hoy en día, como el paquete NbClust en R, o la librería scikit-learn en Python, ya que dichas herramientas no están preparadas para trabajar de modo distribuido. Por lo tanto, es necesario utilizar frameworks que sean capaces de tratar grandes cantidades de datos de forma distribuida haciendo uso de varios nodos, ya que si nos limitamos a utilizar los recursos de una sola máquina, nuestro trabajo puede prolongarse tanto en el tiempo que haga prácticamente imposible trabajar, o incluso puede obtener la respuesta cuando dicha respuesta ya no es útil. Por ejemplo, imagina que queremos diseñar un algoritmo para predecir el tiempo que hará el próximo año, y para entrenar dicho modelo tardamos 2 años.

Necesitamos buscar una herramienta que cumpla dos requisitos: tenga suficientes algoritmos de clusterización implementados y probados para poder ser usados, y además pueda funcionar de forma distribuida. Actualmente, uno de los framework de computación distribuida más utilizados es "Apache Spark", el cual además tiene un modulo específico para aprendizaje automático llamado "MLlib", y cuenta con varios algoritmos de clusterización, como K-means, bisecting K-means, y mezclas gaussianas.

Otra herramienta de computación distribuida que puede utilizarse para tareas de aprendizaje automático y clusterización es "Apache Mahout", el cual está basado en Hadoop MapReduce. Sin embargo, este framework se ha visto claramente superado por "Apache Spark". Y de hecho, en las últimas distribuciones de Mahout, se han empezado a integrar ciertas características propias de Spark para tratar de mejorar el rendimiento de Mahout.

En la siguiente gráfica, procedente del artículo *Meng X., Bradley J., Yavuz B., Sparks E., Venkataraman S., Liu D., Freeman J., Zaharia M. and cols (2016)*, podemos ver una comparativa de rendimiento usando Mahout y el módulo MLlib de aprendizaje automático de Spark para correr un algoritmo de filtro colaborativo (usado habitualmente en sistemas recomendadores), concretamente el algoritmo ALS (alternating least squares), empleando varios datasets de distintos tamaños:

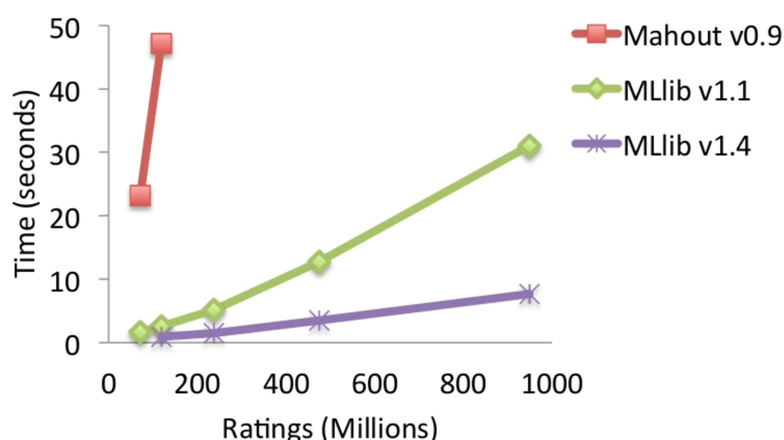


Figura 7: Comparación MLlib vs Mahout con algoritmo ALS

Como podemos observar en la figura 7, Spark no sólo tiene un mejor rendimiento que Mahout, sino que además escala mucho mejor cuando aumentamos el número de datos que tenemos que tratar. Mahout empieza a tener problemas cuando tiene que tratar más de 200 millones de elementos, y su tiempo se incrementa con una pendiente mucho más abrupta cuando aumentamos el número de registros.

Otra ventaja de "Spark", aparte del rendimiento, respecto a "Apache Mahout", es que tiene API's en varios lenguajes de programación muy utilizados por la comunidad de científicos de datos, como son "R" y "python". Y también tiene un API de Java, uno de los lenguajes de programación más empleados actualmente en el mundo. Para este trabajo se ha preferido utilizar directamente Scala, que es el lenguaje de programación en el cual está desarrollado Spark, y que nos ofrece varias ventajas respecto a los otros API's según Ryza, S., Laserson, U., Owen, S., Wills, J. (2015):

- Mayor eficiencia: no es necesario hacer una transformación de un lenguaje de programación a otro, ya que Spark entiende directamente Scala, al estar desarrollado en dicho lenguaje.

- Acceso directo a los últimos avances hechos en Spark: los nuevos cambios primero llegan a Scala, y luego poco a poco se van implementando para los otros API's.

- Facilita el desarrollo: ayuda a entender como está desarrollado Spark, y por tanto, permite entender cómo se comportara el framework cuando usemos nuestro código.

La potencia de Spark se debe principalmente a dos características importantes, una de ellas es que para trabajar intenta cargar los datos en memoria (intentará cargar en memoria todos los datos que necesite siempre que la capacidad de RAM del cluster lo permita), lo cual hace que en procesos iterativos como suelen ser los algoritmos de aprendizaje automático, mejore mucho el rendimiento respecto a otros frameworks como Mahout que deben ir constantemente a disco a buscar los datos.

Otra de las características principales de Spark, es que éste hace uso de un grafo dirigido acíclico (conocido por sus siglas en ingles como DAG), y usa un modo de ejecución perezoso, de forma que cuando le pedimos que ejecute una orden, espera hasta que no le queda más remedio que llevar a cabo una acción y devolver algún resultado. De esta forma, Spark es capaz de usar su motor de ejecución para programar todas las transformaciones que se han solicitado hasta ese momento, y ejecutarlas de la forma más eficiente posible, usando para ello un optimizador conocido como Catalyst, que es capaz de llevar a cabo optimizaciones lógicas y crear un plan de ejecución a partir de todas las operaciones que hemos solicitado. Esto hace que Spark sea también más eficiente que Hadoop, incluso cuando no se carga todo el dataset en memoria, y debe trabajar también con datos procedentes de disco.

En cuanto a las desventajas del módulo MLlib de "Spark", podemos citar principalmente dos:

1. Limitado número de algoritmos implementados actualmente: aunque es cierto que con cada nueva versión de Spark, continúan sacando nuevos algoritmos de aprendizaje automático, aún se echan en falta algunos algoritmos importantes como el k-vecinos para ayudarnos a la selección de variables, o la posibilidad de usar redes neuronales convolutivas o recurrentes.

2. <u>Framework de demasiado alto nivel con poco margen para configurar algoritmos:</u> la mayoría de algoritmos de aprendizaje automático implementados en Spark, permiten poco juego a la hora de llevar a cabo modificaciones de bajo nivel en ellos. Por ejemplo, el algoritmo k-means que trae Spark usa por defecto la distancia euclídea, y no permite usar otra métrica de distancia. Y en el caso de las redes neuronales, se utiliza por defecto la función de activación sigmoide para los nodos de las capas intermedias, y la función softmax a la salida, pero no podemos usar un rectificador lineal (ReLU) como función de activación, lo cual es un problema si queremos usar una red neuronal para un problema de regresión.

No obstante, estas dos desventajas citadas podrían ser solventadas en próximas versiones de Spark. Y en muchos casos, podemos recurrir a paquetes para Spark (no oficiales) ya creados por otros usuarios de la comunidad, puesto que al tratarse de software libre, el código de Apache Spark se encuentra a disposición de cualquier usuario,y la comunidad de Spark se ha mostrado muy activa en los últimos años.

Por otra parte, si queremos trabajar con algoritmos de machine learning en distribuido a un nivel más bajo de abstracción, una de las mejores opciones existentes hoy en día es TensorFlow, que es una librería de Machine Learning para python liberada por Google, que rápidamente esta ganando un gran número de adeptos, sobre todo para usarla en proyectos relacionados con Deep Learning.

A continuación vamos a ver aquellos algoritmos de clusterización que hemos seleccionado para este trabajo porque cumplen con los siguientes requisitos:

1) Son algoritmos para los cuales ya existen publicados artículos donde se detalla alguna implementación para su funcionamiento en distribuido.

2) Tienen un orden de complejidad lineal o logarítmico respecto al tamaño del dataset, lo cual hace que escalen bien cuando tenemos una gran cantidad de datos.

Una excepción al segundo punto serán los algoritmos de clustering espectral que tienen un orden de complejidad aproximadamente de $O(n^3)$, pero existen implementaciones en distribuido para dichos algoritmos, como por ejemplo el algoritmo PIC (Power Iteration Clustering), ya que el orden de complejidad no es tan elevado cuando contamos de antemano con la matriz de similitud.

K-Means ||

Ha pasado ya medio siglo desde que el algoritmo k-means fuera propuesto por primera vez, y a pesar de todo este tiempo aún sigue siendo uno de los algoritmos de clustering más utilizados hoy en día (de hecho se encuentra en el top 10 de algoritmos más usados en minería de datos actualmente). El algoritmo K-Means tiene un orden de complejidad del orden de O(log n). Sin embargo, el algoritmo k-means estándar adolece de algunos problema. Uno de ellos es que selecciona los centroides iniciales de forma aleatoria, seleccionando aleatoriamente “k” elementos del dataset. Y si por azar, hacemos una mala selección de los centroides iniciales, esto repercutirá en la calidad de los grupos creados. Para prevenir este problema podemos utilizar una versión de k-means denominada k-means++ que fue propuesta por Arthur y Vassilvitskii en 2007, y que permite una mejor elección de los centroides iniciales. El algoritmo k-means++ funciona de la siguiente forma:

1. Se escoge un centroide aleatoriamente de entre todos los puntos del dataset.

2. Para cada punto del dataset, calcula su distancia al centroide más cercano.

3. Se escoge un nuevo punto del dataset como nuevo centroide, pero utilizando una distribución de probabilidad ponderada donde cada punto del dataset tiene una probabilidad de ser elegido proporcional a la distancia a su centroide más cercano. Es decir, que cuanto más alejado este un punto de su centroide más cercano, mayor probabilidad tendrá de ser elegido como nuevo centroide.

4. Repetimos los pasos 2 y 3 hasta que se hayan seleccionado “k” centros.

5. Una vez se han seleccionado los centroides iniciales utilizando este método, se continua utilizando el algoritmo k-means estándar. Cada punto del dataset, se asigna al centroide más cercano de los actuales, y pasa a formar parte del grupo al que representa dicho centroide. Tras haber iterado todos los ejemplos del dataset, se recalcula el nuevo centroide para cada grupo. Esto se repite varias veces con todos los elementos del dataset, hasta que o bien ya no se producen cambios en los centroides con cada nueva iteración, o bien se alcanza el máximo número de iteraciones que hemos decidido utilizar.

Obviamente, el tener que realizar esta selección inicial de centroides consume mucho más tiempo que realizar una selección aleatoria de los centroides iniciales. Sin embargo, el algoritmo k-means++ converge más rápidamente que el algoritmo k-means estándar, por lo que muchas veces acaba siendo incluso más rápido.

Un problema que tiene el algoritmo k-means++ es que funciona de forma secuencial (la elección del siguiente centroide depende del conjunto actual de centroides en cada momento), y esto impide que se pueda paralelizar. Mientras que el algoritmo k-means estándar sí que es fácilmente paralelizable, ya que dado un conjunto de centroides, cada punto del dataset puede decidir de forma independiente cual es su centroide más cercano, puesto que los centroides no se van a ver modificados hasta que no se haya producido una primera iteración completa con todos los puntos del dataset. Por tanto, si queremos utilizar un mejor método de selección inicial de centroides que el algoritmo k-means estándar pero usando un método escalable, podemos recurrir a la versión paralelizable de k-means++, que es conocida como k-means|| y fue propuesta en *Bahmani, B., Moseley, B., Vattani, A., Kumar, R., Vassilvitskii, S. (2012)*.

El algoritmo para selección inicial de centroides k-means|| funciona de la siguiente manera:

1. Se elige aleatoriamente un centroide de entre todos los puntos del dataset y se añade a una variable *C* donde almacenaremos nuestros centroides.

2. Calculamos el coste inicial de la clusterización, que representaremos con el símbolo *ψ*
   En este caso, como sólo hemos seleccionado un centroide, calculamos la suma de la distancia euclídea de cada punto del dataset a dicho centroide.

3. Realizamos *log ψ* iteraciones de los pasos 4 y 5, donde *ψ* es el coste inicial de clusterización que obtuvimos en el paso anterior.

4. En cada iteración obtenemos una cantidad de nuevos puntos del dataset para añadir a nuestra lista de centroides *C*, que vendrá determinada por la variable de sobremuestreo *l*, por la distancia de cada punto a su centroide más cercano, y por el coste de clusterización actual que vendrá representado por *ΦX(C)*. Cada punto del dataset tiene una probabilidad mayor de ser seleccionado como nuevo centroide cuanto mayor es la distancia entre dicho punto y su centroide más cercano de los centroides actualmente almacenados en *C*. El valor de *l* debería estar en el orden del número de grupos *k* que utilicemos, en el artículo *Bahmani, B., Moseley, B., Vattani, A., Kumar, R., Vassilvitskii, S. (2012)* se hacen pruebas con varios valores de *l* (0.1k, 0.5k, 2k, o 10k), y Spark por defecto usa un valor de *l* de 2k.

   $$p_x = \frac{\ell \cdot d^2(x, \mathcal{C})}{\phi_X(\mathcal{C})}$$

   Para tener una visión más clara de cómo se produce este proceso de selección de nuevos centroides, podemos analizar como está implementado en la librería MLlib de Spark. Para cada punto del dataset se realiza la siguiente comprobación para ver si es seleccionado como nuevo centroide, donde *rand.nextDouble* es un número aleatorio decimal entre 0.0 y 1.0, *c* es la distancia euclídea de dicho punto a su centroide más cercano, *k* es el número de grupos, y *sumCosts* es el coste total de clusterización que hemos definido como *ΦX(C)*:

   *rand.nextDouble < 2.0 * c * k / sumCosts*

5. Añadimos los nuevos centroides calculados en el paso 4 a la lista de centroides *C*, y recalculamos el nuevo coste de clusterización *ΦX(C)* dada la nueva lista de centroides.

6. Tras finalizar las iteraciones, ahora tendremos un número de centroides mayor que *k*, por lo que el siguiente paso será asignar un peso a cada centroide en función del número de puntos del dataset que están más cerca de dicho centroide que de cualquier otro centroide.

7. Por último, hacemos una reclusterización de los centroides obtenidos en el punto 6 teniendo en cuenta sus pesos, para quedarnos finalmente con *k* centroides. En el caso de Spark, se utiliza directamente el algoritmo k-means++, que no es escalable, pero esto no debería ser un problema, puesto que la clusterización es de un número de puntos muy pequeño.

El algoritmo k-means es un algoritmo rápido, sencillo, y paralelizable, lo cual ha contribuido a su éxito. Sin embargo, k-means tiene varios problemas, no sólo a la hora de inicializar sus centroides, sino en otros aspectos, y conviene tener en cuenta dichos problemas a la hora de seleccionar el mejor algoritmo de clusterización que podemos usar para la tarea que tengamos que abordar.

El algoritmo k-means funciona correctamente cuando los grupos que tratamos de encontrar son grupos convexos, es decir, grupos en los cuales podemos unir cada par de puntos con una línea recta, y dicha linea sigue dentro de los límites del grupo. Sin embargo, cuando intentamos usar k-means para encontrar grupos que tienen formas no convexas, entonces empiezan los problemas, como veremos a continuación.

En la figura de abajo, el objeto verde sería no convexo, porque como podemos observar, si trazamos una linea entre dos puntos del objeto, dicha línea recorre un espacio que no pertenece a dicho objeto. Mientras que el objeto azul seria convexo, puesto que no existe ninguna selección de puntos que pertenezcan a dicho objeto, tales que al trazar una línea de un punto a otro, dicha línea salga irremediablemente fuera de dicho objeto.

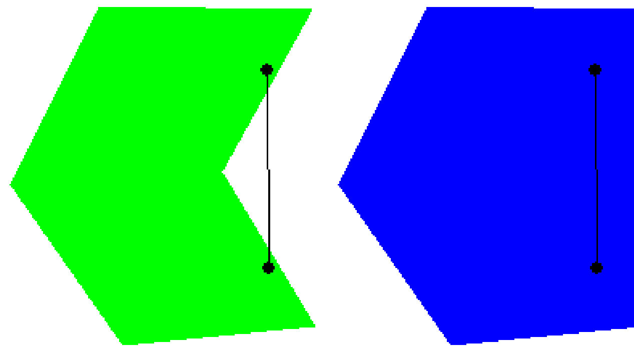


Figura 8: Objeto no convexo en verde y objeto convexo en azul

El algoritmo k-means no es capaz de agrupar correctamente un dataset donde los puntos de cada grupo pertenecen a objetos no convexos, como por ejemplo los grupos que aparecen en la figura 6, que quedarían agrupados por k-means como aparecen en la figura 7.

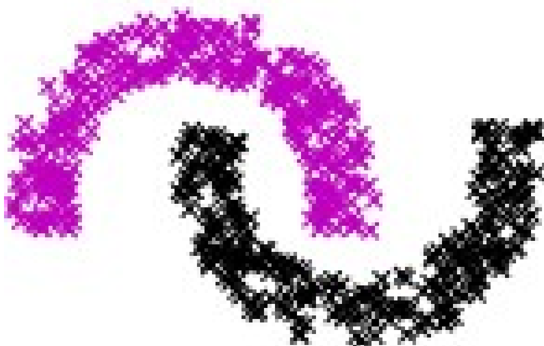


Figura 9: Grupos no convexos

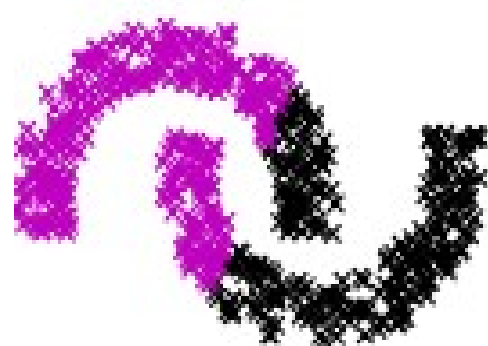


Figura 10: Clusterizacion de grupos no convexos por k-means

Por tanto, si tenemos un dataset que queremos clusterizar y pensamos que podríamos tener grupos no convexos, entonces deberíamos tratar de buscar otras alternativas a k-means.

El algoritmo k-means también tiene problemas para llevar a cabo su tarea cuando tenemos clusters de distinto tamaño o de diferente densidad. Por ejemplo, imaginemos que quisiéramos realizar una clusterización del dataset de la figura 8, compuesto por 3 grupos claramente diferenciados, pero que contiene un grupo muy grande (en rojo), y otros dos grupos mucho más pequeños (en azul y verde):

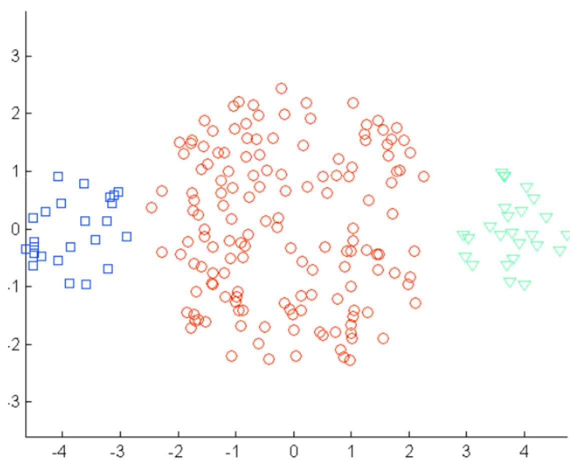


Figura 11: Dataset de 3 grupos de distintos tamaños

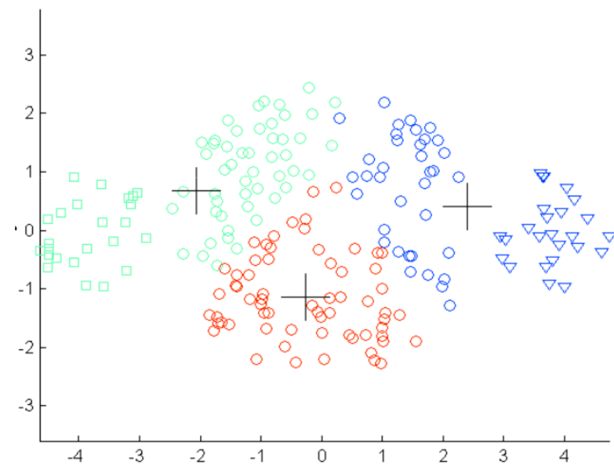


Figura 12: Agrupacion realizada por k-means

Al aplicar k-means con k=3, obtenemos la clusterización de la figura 9, con 3 grupos de un tamaño más o menos similar, lo cual no sería correcto. Esto se debe a que en un intento de minimizar la suma de cuadrados dentro de cada grupo, k-means da mayor prioridad a los grupos grandes, y no le importa que unos pocos ejemplos de otros grupos queden muy alejados de su centroide.

Por otra parte, si tenemos grupos de distinta densidad, como los de la figura 10, también podemos encontrarnos problemas a la hora de utilizar el algoritmo k-means, ya que en función de la suerte que tengamos a la hora de inicializar los centroides, podríamos obtener una agrupación similar a la de la figura 11.

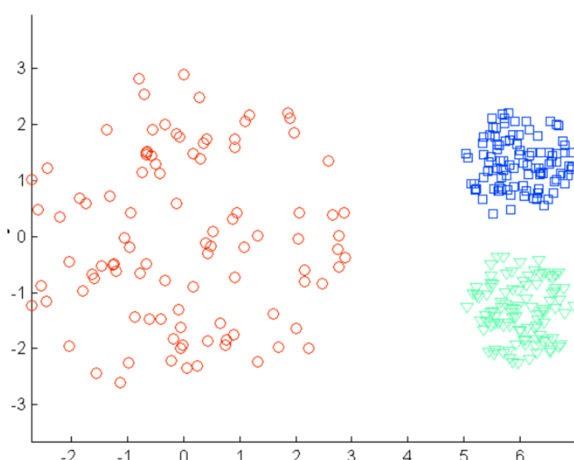


Figura 13: Dataset de 3 grupos de diferente densidad

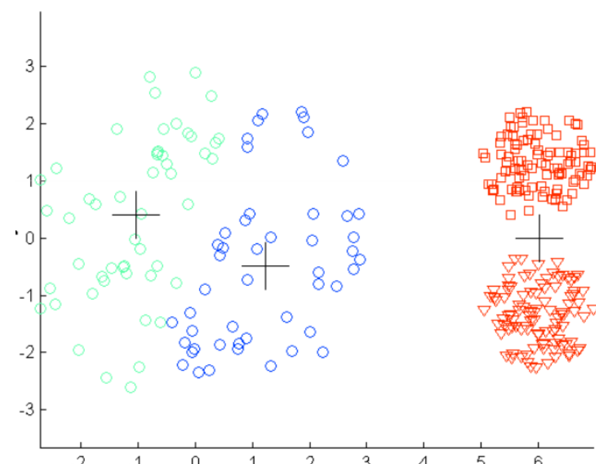


Figura 14: Clusterizacion con k-means

En resumen, podríamos decir que k-means presenta los siguientes problemas o limitaciones:

- La selección de centroides iniciales aleatoria que se realiza en el algoritmo k-means estándar puede acarrear muchos problemas, como por ejemplo, hacernos caer en un mínimo local, por lo que es aconsejable usar otras alternativas para la selección inicial de centroides como k-means++ o su versión paralelizable k-means|| si vamos a tratar con muchos datos.

- El algoritmo k-means no funciona correctamente cuando tenemos grupos no convexos. Por lo que si sabemos o intuimos que nuestro dataset podría contener grupos no convexos, entonces deberíamos plantearnos utilizar otro algoritmo de clusterización, como por ejemplo, un algoritmo de clustering espectral.

- Podemos tener problemas cuando nos encontramos con grupos de distinto tamaño o distinta densidad. Este problema puede ser más o menos grave en función de la selección inicial de centroides que hagamos, y podemos ayudarnos para ello por ejemplo de k-means++ (o k-means||). No obstante, otros algoritmos de clusterización como las mezclas gaussianas son capaces de tratar mejor estas diferencias respecto a la densidad y el tamaño de los distintos grupos.

- El algoritmo k-means es sensible al ruido y a outliers (valores atípicos), ya que pueden desplazar nuestro centroide, y dificultar la asignación de puntos a cada grupo. Una posible solución a este problema es utilizar un algoritmo conocido como k-medoids, que es similar a k-means, pero con la diferencia de que en vez de usar como centroide de un grupo, un punto no existente en el dataset, utiliza un medoide que es un punto del propio dataset. El principal problema del algoritmo k-medoids es que es mucho más costoso computacionalmente que el algoritmo k-means, y además tiene un orden de complejidad superior, haciendo difícil usar dicho algoritmo cuando tenemos datasets muy grandes. No obstante, actualmente la comunidad científica trabaja para encontrar alternativas que permitan crear una versión más escalable del algoritmo k-medoids, como por ejemplo podemos leer en el artículo Barioni, M., Razente, H., Traina, A. (2006). An efficient approach to scale up k-medoid based algorithms in large databases.

- Otro problema al que se enfrenta el algoritmo k-means es que los ejemplos de nuestro dataset siempre van a pertenecer única y exclusivamente a un único grupo, mientras que en otros algoritmos como mezclas gaussianas podemos tener un ejemplo que tenga un 51% de probabilidades de pertenecer a un grupo, pero también un 49% de probabilidades de pertenecer a otro grupo, y esta información puede ser muy valiosa, por ejemplo, para dar un tratamiento especifico a dichos ejemplos, o simplemente para que tengamos en cuenta que hay ciertos ejemplos cuyo grupo de pertenencia es dudoso.

- Y por último, tenemos una limitación respecto a la forma de los grupos, ya que k-means podría ser visto como un caso especial de mezclas gaussianas, donde todas las dimensiones del dataset tienen la misma varianza, lo que hace que la forma de los grupos producidos por k-means sea esférica.

Bisecting K-Means

El algoritmo bisecting k-means es un algoritmo de clustering jerárquico. Los grupos se van formando poco a poco, dividiendo en cada momento, uno de los clusters que tenemos en dos nuevos clusters utilizando para ello el algoritmo k-means estándar (o su versión distribuida, k-means ||). Es decir, que se trata de un algoritmo de clustering jerárquico divisivo, de arriba a abajo. Los algoritmos jerárquicos divisivos, que van de arriba hacia abajo, pueden computarse en paralelo sin problemas, ya que tras decidir qué grupo será el siguiente en dividirse en dos, dicha clusterización en dos nuevos grupos puede hacerse utilizando un algoritmo de clusterización escalable (como k-means ||). Mientras que en el caso de un algoritmo jerárquico aglomerativo que va de abajo hacia arriba es mucho más complicado hacer que dicho algoritmo trabaje de forma distribuida, puesto que necesitamos recalcular continuamente los centroides por cada unión de cada nuevo elemento a un grupo, y no podemos realizar ningún nuevo calculo hasta que esto ocurre.

El algoritmo bisecting k-means funciona de la siguiente manera:

1. Empieza con todos los elementos de entrada formando un sólo grupo, y se divide dicho grupo en dos nuevos grupos usando el algoritmo k-means estándar

2. A continuación se selecciona un grupo de los existentes (que supere el tamaño mínimo establecido para permitir continuar dividiendo dicho cluster) de acuerdo a algún tipo de medida de evaluación (por ejemplo se podría seleccionar el grupo más grande, o aquel cuya suma de distancias intragrupo es mayor), y se divide dicho grupo en dos nuevos grupos usando el algoritmo k-means estándar.

3. Si ya hemos alcanzado el número de grupos deseado, podemos terminar la ejecución. De lo contrario, volvemos al punto 2, para volver a dividir alguno de los grupos existentes en dos nuevos grupos.

El algoritmo bisecting k-means, a parte del parámetro “k” que indica el número de clusters que queremos obtener, también requiere que le indiquemos cuál queremos que sea el tamaño mínimo de un cluster para permitirle que pueda seguir dividiéndose en nuevos clusters.

El algoritmo bisecting k-means suele producir resultados algo mejores que el algoritmo k-means estándar, al menos en algunos campos como el de la clusterización de documentos según recoge el artículo Steinbach, M., Karypis, G., Kumar, V. (2000). Sin embargo, este algoritmo tiene un orden de complejidad de O(n), superior al orden de complejidad del algoritmo k-means que sería O(log n), lo que hace que la ejecución del algoritmo k-means sea más rápida que la del algoritmo divisivo bisecting-kmeans, y además escale peor según vamos añadiendo más ejemplos a nuestro dataset.

Mezclas gaussianas

Una distribución gaussiana es una distribución de probabilidad, y se define por una media y una desviación típica. Imaginemos que tenemos un dataset de dos dimensiones, y la probabilidad de pertenencia a un grupo concreto viene dada por la distribución gaussiana de la figura 12.

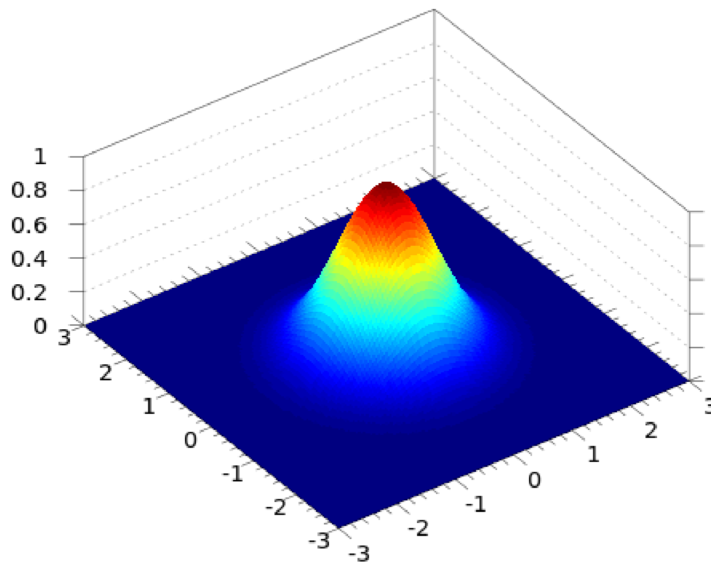


Figura 15: Distribución gaussiana con dos dimensiones

En tal caso, si un ejemplo de nuestro dataset fuera (0, 0), entonces la probabilidad de pertenencia a este grupo sería muy alta. Mientras que otro ejemplo que fuera (0, -1), tendría una probabilidad de pertenencia al grupo más baja. Y un ejemplo que fuera (3,3) tendría una probabilidad muy baja de pertenecer a este grupo.

En el modelo de mezclas de gaussianas, tendremos tantas distribuciones gaussianas como grupos, es decir, “k” distribuciones gaussianas, y además tendremos un peso o una probabilidad a priori de pertenencia a cada grupo, que vendrá dada por la proporción de elementos que han sido asignados a cada gaussiana.

El algoritmo de mezclas gaussianas es más potente que k-means. De hecho, k-means se puede considerar un caso particular de mezclas gaussianas donde todas las variables tienen la misma covarianza. En k-means, cada grupo quedaba definido por un centroide (que básicamente era la media entre todos los ejemplos que pertenecian al grupo). Sin embargo, en el modelo de mezclas gaussianas, cada grupo va a quedar definido por una gaussiana (con una media y una matriz de covarianzas), y también por un peso. La suma de todos los pesos debe ser igual a 1, y representan las probabilidades a priori de que un ejemplo del dataset pertenezca a un determinado grupo. Esto nos da una mayor flexibilidad a la hora de definir la forma de los grupos (que ya no estará limitada a una forma esférica), y también a la hora de tratar con grupos de diferente tamaño.

Otra de las ventajas de un modelo de mezclas de gaussianas respecto a k-means es que mientras que con k-means obtenemos una asignación “dura” de un ejemplo a un grupo (un individuo pertenece única y exclusivamente a un grupo); en el caso de la mezcla de gaussianas, podemos obtener una asignación “blanda” de un ejemplo del dataset a un grupo determinado, de forma que tengamos una probabilidad de pertenencia de dicho ejemplo a cada uno de los distintos grupos que hemos formado. Esto puede ser muy útil, pues refleja mejor la situación en que se encuentran aquellos ejemplos que están en la frontera de varios grupos, y que podrían ser outliers o datos anómalos.

Por ejemplo, los dos puntos que aparecen cerca de la línea de puntos en la figura siguiente, se encuentran en la frontera de pertenencia de los dos grupos, y su pertenencia a un grupo u otro queda definida por un margen muy pequeño. Sin embargo, al usar k-means, el resultado que obtendríamos sería que cada uno de esos puntos pertenece al grupo asignado con la misma probabilidad que otros puntos del grupo más cercanos al centroide, y que parece más claro que sí que pertenecen con seguridad a dichos grupos.

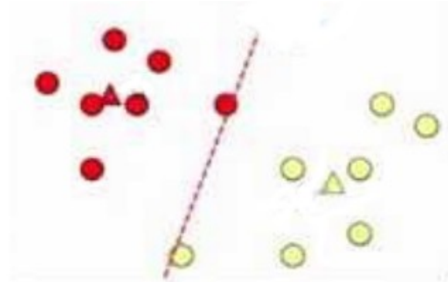


Figura 16: Puntos cercanos a la frontera

Si usáramos un algoritmo de mezclas gaussianas, en el cual podemos obtener asignaciones “blandas” de los objetos a cada grupo, al ver la probabilidad de pertenencia a cada grupo de esos puntos que se encuentran en la frontera, quizás obtendríamos que el punto rojo cercano a la línea de puntos pertenece al grupo rojo, pero con una probabilidad del 55%, ya que también tiene un 45% de probabilidades de pertenecer al grupo blanco.

El algoritmo de mezcla de gaussianas es un algoritmo iterativo que basa su funcionamiento en el algoritmo de EM (esperanza-maximización) para ir adaptándose poco a poco a los datos. El algoritmo EM, primero estima las variables latentes (el grupo al que pertenece cada observación) dados los parámetros actuales de las gaussianas y los ejemplos del dataset (fase “E”). A continuación, estima de nuevo los parámetros de cada grupo, a partir de los datos observados y los grupos que les hemos asignado en el paso anterior (fase “M”).

A continuación vamos a ver paso a paso cómo funciona a grandes rasgos el algoritmo de mezcla de gaussianas:

1. Primero, inicializamos los grupos creando “k” gaussianas cuyos parámetros elegimos de forma aleatoria, y cuyos pesos o probabilidades a priori w_j (proporción de elementos del dataset que pertenecen a dicho grupo) se establecen inicialmente de forma equitativa, puesto que no sabemos que proporción de elementos tendrá cada grupo. A continuación calculamos la función de verosimilitud para los parámetros iniciales usando la siguiente fórmula, donde “n” es el número de elementos del dataset, “k” es el número de grupos, el superíndice (0) indica que es la iteración inicial, y_i representa cada uno de los puntos del dataset, μ_j representa la media de la gaussiana j, y Σ_j representa la covarianza de la gaussiana j:

$$L^{(0)} = \frac{1}{n} \sum_{i=1}^{n} \log \left(\sum_{j=1}^{k} w_j^{(0)} \phi(y_i \mid \mu_j^{(0)}, \Sigma_j^{(0)}) \right)$$

2. A continuación, para cada ejemplo “i” del dataset, calculamos su probabilidad de pertenencia a cada uno de los grupos “j” para la iteración actual “m”, usando para ello la fórmula que veremos a continuación, donde tendremos en cuenta los parámetros de las gaussianas (la media y la covarianza), pero también el peso w_j asociado a cada grupo “j” que vendrá dado por la proporción de elementos del dataset que pertenecen a cada grupo, y sería algo así como una probabilidad a priori. Este paso se corresponde con el paso “E” del algoritmo de esperanza-maximización que se emplea para el algoritmo de mezclas gaussianas.

$$\gamma_{ij}^{(m)} = \frac{w_j^{(m)} \phi(y_i \mid \mu_j^{(m)}, \Sigma_j^{(m)})}{\sum_{l=1}^{k} w_l^{(m)} \phi(y_i \mid \mu_l^{(m)}, \Sigma_l^{(m)})}, \; i = 1, \ldots, n, \; j = 1, \ldots, k$$

3. Una vez que hemos asignado todos los ejemplos del dataset a uno de los grupos, recalculamos los nuevos parámetros para cada grupo. Este paso se correspondería con la fase “M” del algoritmo de esperanza-maximización, y utilizaríamos las siguientes fórmulas para calcular los nuevos parámetros:

$$w_j^{(m+1)} = \frac{n_j^{(m)}}{n}, \; j = 1, \ldots, k,$$

$$\mu_j^{(m+1)} = \frac{1}{n_j^{(m)}} \sum_{i=1}^{n} \gamma_{ij}^{(m)} y_i, \; j = 1, \ldots, k,$$

$$\Sigma_j^{(m+1)} = \frac{1}{n_j^{(m)}} \sum_{i=1}^{n} \gamma_{ij}^{(m)} \left(y_i - \mu_j^{(m+1)} \right) \left(y_i - \mu_j^{(m+1)} \right)^T, \; j = 1, \ldots, k$$

La primera fórmula, se refiere al cálculo de los pesos o probabilidades a priori para cada grupo “j” en la siguiente iteración “m+1”, y se calcula dividiendo el número de elementos que hay en la iteración actual “m” para el grupo “j” entre el número total de elementos “n” del dataset. Mientras que las otras dos fórmulas se emplean para el cálculo de la media y la covarianza de cada grupo o gaussiana “j” en la próxima iteración “m+1”.

4. Por último, realizamos una comprobación para ver si hemos alcanzado el número máximo de iteraciones que hemos definido, o si hemos alcanzado la convergencia, en cuyo caso no habría que seguir realizando más iteraciones. Sin embargo, si no se ha cumplido en esta iteración ninguno de los dos criterios de parada, entonces volveríamos al paso 2 para llevar a cabo una nueva iteración. El criterio de convergencia se define en función de los cambios producidos en la probabilidad de pertenencia de los puntos del dataset a los distintos grupos desde la última iteración a la actual, y para calcular dicho criterio, primero tenemos que volver a calcular la función de verosimilitud para la iteración “m+1”, usando la siguiente fórmula:

$$L^{(m+1)} = \frac{1}{n} \sum_{i=1}^{n} \log \left(\sum_{j=1}^{k} w_j^{(m+1)} \phi(y_i \,|\, \mu_j^{(m+1)}, \Sigma_j^{(m+1)}) \right)$$

Y usando el resultado de la función de verosimilitud obtenida en la iteración anterior, y el nuevo resultado obtenido en esta iteración, calculamos la diferencia entre ambas mediante la siguiente fórmula:

$$|L^{(m+1)} - L^{(m)}| > \delta$$

En caso que la diferencia sea mayor que el umbral establecido para la convergencia, este criterio de parada no se habría cumplido, y si tampoco hemos llegado al número máximo de iteraciones, deberíamos volver a realizar una nueva iteración. De lo contrario, habríamos alcanzado la convergencia, y daríamos por finalizado el algoritmo.

Algunos de los problemas o limitaciones que tiene el utilizar un modelo de mezclas de gaussianas para la clusterización de un dataset son las siguientes:

- El modelo parte de la premisa de que todas las variables o dimensiones de nuestro dataset siguen una distribución normal o gaussiana, lo cual no siempre tiene porque ser así. Y por tanto, puede haber distribuciones que no queden representadas de forma demasiado precisa cuando usamos una gaussiana.

- Comparado con k-means, el algoritmo de mezclas gaussianas es mucho más costoso computacionalmente, y el tiempo de ejecución dado un dataset de un tamaño específico es superior. Además, cuanto más aumentamos el tamaño de dicho dataset, el tiempo de ejecución del algoritmo de mezclas gaussianas se incrementa en la misma medida, ya que tiene un orden de complejidad lineal O(n), mientras que k-means tiene un orden de complejidad logarítmico O(log n).

- El algoritmo de mezclas gaussianas, al igual que sucedía con k-means tiene problemas para tratar con grupos no convexos. Por lo que si pensamos que nuestros grupos podrían tener forma no convexa, entonces deberíamos decantarnos por utilizar algún algoritmo de clustering espectral.

- El algoritmo de mezclas gaussianas no escala bien cuando se aumenta el número de dimensiones, por lo tanto, si vamos a utilizar un dataset de gran tamaño con un elevado número de variables, es posible que usar mezclas gaussianas no sea la mejor opción.

Clustering espectral

Los algoritmos de clustering espectral están basados en el uso de grafos no dirigidos, donde los puntos del dataset aparecen representados como vértices en dicho grafo, y las lineas que unen dichos vértices tienen un peso *W* que representa la similitud entre el par de vértices que unen. Los algoritmos de clustering espectral tratan la tarea de clusterización como un problema de particionado de un grafo sin presuponer nada acerca de la forma de los grupos.

A diferencia de los algoritmos de clusterización vistos hasta ahora (k-means, bisecting-k-means, mezclas gaussianas), donde necesitábamos que los grupos fueran compactos y convexos, en el caso de los algoritmos de clustering espectral lo que pasa a ser más relevante es la conectividad existente entre los distintos puntos del dataset. Podemos ver la diferencia entre grupos compactos (figura 17) y grupos con alta conectividad (figura 18) en las siguientes figuras:

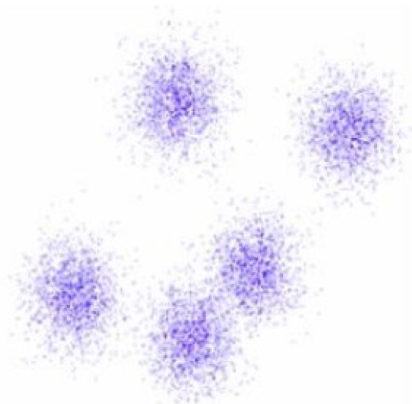


Figura 17: Grupos compactos

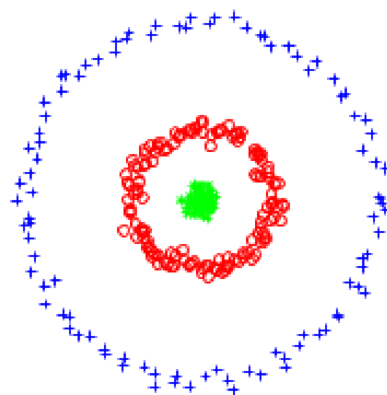


Figura 18: Grupos con alta conectividad

El funcionamiento básico de los algoritmos de clustering espectral es el siguiente:

1. Definir una matriz de similitud *A* a partir de nuestro dataset, donde cada elemento de la matriz A_{ij} representa la afinidad entre el punto *i* y el punto *j*. Para medir la afinidad entre dos puntos podemos emplear por ejemplo la medida de similitud basada en kernel gaussiano que viene dada por la siguiente fórmula, donde x_i representa un punto del dataset y x_j otro punto distinto del dataset, y σ^2 se refiere a la varianza de nuestro dataset:

$$s(i,j) = \exp\left(-\frac{\|x_i - x_j\|}{2\sigma^2}\right)$$

2. Construir una matriz laplaciana (que se utiliza para crear la representación de un grafo en forma de matriz) a partir de la matriz de similitud *A*.

3. Resolver un problema de valores propios, y seleccionar los *k* vectores propios correspondientes a los *k* valores propios más pequeños (pero distintos de cero), para de esta forma definir un subespacio de *k* dimensiones.

4. A partir del subespacio creado en el apartado anterior utilizar una técnica de clusterización, como por ejemplo k-means, para obtener finalmente los grupos.

Uno de los principales problemas que presentan los algoritmos de clustering espectral es que son muy costosos computacionalmente, ya que cada vez que añadimos un nuevo punto al dataset, debemos calcular la similitud entre dicho punto y el resto de puntos del dataset. Esto hace que los algoritmos de clustering espectral no sean muy escalables, y por tanto resulten poco viables cuando tenemos datasets que son demasiado grandes. De hecho, el orden de complejidad de los algoritmos de clustering espectral son del orden de $O(n^3)$ tal como aparece reflejado en el artículo *Yan, D., Huang, L., Jordan, M. (2009)*. Hay que tener en cuenta que simplemente preparar los datos para poder usar un algoritmo de clustering espectral, ya requiere en la mayoría de los casos que creemos una matriz de afinidad donde aparezca representada la similitud entre cada punto del dataset y el resto de puntos del dataset, lo cual hace que dicha preparación de los datos sea muy costosa computacionalmente, y además muy costosa en también en términos de memoria necesitada para almacenar dicha matriz en la memoria del ordenador.

Por otra parte, los algoritmos de clustering espectral son muy sensibles a la elección de parámetros, por tanto, muchas veces requieren un cuidadoso proceso de selección de parámetros, que en la mayoría de los casos debe realizarse mediante prueba y error, como por ejemplo el punto de corte a partir del cual dividimos los grafos en distintos grupos.

No profundizare más en los algoritmos de clustering espectral, puesto que si bien pueden resultar útiles cuando tenemos grupos donde la conectividad entre los puntos es más importante que la compactación de los grupos que forman dichos puntos, estos algoritmos no pueden utilizarse para datasets de miles de millones de objetos como el que pretendemos utilizar, al no ser que tengamos de antemano una matriz de similitud ya creada, que es uno de los procesos más costosos que requieren este tipo de algoritmos. No obstante, dado que el clustering espectral es un tipo de clustering con unas propiedades claramente diferenciadoras respecto a otros tipos de algoritmos de clusterización, y que la librería MLlib de Apache Spark incluye una implementación paralelizable del algoritmo de clustering espectral Power Iteration Clustering (PIC), he decidido incluir en la memoria este pequeño apartado sobre este tipo de clusterización.

2.2.4. Evaluación e interpretación

Una vez que hemos generado un modelo a partir de un conjunto de datos y de un algoritmo de minería de datos llega la hora de evaluar dicho modelo para ver si realmente es capaz de cumplir la tarea para la cual fue creado. Existen multitud de métricas que pueden utilizarse para evaluar la capacidad de un modelo dado, y además dichas métricas dependen del tipo de tarea de minería de datos que estemos llevando a cabo, ya que no usaremos la misma medida de evaluación para una tarea de clasificación, que para una tarea de regresión, o de clusterización.

En cuanto a la interpretación, debemos tener en cuenta que hay algunos modelos de minería de datos que son más fácilmente interpretables que otros, como por ejemplo los arboles de decisión que básicamente acaban generando reglas del tipo:

Sí feature-1 >= 350.0 y feature-2 = 'rojo' y feature-3 < 155.0 → clase = 'Ferrari'

Mientras que otros modelos, como los generados por redes neuronales artificiales, tienen la forma de una matriz de pesos, y una topología de red, lo cual no suele ser comprensible o interpretable para un humano.

En cuanto a la evaluación de un modelo de clusterización debemos tener en cuenta con que no contamos con ninguna etiqueta que nos indique si las agrupaciones que hemos creado son correctas o no, lo cual dificulta bastante la evaluación de este tipo de modelos. Existen dos tipos de validación que podemos emplear para hacernos una idea de la calidad de la agrupación realizada por nuestro algoritmo de clusterización:

- Validación interna: este tipo de validación se lleva a cabo usando los datos de entrada y las agrupaciones obtenidas, sin ningún otro tipo de información adicional. Básicamente, lo que intentamos conseguir son grupos que tengan miembros muy similares entre sí, y que a su vez sean lo más diferentes posibles a los miembros de otros grupos. Existen distintos índices para obtener una medida de la calidad de los grupos creados: índice de Hartigan, índice de Ball y Hall, índice Silhouette, índice de Ratkowsky, índice de Krzanowsky y Lai, etc.

- Validación externa: para llevar a cabo este tipo de validación necesitamos saber que una serie de elementos del dataset de entrada siempre deben ir juntos en los grupos que formemos. De esta forma, usaremos dicha información para asegurarnos que "al menos" esos elementos forman parte del mismo grupo, y no han acabado en grupos separados. Lo mismo puede decirse si tenemos el caso contrario, sabemos que hay ciertos elementos que nunca pueden ir juntos, por lo que podemos evaluar si en nuestros grupos creados esos elementos han acabado en distintos grupos o no. Un índice que nos puede ayudar a obtener una medida de la calidad de los grupos creados usando validación externa es el índice de Rand, que se basa en el número de verdaderos positivos, falsos positivos, verdaderos negativos, y falsos negativos, para obtener un índice de 0 a 1, donde 1 sería la mejor puntuación, y significaría que todos los elementos que podemos validar están en su grupo correcto.

Existen implementaciones de índices de evaluación interna y externa de clusters en la mayoría de lenguajes de programación. Pero en nuestro caso, necesitaríamos que dichos índices se encuentren implementados en un framework de computación distribuida, ya que usaremos dichos índices para evaluar grupos formados a partir de datasets de varios millones de elementos.

En Apache Mahout, existen varios índices ya creados, como son el índice de Davies y Bouldin (DB Index), el índice de Dunn, y el índice de Rand. Sin embargo, en Apache Spark actualmente no existe aún ningún índice de clusterización implementado. Por lo tanto, he decidido implementar algunos de los índices que son más escalables, a partir de los artículos que dieron lugar a cada uno de estos índices, y usando las fórmulas correspondientes. En concreto, he creado una versión escalable de los siguientes índices de evaluación interna en Spark (usando el API de Scala):

- Ball and Hall Index (Ball Index) - 1965
- Calinski and Harabasz Index (CH Index) - 1974
- Hartigan Index - 1975
- Ratkowsky and Lance Index (Ratkowsky Index) - 1978
- Davies and Boulding Index (DB Index) - 1979
- Krzanowski and Lai Index (KL Index) - 1988

Además, también he implementado en Spark (usando el API de Scala) uno de los índices de evaluación externa más utilizados actualmente:

- Rand Index - 1971

Para las pruebas de validación de los índices que he implementado, he utilizado el dataset de Iris que fue introducido por Fisher en su artículo "*Fisher, RA. (1936). The use of multiple measurements in taxonomic problems*", y está compuesto por 150 instancias, 50 pertenecientes a cada una de tres especies de flores distintas: Iris setosa, Iris virginica, y Iris versicolor. Y cada ejemplo está compuesto de 4 variables, la longitud y el ancho del sépalo, y la longitud y el ancho del pétalo.

La implementación que he realizado de estos índices de validación en Spark, y la interfaz que he desarrollado para obtener el mejor *k* de un dataset dadas una secuencia de *k*, una lista de los índices a emplear, y el dataset a emplear, pueden encontrarse en mi repositorio de GitHub cuya ruta es la siguiente:

https://github.com/DanielTizon/ClusteringMetrics

Además, dicho proyecto ha sido incluido como un paquete de Spark, y se encuentra disponible en la siguiente ruta, bajo la licencia Apache 2.0:

https://spark-packages.org/package/DanielTizon/ClusteringMetrics

Ball and Hall Index

Este índice fue propuesto en 1965 por Ball y Hall en *Ball, GH., Hall, DJ. (1965)*. Dicho índice esta basado en la distancia de cada objeto del dataset a sus respectivos centroides, y viene dado por la siguiente fórmula que aparece en *Charrad, M., Ghazzali N., Boiteau V., Niknafs A. (2014)*:

$$\mathrm{Ball} = \frac{W_q}{q}$$

Donde q es el número de grupos, y W_q es la suma de los cuadrados de los errores intra-grupos (diferencias entre cada punto perteneciente a un grupo determinado y su correspondiente centroide), y viene definida por la siguiente fórmula, donde c_k se refiere al centroide correspondiente al grupo k:

$$W_q = \sum_{k=1}^{q} \sum_{i \in C_k} (x_i - c_k)(x_i - c_k)^{\top}$$

Tras calcular los índices de Ball para todas las k que deseamos probar, nos quedaríamos con aquella "k" que produce el mayor salto en el índice de Ball desde la k anterior a la actual. Podemos ver un ejemplo usando el dataset de Iris, y calculando los índices de Ball para k = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}, con lo cual obtendríamos la siguiente gráfica:

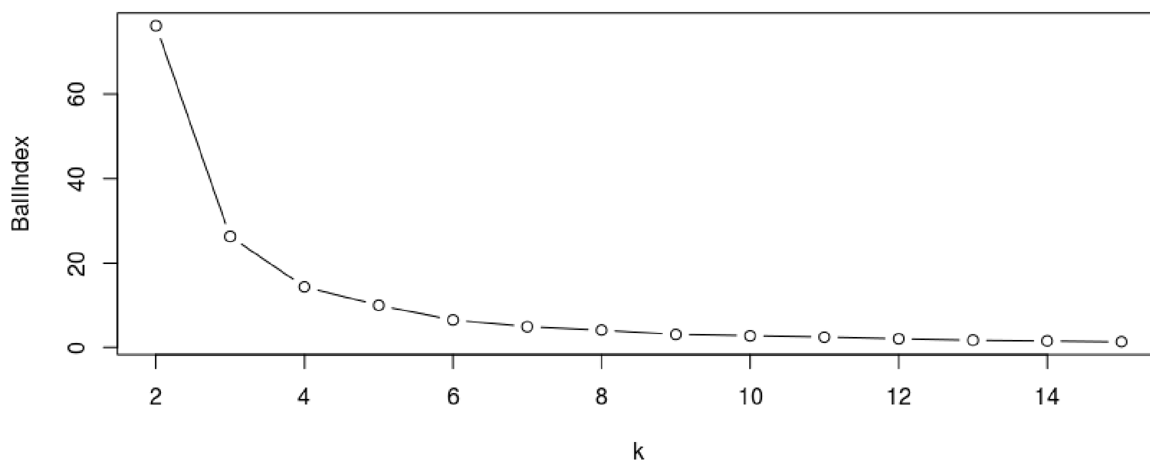


Figura 20: Gráfica del índice de Ball para dataset de Iris

Como podemos ver el mayor salto se produce cuando pasamos de k = 2 a k = 3, donde se produce exactamente un salto desde un BallIndex de 76,17 para k = 2 a un BallIndex de 26,28 para k = 3, lo cual da lugar a un salto de 49,89.

Calinski and Harabasz Index

Este índice fue propuesto en 1974 por Calinski y Harabasz en el artículo *Calinski, T., Harabasz, J. (1974)*. A diferencia del índice de Ball y Hall que sólo tenia en cuenta las distancias intra-grupo a través de la suma de los cuadrados de los errores de cada punto del dataset a su centroide (W_q), el índice de Calinski y Harabasz también tiene en cuenta la dispersión entre grupos que viene definida por B_q. La fórmula que usaremos que aparece en el artículo *Charrad, M., Ghazzali N., Boiteau V., Niknafs A. (2014)* es la siguiente donde *n* es el número de puntos del dataset, y *q* es el número de grupos:

$$\mathrm{CH}(q) = \frac{\mathrm{trace}(B_q)/(q-1)}{\mathrm{trace}(W_q)/(n-q)}$$

Y para calcular B_q usaremos la siguiente fórmula, donde *k* va iterando por todos los elementos del grupo *q*, y n_k es el número de puntos del grupo *k*, que se multiplica por el cuadrado de las diferencias entre cada centroide c_k y el centroide que obtendríamos si todo el dataset al completo (no sólo los elementos del grupo que estamos tratando) fuera un sólo grupo que sería igual a la media de todos los puntos del dataset $\overline{x}$:

$$B_q = \sum_{k=1}^{q} n_k \left(c_k - \overline{x}\right)\left(c_k - \overline{x}\right)^\top$$

Tras calcular los índices CH de cada una de las *k* que queremos probar, seleccionaremos aquella *k* para la cual obtengamos un índice de CH mayor.

Como podemos apreciar por la fórmula del índice de CH, este índice tiene en cuenta que los centroides están lo más separados posibles entre sí a través de B_q y a su vez que los puntos de un mismo grupo estén lo más cerca posible de su centroide a través de W_q . Además, también se tiene en cuenta el número de grupos *q* que elegimos, puesto que si *q* es un valor demasiado alto, y esto no mejora significativamente la calidad de los grupos, esto hará que obtengamos un peor resultado en el índice de CH.

Hartigan Index

Este índice fue propuesto en 1975 por Hartigan en el siguiente libro *"Hartigan, J. (1974). Clustering Algorithms"*. La fórmula que usaremos que aparece en el artículo *Charrad, M., Ghazzali N., Boiteau V., Niknafs A. (2014)* es la siguiente:

$$\text{Hartigan} = \left(\frac{\text{trace}(W_q)}{\text{trace}(W_{q+1})} - 1 \right) (n - q - 1)$$

Como podemos observar, al igual que pasaba con el índice de Ball y Hall, en el índice de Hartigan la suma de los cuadrados de los errores intra-grupos (W_q) tiene un papel muy relevante, y en este caso, además se tiene en cuenta la evolución producida en W_q al aumentar en uno el número de grupos (q).

Para elegir el número de grupos adecuado para nuestro dataset, Hartigan sugirió empezar con un sólo grupo (q = 1), e ir aumentando el número de grupos siempre que el índice de Hartigan para dicho número de grupos sea lo suficientemente grande. Para Hartigan, un valor del índice era suficientemente grande si era mayor que 10, en cuyo caso debíamos añadir un nuevo cluster, de lo contrario ya habíamos llegado al número de grupos optimo para nuestro dataset.

No obstante, se pueden utilizar otras heurísticas para seleccionar el mejor número de grupos empleando el índice de Hartigan. Por ejemplo, en el artículo *Charrad, M., Ghazzali N., Boiteau V., Niknafs A. (2014)* utilizan como mejor número de grupos, aquel q en el cual se produce un mayor salto entre el índice de Hartigan para el número de grupos anterior y el índice de Hartigan para el número de grupos actual.

Veamos un ejemplo de ambas heurísticas, usando el dataset de Iris, con el cual obtendríamos los siguientes índices de Hartigan:

Num. grupos	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hartigan Index	137,9	55,5	21,7	25,1	31,9	22,7	6,3	9,4	7,6	2,9	12,2	7,4	8,6	7,3

Por tanto, según el criterio sugerido por Hartigan, tendríamos que coger k = 7 como mejor número de grupos, puesto que el índice de Hartigan para k = 8 ya sería menor de 10, y para todo el resto de *k* menores de 7, todos son mayores de 10.

Mientras que siguiendo el criterio sugerido por el artículo *Charrad, M., Ghazzali N., Boiteau V., Niknafs A. (2014)* elegiríamos k = 3, ya que es donde se produce el mayor salto 137,9 – 55,5 = 82.4. A la hora de calcular el mejor número de grupos a partir del índice de Hartigan, he implementado esta heurística en el índice de Hartigan que he implementado para Apache Spark.

Ratkowsky and Lance Index

Este índice fue propuesto en 1978 por Ratkowsky y Lance en el artículo *Ratkowsky, DA., Lance, GN. (1978)*. La fórmula que usaremos que aparece en el artículo *Charrad, M., Ghazzali N., Boiteau V., Niknafs A. (2014)* es la siguiente, donde *p* se refiere al número de variables:

$$\text{Ratkowsky} = \frac{\overline{S}}{q^{1/2}} \qquad \overline{S}^2 = \frac{1}{p}\sum_{j=1}^{p}\frac{BGSS_j}{TSS_j}$$

El índice de Ratkowsky y Lance es muy diferente a los índices que hemos visto hasta ahora, ya que no tiene en cuenta la suma de los cuadrados de los errores intra-grupos (W_q) que usaban todos los índices vistos hasta ahora, sino que se basa en el ratio entre $BGSS_j$ y TSS_j, donde $BGSS_j$ es la suma de los cuadrados de las diferencias entre la media de la variable *j* para el dataset, y el valor de esa variable para cada centroide. Por su parte, TSS_j es la suma de los cuadrados de las diferencias entre la media de la variable *j* para el dataset, y cada uno de los puntos del dataset. Para cada variable *j* tenemos c_{kj} que representa el valor que tiene la variable *j* en el centroide del grupo *k*, y $\overline{x}_j$ que representa la media de la variable *j* en nuestro dataset.

$$BGSS_j = \sum_{k=1}^{q} n_k (c_{kj} - \overline{x}_j)^2 \qquad TSS_j = \sum_{i=1}^{n} (x_{ij} - \overline{x}_j)^2$$

Aquella *q* (número de grupos) para la cual obtenemos un índice de Ratkowsky de mayor valor indicaría el número óptimo de grupos.

Por ejemplo, usando el dataset de Iris obtendríamos los siguientes valores de los índices de Ratkowsky:

Num. grupos	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Ratkowsky Index	0,55	0.50	0,44	0,41	0,37	0,35	0,33	0,31	0,30	0,29	0,27	0,26	0,25	0,25

Por tanto, según el índice de Ratkowsky, el mejor número de grupos para el dataset de Iris sería k = 2 ya que tiene el mayor valor 0,55, y el siguiente mejor número de grupos habría sido k = 3, para el cual tenemos un índice de Ratkowsky de 0,50.

Davies and Boulding Index

Este índice fue propuesto en 1979 por Davies y Bouldin en el artículo *Davies, DL., Bouldin, DW. (1979)*. La fórmula que usaremos que aparece en el artículo *Charrad, M., Ghazzali N., Boiteau V., Niknafs A. (2014)* es la siguiente, donde *q* se refiere al número de grupos:

$$\mathrm{DB}(q) = \frac{1}{q}\sum_{k=1}^{q} \max_{k \neq l} \left(\frac{\delta_k + \delta_l}{d_{kl}} \right)$$

Con δ_k y δ_l refiriéndose a la dispersión de los individuos del grupo *k* y el grupo *l* respectivamente. Mientras que d_{kl} sería la distancia entre los centroides del grupo *k* y el grupo *l*. Lo que la fórmula indica con el máximo *k* distinto de *l* es que para cada grupo *k*, debemos buscar otro grupo *l* diferente de *k* que nos devuelva el máximo resultado en la división indicada en la fórmula. La distancia entre los centroides de dos grupos, y la dispersión de los individuos de un grupo *k*, vienen dados por las siguientes fórmulas, donde *p* es el número de variables del dataset, c_{kj} representa el valor que tiene la variable *j* en el centroide del grupo *k*, c_{lj} representa el valor que tiene la variable *j* en el centroide del grupo *l*, n_k es el número de elementos en el cluster *k*, y por último *u* y *v* son dos variables cuyo valor debemos definir nosotros mismos en función de las características de nuestro dataset. En el índice que he implementado en Spark, he utilizado *"u = 2"* y *"v = 2"*:

$$d_{kl} = \sqrt[v]{\sum_{j=1}^{p} |c_{kj} - c_{lj}|^v} \qquad \delta_k = \sqrt[u]{\frac{1}{n_k} \sum_{i \in C_k} \sum_{j=1}^{p} |x_{ij} - c_{kj}|^u}$$

El índice de Davies y Bouldin se basa en el ratio entre la dispersión de los individuos de cada grupo, y la distancia existente entre los centroides de cada grupo. Y el menor valor del índice de Davies y Bouldin nos indicaría cual es el mejor número de grupos para nuestro dataset.

Krzanowski and Lai Index

Este índice fue propuesto en 1988 por Krzanowski y Lai en el artículo *Krzanowski, WJ., Lai, YT. (1988)*. La fórmula que usaremos que aparece en el artículo *Charrad, M., Ghazzali N., Boiteau V., Niknafs A. (2014)* es la siguiente, donde q se refiere al número de grupos :

$$\mathrm{KL}(q) = \left| \frac{\mathrm{DIFF}_q}{\mathrm{DIFF}_{q+1}} \right|$$

Y para calcular $DIFF_q$ usaremos la siguiente fórmula, donde p es el número de variables, y W_q es la suma de los cuadrados de los errores intra-grupos (la distancia entre cada punto perteneciente a un grupo determinado y su correspondiente centroide):

$$\mathrm{DIFF}_q = (q-1)^{2/p}\,\mathrm{trace}\left(W_{q-1}\right) - q^{2/p}\mathrm{trace}\left(W_q\right)$$

El mayor valor del índice de KL para un determinado número de grupos indica cuál sería el mejor número de grupos para nuestro dataset según el índice de Krzanowski y Lai.

Rand Index

El índice de Rand es un método de evaluación externa, y fue propuesto en 1971 por William M. Rand en el artículo *Rand, WM. (1971)*. Para usar este índice, debemos tener en cuenta que nuestro problema debe cumplir los siguientes tres puntos:

- Cada punto debe pertenecer única y exclusivamente a un cluster
- Los clusters deben quedar definidos tanto por los puntos que forman parte de ellos, como por aquellos puntos que no forman parte del cluster.
- Todos los puntos tienen la misma importancia a la hora de generar los clusters.

Y la fórmula que he utilizado para calcular el índice de Rand es la siguiente, donde *TP* se refiere a aquellos elementos de nuestro dataset que deben ir en el mismo grupo, y que realmente han ido a parar al mismo grupo tras realizar la clusterización, *TN* son aquellos elementos del dataset que deben ir en diferentes grupos, y efectivamente han ido a grupos diferentes tras la clusterización, *FP* se refiere a aquellos elementos que tras la clusterización han ido a parar al mismo grupo y no deberían estar en el mismo grupo, y *FN* se refiere a elementos del dataset que tras la clusterización han acabado en grupos diferentes, y deberían haber estado en el mismo grupo:

$$Rand\ Index = \frac{TP+TN}{TP+FP+TN+FN}$$

Hay que tener en cuenta que para calcular el índice de Rand, necesitamos conocer información sobre qué puntos del dataset deberían ir en el mismo grupo y qué puntos del dataset no deben ir en el mismo grupo. No es necesario conocer esta información para todos los puntos del dataset, pero cuanta más información tengamos al respecto, más fiable será el índice de Rand. Para la implementación que he realizado del índice en Apache Spark, necesitaremos un archivo CSV que tenga la siguiente estructura:

GRUPO,ID

HERCULES-LYRA,1735-927-1

HERCULES-LYRA,4687-325-1

HERCULES-LYRA,183-626-1

AB DORADUS,2261-1518-1

AB DORADUS,5276-413-1

…

El primer campo del CSV llamado “GRUPO” es un identificador de grupo, que puede ser cualquier alfanumérico que decidamos usar para identificar a un determinado grupo. Mientras que el segundo campo del CSV llamado “ID” es el identificador de un objeto del dataset.

Capitulo 3: Caso de estudio

En este trabajo vamos a utilizar datos de la sonda Gaia sobre 2 millones de cuerpos celestes (TGAS), pero vamos a tener muy presente su escalabilidad y rendimiento, ya que el objetivo final será tratar más de 1.000 millones de objetos, que serán los que tengamos cuando Gaia termine su misión y dispongamos de todos los datos listos para ser procesados. En cuanto a los atributos, vamos a utilizar sólo aquellos relacionados con la astrometría, que serían la ascensión recta, la declinación, los movimientos propios en ascensión recta y declinación, y la paralaje, por lo que tendríamos un total de 5 atributos.

Veamos con un poco más de detalle las variables de nuestro dataset que utilizaremos inicialmente para llevar a cabo la clusterización:

- Ascensión recta (α): es el ángulo medido hacia el este a lo largo del ecuador celeste, desde el punto vernal hasta el meridiano celeste que pasa por el astro. La ascensión recta se mide de 0 a 24 horas. No obstante, para tratar con las mismas unidades, en nuestro dataset la ascensión recta ya vendrá transformada en grados, y por lo tanto el dominio de esta variable será de 0º a 360º.

- Declinación (δ): es el ángulo entre el ecuador celeste y la posición de un astro, medido a lo largo del meridiano celeste que pasa por el astro. Este ángulo varía entre 0º y 90º en el hemisferio celeste boreal, y entre 0º y -90º en el hemisferio austral. Por tanto, en el caso de la sonda Gaia que se encuentra en el espacio, el dominio de esta variable será de -90º a +90º.

- Movimiento propio en declinación: el movimiento propio mide los cambios observados en las posiciones aparentes de los cuerpos celestes. En nuestro dataset va a venir expresado en mili-segundos de arco por año (mas/yr), y normalmente tendrá valores muy pequeños. Respecto al movimiento propio en declinación, en nuestro dataset la media es de -5,25 mas/yr, el mínimo es de -3.159,34 mas/yr y el máximo 1.858,52 mas/yr.

- Movimiento propio en ascensión recta: esta variable se refiere al movimiento propio ya comentado en el apartado anterior, pero esta vez referido a la ascensión recta. Teniendo una media de -1,74 mas/yr, un mínimo de -3.259,36 mas/yr y un máximo de 2.890, 43 mas/yr.

- Paralaje: medir una paralaje consiste en determinar el ángulo que se forma al observar un objeto desde dos puntos de vista distintos. Este ángulo depende de la distancia que medie entre el objeto y el observador, de manera que cuanto más lejos esté el objeto, menor será el ángulo. Por tanto, la medida de ese ángulo permite conocer la distancia a la que se encuentra un objeto, y a mayor paralaje menor será la distancia a la que se encuentra un determinado objeto. En el dataset GAIA-DR1, la paralaje viene expresada en milisegundos de arco (mas), y el rango de esta variable va desde un mínimo de -24,82 milisegundos de arco hasta un máximo de 295,80 mas, siendo la media de 2,47 milisegundos de arco, y la desviación típica de 2,91 milisegundos de arco.

Para llevar a cabo los distintos experimentos, vamos a utilizar dos datasets, en el primero de ellos, usaremos las variables astrométricas originales que vienen en el dataset de TGAS, y lo etiquetaremos como “Dataset 1”. Y por otra parte, utilizaremos un segundo dataset, al cual etiquetaremos como “Dataset 2”, y el cual estará compuesto por las coordenadas espaciales *X*, *Y*, *Z*, y las velocidades transversales para la ascensión recta y la declinación. A continuación, vamos a ver las fórmulas que he utilizado para calcular las nuevas variables que utilizaremos en el “Dataset 2”. Para el cálculo de las coordenadas, utilizaremos en todos los casos, una variable de distancia que vendrá dada como la inversa de la paralaje:

$$distancia = \frac{1}{paralaje}$$

Ahora veamos qué formulas he utilizado para obtener cada una de las coordenadas:

$$X = distancia \cdot \cos(ascensionRecta) \cdot \cos(declinacion)$$

$$Y = distancia \cdot \sin(ascensionRecta) \cdot \cos(declinacion)$$

$$Z = distancia \cdot \sin(declinacion)$$

Y para obtener las velocidades transversales en km/s, he vuelto a utilizar la distancia como inversa de la paralaje, y he realizado una conversión de los movimientos propios que se encuentran en milisegundos en el dataset a segundos. A continuación he utilizado la siguiente fórmula:

$$Velocidad\,Transversal = \frac{Movimiento\,Propio(ms)}{1000} \cdot distancia \cdot 4.74$$

Con el segundo dataset tratamos de solucionar un problema que afecta por ejemplo a la ascensión recta (cuyo rango va de 0º a 360º), y que presenta el inconveniente de que un punto que se ubique en 0.1º y otro que se ubique en 359.9º se encontraran muy juntos el uno del otro en el espacio, y por tanto debería ser más probable que aparecieran relacionados en el mismo grupo, puesto que aunque la magnitud de sus variables indican lo contrario, realmente serían puntos muy próximos entre sí.

Para tratar de mejorar la clusterización de nuestros datasets, voy a realizar diferentes pruebas, como por ejemplo, llevar a cabo una estandarización de los datasets (restando la media y dividiendo por la desviación típica cada punto del dataset).

La normalización o estandarización es un procedimiento muy útil en una tarea de clusterización en la cual las variables que utilicemos no tengan la misma unidad de medida (como ocurre en nuestro caso de estudio), y más aún si su dominio es muy diferente, lo cual podría hacer que algunas variables con valores más grandes dominen la clusterización.

Por ejemplo, una estrella con un movimiento propio en ascensión recta de 2.500 mas/yr, va a hacer que la ascensión recta (cuyo dominio es de 0º a 360º), la declinación (cuyo dominio es de -90º a +90º), y la paralaje (cuyos mínimos y máximos en nuestro dataset van de -24,82 mas a 295,80 mas), tengan menos peso a la hora de agrupar dicha estrella en uno u otro grupo, y al final acabaremos con grupos que han sido creados basándose básicamente en los movimientos propios, que son las variables con unos valores más altos respecto al resto.

Para este proyecto vamos a utilizar Apache Spark (MLlib) debido a que, como vimos en el apartado anterior, actualmente es una de las mejores opciones a emplear si queremos aplicar algoritmos de aprendizaje automático usando una cantidad ingente de datos.

Además, debemos de tener en cuenta que para trabajar con Big Data, necesitamos de una infraestructura con varias máquinas a nuestra disposición para llevar a cabo las tareas necesarias, y en nuestro caso, vamos a correr nuestros algoritmos en la infraestructura facilitada por el DPAC (Data Processing and Analysis Consortium), consistente en un cluster de Apache Spark formado por 6 nodos con 16 cores y 64Gb de RAM cada uno, por lo que debemos ceñirnos a los algoritmos de clusterización actualmente implementados en Apache Spark para poder trabajar en nuestro caso de estudio.

En el caso de las métricas de evaluación interna y externa, vamos a utilizar unas métricas que he implementado en Apache Spark, basándome en una selección de aquellos índices de evaluación que tienen un orden de complejidad logarítmico o a lo sumo lineal, y que por tanto son capaces de escalar bien cuando crezca nuestro dataset.

Conviene resaltar que todos los algoritmos de clusterización y los índices de evaluación que emplearemos en este caso de estudio, utilizaran la distancia euclídea para el calculo de la distancia entre los distintos puntos del dataset. Esto se debe a que utilizaremos Apache Spark, y dicho framework actualmente (versión 2.1.0) sólo admite el uso de la distancia euclídea para las tareas de clusterización. Por otra parte, el uso de la distancia euclídea encaja bastante bien cuando empleemos el "Dataset 2", en el cual estamos utilizando las coordenadas X, Y, Z, ya que cada objeto de nuestro dataset representa un objeto celeste que se encuentra en el universo, y nuestro universo se puede aproximar a la geometría euclidiana.

Capitulo 4: Resultados

Antes de empezar a trabajar con nuestro dataset de TGAS, vamos a realizar una serie de comprobaciones para asegurarnos de que la elección tomada respecto a los algoritmos de clusterización que emplearemos es acertada y que escalan de forma razonable para un elevado número de datos. Así mismo, realizaremos las mismas pruebas para el caso de los índices de evaluación que he creado en Apache Spark.

4.1. Escalabilidad de los algoritmos de clusterización

A continuación vamos a ver como varían los tiempos de ejecución de varios algoritmos de clusterización que se encuentran en la librería Mllib de Apache Spark (K-means, bisecting-k-means, y mezclas gaussianas), en función del tamaño del dataset. Para ello emplearé el dataset de TGAS, que tiene 5 variables y 2.057.050 registros. Para obtener el número de registros requerido en cada momento, iré realizando un sobremuestreo sobre dicho dataset en cada ejecución.

Para la realización de esta prueba utilizare un cluster de Apache Spark del DPAC (Data Processing and Analysis Consortium), el cual se adquirió para hacer las pruebas de desarrollo del WP 973 de la unidad de coordinación 9 de DPAC, y consta de 6 nodos con 16 cores y 64Gb de RAM cada uno. Empezaremos con un tamaño del dataset igual a N = 2.057.050 (todo el dataset de TGAS) y poco a poco iremos incrementando el tamaño del dataset (realizando sobremuestreo) hasta llegar a 10N = 20.570.500 elementos, *k* será igual a 10 en todos los casos, y el número máximo de iteraciones será fijado a 20 en todos los casos.

Por otra parte, cabe resaltar que el dataset será cacheado antes de utilizar el algoritmo de clusterización, y sólo mediremos el tiempo utilizado para la creación del modelo, y no el tiempo necesario para la lectura y cacheado en memoria del dataset.

Algoritmo clustering \ Tamaño Dataset (tiempo en segundos)	**2.057.050 (N)**	**4.114.100 (2N)**	**10.285.250 (5N)**	**20.570.500 (10N)**
K-Means (k = 10 grupos)	14.46	14.82	16.69	24.94
Bisecting K-Means (k = 10 grupos)	21.40	31.34	53.44	90.61
Mezclas gaussianas (k = 10 grupos)	37.11	59.70	133.47	248.78

En la figura 19 podemos ver una gráfica comparativa de cada uno de los tres métodos de clusterización utilizados, representando en el eje de las X, el tamaño del dataset con N = 2.057.050 registros, 2N = 4.114.100 registros, …, 10N = 20.570.500 registros, y en el eje de las Y aparece el tiempo en segundos.

En la gráfica de la figura 19 se puede apreciar claramente como el algoritmo k-means es el más eficiente de los tres, aunque también es cierto que es el más simple de ellos, y su orden de complejidad se situaría en el orden logarítmico. El algoritmo bisecting-kmeans también tiene un orden de complejidad de orden logarítmico, aunque algo superior que el algoritmo estándar k-means. Por su parte, el algoritmo de mezclas gaussianas tiene un orden de complejidad aproximadamente de O(n), es decir, que escala linealmente con el tamaño del dataset, y aunque el tiempo de ejecución respecto a los otros algoritmos es superior, hay que tener en cuenta que su expresividad también es mayor, y nos permite obtener grupos con formas no sólo esféricas.

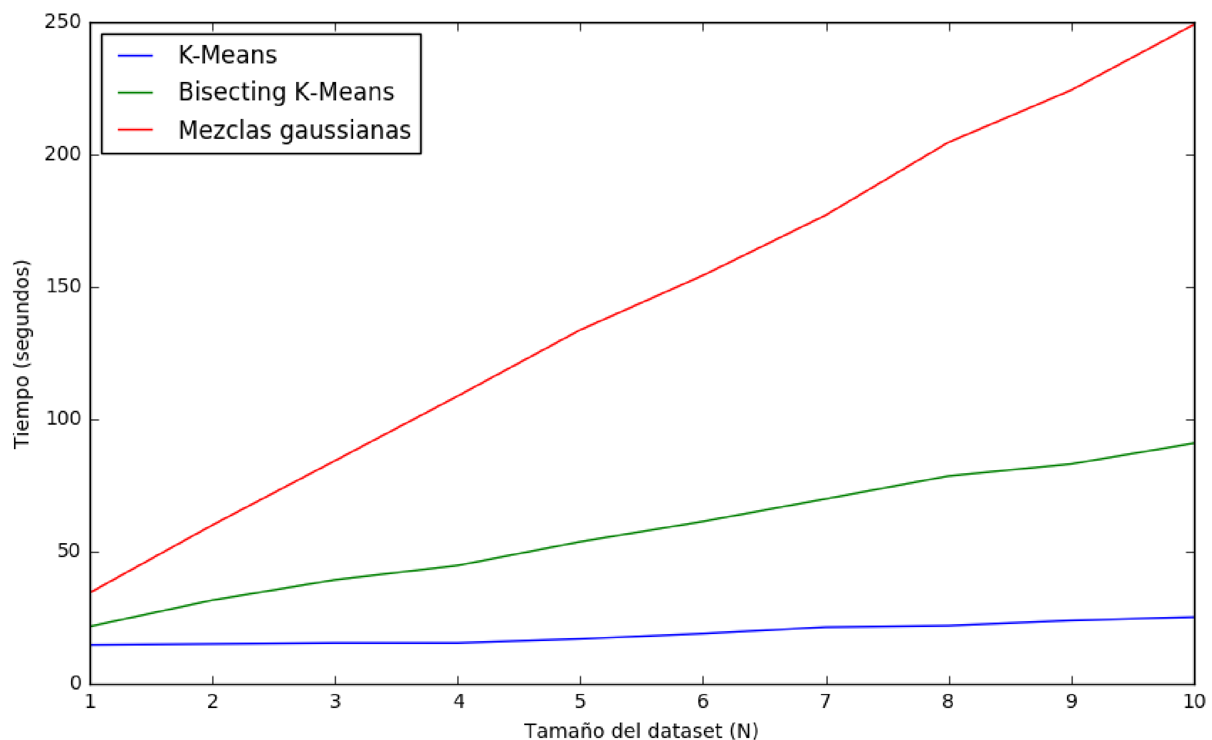


Figura 19: Gráfica comparativa de tiempos de ejecución de algoritmos de clusterización

Por último, vamos a ver como varían los tiempos de k-means, bisecting-kmeans, y mezclas gaussianas en función del número de grupos y del número de atributos que utilicemos, para hacernos una idea general de que podríamos esperar al incrementar dichas variables. Para ello volveremos a utilizar el dataset de TGAS (2.057.050 instancias) y el cluster del DPAC utilizado en la prueba anterior. El número de grupos que usaremos será 50, 100, y 500 grupos, mientras que el número de atributos será 3, 4, y 5. En las siguientes tablas podemos ver los resultados obtenidos por cada algoritmo:

K-MEANS

Nº variables \ Nº grupos (tiempo en segundos)	**50 grupos**	**100 grupos**	**500 grupos**
3	12.44	15.44	25.22
4	12.55	15.07	26.83
5	14.86	15.03	31.29

BISECTING K-MEANS

Nº variables \ Nº grupos (tiempo en segundos)	**50 grupos**	**100 grupos**	**500 grupos**
3	34.97	41.48	52.51
4	34.96	40.78	52.16
5	34.35	40.53	51.91

MEZCLAS GAUSSIANAS

Nº variables \ Nº grupos (tiempo en segundos)	**50 grupos**	**100 grupos**	**500 grupos**
3	95.94	177.27	824.39
4	108.42	208.08	989.58
5	132.44	251.24	1209.24

Como podemos observar en las tablas, los algoritmos k-means y bisecting k-means no se ven demasiado influenciados por el aumento del número de variables, mientras que en el caso de las mezclas gaussianas, sí que tiene un efecto mucho más visible, y de hecho esta es una de las limitaciones de las mezclas de gaussianas, que no escalan bien cuando tenemos muchas dimensiones.

4.2. Escalabilidad de los índices de validación

Ahora vamos a comprobar cómo escalan en tiempo cada uno de los índices de validación interna que he implementado en Apache Spark, para ello haremos uso del algoritmo de mezclas gaussianas implementado en la librería MLlib de Spark, y dichos modelos serán generados y cacheados previamente al calculo de los índices de validación, para de esta forma asegurarnos de que el tiempo de ejecución que estamos midiendo se refiere única y exclusivamente al tiempo empleado en el cálculo del índice.

También he implementado en Spark el índice de evaluación externa de Rand, pero dado que la velocidad de ejecución de dicho índice dependerá en gran parte de la evidencia que suministremos, y para nuestro caso de uso, tenemos muy poca evidencia, he decidido no incluir pruebas sobre tiempos de ejecución relativas a este índice de evaluación.

El dataset empleado será el dataset de TGAS (2.057.050), sobre el cual iré realizando un sobremuestreo con reemplazamiento para quedarme cada vez con una determinada cantidad de datos, de forma que vayamos viendo como escala el calculo de los índices de validación en función del volumen del dataset. Las pruebas se han llevado a cabo utilizando el cluster de la DPAC que consta de 6 nodos con 16 cores y 64Gb de RAM cada uno.

Empezaremos con un tamaño del dataset igual a N= 2.057.050 (todo el dataset de Gaia) y poco a poco iremos incrementando el tamaño del dataset hasta llegar a 10N = 20.570.500, mientras que k será igual a 10, 20, y 30, en todos los casos. El uso de tres valores distintos de “k” se debe a que algunos índices, como por ejemplo el índice de Ball o el índice de Hartigan, se basan en la magnitud del cambio producido en alguna medida cuando usamos un k u otro, y por tanto se necesita más de una sola “k” para ser capaces de obtener un resultado con dichos índices.

K = [10, 20, 30 grupos] (tiempo en segundos)	**2.057.050 (N)**	**4.114.100 (2N)**	**10.285.250 (5N)**	**20.570.500 (10N)**
Ball Index	28.70	28.66	30.01	49.77
CH Index	26.37	32.57	35.49	50.51
DB Index	15.53	16.64	20.04	30.76
Hartigan Index	13.07	13.02	16.85	33.29
Ratkowsky Index	11.52	15.86	18.01	30.92
KL Index	20.75	23.26	31.28	53.46

En la figura 21 podemos ver la gráfica que representaría el tiempo de ejecución en segundos para cada uno de los índices (en el eje y), frente al tamaño del dataset de N hasta 10N (en el eje x).

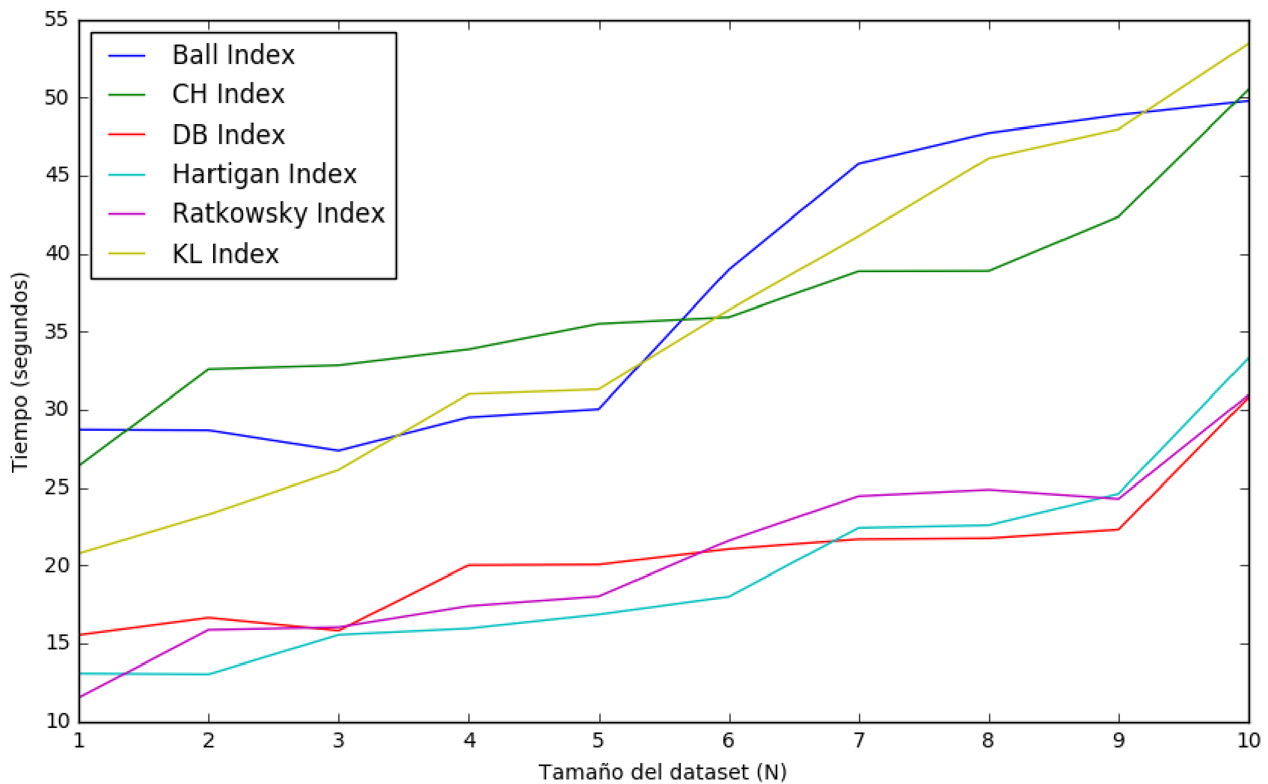


Figura 21: Gráfica con tiempos de ejecución para cada índice según incrementamos el tamaño del dataset

Como podemos ver en la gráfica, todos los índices implementados tienen un orden de complejidad menor a O(n), lo cual hace que escalen bastante bien con el tamaño del dataset, y además debemos tener en cuenta que al ser una versión distribuida implementada en Spark, podríamos añadir más nodos al cluster, para conseguir tiempos de ejecución aún mejores.

Algo que llama la atención en la gráfica es observar como a veces se producen pequeñas fluctuaciones respecto al tiempo de ejecución, y vemos como por ejemplo el índice de Ball tarda menos con un dataset mayor de 3N respecto a un dataset menor de 2N. Esto se debe a que al estar trabajando con varios nodos a la vez, y tras llevar a cabo un sobremuestreo sobre los datos originales de TGAS, en función de lo bien que se distribuya la información a lo largo de los diferentes nodos, el trabajo se va a poder realizar más o menos rápido, debido a que cuando la información se encuentra más homogéneamente distribuida a través de los nodos, dichos nodos se pueden aprovechar de la localidad del dato, y su velocidad es mayor que cuando deben enviar información a través de la red de un nodo a otro para su tratamiento, o cuando un nodo tiene mucha carga, y el resto de nodos deben esperar a que acabe su tarea.

No obstante, al haber cacheado los datos antes de la realización del cálculo de cada índice, el efecto debería ser muy pequeño, del orden de segundos, y además para tratar de disminuir aún más dicho efecto, he realizado 3 repeticiones de cada una de las pruebas para cada índice, y posteriormente he obtenido la media para cada caso.

Si miramos los tiempos de ejecución que hemos obtenido para los índices (en ningún caso el calculo de algún índice supera el minuto de ejecución), podríamos pensar si es relevante o no, tener una implementación distribuida de dichos índices. Pues bien, hay que tener en cuenta que lo relevante de tener estos índices implementados en Apache Spark no es el ahorro de tiempo que suponen respecto a su versión no distribuida implementada en otros lenguajes como R o python, sino directamente tenemos que valorar la posibilidad de emplear dichos índices cuando tenemos datasets de gran tamaño. Como hemos podido observar en el caso de los índices implementados en Spark no supone ningún problema trabajar con datasets de varios millones de registros, y obtener unos tiempos de ejecución bastante razonables. Mientras que, por ejemplo, si tratamos de utilizar un dataset de “tan sólo” medio millón de registros con el paquete NbClust de R, obtendríamos los siguientes mensajes de error:

```
Error: no se puede ubicar un vector de tamaño  1157.7 Gb

Error in t(jeu) %*% jeu : requires numeric/complex matrix/vector arguments
```

Es decir, que la implementación de estos índices de validación en Spark nos permite emplear dichos índices con datasets de gran tamaño, que de otra forma no podríamos validar. Por otra parte, los tiempos de ejecución obtenidos en el cálculo de dichos índices es tan pequeño entre otras cosas porque hemos precalculado y cacheado en memoria los datasets ya clusterizados antes de llevar a cabo ningún calculo, por lo tanto los índices pueden aprovechar el disponer de toda la información que necesitan en la memoria de los nodos del cluster, y por tanto pueden llevar a cabo su trabajo con mayor rapidez.

4.3 Experimentos

4.3.1. Introducción a los experimentos realizados

Para este estudio he utilizado el dataset de TGAS, que tiene 5 variables que ofrecen información sobre diferentes medidas astrométricas: ascensión recta (ra), declinación (dec), movimiento propio en ascensión recta (pmra), movimiento propio en declinación (pmdec), y paralaje (parallax). Estas variables tienen las siguientes características:

Datos en bruto	ra	dec	pmra	pmdec	parallax
Mínimo	0,00	-89,89	-3359,36	-3159,34	-24,82
Media	187,17	-3,21	-1,74	-5,25	2,48
Máximo	360.00	89,83	2890,43	1858,53	295,80

Mientras que con los datos estandarizados, tendríamos las siguientes estadísticas:

Datos estandarizados	ra	dec	pmra	pmdec	parallax
Mínimo	-1,87	-2,01	-129,40	-133,89	-9,38
Media	0,00	0,00	0,00	0,00	0,00
Máximo	1,73	2,16	114,88	79,11	100,79

Para estandarizar los datos, a cada punto del dataset le restaremos la media para cada variable, de forma que centremos los datos en 0 para cada variable, y posteriormente le dividiremos por la desviación típica de cada variable. La decisión de estandarizar los datos se debe a que intentamos dar a todas las variables el mismo peso, ya que si no estandarizásemos los datos, las variables relativas a los movimientos propios que como hemos visto tenían unos mínimos y máximos más elevados respecto al resto de variables, podrían ser los que lleven todo el peso de la división en grupos de los distintos puntos.

4.3.2. Fase 1: Primeras pruebas

Dado que no sabemos en cuantos grupos se divide el dataset de TGAS, vamos a recurrir a los índices de evaluación interna para tratar de inferir un número de grupos lo más consistente posible con el dataset que tenemos.

Primero, vamos a ver que valor de “k” nos sugieren los distintos índices de evaluación interna implementados en Spark, usando para ello los modelos obtenidos usando los tres algoritmos de clusterización de los que disponemos en Apache Spark (k-means, bisecting k-means, y mezclas gaussianas). Es necesario usar una lista de posibles números de grupos para que los índices nos sugieran cual de entre todos ellos se adapta mejor a la agrupación natural de nuestro dataset. Por tanto, he recurrido a un experto en el dominio que me ha sugerido la siguiente lista:

[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

Para estos experimentos utilizaremos 20 como el número máximo de iteraciones. Por otra parte, utilizaremos dos datasets, uno formado por las variables originales que aparecen en TGAS (ra, dec, pmra, pmdec, parallax), y que etiquetaremos como “Dataset 1”, y por otra parte el dataset creado a partir del original (x, y, z, vta, vtd) al cual etiquetaremos como “Dataset 2”.

Primero veamos los resultados obtenidos cuando hemos utilizado los datasets de los experimentos 1 y 2, sin llevar a cabo ninguna estandarización ni eliminación de outliers.

BALL AND HALL INDEX

Dataset \ Algoritmo	K-Means	Bisecting K-Means	Mezclas Gaussianas
Dataset 1	20 grupos	20 grupos	16 grupos
Dataset 2	20 grupos	20 grupos	4 grupos

CALINSKI AND HARABASZ INDEX

Dataset \ Algoritmo	K-Means	Bisecting K-Means	Mezclas Gaussianas
Dataset 1	10 grupos	10 grupos	16 grupos
Dataset 2	90 grupos	100 grupos	3 grupos

DAVIES AND BOULDIN INDEX

Dataset \ Algoritmo	K-Means	Bisecting K-Means	Mezclas Gaussianas
Dataset 1	80 grupos	10 grupos	10 grupos
Dataset 2	10 grupos	20 grupos	3 grupos

HARTIGAN INDEX

Dataset \ Algoritmo	K-Means	Bisecting K-Means	Mezclas Gaussianas
Dataset 1	20 grupos	20 grupos	16 grupos
Dataset 2	20 grupos	20 grupos	3 grupos

RATKOWSKY INDEX

Dataset \ Algoritmo	K-Means	Bisecting K-Means	Mezclas Gaussianas
Dataset 1	10 grupos	10 grupos	16 grupos
Dataset 2	10 grupos	10 grupos	3 grupos

KL INDEX

Dataset \ Algoritmo	K-Means	Bisecting K-Means	Mezclas Gaussianas
Dataset 1	100 grupos	100 grupos	22 grupos
Dataset 2	90 grupos	100 grupos	3 grupos

Lo primero que llama la atención tras realizar este primer experimento es la diferencia existente entre el número de grupos sugerido por cada uno de los índices. Por ejemplo, para el "Dataset 1" el índice KL nos sugiere k = 100 grupos como mejor número de grupos, mientras que el índice de Ball nos da k = 20 grupos como mejor *k* para ese mismo dataset. Esto se debe a qué cómo vimos en la definición de los distintos índices, cada índice utiliza distintas medidas para decidir qué *k* es mejor para nuestro dataset. Por ejemplo, el índice de Ball se basa en encontrar aquel *k* que lleva a un mayor salto en la disminución de la suma de los cuadrados de los errores intra-grupos, respecto al anterior *k*. Es decir, que para nuestro "Dataset 1" con k = 20, se puede haber producido el mayor salto, y por eso nos lo sugiere el índice de Ball, pero ello no quiere decir que no podamos tener otros *k* distintos para los cuales no sigan produciendo descensos en la suma de los cuadrados de los errores intra-grupos, o mejoras en cualquier otra medida relevante para la calidad de los grupos, como puede ser por ejemplo la dispersión de los grupos.

Por otra parte, el algoritmo de mezclas de gaussianas, nos devuelve en algunos casos, valores de *k* que no hemos utilizado inicialmente (recordemos que en esta primera fase sólo hemos usado los múltiplos de 10 que van de 10 a 100). Ello se debe a que inicialmente hemos empleado tan sólo 20 iteraciones (valor por defecto en la librería MLlib de Spark, que posteriormente modificaremos) como máximo a la hora de ejecutar el algoritmo de mezclas de gaussianas, y aunque en la primera iteración se generen por ejemplo 10 gaussianas para k=10, posteriormente, según se van modificando las distribuciones, puede que haya gaussianas tan próximas entre sí que se solapen, y debido al peso que asignamos a cada una de las gaussianas, una de ellas domine a la otra por tener un mayor número de elementos asociados. De forma, que a la hora de calcular la probabilidad de pertenencia de un punto del dataset a cada una de esas dos gaussianas, la gaussiana dominante acabe obteniendo todos los puntos, porque la probabilidad de pertenencia de un elemento a dicha gaussiana va a ser mayor que el de pertenecer a la otra debido a ese peso o probabilidad a priori mayor en la gaussiana dominante.

Resumiendo las pruebas realizadas en este primer experimento de la primera fase de experimentación, el número de grupos más veces recomendado por los índices de evaluación interna ha sido 10 y 20 grupos, como podemos observar en la siguiente tabla de resultados:

N.º de grupos	**N.º de veces recomendado**
10	**9**
20	**9**
3	5
16	4
100	4
90	2
4	1
22	1
80	1

A continuación, he vuelto a realizar el mismo experimento, pero ahora he estandarizado ambos datasets usando para ello la siguiente fórmula: $[(x_i - \mu) / \sigma]$, y los resultados obtenidos han sido los siguientes:

BALL AND HALL INDEX

Dataset \ Algoritmo	**K-Means**	**Bisecting K-Means**	**Mezclas Gaussianas**
Dataset 1	20 grupos	20 grupos	20 grupos
Dataset 2	20 grupos	20 grupos	4 grupos

CALINSKI AND HARABASZ INDEX

Dataset \ Algoritmo	**K-Means**	**Bisecting K-Means**	**Mezclas Gaussianas**
Dataset 1	10 grupos	10 grupos	10 grupos
Dataset 2	100 grupos	100 grupos	4 grupos

DAVIES AND BOULDIN INDEX (20 iteraciones max.)

Dataset \ Algoritmo	K-Means	Bisecting K-Means	Mezclas Gaussianas
Dataset 1	90 grupos	70 grupos	10 grupos
Dataset 2	10 grupos	90 grupos	4 grupos

HARTIGAN INDEX

Dataset \ Algoritmo	K-Means	Bisecting K-Means	Mezclas Gaussianas
Dataset 1	20 grupos	70 grupos	40 grupos
Dataset 2	20 grupos	40 grupos	4 grupos

RATKOWSKY INDEX

Dataset \ Algoritmo	K-Means	Bisecting K-Means	Mezclas Gaussianas
Dataset 1	10 grupos	10 grupos	10 grupos
Dataset 2	10 grupos	10 grupos	4 grupos

KL INDEX

Dataset \ Algoritmo	K-Means	Bisecting K-Means	Mezclas Gaussianas
Dataset 1	100 grupos	40 grupos	59 grupos
Dataset 2	100 grupos	100 grupos	4 grupos

Es decir, que usando ahora el dataset estandarizado y los mismos índices de evaluación interna, hemos obtenido las siguientes recomendaciones acerca del número de grupos:

N.º de grupos	N.º de veces recomendado
10	**10**
20	**7**
4	6
100	5
40	3
70	2
90	2
59	1

Si comparamos los resultados obtenidos en los dos experimentos realizados en esta primera fase de experimentación, podemos observar como en ambos casos (tanto cuando usamos los datasets estandarizados como sin estandarizar), los índices de evaluación interna utilizados parecen sugerir que el dataset de TGAS debería dividirse en 10-20 grupos.

Sin embargo, para tratar de mejorar el proceso de detección del mejor número de grupos en el cual se puede dividir nuestro dataset, voy a llevar a cabo nuevos experimentos donde realizaremos múltiples cambios que esperemos puedan acercarnos a una solución más fiable.

4.3.3. Fase 2: mezclas gaussianas, aumento de la precisión de "k", probabilidad de pertenencia al grupo alta, y uso de sistema de scoring

A continuación vamos a llevar a cabo un nuevo experimento, introduciendo varios cambios respecto a los experimentos realizados hasta el momento. Primero, llevaremos a cabo ciertas simplificaciones respecto a los experimentos realizados en la fase 1:

- Los modelos generados por el algoritmo k-means, pueden considerarse un caso particular de los modelos generados mediante mezclas gaussianas, en el cual todas las variables tienen la misma covarianza. Por tanto, para los experimentos de la fase 2 utilizaremos sólo mezclas gaussianas como algoritmo de clusterización.

- Utilizaremos sólo el "Dataset 2" estandarizado usando la misma fórmula empleada en los últimos experimentos de la fase 1 [$(x_i - \mu) / \sigma$]. Usaremos sólo el "Dataset 2", ya que nos ofrece una ventaja importante respecto al "Dataset 1", y es que nos ayuda a solucionar el problema creado por puntos que están muy cercanos pero tienen valores muy dispares (por ejemplo 0.1º y 359.9º). Por otra parte, el estandarizar los datos siempre es una buena práctica cuando estamos ante una tarea de clusterización y no queremos que una variable del dataset domine sobre el resto sólo por tener valores más extremos.

Partiendo de las dos simplificaciones comentadas en los puntos anteriores, vamos a realizar algunos cambios importantes para tratar de obtener un mejor resultado en nuestros experimentos:

- En primer lugar, el número de *k* que consideraremos ahora será mucho mayor que antes, ya que irá de 10 a 100 de uno en uno, es decir que ahora tendremos 90 número de grupos distintos a probar, frente a la lista de 10 números de grupos que probamos en los experimentos de la fase 1.

- En segundo lugar, aprovecharemos la asignación blanda que permite el uso del algoritmo de mezclas gaussianas, y antes de evaluar cada dataset por los índices de validación interna, sólo tendremos en cuenta aquellos elementos del dataset que han sido propuestos como elementos de un determinado grupo con una probabilidad de pertenencia a dicho grupo de al menos un 75%. De esta forma, habrá elementos del dataset a los que no asignaremos ningún grupo, pero tendremos una mayor seguridad de que los elementos que obtengan un grupo determinado son miembros más legítimos de dicho grupo.

- En tercer lugar, eliminaremos los outliers que detectemos del dataset. Para detectarlos, usaremos el dataset estandarizado, y filtraremos aquellos elementos que tienen alguna variable con un valor superior a 5 o inferior a -5, lo cual indicaría valores muy extremos. De esta forma, sólo eliminaríamos 312 elementos del dataset.

- En cuarto lugar, empezaremos a ver gráficas en 3d, cuyo objetivo no es mostrar con precisión la posición de cada elemento en un eje de coordenadas, sino que podamos observar la forma de los grupos, y de esta forma tener una primera estimación sobre la influencia que están teniendo la posición o el movimiento en la formación de los grupos. Por ejemplo, cuando vemos que un grupo tiene elementos muy separados entre sí, pero aún así forman parte del mismo grupo, puede ser un indicativo de que en dicho grupo están primando más los movimientos propios o las velocidades transversales.

- En quinto lugar, aumentaremos el número máximo de iteraciones que va a realizar el algoritmo de mezclas gaussianas a 100. El aumento del número máximo de iteraciones se debe a que tras realizar un pequeño experimento en el cual aplicamos mezclas gaussianas al “Dataset 2” con 20 iteraciones y usando k=90, el número de grupos que realmente obtenemos son 63, y si nos quedamos sólo con aquellos elementos que tienen una probabilidad de pertenencia a su grupo de al menos un 75% nos quedamos con tan sólo 7 grupos. Mientras que si aumentamos el número máximo de iteraciones a 100, entonces el número de grupos final que obtenemos son exactamente 90 grupos, y si aplicamos la probabilidad de al menos un 75%, nos quedamos con 89 grupos.

- Por último, en vez de quedarnos con una sola recomendación de *k* por cada índice de evaluación interna, ahora obtendremos una lista con un scoring para cada *k* por parte de cada índice. De forma que para cada uno de los índices, el primer *k* sugerido por un índice obtendrá 90 puntos, el siguiente mejor *k* para dicho índice obtendrá 89 puntos, y así sucesivamente hasta el peor *k* que obtendrá 0 puntos. Y posteriormente sumaremos los puntos obtenidos por cada *k*, para de esta forma tratar de utilizar toda la información de la que disponemos a la hora de elegir cuál es el valor de *k* más apropiado para nuestro dataset.

  Sumar los scorings obtenidos por cada índice permite seleccionar aquellos *k* que cumplen más criterios como mejor *k* para nuestro dataset, ya que hay índices que se basan en la disminución que se produce en los errores intra-grupo al aumentar el número de grupos (como el índice de Ball y Hall), otros se basan en la dispersión de los grupos (como el índice de Calinski y Harabasz o el índice de Davies y Boulding), y otros como el índice de Ratkowsky y Lance se basan en otros aspectos de los grupos como la distancia de los centroides de cada grupo a un hipótetico grupo que contuviera todos los ejemplos del dataset. Por tanto, parece lógico pensar, que cuantos más criterios de validación cumpla una clusterización, mayor valor deberíamos darla. Por ejemplo, si tenemos grupos muy compactos, que además se encuentran muy dispersos unos de otros, y muy alejados del centro del dataset total, dicha clusterización debería ser más premiada, que una clusterización en la cual los grupos son muy compactos, pero se encuentran muy cerca unos de otros.

Tras los nuevos cambios realizados, los resultados obtenidos en el experimento han sido los siguientes (quedándonos sólo con los 20 número de grupos mejor puntuado para cada índice):

index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
indexDB	96	91	93	98	85	97	92	84	79	95	99	94	82	90	75	77	87	100	86	89
indexBall	6	7	8	9	11	10	12	20	22	14	13	16	17	27	15	38	18	30	69	24
indexRatkowsky	45	64	86	63	87	68	96	94	100	79	89	95	75	97	37	61	72	52	48	34
indexKL	45	93	80	86	67	58	41	34	11	15	98	60	62	23	74	94	39	95	50	27
indexHartigan	11	69	79	72	81	73	10	71	20	70	21	91	75	22	49	92	19	68	26	56
indexCH	45	64	86	63	87	97	96	100	37	95	94	93	89	60	68	53	99	34	72	82

Si sumamos ahora el scoring de todos los número de grupos, obtenemos el siguiente ranking:

k	score
79	**470.0**
72	**447.0**
75	**441.0**
86	438.0
92	436.0
58	428.0
93	418.0
70	411.0
73	409.0
80	389.0
64	386.0
99	384.0
94	384.0

Es decir, que los resultados obtenidos en esta segunda fase de experimentación nos ofrecen como mejor número de grupos k = 79. Aunque k = 45 también resulta interesante, al haber sido el mejor “k” para tres índices (Ratkowsky, KL, y CH). A continuación vamos a realizar varias gráficas que nos permitan intuir como sería la distribución de dichos grupos para este valor.

En la figura 22 podemos observar en una gráfica en 3d el resultado de la clusterización usando k = 79. Uno de los primeros problemas que podemos encontrar en esta gráfica es que algunos puntos parecen encontrarse muy alejados respecto a la mayoría de puntos que se hayan en la parte central de la gráfica. Dichos puntos aparecen bastante aislados, lo cual podría deberse, por ejemplo, a errores en la medida del paralaje, o a que sean objetos celestes que no pertenecen a la Vía Láctea. Aunque también podría tratarse de elementos de nuestra vía láctea que se encuentran en los límites de nuestra galaxia.

En el próximo experimento que realizaremos en la fase 3, trataremos de eliminar aquellos puntos que tienen un error de paralaje demasiado grande respecto a su valor asociado, para de esta forma tener una mayor certeza sobre la posición que ocupan los puntos de nuestro dataset.

Por otra parte, en este experimento 53 de los 79 grupos que hemos obtenido tienen menos de 50 elementos, mientras que el grupo más numeroso consta de unos 700.000 elementos. Esta información es importante contrastarla con un experto en el dominio para ser capaces de mejorar la calidad de la clusterización.

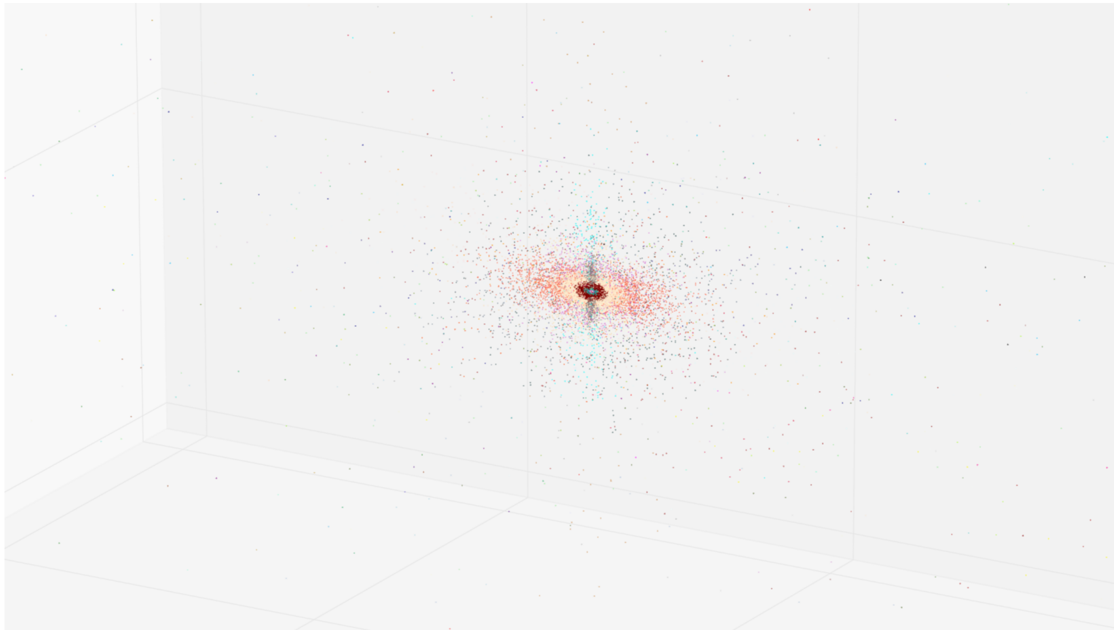


Figura 22: Representación en 3d de una clusterización de 79 grupos con GMM

Y en la figura 23, vemos una representación en 3d de una clusterización con k = 45 grupos.

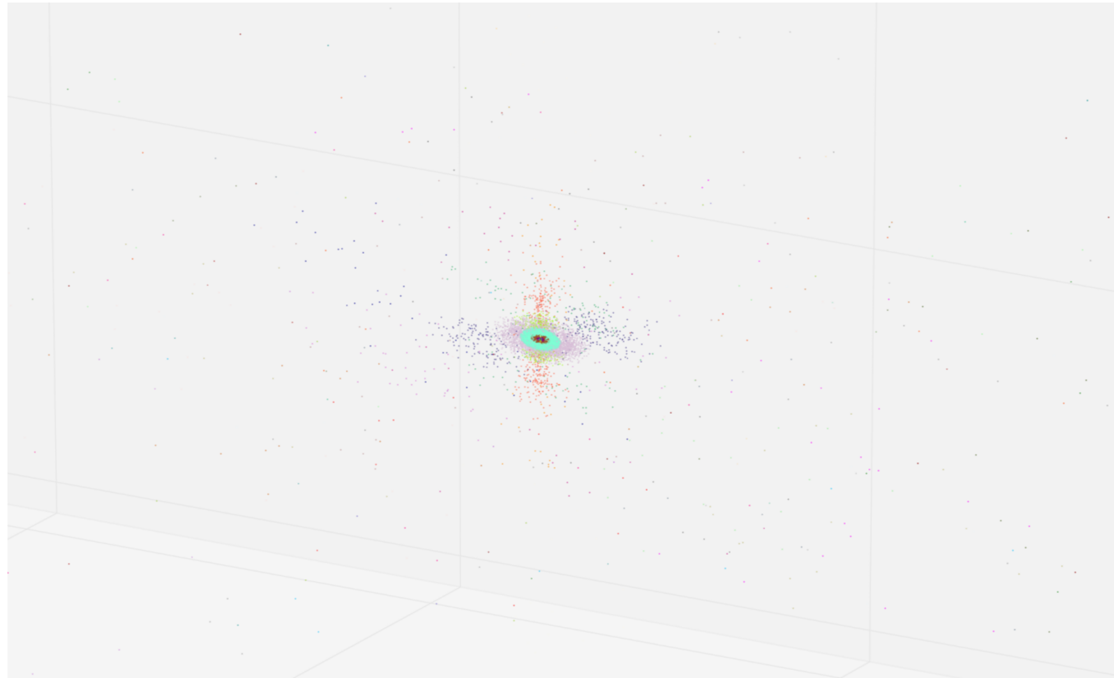


Figura 23: Representación en 3d de una clusterización de 45 grupos con GMM

Representar un número elevado de puntos (en nuestro caso unos 2 millones), y con muchos grupos, hace difícil sacar conclusiones a partir de las gráficas. No obstante, pueden ayudarnos a saber como se esta realizando la clusterización. Por ejemplo, saber si para la clusterización de un determinado grupo está teniendo mayor peso la posición del objeto, o sus movimientos propios (o velocidad transversal).

4.3.4. Fase 3: error relativo paralaje máximo 20%, nuevo sistema de scoring, más iteraciones, y uso del índice de Rand

En esta nueva fase vamos a mantener los cambios realizados en los experimentos de la fase 2, pero introduciendo las siguientes novedades:

- Calcularemos el error relativo de la paralaje (error absoluto en el paralaje dividido entre el valor de la paralaje), y nos quedaremos sólo con aquellos elementos del dataset de TGAS-DR1 en los cuales, el error relativo sea inferior al 20%. Esto nos elimina prácticamente la mitad de elementos del dataset, quedándonos ahora con tan sólo 1.067.920 registros. No obstante, así nos aseguramos que no estamos usando ningún elemento con valores anómalos relativos a la distancia a la que se encuentra dicho objeto.

- Debido a que usando el sistema de scoring visto en la fase 2, incluso obteniendo la mejor puntuación en tres de los seis índices, dicho valor de *k* (en nuestro caso era 45) no obtenía una buena posición en el ranking. Ahora, el sistema de puntuación será diferente, de forma que sólo tendremos en cuenta a los 10 mejores *k* por cada índice, y además los tres mejores para cada índice, obtendrán un "premio", que será +5 puntos para el primero, +3 puntos para el segundo, y +1 punto para el tercero, para de esta forma dar aún más valor al *k* ganador en cada caso. Asignar 5, 3, y 1 a los tres primeros es una forma de recompensar a los mejores, no obstante, al igual que pasa con muchos hiperparámetros utilizados en minería de datos, es algo que debería ser comprobado experimentalmente, y cambiado en caso de encontrar otro valor con el que obtuviéramos mejores resultados.

- Subiremos el número máximo de iteraciones a 200 para asegurarnos de lograr la convergencia, y que las gaussianas se ajusten lo mejor posible a los datos. Y el parámetro "convergenceTol" del algoritmo de mezclas gaussianas le seguiremos dejando a su valor por defecto que es 0.01. Este parámetro actúa como criterio de parada, de forma que se dejarán de llevar a cabo más iteraciones, bien cuando alcancemos el número máximo de iteraciones propuesto, o bien cuando en una iteración no podamos maximizar la probabilidad respecto a la anterior iteración según el parámetro "convergenceTol", y por tanto consideremos que ya hemos alcanzado la convergencia y podemos parar de realizar más iteraciones.

- Por último, utilizaremos el índice de Rand de evaluación externa que he implementado en Spark, para de esta forma tratar de obtener una medida más objetiva de la calidad de los grupos. Para ello, vamos a usar información sobre una serie de estrellas que sabemos que forman parte de un cúmulo estelar, y que por tanto, deberían aparecer en el mismo grupo si la clusterización se ha realizado correctamente.

Para cada índice, el ranking de las 10 mejores “k” ha sido el siguiente:

index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
indexDB	7	6	5	91	100	97	93	95	94	77
indexBall	8	6	7	11	9	10	14	12	34	21
indexRatkowsky	5	6	7	8	9	10	11	12	16	14
indexKL	78	8	97	38	39	35	28	22	74	11
indexRand	8	9	44	10	26	91	68	99	96	94
indexHartigan	7	8	11	10	99	72	37	71	34	68
indexCH	11	8	9	5	12	100	10	16	20	6

Y aplicando el sistema de puntuación propuesto en la página anterior, ahora el scoring final obtenido para cada “k” es el siguiente:

k	**score**
8	**73**
7	**48**
6	**37**
11	36
9	33
5	31
10	28
78	15
97	14
91	12
12	12
100	11
44	9
99	9
38	7
39	6
26	6
16	5
68	5
35	5

Si analizamos los resultados obtenidos por los distintos índices, parece que con k=8, obtenemos muy buena puntuación en todos los índices, salvo en el índice de Ratkowsky. El índice de Ball y el índice de Rand directamente nos lo ofrecen como mejor *k* posible. Mientras que los índices de Hartigan, de KL, y de CH, aparece como segundo mejor *k*.

Una de las cosas que llama la atención es que ahora a diferencia de los experimentos realizados en la fase 2, el número de grupos sugerido por los índices empleados es mucho menor que antes. Al haber eliminado aquellos elementos del dataset con un error relativo de la paralaje superior al 20%, parece que hemos conseguido deshacernos de muchos pequeños grupos que se formaban en los experimentos anteriores, y que contaban en muchos casos con muy pocos elementos. Por ejemplo, en el último experimento realizado en la fase 2, el 67.09% de los grupos formados para el sugerido como mejor *k* (k = 79) tenían menos de 50 elementos.

En cuanto al número de elementos por grupo, generando el modelo de mezclas gaussianas con k = 8, y las mismas condiciones del experimento realizado en la fase 3, ahora el número de elementos por grupo que obtenemos es el siguiente:

IdGrupo	Num. Elementos del grupo
Grupo5	385.867
Grupo2	123.867
Grupo4	84.215
Grupo0	8.125
Grupo1	7.262
Grupo6	4.214
Grupo7	2.347
Grupo3	943

Veamos a continuación como aparecen representados los distintos grupos para k = 8 en una gráfica:

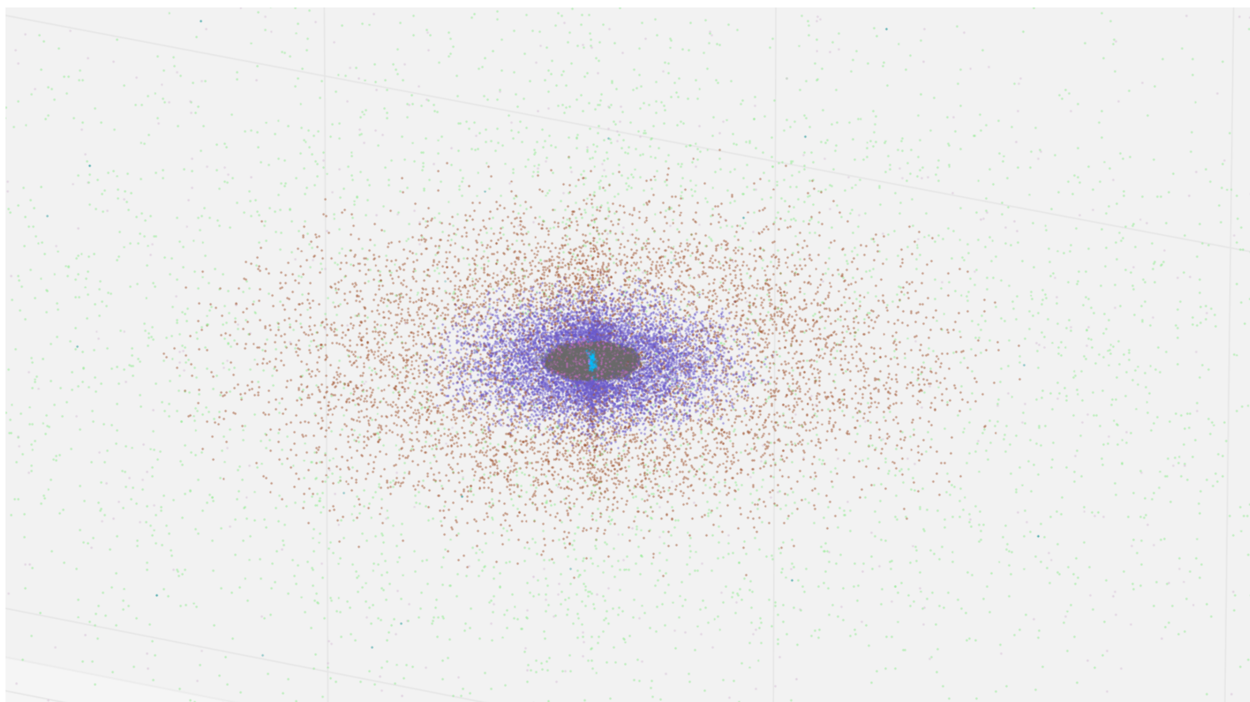


Figura 24: Representación en 3d de una clusterización de 8 grupos con GMM

Uno de los problemas que presenta la figura 24 es que el grupo identificado como “Grupo5” que aparece en color gris y se encuentra en el centro de nuestra galaxia, está compuesto por una gran cantidad de estrellas, exactamente 385.867 estrellas, lo cual hace que perdamos visibilidad del resto de grupos. Por tanto, vamos a visualizar de nuevo la misma gráfica en la figura 25, pero eliminando ahora todos los elementos que pertenecen al “Grupo5”.

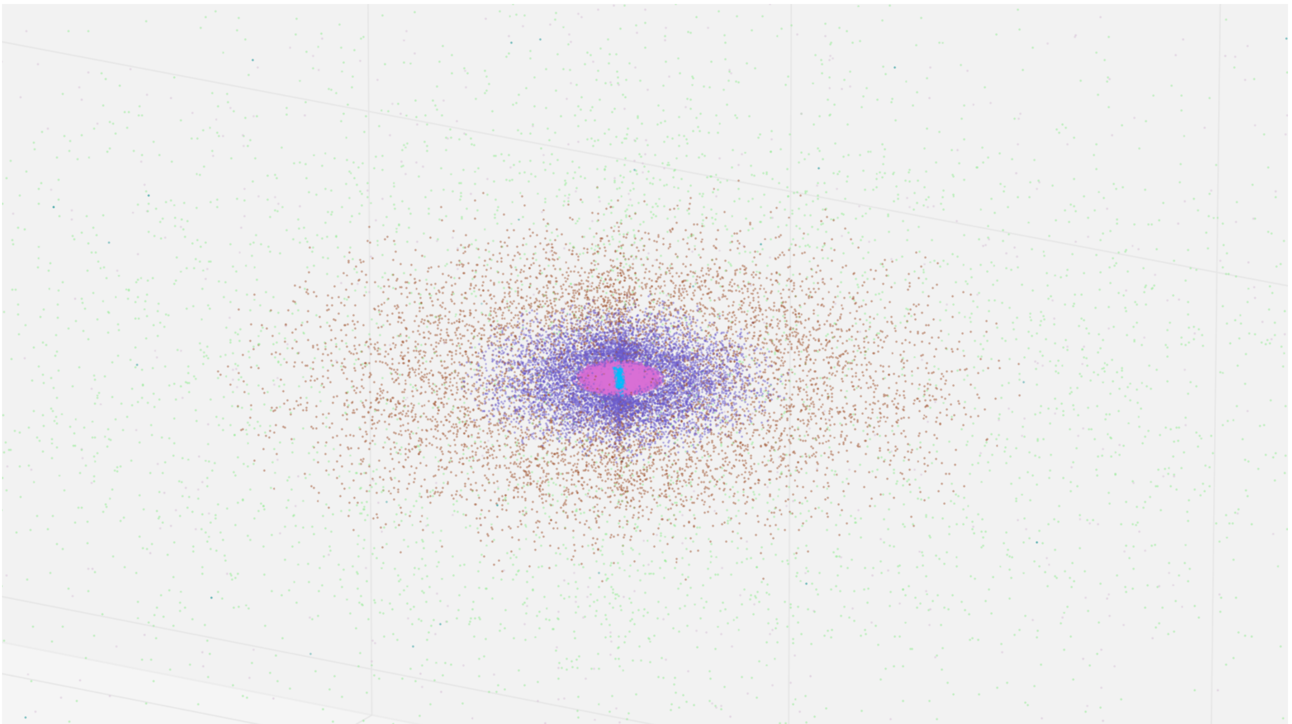


Figura 25: Representación en 3d de una clusterización de 8 grupos con GMM (quitando Grupo5)

En la figura 25 podemos ver claramente como el “Grupo5” no nos dejaba ver que justo en la misma posición, se encuentra el “Grupo4” en color orquídea (muy parecido al rosa), el cual estaba compuesto por 84.215 elementos. Para ver si aún seguimos teniendo grupos ocultos que no podemos ver por la gran concentración de elementos en el bulbo de la galaxia, vamos a eliminar también todos los elementos del “Grupo4” para ver si podemos obtener más información sobre la agrupación obtenida para k = 8.

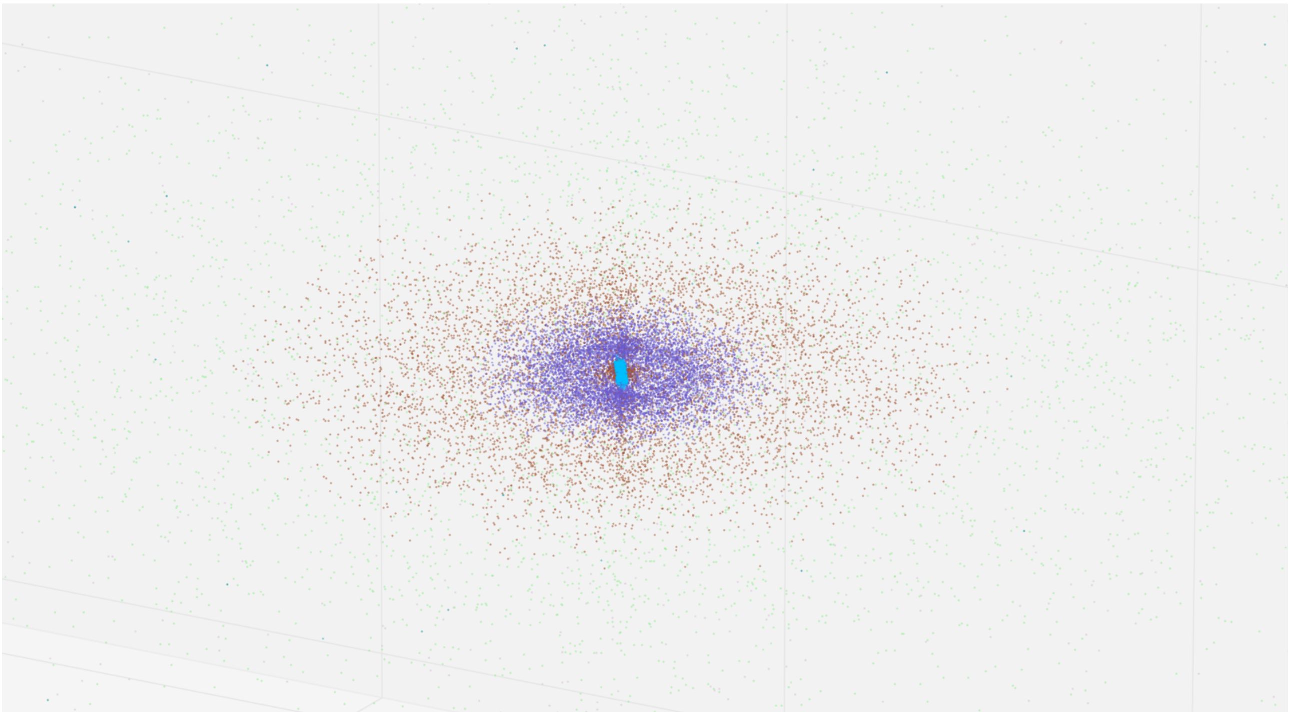


Figura 26: Representación en 3d de una clusterización de 8 grupos con GMM (quitando Grupo5 y Grupo4)

En la figura 26 se observa como en el centro de la galaxia, de forma mucho más concentrada que en cualquier otro grupo, aparece en color azul el grupo identificado como “Grupo2”, que contiene 123.867 elementos. Por otra parte, llama la atención como el grupo identificado como “Grupo1” en color marrón, y que contiene 7.262 elementos, tiene una gran concentración de elementos en el centro de la galaxia, pero también aparecen muchos de sus elementos difuminados alrededor de dicho centro.

Por otra parte, el grupo identificado como “Grupo0” que contiene 8.125 elementos, y aparece representado en color morado, es también un grupo con bastante visibilidad y bastante compacto. Mientras que los tres grupos identificados como “Grupo6” (en color verde con 4.214 elementos), “Grupo7” (en color rosa claro con 2.347 elementos), y “Grupo3” (en color cían oscuro con 943 elementos), aparecen muy dispersos en la gráfica.

4.3.5. Fase 4: eliminación de grupos muy dispersos, y re-clusterización individual de los grupos más compactos

Partiendo del dataset clusterizado en 8 grupos que obtuvimos en la fase anterior, vamos a eliminar de dicho dataset los elementos pertenecientes a los grupos identificados como “Grupo3”, “Grupo6”, y “Grupo7”, ya que como veíamos en la figura 26 los elementos pertenecientes a dichos grupos aparecían muy dispersos, y podrían ser identificados como outliers. Y de los elementos restantes, también eliminaremos 1.230 elementos (pertenecientes a distintos grupos) de los cuales no tenemos el identificador de tycho2. Por lo tanto, finalmente nos quedaremos con un dataset formado por tan sólo 5 grupos y un total de 608.109 elementos, que aparecen repartidos de la siguiente forma:

IdGrupo	**Num. Elementos del grupo**
Grupo5	385.479
Grupo2	123.202
Grupo4	84.076
Grupo0	8.121
Grupo1	7.231

Dado que en estos 5 grupos había acumulaciones bastante importantes de elementos que se encontraban muy juntos entre sí (como se puede ver en las figuras 24, 25, y 26), vamos a llevar a cabo una nueva clusterización para cada uno de estos grupos por separado, usando los índices de validación interna y externa para comprobar cual es la mejor división para cada uno de ellos. Hay que tener en cuenta que el índice de validación externa no le podremos usar en todos los casos, ya que habrá algunos grupos, para los cuales no tengamos ninguna evidencia sobre como deberían agruparse sus elementos.

Para los tres grupos con mayor número de elementos (Grupo5, Grupo2, y Grupo4), voy a realizar la misma estandarización de las variables (X, Y, Z, VTA, VTD) que hemos realizado hasta ahora [$(x_i - \mu) / \sigma$], pero de forma independiente para cada grupo. Dado que al haber eliminado los elementos más “diferentes” y quedarnos sólo con los de un grupo concreto, la estandarización puede variar sustancialmente, respecto a la realizada inicialmente.

Empezaremos con el grupo más numeroso identificado como “Grupo5”, y que cuenta con 385.479 elementos. Para dicho grupo, no tenemos ninguna evidencia sobre como deben agruparse sus elementos, por tanto, no podremos usar el índice de Rand para dicho grupo.

Para los elementos del “Grupo5”, tenemos que los mejores 10 *k* para cada índice han sido los siguientes:

index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
indexDB	5	84	54	85	90	6	43	94	87	62
indexBall	6	7	8	10	9	12	13	18	15	20
indexRatkowsky	6	7	8	9	10	11	12	14	13	15
indexKL	6	27	79	86	59	37	40	67	22	29
indexHartigan	6	61	69	20	78	66	57	93	51	19
indexCH	6	7	8	9	5	10	11	12	13	14

Y ahora si aplicamos el scoring por puntos, aplicado igualmente en los experimentos de la fase 3, obtenemos el siguiente resultado:

k	**score**
6	**80**
7	**36**
8	**27**
5	21
9	20
10	18
27	12
84	12
61	12
12	12
69	9
11	9
79	9
54	9
20	8
13	8
85	7
86	7
90	6
59	6

Parece que k = 6 es claramente la mejor opción para subdividir nuevamente el grupo identificado como “Grupo5”. Al quedarnos sólo con aquellos elementos cuya probabilidad de pertenencia al grupo sea superior al 75%, al igual que hicimos en anteriores experimentos, el número de elementos que finalmente conforman el dataset son 231.296, los cuales se dividen de la siguiente forma:

IdGrupo	Num. Elementos del grupo
Grupo5-1	39.663
Grupo5-3	61.523
Grupo5-5	37.876
Grupo5-4	17.383
Grupo5-2	74.366
Grupo5-0	485

La figura 27 muestra como quedarían gráficamente representados dichos grupos:

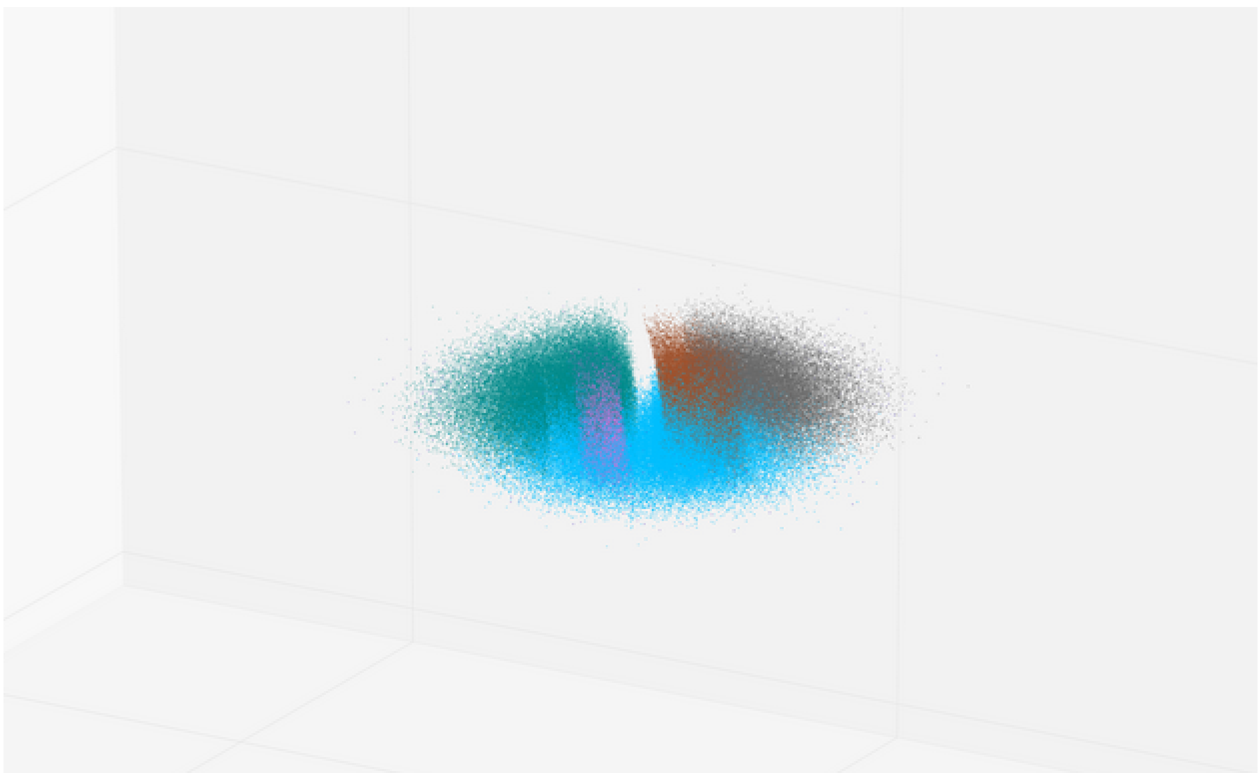


Figura 27: Representación en 3d de una clusterización de 6 grupos con GMM del dataset formado por Grupo5

En el caso del "Grupo2", sí que contamos con 16 elementos del dataset de los cuales tenemos evidencia sobre cómo deberían ir agrupados. En concreto son estrellas pertenecientes a los grupos de AB DORADUS (11), HERCULES-LYRA (3), y BETA PICTORIS (2). Por tanto, ahora sí podemos emplear el índice de Rand. Las mejores 10 *k* para cada índice han sido las siguientes:

index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
indexDB	5	7	43	27	13	39	57	65	22	37
indexBall	6	8	12	10	7	9	11	16	13	28
indexRatkowsky	5	7	6	8	9	11	10	12	13	15
indexKL	50	12	66	72	65	39	37	41	84	93
indexRand	96	94	88	79	68	66	54	53	48	7
indexHartigan	6	11	27	7	28	56	64	26	12	55
indexCH	5	6	7	8	9	10	11	13	12	14

Teniendo en cuenta que en el índice de Rand, sólo ha habido 10 *k* para las cuales hemos obtenido el valor máximo 1.0, y que son exactamente las "k" que aparecen representadas en la tabla de arriba. Por lo que todas tienen la misma puntuación, y a la hora de asignarlas un score, todas obtendrán un 15. Ahora si obtenemos el scoring para cada k, obtenemos la siguiente puntuación:

7	**61**
6	**51**
5	**45**
12	28
8	26
11	25
66	24
9	17
10	16
27	16
88	15
53	15
68	15
54	15
79	15
48	15
96	15
94	15
50	15
13	13

Según el score que obtenemos k = 7 sería la mejor elección para dividir el dataset identificado como "Grupo2". Aunque 5 y 6 también parecen buenas opciones, una de las razones por las que k = 7 obtiene la mejor puntuación es debido a que con dicha subdivisión de grupos, conseguimos confirmar todas las evidencias que hemos recabado respecto a los grupos de AB DORADUS (11), HERCULES-LYRA (3), y BETA PICTORIS (2).

Al quedarnos sólo con aquellos elementos cuya probabilidad de pertenencia al grupo sea superior al 75%, al igual que hicimos en anteriores experimentos, el número de elementos que finalmente conforman el dataset son 74.599, los cuales se dividen de la siguiente forma:

IdGrupo	Num. Elementos del grupo
Grupo2-1	19.699
Grupo2-2	17.222
Grupo2-3	16.247
Grupo2-4	8.589
Grupo2-0	5.576
Grupo2-6	5.202
Grupo2-5	2.064

La figura 28 muestra la distribución de los 7 subgrupos en los cuales quedaría dividido el dataset identificado como “Grupo2”:

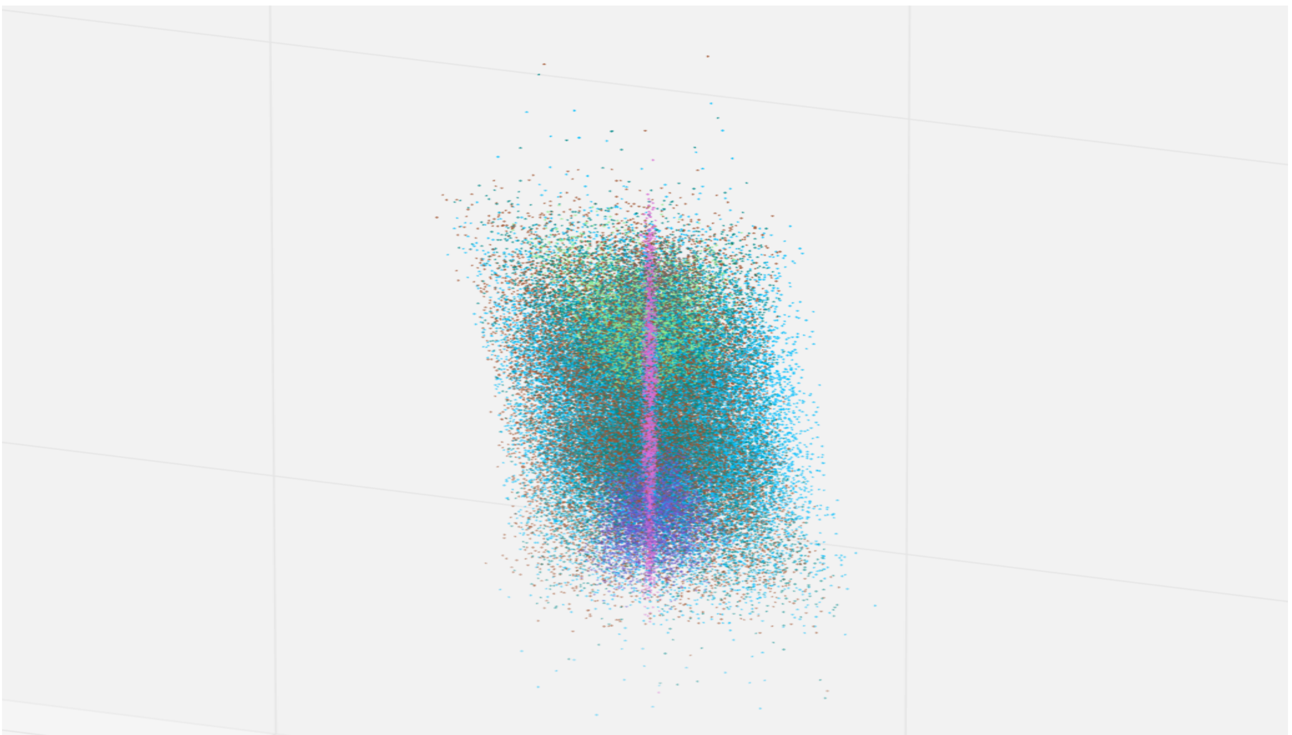


Figura 28: Representación en 3d de una clusterización de 7 grupos con GMM del dataset formado por Grupo2

En el caso del “Grupo4”, al igual que pasaba con el dataset del “Grupo5” no tenemos ninguna evidencia externa respecto a como se deben agrupar los elementos de dicho grupo. Por tanto, no utilizaremos ningún índice de evaluación externa en este caso. El número de elementos de este dataset es de tan sólo 84.215 elementos, y tras un primer experimento utilizando el mismo rango de *k* que antes (5-100), la mayoría de los índices han señalado k = 5 como el mejor número de subgrupos en el cual podemos dividir el dataset, pero dado que dicho número coincide con el limite inferior de “k” que hemos usado, vamos a realizar un nuevo experimento con el limite inferior de k igual a 2, para ver si realmente k = 5 es la mejor agrupación para este dataset o dicho número ha sido seleccionado simplemente por ser el más bajo. Las mejores 10 *k* para cada índice han sido las siguientes:

index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
indexDB	3	2	5	6	7	4	8	56	86	82
indexBall	3	4	5	6	8	7	11	12	10	18
indexRatkowsky	3	2	4	5	6	7	9	8	10	11
indexKL	3	49	25	35	38	47	73	68	59	99
indexHartigan	67	62	37	86	24	66	87	63	46	8
indexCH	2	3	4	5	6	7	8	9	10	11

Ahora si obtenemos el scoring para cada k, obtenemos la siguiente puntuación:

3	**72**
2	**39**
4	**35**
5	32
6	26
7	21
8	18
67	15
49	12
62	12
86	9
37	9
25	9
35	7
9	7
10	6
11	6
38	6
24	6
47	5

Si usamos el mejor score que ha sido k=3 grupos, para dividir el dataset etiquetado como “Grupo4”, la representación gráfica que obtenemos es la siguiente:

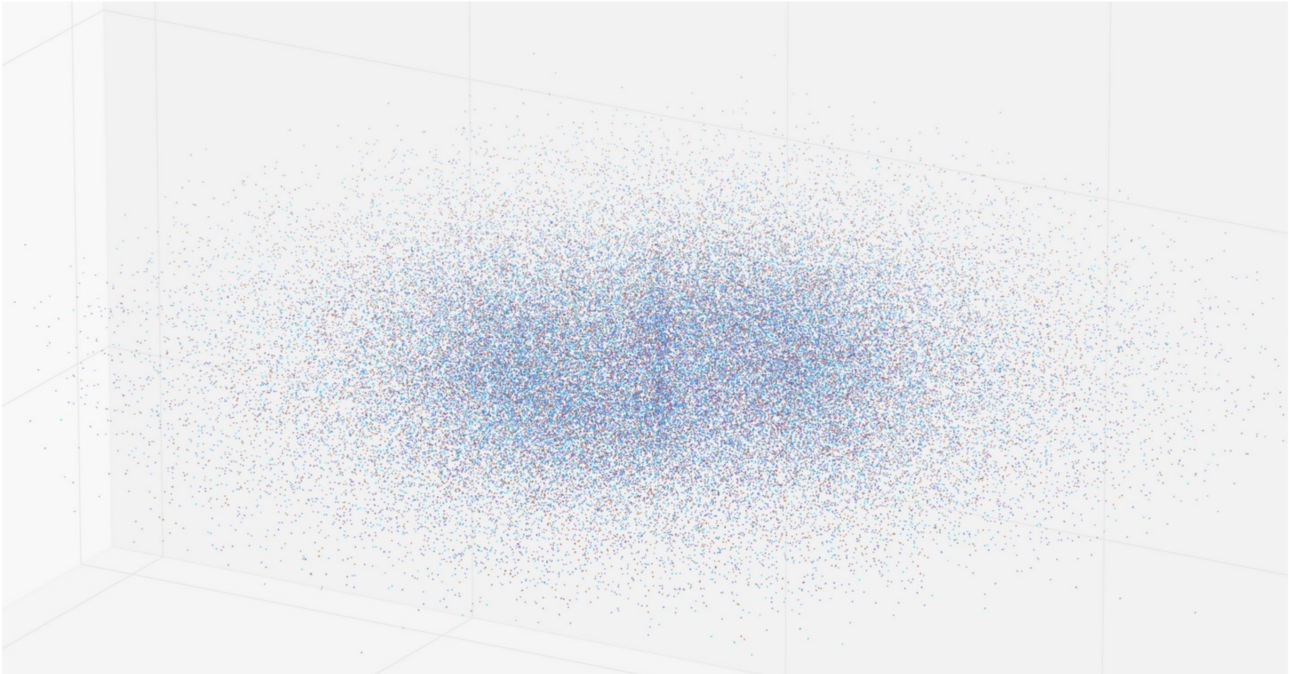


Figura 29: Representación en 3d de una clusterización de 3 grupos con GMM del dataset formado por Grupo4

Al observar la gráfica, podemos ver como los elementos de los tres grupos se encuentran muy mezclados entre sí, y es que en este caso, las variables relativas a las velocidades transversales han sido las que han llevado el mayor peso a la hora de asignar un grupo u otro a cada elemento del dataset.

Capitulo 5: Conclusiones

Los resultados obtenidos respecto al caso de uso que nos habíamos planteado no han sido relevantes desde el punto de vista de la astrofísica, ya que no hemos tenido en cuenta que los datos que teníamos hasta el momento se adaptaban a una gaussiana, y para obtener grupos relevantes, deberíamos haber eliminado los datos de los elementos que se encontraban en el centro de dicha gaussiana para de esa forma haber obtenido una clusterización más eficiente del resto de elementos del dataset.

El principal problema que se nos presenta en una tarea de clusterización es que al tratarse de un tipo de tarea no supervisada, realmente no sabemos si la solución que estamos obteniendo es correcta o no, y tampoco tenemos una forma clara de evaluar si una determinada clusterización es mejor o peor que otra. Por si esto no fuera suficiente, el problema se complica aún más si tampoco sabemos cuál es el número de grupos en el que se divide nuestro dataset, aunque sea de forma aproximada.

Para ayudarnos a resolver este problema podemos acudir a índices de evaluación interna basados exclusivamente en los datos que disponemos, y que se basan en distintas medidas como la compactación o la dispersión de los grupos. Además, si el problema que tratamos de resolver nos permite hacerlo, podemos usar algún índice de evaluación externa basado en la evidencia que tenemos sobre nuestro dataset, como por ejemplo elementos que estamos seguros que deben ir en el mismo grupo o elementos que sabemos que deben ir en grupos separados. En nuestro caso, la evidencia era muy pequeña, puesto que sólo hemos utilizado 42 elementos del dataset que sabíamos que debían ir en unos grupos determinados, por lo tanto sólo teníamos una evidencia sobre el 0,002% de elementos del dataset, lo cual hace casi insignificante el aporte del índice de evaluación externa empleado a la hora de arrojar luz sobre cual es el mejor número de grupos para nuestro dataset. No obstante, aún usando estos índices, salvo que tengamos una evidencia externa bastante importante, sólo vamos a obtener pequeñas pistas que puedan ayudarnos a intuir como agrupar nuestro dataset.

Y en cuanto a los algoritmos de clusterización que tenemos disponibles cuando necesitamos trabajar con una gran cantidad de datos, he decidido utilizar sólo aquellos algoritmos que tienen un orden de complejidad en cuanto a tiempo de ejecución que esté en torno a O(n) o menor, ya que si usáramos otros algoritmos de clusterización que tienen una complejidad superior, por ejemplo de $O(n^3)$, como algunos basados en clustering espectral, entonces tendríamos importantes problemas de rendimiento cuando tratásemos muchos datos, que es el objetivo final al cual he orientado este trabajo, proporcionar una base para que cuando tengamos que trabajar con un dataset de más de 1.000 millones de objetos, podamos usar los mismos algoritmos de clusterización e índices de evaluación empleados en este trabajo y aún así obtener feedback en un tiempo razonable.

Por otra parte, conviene destacar que el proceso de clusterización es siempre un proceso subjetivo, que depende de las variables que seleccionemos para llevar a cabo la agrupación, y de la escala en la que se encuentren dichas variables. Para tratar de minimizar el impacto relativo a la distinta escala en que se encuentra cada una de las variables que empleemos, suele ser aconsejable normalizar nuestro dataset antes de llevar a cabo una clusterización. Sin embargo, cuantas más variables de distinta naturaleza empleemos a la vez, más difícil será obtener resultados coherentes.

Por último, usando el mismo cluster de Spark empleado para llevar a cabo todos los experimentos del capitulo 4, he realizado una prueba de clusterización usando todo el dataset de Gaia, compuesto por más de mil millones de elementos (concretamente 1.142.679.769 ejemplos), pero cogiendo solamente los datos astrométricos de los que disponemos actualmente para todos los ejemplos de dicho dataset (la ascensión recta y la declinación). La idea es comprobar si cuando contemos con todos los datos de astrometría para este dataset vamos a ser capaces de realizar una clusterización empleando los algoritmos de la librería MLlib de Spark que hemos utilizado en los experimentos llevados a cabo en este trabajo. El número máximo de iteraciones que he empleado ha sido de 20, y la “k” utilizada ha sido 50.

Los tiempos obtenidos han sido los siguientes:

- En el caso de K-Means, el algoritmo ha terminado correctamente, y ha tardado 873,39 segundos (aproximadamente 15 minutos) en realizar la clusterización.

- Para el caso de Bisecting K-Means, el tiempo empleado por el algoritmo ha sido de 5.765,44 segundos (aproximadamente 1 hora y media).

- Y por último, para el algoritmo de mezclas gaussianas, el tiempo empleado en llevar a cabo la clusterización ha sido de 11.431,26 segundos (aproximadamente 3 horas y cuarto).

Por lo tanto, las pruebas realizadas parecen indicar que podemos clusterizar sin problemas un dataset de más de 1.000 millones de ejemplos utilizando la librería MLlib de Apache Spark. Aunque es posible que cuando añadamos más dimensiones, el dataset no pueda ser almacenado en la memoria RAM de los executors de Spark, y por tanto, se tenga que realizar volcado a disco, lo cual sin duda influirá en el rendimiento de los algoritmos y en el tiempo de ejecución.

También hay que tener en cuenta que si queremos usar diferentes valores de *k* para comprobar cual es la mejor *k* para dividir nuestro dataset, el uso de 90 valores de *k* diferentes como los que empleábamos en los experimentos realizados en las fases 2, 3, y 4, el tiempo que necesitaríamos para que finalizará esta ejecución en el caso de querer emplear mezclas gaussianas sería de más de 286 horas (eso sin tener en cuenta que al aumentar el valor de k aumenta considerablemente el tiempo de ejecución en el caso de mezclas gaussianas como vimos en los experimentos realizados), lo cual sería unos 12 días de ejecución continuada en un cluster compuesto por 6 nodos con 16 cores y 64Gb de RAM cada uno.

Otra cosa que debe valorarse es que si para un dataset con 2 millones de estrellas, hemos probado tan sólo 90 valores de *k* distintos, en el caso de un dataset de más de 1.000 millones de estrellas, puede que tengamos que probar con muchos más valores de *k*, ya que lo normal es que existieran más cúmulos de estrellas en un dataset de esas dimensiones que engloba aproximadamente el 1% de estrellas de nuestra galaxia. Y por otra parte, el aumentar el número de grupos, afecta de una forma negativa a la ejecución de los algoritmos de clusterización, especialmente al algoritmo de mezclas de gaussianas, lo cual puede llevarnos a tiempos de ejecución aún más elevados.

Por tanto, si tenemos en cuenta estos detalles, el tiempo de ejecución para obtener cuál es la mejor clusterización del dataset de Gaia, utilizando un rango de *k* lo suficientemente grande, podría llevarnos varios meses de ejecución continuada. Sin embargo, al estar usando Apache Spark, siempre podríamos recurrir a un incremento del número de nodos, lo cual redundaría en una mayor velocidad de ejecución, y en la posibilidad de obtener nueva información sobre la estructura de la vía láctea.

Bibliografía

Flach, P. (2012). Machine Learning: The Art and Science of Algorithms that Make Sense of Data

Meng X., Bradley J., Yavuz B., Sparks E., Venkataraman S., Liu D., Freeman J., Zaharia M. and cols (2016). MLlib: Machine Learning in Apache Spark

Mirtaheri, S.M. y cols (2014). A Brief History of Web Crawlers

Provost, Kolluri. (1999). A Survey of Methods for Scaling Up Inductive Algorithms

Hernandez Orallo J., Ramirez Quintana M., Ferri Ramirez C. (2005). Introducción a la Minería de Datos. Pearson.

Web oficial de Apache Spark: http://spark.apache.org

Sujatha R. Upadhyaya (2013). Parallel approaches to machine learning - A comprehensive survey

Charrad, M., Ghazzali N., Boiteau V., Niknafs A. (2014). NNbClust: An R Package for Determining the Relevant Number of Clusters in a Data Set

Liu Y., Zhongmou L., Hui X. (2010). Understanding of Internal Clustering Validation Measures

Sanchez-Maroño, N., Alonso-Betanzos A., Tombilla-Sanromán, M. (2007). Filter methods for feature selection. A comparative study

Karau, H., Konwinski, A., Wendel, P., Zaharia, M. (2015). Learning Spark: Lightning-Fast Big Data Analysis

Ryza, S., Laserson, U., Owen, S., Wills, J. (2015). Advanced Analytics with Spark: Patterns for Learning from Data at Scale

Wong, K. (2015). A Short Survey on Data Clustering Algorithms

Domingos, P. (2012). A few useful things to know about Machine Learning

Luxburg, U. (2007). A Tutorial on Spectral Clustering

Lim, F., Cohen, W. (2010). Power Iteration Clustering

Berzal, F. (2012). Clustering: http://elvex.ugr.es/decsai/intelligent/slides/dm/D3%20Clustering.pdf

Bahmani, B., Moseley, B., Vattani, A., Kumar, R., Vassilvitskii, S. (2012). Scalable K-Means++

Sankararaman, S. (2009). Computer Science 294. Practical Machine Learning

Dean, J., Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters

Ghemawat, S., Gobioff, H., Leung, S. (2003). The Google File System

Barioni, M., Razente, H., Traina, A. (2006). An efficient approach to scale up k-medoid based algorithms in large databases

Martínez, V., Miralles, J., Marco, E., Galadí-Enríquez, D. (2005). Astronomía fundamental

Documentación oficial de Apache Hadoop sobre la arquitectura del HDF: https://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

Zecević, P., Bonaći, M. (2017). Spark in Action

Kauffman, L., Rousseeuw, P. (2005). Finding Groups in Data: An Introduction To Cluster Analysis

Modha, D., Spangler, W. (2002). Feature Weighting in k-Means Clustering

Ball, GH., Hall, DJ. (1965). ISODATA: A Novel Method of Data Analysis and Pattern Classification

Calinski, T., Harabasz, J. (1974). A Dendrite Method for Cluster Analysis

Hartigan, J. (1975). Clustering Algorithms

Yan, D., Huang, L., Jordan, M. (2009). Fast Approximate Spectral Clustering

Ratkowsky, DA., Lance, GN. (1978). A Criterion for Determining the Number of Groups in a Classification

Fisher, RA. (1936). The use of multiple measurements in taxonomic problems

Krzanowski, WJ., Lai, YT. (1988). A Criterion for Determining the Number of Groups in a Data Set Using Sum-of-Squares Clustering

Rand, WM. (1971). Objective criteria for the evaluation of clustering methods

Petrovsky, M.I. (2003). Outlier Detection Algorithms in Data Mining Systems

Hawkins, S., He, H., Williams, G., and Baxter, R. (2002). Outlier Detection Using Replicator Neural Networks

Borne, K. D. (2009). Scientific Data Mining in Astronomy

Chawla, S., Gionis, A. (2013). k-means-- A unified approach to clustering and outlier detection