# Clustering con Incertidumbre

Víctor Muñoz Ramírez
*Dpto. Ciencias de la Computación e Inteligencia Artificial*
*Universidad de Sevilla*
Sevilla, España
vicmunram@us.es

Enrique Reina Gutiérrez
*Dpto. Ciencias de la Computación e Inteligencia Artificial*
*Universidad de Sevilla*
Sevilla, España
enrreigut@us.es

***Resumen* – El objetivo principal del trabajo es tratar de dar un catálogo de las posibles configuraciones con el objetivo de resolver diferentes escenarios de clustering**

***Palabras claves* –**

## I. Introducción

En la gran mayoría de ámbitos, siempre hemos encontrado de forma muy útil la clasificación de datos con varios objetivos en mente como puede ser visualizar estos y ver que características comparten entre si.

Sin embargo, existe un problema, el tiempo. Dado un determinado conjunto de datos el cual posee un tamaño extenso, el tiempo que se puede llegar a invertir intentando clasificarlos de forma manual puede ser absurdamente grande.

Aquí es donde entra la ventaja de los ordenadores, pero ¿cómo? ¿Qué beneficio supondrá un ordenador en estos casos?

Es cuando entramos en la rama de Inteligencia Artificial: Machine Learning, dentro del campo de aprendizaje no supervisado, donde aparece la técnica conocida como Clustering. Esta consiste en agrupar un conjunto de objetos no "etiquetados" en subconjuntos de objetos llamados clusters. Esa agrupación se realiza bajo la premisa que son similares.

El objetivo por abordar es poder encontrar un patrón en el comportamiento del algoritmo propuesto, para poder optimizar los tiempos de ejecución y obtener a modo de catálogo que opciones nos interesan más dados ciertos escenarios.

Esto resultará en la realización del código propuesto para resolver el problema de agrupación de puntos que conformen una circunferencia junto a una batería de ejemplos con diferentes configuraciones las cuales explicaremos con el fin de dar a entender lo que sucede, por qué y cómo mejorarlo.

De esta forma, tendremos como conclusión un conjunto de decisiones empíricas para diferentes configuraciones y diferentes escenarios donde podremos ver que nos interesa realizar.

## II. Investigación

Para la realización de nuestro trabajo, investigamos respecto al funcionamiento de los diferentes algoritmos utilizados.

Según la Referencia 1:

Empieza con una pequeña introducción respecto a lo que es Clustering en Machine Learning. Una técnica para poder separar un conjunto de puntos dados. Estos puntos deben de poseer algún tipo de característica común que nos permita poder identificar unos de otros y así poder agruparlos.

Habla de 5 técnicas de algoritmos de Clustering: K-Means, Mean-Shift, Density-Based Spatial con Ruido, Expectation-Maximization (EM) usando mezcla de modelos Gaussianos (GMM) y Aglomerative Hierarchical Clustering. Cada uno presenta sus ventajas y desventajas, que iré desarrollando a lo largo de los siguientes puntos:

**K-Means:**

Este es el algoritmo que es presentado en las transparencias de teoría. Según el artículo es uno de los métodos más fácil de implementar y de entender. Lo primordialmente destacable es que el input inicial es tuyo, es decir, debes primero observar los datos y tratar de identificar el posible número o cantidad de grupos(clústeres) que puedes identificar según las densidades de los puntos.

Este algoritmo consiste, en como ya se mencionó anteriormente, seleccionar tantos puntos considerados como centros de los clusters que consideramos que hay según la representación dimensional de los datos. El algoritmo es iterativo, es decir, con cada iteración los centros se van desplazando de forma que se sitúen en el centro de cada clúster. Esto se consigue midiendo la distancia de los puntos con cada centro para ver a cuáles corresponden.

La ventaja de este método es que un algoritmo que es considerado bastante rápido con una complejidad de $\mathcal{O}(n)$. Sin embargo, partes con la premisa de que debes seleccionar una serie de clúster. Esta decisión no siempre es trivial. Una alternativa es realizar aproximaciones como, en función de la cantidad de puntos, introducir x centros, aunque puede provocar resultados diferentes, por ende resultando en una falta de consistencia.

Otra posible optimización de este algoritmo es ahorrarnos el tener que situar los puntos manualmente y simplemente indicar cuantos queremos. Esto suele realizarse mediante la inicialización aleatoria de estos puntos, no obstante, volvemos a encontrarnos con el problema de la falta de consistencia.

La solución a esta falta de consistencia sería el uso de generaciones (Referencia 2). Instanciamos varias veces los clusteres con posiciones aleatorias y nos quedamos con el que de todas las generaciones agrupe de mejor forma los puntos.

Por otro lado, la Referencia ofrece una alternativa a K-Means conocida como K-Medians. La principal diferencia es que a la hora de recomputar los centros nos basamos en la mediana y no en la media. Este método es menos sensitivo a valores atípicos sin embargo es más costoso de forma computacional ya que requiere *sorting* para computar la mediana del vector de los datos.

**Mean-Shift:**

Este algoritmo es un algoritmo basado en un enventanado que trata de encontrar las áreas más densas de puntos con el fin de determinar el centro de los clusters de puntos dados. Este algoritmo es denominado como *centroid-base algorithym*. La función de este algoritmo es encontrar el centro de los clusters formado por el conjunto de puntos dados en función de la densidad de estos.

Lo primero que se hacer es inicializar un punto de forma aleatoria en el espacio de puntos con un radio de búsqueda. La superficie cubierta por el círculo (ventana circular la cual se le conoce como*kernel* o núcleo) se encarga de contar los puntos incluidos en este. El vector dirección en el que se mueve se hace en función del centro de masa, es decir, donde se vayan concentrando los puntos.

La forma en la que este algoritmo converge es que, en vez de inicializar una única ventana, se inicializan varias distribuidas de forma uniforme por el espacio de puntos. Vamos iterando hasta que las distancia entre el centro de las ventanas sean cubiertas por todas las otras ventanas.

La ventaja de este algoritmo respecto a K-Means es que no requerimos de indicar el número de centros que vamos a necesitar, sino que se detectan solos basados en la densidad de puntos, sin embargo, esta ventaja que presenta puede llegar a ser una desventaja. Este método es perfecto para encontrar el centro de los puntos que formen parte de clusters que estén separados y sean densos. En el escenario de que los cluster conformen figuras, tipo el perímetro de un círculo, el algoritmo puede que no agrupe los puntos en único conjunto, sino que haga una subdivisión de estos. Otra desventaja que presenta es que se deben escoger la longitud del radio resultando factor determinista que puede provocar diferentes resultados.

**Density-Based Spatial Clustering con ruidos (DBSCAN):**

Este algoritmo también está basado en la agrupación de los puntos en función de su densidad. La gran diferencia que presenta respecto a Mean-Shift es que no trata de localizar el centro del conjunto de los puntos, si no que va generando la agrupación al vuelo.

El funcionamiento del algoritmo es muy simple. Sen encarga de ir recorriendo todos los puntos del conjunto de puntos y mira en un radio alrededor de este. Si se encuentra un punto dentro de este radio, se le asigna como parte del clúster. Los puntos que se leen se marcan como visitados. En el momento que ha acabado de ”agrupar un conjunto de puntos” pasa al siguiente en la lista de puntos.

Este algoritmo presenta ventajas respecto a K-Means ya que como en Mean-Shift no es necesario indicar de antemano el número de agrupaciones que identificamos o vamos a querer, además de que a difrencia de Mean-Shift es capaz de identificar figuras. Sin embargo, también tiene su desventaja la cual es que requiere cierta distancia mínima entre los puntos para poder agrupar todo en la misma figura, ya que la distancia a la que miramos, conocida como Epsilon, varía en función de la densidad de puntos en esa zona.

El artículo habla de otros dos algoritmos los cuales no consideramos relevantes para el objetivo que se nos propone, pero son muy interesantes y presentan alternativas a la hora de agrupar diferentes grupos de puntos.

## III. Metodología

### A. Dominio del problema y generación de ejemplos

En primer lugar, se implementaron aquellas clases relativas a los elementos del dominio del problema, *Point* y *Circunference*, que sirven para representar el conjunto de puntos de entrada y las resultantes circunferencias que agrupan a estos.

La primera solo almacena las coordenadas x e y de un punto, mientras que la segunda almacena el centro y radio de una circunferencia; además del conjunto de puntos usados para su representación gráfica. Ambas clases, durante el desarrollo del proyecto, fueron aumentando en tamaño al incluirse en ellas métodos necesarios para la implementación del algoritmo.

Tras la elaboración de estas clases, se decidió diseñar las necesarias para la generación de ejemplos y representación gráfica de estos mismos. El objetivo era poder comprobar la eficacia de las distintas partes del algoritmo a medida que se desarrollaba con una buena variedad de casos de prueba; pudiéndose encontrar los fallos que podrían haberse cometido y subsanarlos lo antes posible.

Estas clases son *DataSet* y *Canvas*, que nos permiten, respectivamente, realizar la carga de datos de un ejemplo a partir de un archivo .csv y elaborar y representar estos. En cuanto al formato de los archivos .csv, este viene detallado en el archivo README.txt adjunto al código. En cuanto a la generación de ejemplos, se crearon métodos aleatorios para la creación de distribuciones de circunferencias, la eliminación de un porcentaje de puntos y la inserción de ruido desviando estos. Este desvío se hace adaptando el método *numpy.random.normal* el cual proporciona valores aleatorios dentro de una distribución. Para la adaptación solo es necesario aportar la desviación que queremos en dicha distribución.

La generación de un nuevo ejemplo consiste en crear una distribución aleatoria, eliminar algunos puntos de ella y desviar los restantes para tener algo de ruido. El único problema de esta forma de actuar, es que si se queremos un ejemplo muy concreto, es necesario crearlo a mano o bien ejecutar el método hasta tener un caso similar al buscado. Como ya se ha mencionado, *Canvas* nos permite también representar dichos ejemplos mostrándonos de forma conjunta los puntos y circunferencias del problema.

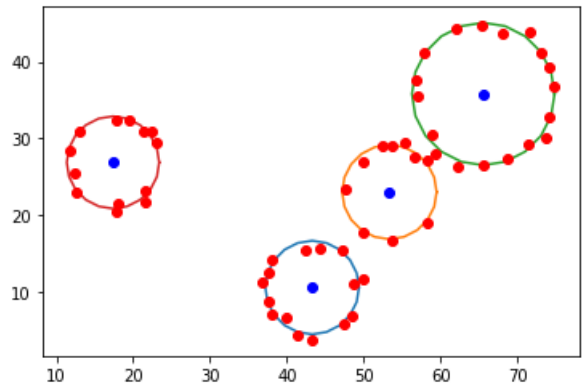


Fig. 1. Ejemplo generado de forma aleatoria, porcentaje de eliminación=30%, escala del desvío=0.5

A continuación, se realizó la implementación de los objetivos específicos relativos al código siguiendo el orden en el que aparecían en el documento; ya que iban de menos a más en la elaboración del algoritmo final.

### B. Estructura de datos

De esta manera, se diseñó para almacenar la información de cada iteración del algoritmo una estructura de datos basada en dos arrays:

- Clusters: Almacena objetos de la clase *Circunference*, los cuales representan. El orden de estos en el array viene definido por como se inicializaron.
- Threshold: Almacena arrays con los grados de pertenencia de cada punto a cada cluster. El orden de estos viene definido por como estuvieran los puntos ordenados en el archivo .csv.

### C. Cálculo de circunferencias

Tras esto, se elaboró un método para calcular centro y radio de una circunferencia a partir de un listado de puntos.

La primera versión de este método calculaba el centro como el baricentro del conjunto y el radio como la distancia media de este a todos los puntos. Este método, aunque muy eficiente incluso con ruido, fallaba cuando se le proporcionaba un arco en lugar de una circunferencia; ya que como el baricentro se sitúa en función de la densidad, el centro queda muy cerca del arco resultando en un radio menor al debido, y quedando los puntos de los extremos muy alejados de la circunferencia.

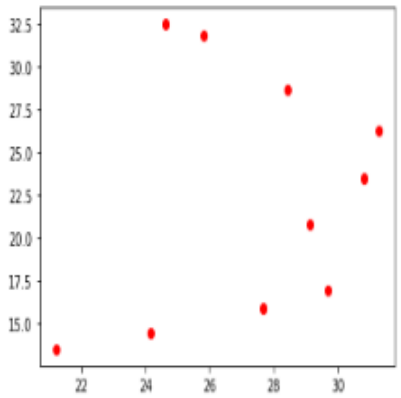


Fig. 2. Ejemplo con los puntos en forma de arco

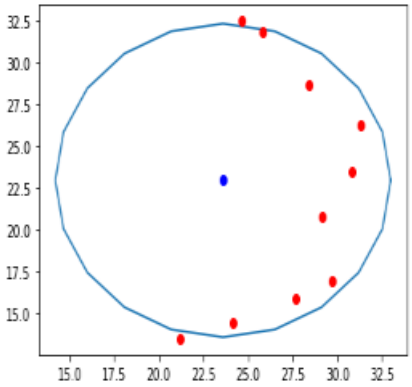


Fig. 3. Circunferencia calculada usando el baricentro

Por ello, se decidió implementar un segundo método que calculara el centro como el circuncentro del conjunto; ya que con la circunferencia circunscrita nos aseguramos que al menos pase por tres puntos de los proporcionados, y el radio como la distancia desde este a uno de esos puntos. Este se calcula como el punto de corte de las mediatrices de los segmentos que forman los puntos seleccionados, Cálculo del circuncentro. El problema de esta forma de calcularlo es que los puntos elegidos hagan que las mediatrices no se corten y no haya solución, lo cual fue solventado haciendo que el cálculo se repitiera hasta que esta existiera.

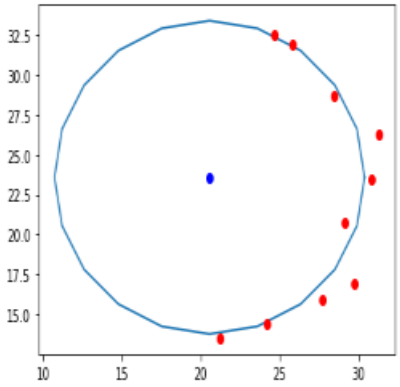


Fig. 4. Circunferencia calculada usando el circuncentro

Como vemos, la selección de los puntos es muy importante y ha sido un elemento que se ha tratado de múltiples formas. El planteamiento inicial consistía en tomar un punto aleatorio, buscar el más lejano a este y por último buscar uno a una distancia intermedia de ambos. Esta forma requería cierto coste computacional adiciona con respecto al baricentro;l ya que había que recorrer el listado de puntos más de una vez, pero reaccionaba bien incluso con arcos. Sin embargo, en situaciones con mucho ruido el resultado se desviaba. Se intentaron otras formas de seleccionar estos puntos, por ejemplo, siguiendo la estructura anterior pero siendo el último punto uno aleatorio también. Sin embargo, ninguna terminaba de funcionar; así que finalmente se optó porque todos fueran aleatorios ya que el resultado era muy similar y requería un coste computacional mucho menor.

Finalmente, se dejaron tanto el método que utiliza el baricentro como el que utiliza el circuncentro para poder comprobar con el algoritmo final como reaccionaba cada uno y cual era más eficiente.

### *D. Cálculo de grados de pertenencia*

Después, se implementó el método correspondiente al cálculo de los grados de pertenencia.

Como primer paso se calculan todas las distancias del punto a los clusters, tras lo cual se realiza un sumatorio de todas ellas. A este se le restan las distancias por separado, para que los grados de pertenencia sean inversamente proporcionales a la distancia. Y para que dichos valores esten normalizados, se dividen los resultados de la operación anterior entre el sumatorio.

Cabe destacar que durante la experimentación con el algoritmo final, en ocasiones al intentar normalizar se producían divisiones por cero; lo cual fue solventado instanciando el sumatorio de los puntos con una cantidad despreciable.

Posteriormente, se diseñó una modificación de este método que calcula los grados de pertenencia para todos los puntos de un problema, devolviendo el array correspondiente a Threshold, ya mencionado antes. Para esto se llama al método anterior por cada uno de los puntos del problema.

### *E. Implementación del esquema general*

Finalmente, se implementó la clase *ClusteringSolver* en la cual encontramos el método *learn*, el cual sigue el esquema general planteado con pequeñas modificaciones que comentaremos más adelante; pero primero vamos a centrarnos en lo que tienen en común ambas.

Para la inicialización, desde un inicio se decidió instanciar las circunferencias de forma aleatoria, concretamente, se toma como centro un punto aleatorio del listado y se le proporciona un radio también aleatorio. Además, se decidió permitir al usuario elegir si utilizar baricentro o circuncentro para recalcular las circunferencias. Aunque es cierto, que si el conjunto de puntos suministrado para un cluster es menor de tres, se utiliza el baricentro ya que el circuncentro necesita como minimo esa cantidad.

En cuanto al criterio de parada, primero se implementó con un número fijo de iteraciones ya que era más sencillo y permitía comprobar el comportamiento del algoritmo con facilidad; aunque finalmente se acabó implementando un criterio de parada basado en que los centros y radios de los clusters no variaran en una cantidad determinada por el usuario, correspondiente al parámetro *precision*. Sin embargo, es necesario aportar un número máximo de iteraciones pues en el caso de utilizar el circuncentro para recalcular los clusters, al tomarse puntos aleatorios, los centros y radios pueden aumentar sustancialmente. Este problema no se produce al utilizar el baricentro.

Por último, para la presentación final de los datos, se creó la clase *Solution*; la cual utiliza la información contenida en un objeto *ClusteringSolver* que ya ha realizado el método *learn*, es decir, ya ha obtenido el resultado del algoritmo. Esta clase nos permite eliminar puntos cuyo grado de pertenencia final sea menor al elegido por el usuario, además de representar los datos de forma gráfica mostrando las circunferencias junto con los puntos coloreados en función del cluster al que pertenecen. Actualmente, esta representación tiene como límite siete clusters, ya que a partir del octavo los colores se repetirán y no se podrá saber con certeza si la asignación es correcta. Aún así, es facilmente escalable.

Las modificaciones se realizaron una vez implementado todo. Realizando la experimentación se observó que la inicialización de los clusters al ser aleatoria condicionaba mucho el resultado final del algoritmo. De modo, que se decidió investigar sobre ello y se encontró que una técnica bastante utilizada era el lanzamiento del algoritmo con múltiples instancias haciendo uso de una heurística para obtener el mejor resultado de todas ellas. En nuestro caso, la heurística considera un mejor resultado aquel en el que el sumatorio de las distancias de los puntos a los clusters es menor.

### *F. Clases principales y sus métodos*

Clase *Point*:

Esta clase como su nombre indica representa un punto. Como constructor recibe las coordenadas del eje OX y el eje OY.

Como métodos adicionales en esta clase tenemos:

- **getFurthestPoints(points)** : recibe como parámetro el conjunto de puntos y devuelve al más lejano de estos.
- **calculateDistanceToCluster(c)**: recibe como parámetro el cluster que queramos y con este llamamos al método auxiliar calculateDistanceToPoint(c) el cual recibe como parámetro el centro de ese cluster. El método deveulve la distancia al centro y simplemente le restamos el radio del cluster en valor absoluto.
- **calculateDistanceToCluster_noAbs(c)**: recibe como parámetro el cluster que queramos y se comporta igual que el método anterior sin embargo en este método la distancia al cluster no está en valor absoluto.
- **calculateThreshold(clusters)**: recibe como parámetro el conjunto de clusters y devuelve el grado de pertenencia a estos. Para cada cluster calculamos la distancia y normalizamos de forma inversa el vector que resulta.
- **calculatePerpendicular(p)**: Recibe como parámetro otro punto y se devuelve el vector perpendicular al vector que resulta del punto al punto p.
- **calculateDistanceToPoint(p)**: recibe como parámetro un punto y se calcula la distancia euclídea a este.
- **equals(p)**: recibe como parámetro un punto y se comparan las componentes una a una para ver si son iguales.
- **print()**: se imprime las componentes del punto con el formato “(x,y)”.

Clase *Circunference*:

Esta clase como su nombre indica, representa una circunferencia. Como constructor recibe un centro, un radio y un número de puntos (precisión a la hora de representar la

circunferencia).

Como métodos adicionales en esta clase tenemos:

- **__calculate()**: Método privado donde calculamos una representación de la circunferencia acorde con los parámetros que se pasan en el constructor.
- **update(centre, radius)**: Recibe como parámetro un centro y un radio que serán los valores a actualizar.
- **draw()**: Coge los valores calculados en el método calculate() y le damos una representación 2D.
- **drawPlot()**: Similar a draw() con la diferencia que se utiliza en otro contexto.
- **drawSubPlot(axs, it)**: Similar a draw() y recibe como parámetro unos eje (tipo de la librería matplotlib) y la iteración por la que va el algoritmo de aprendizaje.
- **calculateWithCircumcenter(points)**: Recibe como parámetro el conjunto de puntos de entre los cuales se escogen tres de forma aleatoria para realizar un circuncentro de estos y hacer una representación del cluster.
- **calculateWithBarycentre(list_points)**: Recibe como parámetro la lista de puntos y se halla el baricentro haciendo una suma de las coordenadas del listado de puntos y dividiendo entre el total de puntos. Para el radio hacemos una media de las distancias entre el baricentro y todos los puntos.

Clase *Clustering Solver*:

Esta clase es la encargada de llevar a cabo el algoritmo de aprendizaje. Como constructor recibe una ruta a un fichero csv que contiene los puntos los cuales le queremos hacer clustering y la cantidad de custers con los que queremos que resuelva los puntos.

Como métodos adicionales tenemos:

- **initRandomClusters()**: Inicializa de forma aleatoria los clusters iniciales.
- **calculateThreshold()**: Calcula el grado de pertenencia de todos los puntos a cada cluster.
- **getClusterAssignment(threshold)**: Recibe como parámetro los grados de pertencia de cada punto a cada cluster. Con estos datos recorremos el vector de grados de pertenencia y a cada punto le asignamos un cluster según su pertenencia.
- **groupPointsByCluster(cluserAssignment)**: Recibe como parámetro el vector de tuplas (punto, cluster asigando) del método getClusterAssignment(threshold). Iteramos por todo el vector y vamos agrupando en función del cluster asignado. Devuelve un array de arrays con tantos grupos como clusteres con los puntos asignado a ese cluster.
- **calculateCluster(mode,points)**: Recibe como parámetro un modo que sirve para decidir si el cluster se va a formar mediante baricentro o circuncentro (mode puede ser 'b' o 'c') y los puntos asignado a ese cluster para formar la circunferencia que los agrupe.
- **learn(generations,precision,limit,mode)**: Recibe como parámetro el número de generaciones del algoritmo (cuantas veces se van a inicializar los clusters), la precisión que indica la mínima variación de una iteración a otra para que se considere que el algoritmos ha convergido, el límite que es el número de iteraciones máximas en el caso de que se encuentre en un bucle y el modo que al igual que en el método calculateCluster(mode,points) que define la técnica para definir la circunferencia que represente al cluster.

  Este es el algorimto principal. Nos limitamos a instanciar un número de inicializaciones (generaciones) y vamos iterando hasta que converjan o se alcance el número de iteraciones máximas. Por cada generación nos guardamos la mejor iteración y de estas, comparamos la mejor para devolver la mejor asignación.

Clase *Solution*:

Esta clase representa la solución final de haber realizado el algoritmo de asignación de clusters a los puntos, por ello el constructor recibe como parámetro la clase ClusteringSolver. Se limita a devolver el resultado aplicando un filtrado de detección de ruido, para que solo se devuelvan los puntos que realmente considera como suyos.

Como métodos adicionales tenemos:

- **eraseNoise(minThreshold)**: Recibe como parámetro el mínimo grado de pertenencia al cluster para que este se represente y no se considere como ruido. En caso de no cumplirse el grado de pertenencia mínimo ese punto es borrado.
- **drawSolution(minThreshold)**: Recibe como parámetro el mínimo grado de pertenencia ya que se llama al método auxiliar eraseNoise(minThreshold). Tras eliminar los puntos simplemente nos limitamos a dibujar los puntos con colores para que se pueda ver como se realizó la asignación de puntos a cada cluster.

### *G. Elementos descartados*

**Método para la selección de los tres puntos para realización del circuncentro:**

Originalmente habíamos diseñado un método que según cierta heurística era capaz de calcular el circuncentro de 3 puntos dados, sin embargo, siempre nos encontramos con el mismo problema de forma recursiva: no se escogían los puntos de la forma que queríamos.

Esto nos empujó a una solución un tanto radical que fue la selección de los puntos de forma aleatoria. Tal como se había planteado el algoritmo, esto no iban a suponer ningún problema ya que introdujimos generaciones, de forma que en alguna de ellas los puntos escogidos serían los ideales y efectivamente funcionaba. Sin embargo, esto provocó un problema: Ciclos. Había un determinado momento en el que, por iteración al escogerse los puntos de forma aleatoria, los clusters variaban lo justo y necesario como para no converger.

Ante este problema teníamos tres soluciones: implementar un número máximo de iteraciones, subir el grado de precisión a valores poco precisos para considerar la convergencia o volver a intentar la selección de 3 puntos alejados entre sí de forma heurística.

Tras algunos intentos fallidos tratando de implementar alguna heurística para escoger siempre los mismos puntos para formar el cluster, decidimos mantener el algoritmo que escoge tres puntos aleatorios introduciendo un número

máximo de iteraciones.

Presenta la ventaja de que el tiempo que invierte en generarse el cluster es siempre constante a expensas de que va a tardar en converger, alcanzando para la gran mayoría de casos las iteraciones máximas permitidas.

**Método iterativo:**

Tal como se indica en la propuest del trabajo, existe la mejora de cambiar el código de iteración para que no se tenga que indicar un número de iteraciones con el fin de optimizar el algoritmo.

Este cambio se realiza simplemente introduciendo la precisión para la cual quieres que el algoritmo deje de iterar.

En nuestro caso, debido al problema explicado anteriormente solo se nos ve beneficiado en el caso en el que los clusters se formen con el total de los puntos (baricentros).

### *H. Mejoras*

Como mejora, se implementó una interfaz gráfica dentro de Jupyter Notebook, que nos permite crear ejemplos aleatorios, eliminar de estos puntos, insertar ruido y guardarlos como fichero .csv en la ruta deseada; así como la resolución de estos indicando el número de instancias, clusters, precisión y máximo de iteraciones. Además, se aporta otra interfaz gráfica en la que se puede cargar un archivo .csv, al cual se le puede eliminar puntos y añadir ruidos, y resolverlo con los parámetros antes mencionados. Estos archivos siguen el formato indicado en el fichero README.txt.

Todo esto se proporciona al usuario a través de la clase *ClusteringUI* y con los métodos *createAndSolve* and *loadAndSolve* que facilitan las funcionalidades anteriores. Esta interfaz deja al usuario de elegir los valores descritos anteriormente a través de sliders; así como insertarlos a mano.

## REFERENCIAS


[1] Seif, G., Undated. The 5 Clustering Algorithms Data Scientists Need To Know. [online] Medium. Available at: https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68 [Accessed 18 April 2020].

[2] Seif, G., 2020. SuperProf Material Didáctico - Circuncentro. [online] Medium. Available at: https://www.superprof.es/apuntes/escolar/matematicas/analitica/recta/circuncentro.html [Accessed 22 April 2020].

[3] Zaiontz C., Undated. The 5 Clustering Algorithms Data Scientists Need To Know. [online] Medium. Available at: http://www.real-statistics.com/multivariate-statistics/cluster-analysis/initializing-clusters-k-means/ [Accessed 11 June 2020].