Universidad de La Laguna

Máster Universitario en Ciberseguridad e Inteligencia de Datos

Algoritmo de agrupamiento espacial: GDBSCAN.

Spatial clustering algorithm: GDBSCAN.

Yeray Expósito García

La Laguna, 4 de abril del 2021

D. **José Marcos Moreno Vega**, con N.I.F. 42.841.047-M Catedrático de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

D. **María Belén Melián Batista**, con N.I.F. 44.311.040-E Catedrática de Universidad adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor.

CERTIFICA(N)

Que la presente memoria titulada:

"Algoritmo de agrupamiento espacial: GDBSCAN"

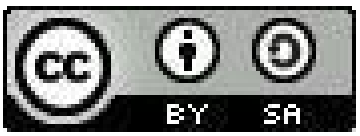
ha sido realizada bajo su dirección por D. **Yeray Expósito García**, con N.I.F. 43.381.717-Z .

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a Fecha

Agradecimientos

Debo de agradecer el apoyo ofrecido por mi familia desde el inicio. A mis amigos del máster por acompañarme y ayudarme en este desafío. Y a mi tutor Marcos y a mi cotutora Belén por compartir su conocimiento conmigo a lo largo del presente proyecto.

Licencia



Resumen

El objetivo de este trabajo ha sido estudiar el algoritmo GDBSCAN y comparar su funcionamiento con el del algoritmo DBSCAN, el cual es una especificación del primero. Para ello se realizará un profundo estudio de ambos algoritmos, haciendo hincapié en sus conceptos específicos, su funcionamiento y sobre cómo estimar correctamente los parámetros que rigen su funcionamiento.

Con el fin de mostrar todo el potencial del algoritmo GDBSCAN se desarrollará un dataset artificial que reúna las características necesarias para tal fin, y se mostrarán las diferentes opciones que ofrece el uso de atributos no espaciales a la hora de detectar clusters de datos. Para proporcionar un enfoque más realista, también se realizarán pruebas sobre una base de datos real.

Por otro lado, se analizará el rendimiento y funcionamiento del algoritmo. Para ello se observará el comportamiento del algoritmo frente a aspectos muy concretos presentes en los conjuntos de datos, como por ejemplo la presencia o ausencia de ruido, el número de clusters existentes, etc.

Por último, se prestará atención a todos aquellos aspectos del algoritmo que sean potencialmente mejorables o que permitan añadir algún componente adicional con el fin de mejorar su rendimiento.

Palabras clave: GDBSCAN, DBSCAN, clustering, agrupamiento, dataset, datos espaciales, aprendizaje no supervisado, vecindad, cardinalidad, atributos no espaciales.

Abstract

The objective of this work is to study the GDBSCAN algorithm and compare it with the DBSCAN algorithm, which is a specification of the former. For this, both algorithms will be studied, emphasizing their specific concepts, their operation and how to correctly estimate the parameters for their operation.

On the other hand, an artificial dataset has been created with the necessary characteristics to show the potential of the GDBSCAN algorithm. GDBSCAN allows the use of non-spatial attributes to detect data clusters, which offers a wide variety of options to work. To provide a more realistic approach, different tests will be performed on a real database.

In the third place, the performance and operation of the algorithm will be analyzed, for this the algorithm behavior will be observed in data sets with very specific aspects such as the presence or absence of noise, the number of clusters, etc.

Finally, the algorithm will be analyzed to find those aspects that can be potentially improved or that allow adding a component to improve its performance.

Keywords: GDBSCAN, DBSCAN, clustering, dataset, spatial data, unsupervised learning, neighborhood, cardinality, non-spatial attributes.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1 Introducción

El mundo en el que vivimos genera de manera incesante grandes cantidades de datos e información. Y solo ha sido cuestión de tiempo que el ser humano depositara sus ojos en este campo y se enfrentara a los desafíos tanto a nivel computacional como de interpretación, que implica el análisis de esa ingente cantidad de información y la posterior extracción de conocimiento. Su potencial es tal, que hoy en día se le ha encontrado aplicación en una gran cantidad de ámbitos, algunos de especial relevancia como la salud. Y es por este motivo, que la minería de datos se está convirtiendo en uno de los trabajos con mayor proyección de futuro.

1.1 Antecedentes

A pesar de que el concepto de minería de datos puede parecer novedoso, en realidad surgió en los años sesenta junto a otros términos como data fishing o data archeology. Sin embargo, fue en los años ochenta cuando comenzó a consolidarse de la mano de Rakesh Agrawal, Gio Wiederhold, Robert Blum y Gregory Piatetsky-Shapiro. La minería de datos apareció con el fin de ayudar a entender las enormes cantidades de datos que se generan en el mundo actual de manera diaria, para así extraer conocimiento que pueda ser aprovechado por las empresas u otros organismos para obtener conclusiones y mejorar su crecimiento. A finales de los ochenta sólo existían unas pocas empresas dedicadas a esta tecnología, sin embargo, con el tiempo el número se ha incrementado, hasta que en el 2002 se contabilizaron más de 100 empresas operando en este ámbito [1].

La minería de datos se puede considerar como una de las fases que componen un proceso mucho más amplio, denominado proceso de extracción de conocimiento en bases de datos. Como se mencionó, su objetivo es explorar grandes bases de datos de forma automática, a través del empleo de diferentes tecnologías, con el fin de localizar reglas, tendencias o patrones que se repitan que permitan entender el comportamiento de los datos recogidos a lo largo del tiempo. La detección de dichos patrones puede lograrse empleando algoritmos de búsqueda cercanos a la inteligencia artificial o mediante el empleo de estadísticas. Las personas que llevan a cabo estas tareas se denominan mineros de datos. En los últimos años, este tipo de trabajos se están realizando en una amplia variedad de campos, como coches inteligentes, procesamiento del lenguaje natural, búsquedas online, finanzas, marketing, salud, identificación de fraude, etc [2].

1.2 Objetivos

Los objetivos principales del presente proyecto son:

1. Entender mejor en qué se basa el agrupamiento espacial y en qué se diferencia del resto de agrupamientos.
2. Estudiar el algoritmo DBSCAN.
3. Estudiar el algoritmo GDBSCAN considerando no sólo aquellos atributos que sean espaciales, sino también los no espaciales, y cómo repercuten a la hora de detectar los clusters de datos.
4. Generar artificialmente un conjunto de datos que reúna las características necesarias para poder mostrar todo el potencial del algoritmo GDBSCAN.
5. Aplicar el algoritmo GDBSCAN sobre un dataset real.
6. Realizar un análisis de rendimiento del algoritmo con grandes bases de datos e intentar proponer una mejora con el fin de lograr que sea más eficiente.

Capítulo 2 Estado del arte

2.1 Proceso de extracción de conocimiento.

El proceso de extracción de conocimiento en base de datos se puede definir como un proceso complejo basado en la identificación de patrones válidos, nuevos, útiles y comprensibles a partir de datos. El conocimiento obtenido debe poseer las siguientes propiedades;

- ➢ Debe poder comprenderse con cierta facilidad.
- ➢ El modelo o patrón identificado debe presentar un nivel de precisión adecuado al emplearse con datos nuevos.
- ➢ El conocimiento adquirido debe servir para mejorar el sistema o para tomar decisiones que beneficien al usuario.
- ➢ El conocimiento obtenido tiene que ser novedoso, es decir, no debe ser conocido antes de su obtención.

Este proceso se puede dividir en cinco fases. La primera, es la fase de **integración y recopilación de datos**, la cual, como su propio nombre indica, abarca la obtención de los datos desde múltiples fuentes y la integración de los mismos en un almacén de datos siguiendo un esquema unificado.

En segundo lugar, se encuentra la fase de **preprocesamiento de datos**, en la cual se busca mejorar la calidad de los datos, para ello se aplican técnicas de limpieza, normalización y transformación de datos, identificación de ruido, selección de características, selección de instancias, reducción de dimensionalidad, etc.

En tercer lugar, se encuentra la fase de **minería de datos**, en la cual se contextualiza el presente proyecto. En esta fase se obtiene nuevo conocimiento a partir de los datos que ya han sido preprocesados, para ello se construye un modelo, el cual tiene como fin describir los patrones y relaciones encontrados. Este modelo puede ser descriptivo, permitiendo entender los datos o explicar situaciones pasadas, o predictivo, permitiendo predecir el valor de una o varias variables. Con el fin de solucionar un problema de predicción, primero se debe identificar la tarea que mejor se adapta, en ese aspecto se distinguen las tareas de clasificación, agrupamiento, detección de outlier, regla de asociación, etc. A continuación, se deberá elegir el modelo que resuelva la tarea, entre los que se encuentran los árboles de decisión, los clasificadores bayesianos, el agrupamiento jerárquico, etc. Y, por último, se deberá seleccionar el algoritmo que construirá el modelo escogido anteriormente.

En cuarto lugar, se encuentra la fase de **evaluación e interpretación**, en la que el modelo se evalúa con el fin de medir su calidad. Para ello, en el caso de los modelos predictivos, se deben definir unas etapas de entrenamiento y validación. En este aspecto, se distingue entre datos de entrenamiento, formados por los datos que se emplearán para construir el modelo, y los datos de validación, que permiten validar el modelo obtenido en la etapa anterior. De esta forma, es posible distinguir diferentes técnicas de evaluación, como la validación simple, la validación cruzada y la validación cruzada con k pliegues.

Por último, se encuentra la fase de **difusión, uso y monitorización**. En esta fase el conocimiento extraído se difunde entre los miembros de la organización, por lo que pasa a formar parte de su "know-how". Además el modelo obtenido y el conocimiento que de él se obtiene deben ser monitorizados para detectar los cambios que se puedan producir[3].

2.2 Introducción a la minería de datos.

Como se indicó anteriormente, la **minería de datos** se puede definir como la tarea de encontrar correlaciones, anomalías y patrones en conjuntos de datos de gran tamaño, con el objetivo de predecir resultados. Para ello se suelen emplear métodos de aprendizaje automático, sistemas de bases de datos y estadística.

El **aprendizaje automático** o **machine learning**, se trata de una rama de la inteligencia artificial y un subcampo de las ciencias de la computación, que tiene como fin la creación de técnicas que logren que las computadoras aprendan [4]. El aprendizaje automático se especializa en la creación de programas informáticos que son capaces de detectar patrones en los datos y ajustar sus acciones en consecuencia [5]. Los algoritmos de machine learning se clasifican habitualmente en algoritmos de aprendizaje supervisado y no supervisado, aunque también es posible distinguir otros algoritmos como los de aprendizaje semisupervisado, aprendizaje por refuerzo y los algoritmos de deep learning.

2.2.1 Aprendizaje supervisado.

El **aprendizaje supervisado** abarca aquellas técnicas que permiten deducir una función a partir de datos de entrenamiento. Dichos datos consisten en conjuntos de objetos caracterizados por el par de atributos (X,Y), donde X es un vector de características (datos de entrada) e Y es el resultado. La salida de la función puede ser un valor numérico (regresión) o una etiqueta de clase (clasificación). Para obtener dicha salida se tiene que generalizar, a partir de los ya mencionados datos de entrenamiento, las situaciones no vistas previamente.

La tarea de **clasificación** consiste en aprender una función F que asocia a cada vector de características X una de las clases predefinidas Y. Existen diferentes modelos que permiten cumplir con dicha tarea, como los árboles de clasificación (algoritmo Hunt, ID3 y C4.5) [6], clasificadores bayesianos (Naive-Bayes), clasificadores basados en reglas (RIPPER), etc. En cambio la **regresión** tiene como objetivo predecir valores numéricos continuos, para ello se emplean modelos como los árboles de regresión, random forest, KNN (K Nearest Neighbors), etc [7].

2.2.2 Aprendizaje no supervisado.

El **aprendizaje no supervisado** se caracteriza por el hecho de no requerir de un entrenamiento previo, es decir, realiza su trabajo directamente con los datos a su disposición [8]. Estos algoritmos operan con datos no etiquetados y su objetivo es la exploración.

El aprendizaje no supervisado intenta encontrar patrones desconocidos en los datos, sin embargo, gran parte de los patrones que se identifican no son más que aproximaciones de lo que es posible conseguir mediante el aprendizaje supervisado, pues se desconocen cuáles deberían ser los resultados y, por consiguiente, no se puede saber la precisión de los mismos. Esto lleva a que el aprendizaje supervisado sea más aplicable.

Teniendo en cuenta estas consideraciones, es posible determinar que la situación más idónea para aplicar el aprendizaje no supervisado es aquella en la que no se posee de los resultados (etiquetas) asociados a cada observación. Indicar que los datos no etiquetados son más fáciles de obtener que los etiquetados.

Dado que en los métodos no supervisados, como ya se mencionó, se desconocen los datos de salida (etiquetas), no es posible emplearlos directamente en una tarea de regresión o clasificación, pues resulta imposible entrenar el algoritmo. Lo cual constituye una de las principales diferencias respecto al aprendizaje supervisado. Sin embargo, el aprendizaje sin supervisión permite realizar tareas de procesamiento más complejas, gracias a que se centra en el descubrimiento de la estructura subyacente de los datos. Un ejemplo de ello es la agrupación de datos no estructurados según sus similitudes y patrones distintos en el conjunto de datos [12].

2.2.3 Aprendizaje semisupervisado.

El aprendizaje semisupervisado es una técnica en la que se emplea un conjunto de datos de entrenamiento que está constituido, habitualmente, por una pequeña cantidad de datos etiquetados y por una gran cantidad de datos no etiquetados. Estos algoritmos analizan los

datos no etiquetados, en concreto su información estructural, con el objetivo de obtener modelos predictivos más eficientes que los que emplean únicamente datos etiquetados. Por lo que el aprendizaje semisupervisado está a medio camino entre el aprendizaje no supervisado y el supervisado [14].

2.2.4 Aprendizaje por refuerzo.

En la psicología conductista se emplea el concepto de "condicionamiento operante", al que se define como un proceso de aprendizaje por el cual una acción en particular es seguida por algo deseable, con el fin de lograr que el sujeto repita la acción, o por algo no deseable, para evitar que se vuelva a repetir. El **aprendizaje por refuerzo** se basa en aplicar dicho concepto a las inteligencias artificiales con el fin de que puedan aprender por sí mismas.

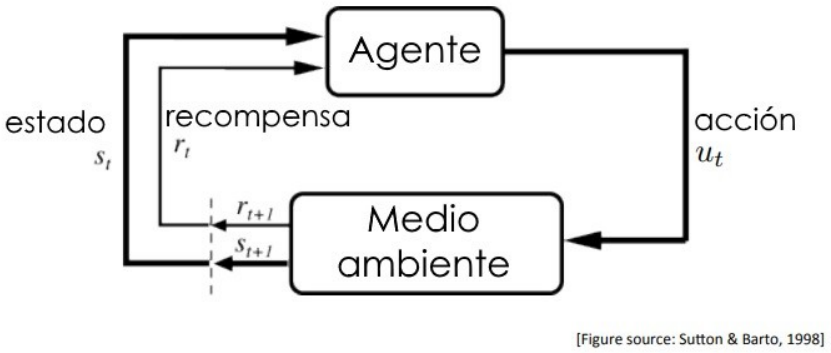


Figura 2.1: Esquema de aprendizaje por refuerzo [15].

Para formular un problema de aprendizaje por refuerzo se necesita un **agente** en un **estado** determinado dentro de un **medio ambiente**, y una **recompensa** positiva y negativa en base a la **acción** que se realice. En conclusión, como se puede ver en la imágen anterior, se basa en un sistema de retroalimentación, en el que sus entradas son las consecuencias de sus respuestas en el exterior, de manera que el sistema aprende a base de ensayo y error, pues para cada una de sus acciones recibe una recompensa o castigo que usa para completar su conocimiento sobre el entorno hasta encontrar la mejor recompensa posible, lo que indica que el sistema ha identificado los pasos necesario para alcanzar el resultado correcto [15].

2.2.5 Deep learning.

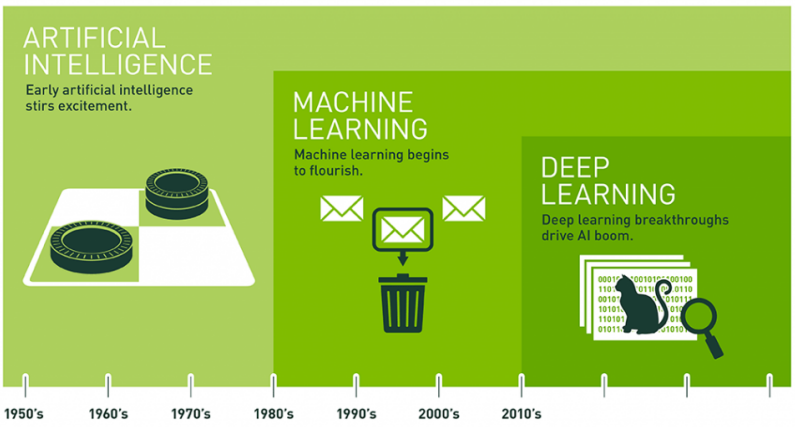


Figura 2.2: Comparación de conceptos [21].

Los algoritmos **deep learning** se caracterizan por emplear redes neuronales para detectar características ocultas de los datos que permitan definir un resultado que almacena en una estructura de neuronas. Esto supone un acercamiento hacia la forma de pensar del ser humano, pues se intenta emular el funcionamiento del sistema nervioso.

Dicha estructura neuronal se organiza en capas o niveles jerárquicos, en las que cada neurona dispone de una conexión con cada una de las presentes en la capa siguiente. Dicha estructura puede estar formada por cientos de capas, cada una de las cuales está dotada de unos pesos que afectan a la transformación que sufren los datos en cada una de ellas a medida que se avanza. Estas capas son capaces de aprender automáticamente mediante el entrenamiento con datos. Para ello se requiere de una **función de pérdida** que se encarga de calcular la distancia que existe entre la predicción de la red neuronal y el resultado esperado, de manera que el valor de retorno de dicha función se emplea para ajustar los pesos de cada capa con el fin de mejorar el resultado ofrecido por la red neuronal. De esta labor se encarga el optimizador, a través de la implementación del método de **propagación hacia atrás**, el cual recibe el valor de error y lo propaga desde la capa de salida hacia las neuronas de la capa oculta (zona intermedia), de manera que cada neurona recibe una fracción de la señal de error proporcional a su contribución en el resultado ofrecido, que usa para mejorar su rendimiento y de esta forma aprender [22].

2.3 Algoritmos de clustering.

Como se indicó anteriormente, los algoritmos de **clustering o agrupamiento**, se encuentran enmarcados dentro de las técnicas de aprendizaje no supervisado. Las tareas de agrupamiento se basan en agrupar los elementos de un conjunto en grupos (cluster) que tengan un significado o sean útiles. Ello implica que dichos grupos están formados por elementos similares o relacionados entre sí. Se pueden distinguir cuatro tipos de técnicas de agrupamiento.

2.3.1 Clustering basado en prototipos.

En primer lugar, se encuentra el **clustering basado en prototipos** [9]. Uno de los algoritmos más conocidos es el K-means, en el cual inicialmente se definen K centroides, de manera que a cada uno se le asignan aquellos puntos más próximos para formar los clusters. Tras ello se recalcula el centroide de cada cluster y se repiten los dos pasos anteriores hasta que los puntos no cambien de cluster.

El comportamiento del algoritmo K-means depende de los centroides iniciales elegidos, el uso de centroides iniciales distintos implica que se obtendrán agrupamientos diferentes. En general, el algoritmo proporciona un agrupamiento que es localmente óptimo.

Es común que los centroides iniciales se generen aleatoriamente, lo que puede suministrar agrupamientos de baja calidad. Con el objetivo de incrementar la calidad del agrupamiento final, se puede ejecutar varias veces el algoritmo K-means con diferentes centroides iniciales, y se selecciona el agrupamiento con menor suma de errores al cuadrado. Otra alternativa consiste en seleccionar, iterativamente, como nuevo centroide al punto más alejado de los centroides actuales.

2.3.2 Clustering jerárquico.

Una de las mayores dificultades al agrupar elementos es encontrar el número apropiado de clusters. Estos algoritmos, generalmente, crean una estructura que permite agrupar objetos en conjuntos de tamaño cada vez mayor hasta que se obtiene un conjunto que engloba todos esos objetos. De manera que lo que se puede observar son las relaciones de proximidad existentes entre los diferentes elementos.

Es posible distinguir dos procedimientos básicos, el **aglomerativo** en el que inicialmente se forman clusters individuales con un solo elemento. En cada iteración se fusionan los dos clusters más cercanos, y se finaliza cuando sólo hay un cluster. Y el **divisivo**, en el que se

parte de un único cluster al que pertenecen todos los elementos. En cada iteración se escoge un cluster y se divide. Finaliza cuando haya tantos clusters como elementos.

Para llevar a cabo la representación del clustering jerárquico es frecuente emplear un dendrograma, el cual permite observar el nivel de proximidad de los agrupamientos que se fusionan y el orden en el que lleva a cabo la unión de dichos clusters. A través del dendrograma es posible generar diferentes agrupamientos realizando una poda a lo largo de uno de los niveles del árbol que forma dicho dendrograma. En función del nivel escogido se conseguirán clusters más o menos compactos [10].

Existen diferentes medidas de proximidad entre clusters, a continuación se definen algunas de ellas;

1. **Enlace simple:** La proximidad entre dos clusters se define como la proximidad entre los dos elementos más próximos que pertenecen a clusters diferentes.
2. **Enlace completo:** La proximidad entre dos clusters se define como la proximidad entre los dos elementos menos próximos que pertenecen a clusters diferentes.
3. **Promedio del grupo:** La proximidad entre dos clusters se define como la proximidad promedio entre todos los pares de elementos que pertenecen a clusters diferentes.
4. **Proximidad basada en centroides:** La proximidad entre dos clusters se define como la proximidad entre los centroides de cada cluster.
5. **Método de Ward:** La medida de proximidad usada es la suma de errores al cuadrado. De esta forma, en cada fase, se unen los dos clusters que dan lugar al cluster con menor suma de errores al cuadrado.

2.3.3 Clustering basado en distribuciones.

En este tipo de algoritmos de clustering lo que se busca, desde el punto de vista bayesiano, es encontrar los clusters más probables según los datos de los que se dispone. En este caso cada dato posee una determinada probabilidad de pertenecer a un grupo. Estos algoritmos se basan en el modelo estadístico *finite mixtures*, según el cual una mezcla es un grupo de X distribuciones, que representan X clusters.

Dichas distribuciones indican la probabilidad de que un dato, si se conociera que forma parte de ese cluster, tome unos valores particulares en sus atributos. El objetivo es obtener las X distribuciones normales (varianzas y medias) y las probabilidades asociadas a cada distribución. Un ejemplo de algoritmo de agrupamiento basado en distribuciones es el EM o *Expectation Maximization* [20].

2.3.4 Clustering basado en densidad.

Este tipo de algoritmos permiten identificar regiones de alta densidad que están rodeadas de áreas poco densas. Es decir, se basan en la detección de aquellas áreas con mayor concentración de puntos que se encuentran separadas por áreas vacías o con escasos puntos. Cada una de las regiones densas identificadas se corresponde con un cluster. Los puntos que no forman parte de un cluster se etiquetan como ruido [11]. Este tipo de clustering es apropiado cuando los clusters no tienen una forma geométrica definida. Algunos ejemplos de algoritmos de agrupamiento basado en densidad son; DBSCAN, OPTICS, EnDBSCAN, DENCLUE y SNN Clustering.

Este es el tipo de algoritmo de agrupamiento en el que se centrará el presente proyecto. A modo de introducción se estudiará primero el algoritmo DBSCAN y, posteriormente, nos centraremos en el algoritmo GDBSCAN, el cual permite emplear características no espaciales en el proceso de identificación de los clusters.

2.4 Algunos algoritmos de clustering basado en densidad.

Además de los algoritmos DBSCAN y GDBSCAN que se estudiarán en este proyecto, existen muchos otros algoritmos de clustering basados en densidad, como los que se mencionan a continuación;

2.4.1 OPTICS.

OPTICS es un algoritmo agrupamiento basado en densidad que fue elaborado por Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel y Jörg Sander. Parte de una idea base muy parecida a DBSCAN, con la diferencia de que este algoritmo pone solución a uno de los principales problemas de DBSCAN a la hora de identificar agrupaciones significativas en datos de densidad variable. Para ello, emplea la distancia entre puntos vecinos para elaborar un diagrama de accesibilidad, el cual se emplea para distinguir los agrupamientos de diferentes densidades respecto al ruido. En dicho diagrama los puntos de la base de datos se ordenan en función de su distancia de accesibilidad al vecino más cercano. Lo que da lugar a la formación de valles, cada uno de los cuales representa a un agrupamiento diferente. Dichos valles se forman debido a que las distancias entre puntos de un clúster son menores que las que presenta el ruido. Lo que da lugar a un resultado como el de la siguiente imagen [16][28].

Figura 2.3: Ejemplo de diagrama de accesibilidad [28].

2.4.2 HDBSCAN.

Se trata de un algoritmo de clustering creado por Campello, Moulavi y Sander. Éste extiende DBSCAN convirtiéndolo en un algoritmo de agrupamiento jerárquico [17]. Mientras DBSCAN utiliza una distancia específica para separar los clústeres densos del ruido más disperso, HDBSCAN utiliza diferentes distancias para llevar a cabo la separación de clusters de densidades variables. HDBSCAN es el algoritmo de agrupamiento que más se basa en los datos y que, por ello, necesita una menor intervención del usuario [18].

2.4.3 DENCLUE.

DENCLUE es un algoritmo de agrupamiento basado en densidad que identifica grupos de áreas densas y poco densas. Emplea un modelo de clúster basado en la estimación de la densidad del kernel. El algoritmo calcula los máximos locales del conjunto de datos, de manera que cada grupo se define por un máximo local de la función de densidad estimada. Las observaciones que llegan al mismo máximo local se colocarán en el mismo grupo. Por ello DENCLUE no funciona bien con datos con distribución uniforme. DENCLUE es un buen algoritmo para conjuntos de datos con mucho ruido porque permite una descripción centralizada de grupos de forma irregular en un conjunto de datos de alta dimensión, identificando valores atípicos como puntos de datos con baja cardinalidad y excluyéndolos de modo que solo se agrupen los puntos de datos relevantes [19].

Capítulo 3 Metodología

Con el fin de llevar a cabo la planificación del proyecto se realizaron diferentes reuniones con el tutor y el cotutor. Primero se definieron los objetivos que se pretendían cumplir y a continuación se ubicaron temporalmente las tareas que serían necesarias realizar para cumplir con las metas previstas. A medida que se iban cumpliendo dichas tareas se realizaban reuniones para mostrar los avances, plantear las dudas y verificar que los pasos dados eran los adecuados. La idea principal es que en cada revisión el proyecto debe mostrar alguna evolución respecto a la comprobación previa, lo que se ajusta al modelo de planificación Scrum. Las diferentes tareas que componen el presente proyecto, se pueden dividir en tres bloques. Indicar que algunas tareas se realizaron en paralelo a otras y que todas ellas se terminaron en los plazos previstos.

3.1 Estado del arte.

Esta fase se basó en la investigación y documentación sobre el contexto que envuelve a las técnicas de aprendizaje no supervisado, entre las que se encuentran los algoritmos de clustering, en concreto los de agrupamiento espacial. También se estudió el proceso de extracción de conocimiento y de minería de datos. Esta tarea abarcó desde el 22 marzo hasta el 30 de junio.

3.2 Estudio de los algoritmos.

Tarea	Inicio	Fin
Estudio del algoritmo DBSCAN.	22/03/2021	20/04/2021
Estudio del algoritmo GDBSCAN.	21/04/2021	29/05/2021

Tabla 3.1: Tareas de la fase de estudio de algoritmos.

El bloque de estudio de los algoritmos está constituído por las dos tareas que se pueden apreciar en la tabla superior. La primera se centra en la documentación e investigación del algoritmo DBSCAN, para lo que se usó principalmente fuentes obtenidas a través de internet y el material de la asignatura de extracción de conocimiento de bases de datos cursada en el máster. La segunda tarea abarcó la investigación del algoritmo GDBSCAN, para lo que se empleó principalmente un artículo proporcionado por el tutor sobre dicho algoritmo y otras fuentes digitales.

3.3 Análisis y mejora.

Tarea	Inicio	Fin
Creación del dataset artificial.	30/05/2021	06/06/2021
Realización de las pruebas con el dataset artificial.	07/06/2021	30/06/2021
Búsqueda y elección del dataset real	01/07/2021	08/07/2021
Realización de las pruebas con el dataset real	09/07/2021	31/07/2021
Análisis de rendimiento y funcionamiento	01/08/2021	19/08/2021
Mejora del algoritmo	20/08/2021	28/08/2021

Tabla 3.2: Tareas de la fase de análisis y mejora.

Por último, se encuentra el bloque de análisis y mejora del algoritmo, el cual está constituido por seis tareas. En la primera se seleccionó el contexto del dataset artificial y se desarrolló el código con el que se generó. En la segunda tarea se realizaron varias pruebas con el dataset artificial y el algoritmo GDBSCAN, en las cuales se emplearon atributos no espaciales. En la tercera tarea se buscó y seleccionó un dataset real sobre el que aplicar, una vez más, el algoritmo GDBSCAN y en la cuarta tarea se realizaron las pruebas pertinentes. La tarea de análisis de rendimiento y funcionamiento se basó en la realización de pruebas con conjuntos de datos que poseen características particulares con el fin de comprobar el comportamiento del algoritmo. Por último, se afrontó la tarea de mejora del algoritmo, en la cual se intentó prestar atención a todos aquellos aspectos potencialmente mejorables o que permitan añadir algún componente adicional con el fin de mejorar su rendimiento.

Capítulo 4 Desarrollo

4.1 Estudio del algoritmo DBSCAN

DBSCAN [26] es un algoritmo de densidad simple que implementa la noción de densidad por medio de un procedimiento basado en centro. La densidad de cada punto se estima contando el número de puntos, incluido él, que se encuentran a una distancia menor o igual que "Eps" del punto. Dependiendo de dicho valor, cada punto es clasificado como central, frontera o ruido.

- ➢ Un punto es central si el número de puntos a una distancia no mayor que "Eps" de él supera el valor mínimo "MinPts".
- ➢ Un punto es frontera si no es central pero hay al menos un punto central a una distancia no mayor que "Eps".
- ➢ Un punto es ruido si no es central ni frontera.

Como se puede observar el funcionamiento de este algoritmo depende de la elección de los parámetros "Eps" y "MinPts". Un procedimiento bastante empleado para fijar el valor del parámetro "MinPts" es el concepto de k-distancia. La k-distancia de un punto se define como la distancia al k-ésimo punto más cercano. Tras calcular la k-distancia de cada punto, se ordenan estas de menor a mayor y se muestran en un gráfico. El valor predeterminado de k es K= 2*dimensión - 1.

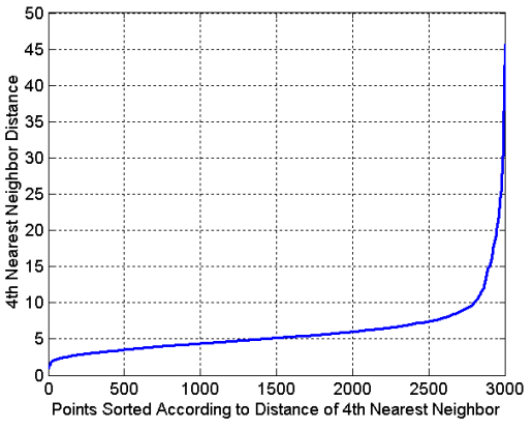


Figura 4.1: Ejemplo de gráfico de K-distancia [23].

El valor en el que se produce un cambio de la curva es un valor apropiado para "Eps". El valor apropiado para "MinPts" debe ser mayor o igual a K + 1. Más adelante, en la sección destinada a la determinación de los parámetros en GDBSCAN se proporcionarán más detalles.

4.2 Estudio del algoritmo GDBSCAN.

El algoritmo GDBSCAN [26] es una generalización del algoritmo DBSCAN. Permite agrupar objetos puntuales y objetos extendidos espacialmente de acuerdo con sus atributos espaciales y no espaciales. Dicho algoritmo se caracteriza por dos aspectos;

1. Permite usar cualquier noción de vecindad de un objeto si la definición de vecindad se basa en un predicado binario que es simétrico y reflexivo.
2. En lugar de simplemente contar los objetos presentes en la vecindad de un objeto, es posible usar otras medidas, como por ejemplo atributos no espaciales para definir la cardinalidad.

Antes de continuar explicando el algoritmo, indicar que al tratarse de una generalización, los parámetros de DBSCAN, es decir; **"**Eps" (distancia que determina la vecindad), la cardinalidad como método para calcular la densidad, y "MinPts" como umbral para determinar si un punto es central, frontera o ruido, pasarán a ser representados en GDBSCAN por;

DBSCAN	GDBSCAN	Descripción
"Eps" como único aspecto considerado para determinar la vecindad de un punto.	NPred(punto)	Función genérica que permite obtener la vecindad de un punto en función de características que pueden estar relacionadas con la distancia o cualquier otro aspecto. El predicado de vecindad no está restringido a vecindades puramente espaciales, se pueden usar atributos no espaciales de los objetos para derivar un predicado de vecindad.
Cardinalidad como método para calcular la densidad.	wCard	Puntero a una función que devuelve la cardinalidad ponderada de un objeto. En este caso es posible usar los atributos no espaciales como un "peso" al calcular la cardinalidad de la vecindad de un objeto.
MinPts	MinCard	Umbral para evaluar la densidad.

Tabla 4.1: Términos empleados en GDBSCAN.

Por otro lado, en el siguiente apartado se mencionará la función **MinWeight(NPred(punto))**, dicha función recibe como parámetro la vecindad de un punto, e internamente, realiza la siguiente comparación **wCard(conjunto_puntos) >= MinCard**, de manera que si se cumple devuelve "Verdadero" y en caso contrario "Falso".

4.2.1 Conceptos.

A través del algoritmo GDBSCAN se introduce el concepto de "conjunto conectado por densidad", el cual a su vez abarca otros conceptos que serán explicados a continuación [26].

- **Directamente alcanzable por densidad:** Un objeto **"p"** es directamente alcanzable por densidad desde un objeto **"q"** si se cumple que;

  1. $p \in NPred(q)$
  2. $MinWeight(NPred(q)) = Verdadero$

  Esto es simétrico para dos objetos centrales, pero no para dos objetos de diferente tipo (central y fronterizo).

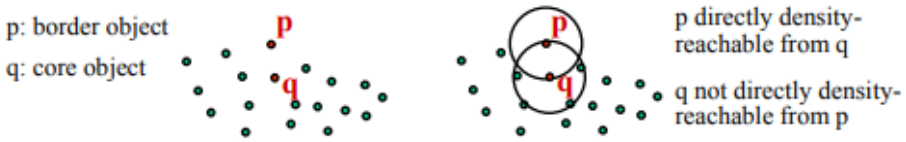


Figura 4.2: Objetos directamente alcanzables por densidad [26].

- **Alcanzable por densidad:** Un objeto **"p"** es alcanzable por densidad desde un objeto **"q"** si hay una cadena de objetos $\mathbf{p_1, p_2, \ldots, p_n}$ (p_1=q, p_n=p) tal que para todo **i=1, …, n:** $\mathbf{p_{i+1}}$ es directamente alcanzable por densidad desde $\mathbf{p_i}$.

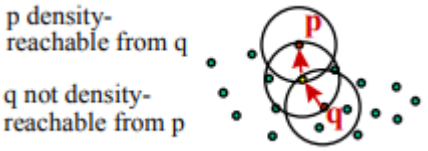


Figura 4.3: Objeto alcanzable por densidad [26].

- **Conectado por densidad:** Un objeto **"p"** está conectado por densidad a un objeto **"q"** si hay un objeto **"o"** tal que tanto **"p"** como **"q"** son alcanzables por densidad desde **"o".** La conectividad de densidad es una relación simétrica.

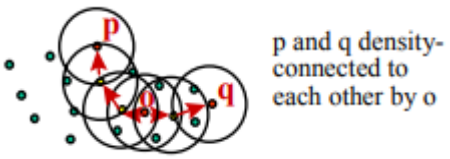


Figura 4.4: Objetos conectados por densidad [26].

- **Conjunto conectado por densidad:** Un conjunto **"C"** conectado por densidad en una base de datos **"D",** es un subconjunto no vacío de **"D"** que debe cumplir las siguientes propiedades;

    1. **Maximidad:** Para todo **p,q ∈ D:** si **p ∈ C** y **"q"** es alcanzable por densidad desde **"p"**, **q ∈ C** .
    2. **Conectividad:** Para todo **p,q ∈ C: "p"** está conectado por densidad a **"q".**

Teniendo en cuenta los conceptos explicados anteriormente,un cluster **"CL"** de **"D"** es un conjunto de conjuntos conectados por densidad en **"D"**, **CL = {C1, ..., CK}**. A su vez, el concepto de ruido se definiría de la siguiente forma: Sea **CL = {C1, ..., CK}** un agrupamiento de la base de datos **"D"**. Se define ruido en **"D"** como el conjunto de objetos que no pertenecen a ningún conjunto **"Ci"** conectado por densidad.

4.2.2 Funcionamiento.

Para encontrar un conjunto conectado por densidad, GDBSCAN sigue los pasos descritos a continuación;

1. Selecciona un objeto arbitrario "p".

2. Recupera todos los objetos alcanzables por densidad desde "p".
    - Si "p" es un objeto central se produce un conjunto conectado por densidad.
    - Si "p" no es un objeto central, ningún objeto será alcanzable por densidad desde "p", y "p" se asignará a "NOISE".

3. Este procedimiento se aplica iterativamente a cada objeto "p" que aún no ha sido clasificado.

Antes de examinar el código del algoritmo GDBSCAN se procederá a explicar las clases "Point" y "Points" [27], ya que ambas son empleadas en dicho algoritmo.

```
#Clase que representa un punto de la base de datos.
class Point:
    def __init__(self, x, y, val):
        self.x = x
        self.y = y
        self.attributes = val
        self.cluster_id = UNCLASSIFIED

    def __repr__(self):
        return '(x:{}, y:{}, attributes:{}, cluster:{})' \
            .format(self.x, self.y, self.attributes, self.cluster_id)
```

Figura 4.5: Clase “Point”.

La clase “Point” se emplea para almacenar o encapsular cada uno de los objetos que componen el dataset sobre el que se desea aplicar GDBSCAN. Dicha clase posee los atributos para almacenar los datos de cada objeto, “x” para la coordenada del eje X, “y” para la coordenada del eje Y, “attributes” para almacenar los atributos del objeto en formato diccionario, y “cluster_id” para almacenar el identificador del cluster al que pertenece dicho objeto, inicialmente tiene como valor “UNCLASSIFIED”. El método “__repr__” tiene como fin facilitar la visualización de los objetos de la clase “Point”.

En cuanto a la clase “Points” se utiliza como contenedor de una lista de objetos “Point”, dicha lista representa todos los datos presentes en el dataset que se pretende usar. El atributo que contiene esa lista se denomina “points”. La clase en cuestión dispone de los métodos específicos para llevar a cabo operaciones sobre los puntos de la mencionada lista. En la siguiente imágen se muestran los principales métodos.

```
def get(self, index):
    return self.points[index]

def neighborhood(self, point, n_pred):
    return list(filter(lambda x: n_pred(point, x), self.points))

def change_cluster_ids(self, points, value):
    for point in points:
        self.change_cluster_id(point, value)

def change_cluster_id(self, point, value):
    index = (self.points).index(point)
    self.points[index].cluster_id = value
```

Figura 4.6: Métodos de la clase “Points”.

El método “neighborhood” devuelve una lista con los puntos presentes en la vecindad del objeto “point” que se pasa como parámetro. El método “change_cluster_ids” actualiza el atributo “cluster_id”, con el valor contenido en “value”, de los puntos presentes en la lista que se pasa a través del parámetro “points”. Por último, mencionar el método “change_cluster_id”, el cual realiza el mismo trabajo que el método anterior con la diferencia de que solo afecta a un punto, el cual se le pasa por parámetro. Y el método “get” que retorna una elemento de la lista de objetos que representa los datos del dataset.

Una vez comentadas las clases “Point” y “Points” se procederá a explicar el algoritmo GDBSCAN [27]. El código en python de la función principal del algoritmo es el siguiente:

```
def GDBSCAN(points, n_pred, min_card, w_card):
    points = copy.deepcopy(points)
    cluster_id = 0
    for point in points:
        if point.cluster_id == UNCLASSIFIED:
            if _expand_cluster(points, point, cluster_id, n_pred, min_card,
                               w_card):
                cluster_id = cluster_id + 1
```

Figura 4.7: Función principal de GDBSCAN.

El parámetro “points” representa los puntos del dataset que se esté empleando, el resto de parámetros de la función corresponden con los parámetros de GDBSCAN explicados anteriormente (NPred, MinCard y wCard).

La variable “cluster_id” se inicializa a 0, y es diferente para cada cluster. Los puntos que pertenezcan a un mismo cluster se les asignará el mismo “cluster_id”, aunque inicialmente dichos puntos tendrán como valor “UNCLASSIFIED” (-2).

A continuación, para cada punto del dataset, se comprobará si su “cluster_id” es igual a “UNCLASSIFIED”, de ser así, para el punto en cuestión se llamará a la función “_expand_cluster”, la cual permitirá detectar el cluster al que pertenece dicho punto. Si efectivamente a través del punto analizado se ha podido detectar un cluster, la función retornará “true” y, a continuación, la variable “cluster_id” se incrementará en una unidad, en caso contrario retornará “false”.

A continuación, se muestra el código en python de la función “_expand_cluster” [27]:

```
def _expand_cluster(points, point, cluster_id, n_pred, min_card, w_card):
    if not _in_selection(w_card, point):
        points.change_cluster_id(point, UNCLASSIFIED)
        return False

    seeds = points.neighborhood(point, n_pred)
    if not _core_point(w_card, min_card, seeds):
        points.change_cluster_id(point, NOISE)
        return False

    points.change_cluster_ids(seeds, cluster_id)
    seeds.remove(point)

    while len(seeds) > 0:
        current_point = seeds[0]
        result = points.neighborhood(current_point, n_pred)
        if w_card(result) >= min_card:
            for p in result:
                if w_card([p]) > 0 and p.cluster_id in [UNCLASSIFIED, NOISE]:
                    if p.cluster_id == UNCLASSIFIED:
                        seeds.append(p)
                    points.change_cluster_id(p, cluster_id)
        seeds.remove(current_point)
    return True
```

Figura 4.8: Función "_expand_cluster".

Esta función es la que se encarga de detectar los clusters. La función "_in_selection()" comprueba si la cardinalidad para el punto seleccionado es mayor que 0. Si no se cumple, retornara "False" y a continuación se establece el "cluster_id" de ese punto a "UNCLASSIFIED".

Tras ello, en la variable "seeds" se almacena la vecindad del punto seleccionado, la cual es obtenida mediante el método "points.neighborhood()". A continuación, se emplea la función "_core_point()" para comprobar si "point" es un punto central o no. Si no lo es, retornará "False" y se establece el "cluster_id" del punto a "NOISE".

En el caso de tratarse de un punto central, se modifica el valor del "cluster_id" asociado a cada punto de la vecindad, asignándoles a todos ellos el mismo identificador de cluster. Tras ello, se elimina de "seeds" el punto "point" que se ha usado hasta ahora, ya que su vecindad ya ha sido estudiada. Tras ello, se procederá a trabajar con el resto de puntos presentes en "seeds".

Para ello, se emplea un bucle “while” en el que se extraerá en cada iteración un punto de “seeds” hasta que se vacíe. En cada iteración, se almacenará en “result” la vecindad del punto correspondiente. Tras ello, se comprobará si el punto actual es central o no. De ser así, para cada uno de los puntos de su vecindad se comprobará si su cardinalidad es mayor que 0 y si el “cluster_id” que tienen asignado es “UNCLASSIFIED” o “NOISE”. Si se cumple la condición, aquellos puntos de la vecindad que sean “UNCLASSIFIED” se añadirán a “seeds” para ser estudiados posteriormente, mientras que los que tengan como “cluster_id” el valor “NOISE” pasarán a tener el valor del identificador de cluster que se le pasó a la función “_expand_cluster”. Finalmente, se eliminará de ”seeds” el punto con el que se ha trabajado y se iniciará una nueva iteración del bucle “while”.

4.2.3 Determinación de parámetros.

Como se vió anteriormente, en DBSCAN se empleó el concepto de k-distancia para fijar el valor de “Eps” y “Minpts”. En el caso de GDBSCAN también se puede emplear el concepto de K-distancia para determinar el valor del parámetro “MinCard” y “Eps” en el caso de que “NPred()” tenga en cuenta la distancia a la hora de determinar la vecindad de un punto.

A la hora de seleccionar un valor para K se debe tener en cuenta que cuanto menor sea dicho valor, menores serán los costos computacionales. Se ha comprobado que pequeños cambios en K no afectan casi al resultado, por lo que se prioriza que tenga un valor bajo. A ser posible K > 1 para evitar el efecto single-link (enlace único).

> El **single-link clustering** es un método de agrupamiento jerárquico. En él se unen los grupos de forma ascendente, combinando en cada paso los dos grupos con el par de elementos más cercanos. El **efecto single-link** es un inconveniente de este método, en el cual los grupos mal separados, pero distintos, se fusionan en una etapa temprana.

Los pasos a seguir para fijar el valor de los parámetros en GDBSCAN son los siguientes;

1. El usuario selecciona un valor para K (el valor predeterminado es K= 2*dimensión - 1)
2. El sistema calcula y muestra la gráfica de K-distancia para una muestra de la base de datos.

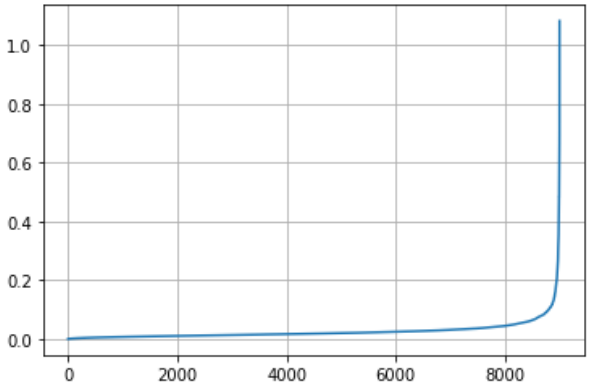


Figura 4.9: Ejemplo de gráfico de K-distancia.

3. El usuario inspecciona la gráfica, selecciona un objeto como objeto umbral, y el valor de k-distancia de ese objeto se utiliza como valor de "Eps". El objeto umbral se determina distinguiendo el primer "valle" en la gráfica. Si fuese posible estimar el porcentaje de ruido también se podría derivar el umbral. En cuanto a "MinCard" se debe establecer en un valor igual o mayor a K+1, dependiendo de las características del dataset, como por ejemplo la cantidad de puntos [26].

4.3 Creación del dataset artificial.

Con el objetivo de mostrar diferentes casos de uso del algoritmo GDBSCAN se ha creado un conjunto de datos artificial que reúne las características necesarias para poder mostrar el máximo potencial del algoritmo. En concreto, el contexto del dataset será el de una área forestal, de la cual se dispone de información sobre cada planta presente en ella, como el nombre de la especie, su altura y su ubicación mediante coordenadas.

Para elaborar dicho dataset artificial se ha desarrollado el siguiente código python:

```
#Función para crear los grupos de datos
def create_cluster(n_samples, centers, cluster_std, attributes):
  X, y = make_blobs(n_samples=n_samples,n_features=2, centers=centers, cluster_std=cluster_std)
  points_csv = []

  for i in range(n_samples):
    points_csv.append(Point(X[i][0],X[i][1],attributes))

  return points_csv
```

Figura 4.10: Función "create_cluster".

Como se puede ver en la imagen, primero se elabora la función "create_cluster" la cual se encarga de crear los grupos de datos para el dataset. Dicha función recibe como parámetro

el número de puntos (n_samples), la posición central del conjunto de datos (centers), la desviación típica para el conjunto (cluster_std) y los atributos no espaciales en forma de diccionario (attributes). Internamente, para crear el conjunto de datos se llama a la función "make_blobs" a la cual se le especifica por parámetro el número de puntos, el número de características, la posición central del conjunto de datos y la desviación estándar. La función retorna dos resultados, en "X" almacena una matriz con las coordenadas de cada punto, mientras que en "y" guarda una etiqueta numérica que indica a qué cluster pertenece cada uno. A continuación, se crea la lista "points_csv" y se almacenan en ella objetos "Point" con las coordenadas y los atributos de cada planta. Tras ello se retorna dicha lista.

```
#Creamos los datos para el dataset artificial
group1 = create_cluster(100,[(20,20)],4,{"Especie":"pino"})
group2 = create_cluster(200,[(12,16)],1.5,{"Especie":"roble"})
group3 = create_cluster(200,[(12,22)],1.5,{"Especie":"brezo"})
group4 = create_cluster(200,[(28,25)],1.5,{"Especie":"nogal"})
group5 = create_cluster(200,[(22,13)],2.5,{"Especie":"haya"})
group6 = create_cluster(200,[(28,15)],2.5,{"Especie":"palmera"})
group7 = create_cluster(50,[(20,27)],1.5,{"Especie":"almendro"})
```

Figura 4.11: Creación de los grupos de datos.

En segundo lugar se crean los grupos de datos del dataset artificial. En total, como se puede apreciar en la imagen anterior, se crean 7 agrupaciones. Para cada una de ellas se especifica el número de puntos, la posición central del agrupamiento, la desviación típica y el nombre de la especie presente en el agrupamiento.

```
#Se unen todos los datos
points_csv = group1 + group2 + group3 + group4 + group5 + group6 + group7
print(len(points_csv))
```

Figura 4.12: Unión de los datos.

```
#Los datos se copian en un objeto dataframe
dataset = pd.DataFrame(dict(X=[point.x for point in points_csv],
                            Y=[point.y for point in points_csv],
                            Especie=[point.attributes["Especie"] for point in points_csv]))
```

Figura 4.13: Los datos se copian en un objeto dataframe.

Una vez generados todos los datos se unen y se almacenan en la variable “points_csv”. Tras ello la información se pasa a un objeto dataframe, ya que éste facilitará el trabajo para añadir la última columna al dataset, además de que posee un método específico para escribir los datos en formato CSV. A continuación, se muestra el código usado para añadir la columna “Altura” con el tamaño de cada planta en centímetros.

```
#Se crea la columna "Altura"
dataset["Altura"] = None

#Se asigna la altura a cada planta
for i in range(len(dataset)):
  if 5 < dataset.iloc[i,0] < 15 and 10 < dataset.iloc[i,1] < 20:
     dataset.iloc[i,3] = random.uniform(200,300)
  else:
     dataset.iloc[i,3] = random.uniform(20,300)

dataset.to_csv("data.csv")
```

Figura 4.14: Código para añadir la columna “Altura” y escribir los datos.

Para escribir los datos en formato CSV se ha empleado la función “to_csv”, de esta forma se tendrán disponibles para el resto de pruebas que se realizarán. A continuación, se muestran las primeras líneas de dicho archivo en formato CSV.

```
,X,Y,Especie,Altura
0,20.82007750962035,20.97395604408334,pino,64.85145371064766
1,20.86708859265843,15.101367615978248,pino,199.21511921443695
2,20.41853375497702,21.482189032368744,pino,39.00341995002376
3,16.35529016606309,18.940511377603,pino,43.7854825583004
4,19.382585155963,21.78737895824518,pino,89.2880675048978
5,18.78946546766101,19.12144035035648,pino,289.45285792508156
6,23.730041694100297,18.5039114597102,pino,80.55319964854775
7,17.188924019435113,20.011016931859547,pino,152.6532661990763
8,22.20489604787097,22.19035531277372,pino,79.604947392373
9,18.88110247917399,16.1004575779187,pino,143.16252827380748
```

Figura 4.15: Archivo CSV.

Para la lectura del fichero creado anteriormente, se emplea el siguiente código:

```
#Leer datos desde fichero
csv_data=pd.read_csv('/content/data.csv', index_col=0)

#Se encapsulan los datos para posteriormente usarlos con el algoritmo
points_csv = []
for i in range(len(csv_data)):
  points_csv.append(Point(csv_data.iloc[i,0],csv_data.iloc[i,1],{"Especie":csv_data.iloc[i,2],
                                                                 "Altura":csv_data.iloc[i,3]}))
```

Figura 4.16: Lectura y encapsulamiento de los datos.

Una vez el fichero ha sido leído y los datos han sido encapsulados en objetos “Point” formando una lista, su visualización se realiza de la siguiente manera;

```
#Se visualizan los datos
for point in points_csv:
  plt.plot(point.x, point.y, marker="o", color="black", markersize=2)
```

Figura 4.17: Visualización de los datos.

El método “plot” de la librería “matplotlib” permite representar en un espacio bidimensional los datos. Para ello se le pasa por parámetro las coordenadas del punto (atributos espaciales), el tipo de marca, su color y su tamaño. El resultado de la visualización de los datos del dataset artificial es el siguiente;

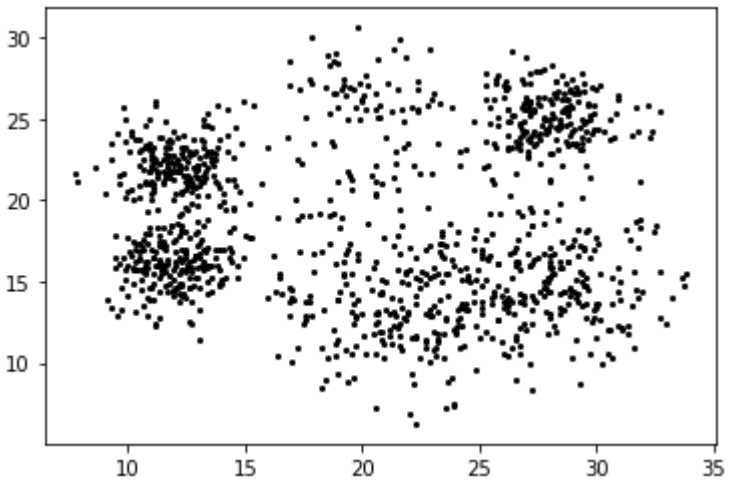


Figura 4.18: Dataset artificial.

Una vez creado el dataset artificial se procederá a estimar el valor de los parámetros, “Eps” y “Mincard” (o “Minpts” en el caso de DBSCAN). Para calcular “Eps” se empleará el concepto de K-distancia, y como valor de “K” se elegirá 3 (K = 2 x dimensión -1 = 2 x 2 - 1 = 3). Para obtener el diagrama de K-distancia se ha empleado el siguiente código:

```
#Leer datos desde fichero
csv_data=pd.read_csv('/content/data.csv', index_col=0)

#Se crea el estimador y se ajusta a la información del dataset para luego
#realizar las consultas
neighbors = NearestNeighbors(n_neighbors=4)
neighbors_fit = neighbors.fit(csv_data[["X","Y"]])
distances, indices = neighbors_fit.kneighbors(csv_data[["X","Y"]])

#Se ordenan las distancias de forma ascendente y se visualizan.
distances = np.sort(distances, axis=0)
plt.plot(distances[:,3])
plt.grid()
```

Figura 4.19: Código para el diagrama de K-distancia.

Como se puede observar, para obtener los "K" vecinos más cercanos de cada punto se ha empleado la librería "sklearn", en concreto, el módulo "NearestNeighbors". A la función "NearestNeighbors" se le pasa por parámetro el número de vecinos que se buscará en cada consulta. Anteriormente, se indicó que el valor de "K" sería 3, sin embargo en el código se ha empleado el valor 4, esto es debido a que el estimador considera el propio punto que se consulta como un vecino más, por lo que si se desea obtener 3 vecinos diferentes se debe incrementar "K" en una unidad. El método "fit" se emplea para ajustar el estimador y se le pasa por parámetro la lista con la información del dataset. Una vez ajustado, se emplea el método "kneighbors" para realizar las consultas. Para ello se le debe pasar la lista de puntos por parámetro. Finalmente, retorna una matriz con las distancias de cada uno de los vecinos para cada punto consultado. En este caso lo que interesa es la distancia al "K" vecino más cercano, así que se cogerán los valores de la última columna.

El resultado obtenido ha sido el que se puede ver en la siguiente imágen:

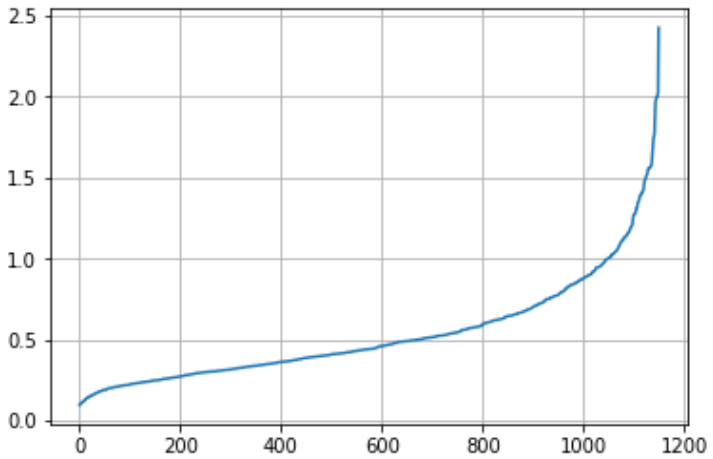


Figura 4.20: Diagrama K-distancia del dataset artificial.

El valor ideal para “Eps” será igual al valor de la distancia que se puede apreciar en el “codo” de la curva o en el punto de máxima curvatura, que en este caso es 1. En cuanto al valor de “Mincard” se probará con 4 (Mindcard = K + 1 = 3 + 1 = 4). Si se aplica DBSCAN sobre el dataset artificial con los valores para los parámetros anteriormente calculados se obtiene el siguiente resultado:

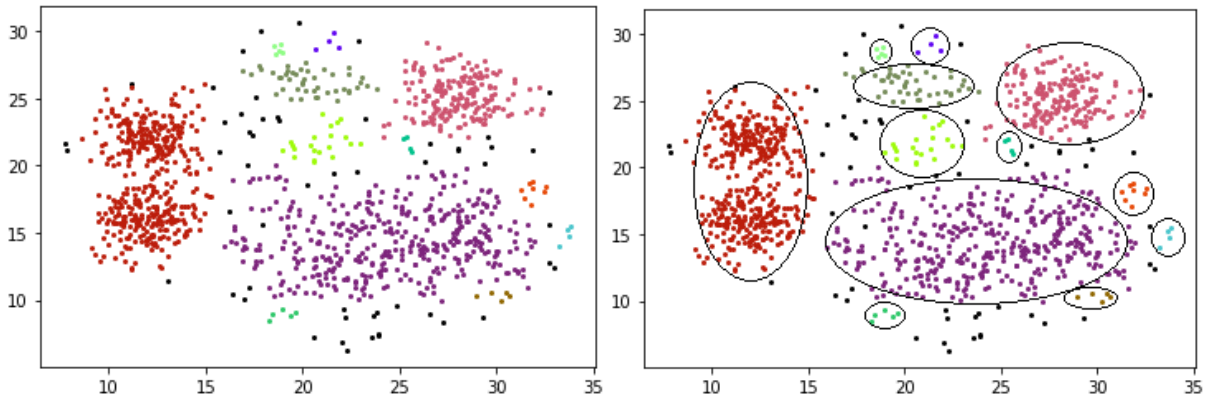


Figura 2.21: Resultado de aplicar DBSCAN (Mincard = 4 Esp = 1).

Como se puede observar, usando como “Mincard” el valor 4 se han detectado 12 clusters, los puntos negros dispersos entre ellos representan el ruido. Algunos de estos clusters son de pequeño tamaño y no aportan información relevante. Por ello, con el fin de no incluirlos se ha incrementado el valor de “Mincard” a 5. Por otro lado, se puede apreciar que existen clusters, como el violeta y el rosado, que se han expandido demasiado abarcando puntos muy distantes, por ello con el fin de obtener unos clusters más compactos y precisos se ha decidido reducir el valor de “Eps” a 0.85. Los resultados obtenidos son los siguientes.

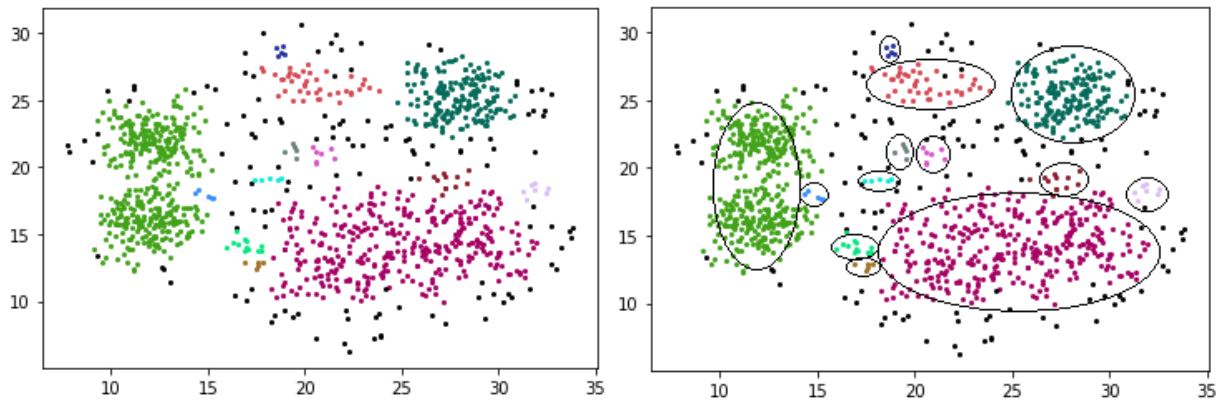


Figura 4.22: Resultado de aplicar DBSCAN (Mincard = 5 Esp = 0.85).

En este caso, se han obtenido 13 clusters, los clusters de tan solo 4 elementos que se detectaron anteriormente han pasado a ser considerados como ruido. Estos serán los valores

definitivos de los parámetros que se emplearán en las pruebas que se muestran en el siguiente apartado.

Para visualizar los agrupamientos identificados anteriormente por DBSCAN se ha empleado el siguiente código:

```
for i in clustered:
  hex_color = "#" + str("%06x" % random.randint(0, 0xFFFFFF))
  if (i == -1):
    hex_color = "black"
  for point in clustered[i]:
    plt.plot(point.x, point.y, marker="o", color=hex_color, markersize=2)

plt.show()
```

Figura 4.23: Código para visualizar los clusters detectados.

4.3.1 Prueba 1: Determinar vecindad usando atributos no espaciales.

Esta prueba se basa en manipular la función "n_pred" que permite determinar la vecindad de un punto. En DBSCAN la condición que permite detectar la vecindad está únicamente relacionada con la distancia "Eps" entre el punto seleccionado y el resto, de manera que aquellos que estén a una distancia menor o igual que "Eps" constituirán dicha vecindad.

En GDBSCAN es posible emplear otras condiciones para seleccionar la vecindad de un punto a través de la combinación de la distancia entre puntos con los atributos no espaciales. La función "n_pred(p1,p2)" se ha implementado de tal forma que retorna "True" si el punto "p2" pertenece a la vecindad de "p1" o "False" en caso contrario, de manera que dicha función deberá ser llamada una vez por punto presente en el dataset para obtener la vecindad de "p1". A continuación, se muestra un ejemplo de función "n_pred" en la que se combina la distancia entre puntos con atributos no espaciales.

```
#Función que comprueba si un punto (p2) forma parte de la vecindad de otro punto (p1).
def n_pred(p1, p2):
    return all([math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2) <= 0.85,
                p1.attributes["Especie"] == p2.attributes["Especie"]])
```

Figura 4.24: Función "n_pred" para clusters de la misma especie.

En este caso el uso de atributos no espaciales permite distinguir nuevos clusters atendiendo al valor que toma dicho atributo. En este caso, el atributo hace referencia a la especie de la planta y se obliga a que la especie de "p2" sea igual a la de "p1", lo que

permitirá descubrir clusters de plantas de la misma especie. También se debe cumplir la condición de distancia entre puntos. A continuación, se muestra el resultado de aplicar GDBSCAN con dicha función “n_pred”.

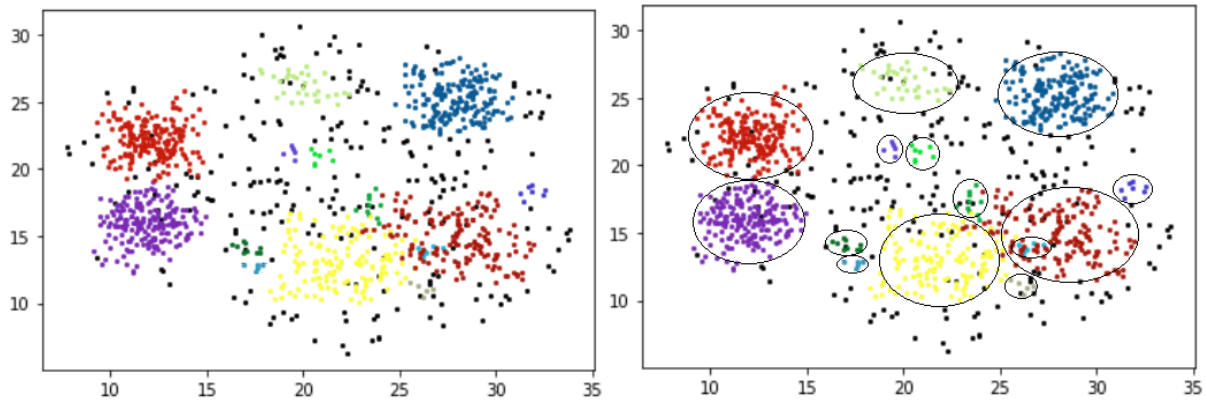


Figura 4.25: Resultado de aplicar GDBSCAN (Prueba 1).

Como se puede apreciar se han detectado 14 clusters. Como ya se dijo, los clusters están asociados a conjuntos de plantas que pertenecen a una misma especie y que cumplen la condición asociada a la distancia “Eps”. Como se puede ver, en el caso de los clusters rojo y violeta a pesar de estar muy próximos en el espacio y cumplir la condición de distancia “Eps” no constituyen un mismo cluster pues las especies de cada uno son distintas. Por otro lado, este aspecto también afecta al ruído. En DBSCAN es imposible que exista ruido en zonas de alta densidad, sin embargo, en este caso GDBSCAN es capaz de detectar ruido en zonas muy densas, el cual es causado por la presencia de una especie distinta a las existentes a su alrededor y que respecto a las plantas de su propia especie se encuentra distante y no cumple las condiciones como para construir un nuevo cluster. Un claro ejemplo de dos clusters de especies diferentes que comparten una misma zona son el cluster azul y el marrón. Si se compara con el resultado de DBSCAN se puede apreciar que se han detectado clusters diferentes.

DBSCAN. **GDBSCAN.**

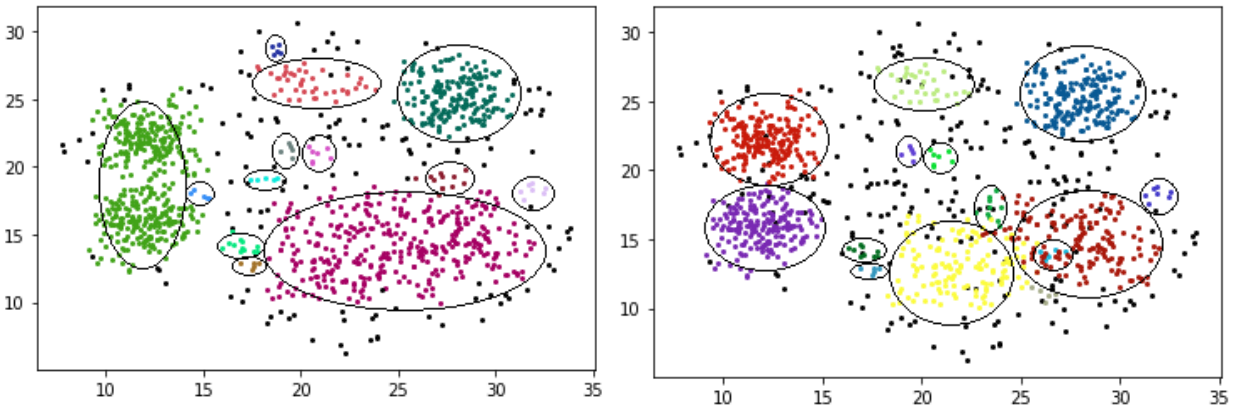


Figura 4.26: Comparativa de los resultados de DBSCAN y GDBSCAN.

Siguiendo esta dinámica, haciendo uso del atributo “Altura” también sería posible detectar aquellos clusters de plantas que cumplan una determinada condición en lo que al tamaño de las plantas se refiere. Por ejemplo, para detectar los agrupamientos de plantas con un tamaño superior o igual a 200 centímetros se debe emplear la siguiente función “n_pred”.

```
#Función que comprueba si un punto (p2) forma parte de la vecindad de otro punto (p1).
def n_pred(p1, p2):
    return all([math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2) <= 0.85,
                p1.attributes["Altura"] >= 200,
                p2.attributes["Altura"] >= 200])
```

Figura 4.27: Función “n_pred” para clusters de plantas de 200 cm o más.

Al aplicar GDBSCAN, se obtiene el siguiente resultado.

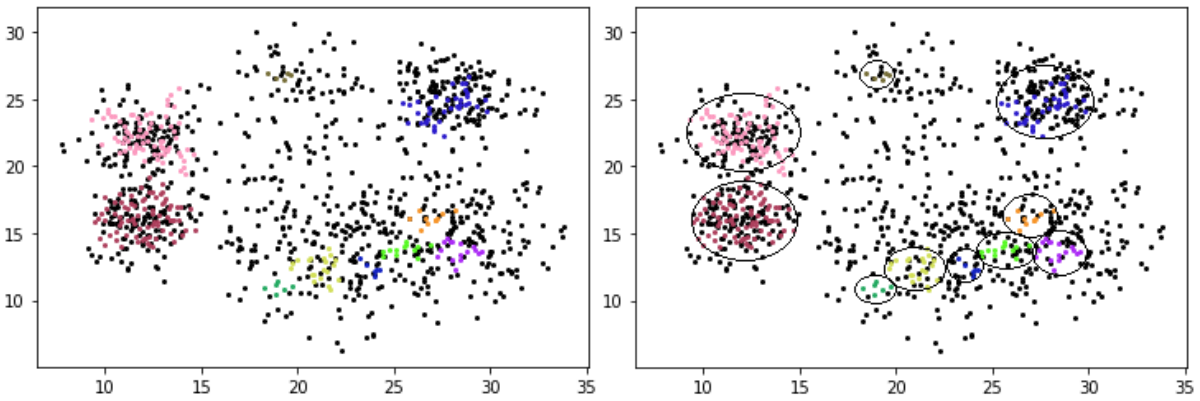


Figura 4.28: Cluster de plantas con una altura mayor o igual a 200 cm.

Por último, indicar que de la misma forma que es posible detectar clusters de plantas de una misma especie, también es posible localizar clusters de plantas pertenecientes a una especie concreta. Este caso, tal vez sea menos interesante que los anteriores, ya que siempre es posible filtrar los datos del dataset para obtener los relativos a una especie y luego aplicar tan solo DBSCAN para detectar los clusters. Pero si lo que se quiere es detectar agrupamientos de especies específicas manteniendo el resto de datos, con GDBSCAN sería posible. Por ejemplo, para obtener los clusters formados por robles se emplearía la siguiente función “n_pred”.

```
#Función que comprueba si un punto (p2) forma parte de la vecindad de otro punto (p1).
def n_pred(p1, p2):
    return all([math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2) <= 0.85,
                p1.attributes["Especie"] == "roble",
                p2.attributes["Especie"] == "roble"])
```

Figura 4.29: Función “n_pred” para clusters de robles.

El resultado es el siguiente.

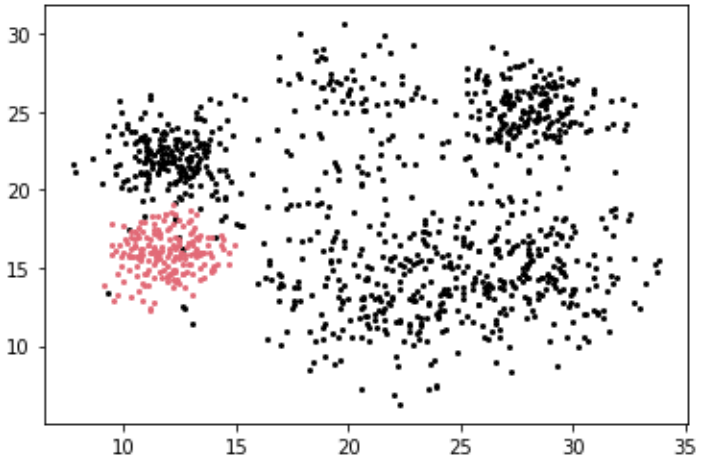


Figura 4.30: Clusters de robles.

4.3.2 Prueba 2: Calcular la cardinalidad usando atributos no espaciales.

En esta segunda prueba se modificará la función “w_card” que se encarga de calcular la cardinalidad de la vecindad de un punto. En DBSCAN el valor devuelto por la función “w_card” coincide con el número de vecinos que componen la vecindad.

Sin embargo, en GDBSCAN, es posible emplear en el cálculo de la cardinalidad atributos no espaciales que actúan como pesos, de manera que no todos los vecinos influirán igual en el cálculo de la cardinalidad. En el contexto del dataset artificial creado, este aspecto permite establecer un número de individuos diferentes para cada especie a la hora de definir un punto como central. Es decir, si el valor de “Mincard” es 5 y a la especie “pino” se le asigna un valor de 0.5, en la vecindad de un punto deberán existir al menos 10 pinos para que dicho punto sea central y forme un cluster. Sin embargo, si su valor fuera de 0.25 se requeriría de 20 pinos en la vecindad. Esto permite dar más o menos importancia a cada planta a la hora de generar un cluster. A continuación se muestra la función “w_card” con diferentes pesos para cada especie de planta:

```
def w_card(points):
    card = 0
    for point in points:
      value = 0.25
      if point.attributes["Especie"] == "haya" or point.attributes["Especie"] == "palmera":
        value = 0.50
      elif point.attributes["Especie"] == "pino":
        value = 1

      card += value

    return card
```

Figura 4.31: Función “w_card” con pesos para cada planta.

El dataset consta de 7 especies de plantas diferentes, a las cuales, como se ha visto en la función “w_card”, se les ha asignado unos determinados pesos para el cálculo de la cardinalidad. En concreto a la haya y la palmera se le ha asignado un valor de 0.5, al pino un valor de 1 y al resto de plantas un valor de 0.25 (roble, brezo, nogal y almendro).

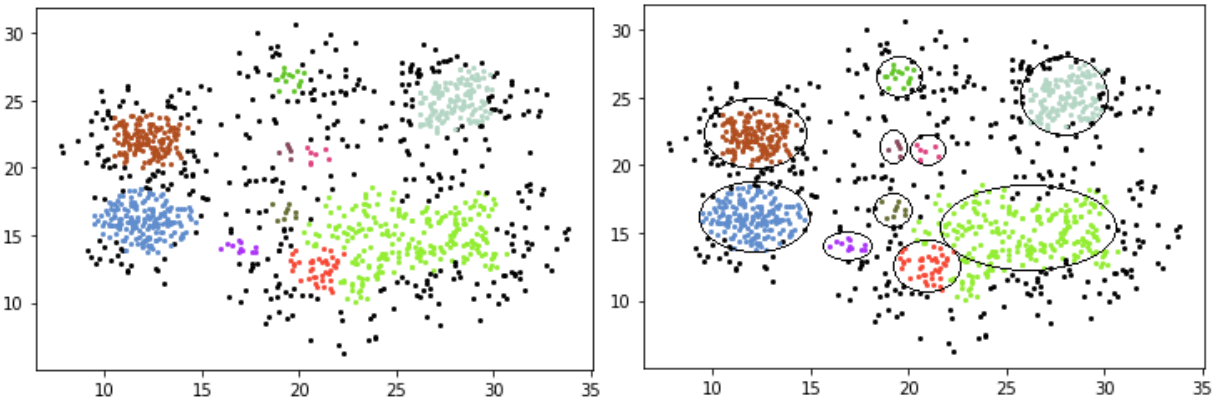


Figura 4.32: Resultado de aplicar GDBSCAN (Prueba 2).

En este caso, se han detectado 10 clusters. Según lo comentado anteriormente, para aquellas plantas que poseen un bajo peso asociado, el algoritmo detectará solo los grandes cúmulos, mientras que las que posean un mayor valor tenderán a crear clusters más fácilmente, pudiendo originar clusters de pequeño tamaño. En este caso, el uso de pesos busca compensar el hecho de que existan plantas que de forma natural crecen muy agrupadas frente a las que crecen de manera más dispersa.

Por último, se unirán los aspectos analizados en las dos pruebas. Es decir, se usará de manera conjunta para detectar clusters con GDBSCAN las funciones “n_pred”, para las especies, y “w_card” vistas previamente.

```
def n_pred(p1, p2):
    return all([math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2) <= 0.85,
                p1.attributes["Especie"] == p2.attributes["Especie"]])

def w_card(points):
    card = 0
    for point in points:
      value = 0.25
      if point.attributes["Especie"] == "haya" or point.attributes["Especie"] == "palmera":
        value = 0.50
      elif point.attributes["Especie"] == "pino":
        value = 1

      card += value

    return card
```

Figura 4.33: Se usan las funciones “n_pred” y “w_card” ya vistas.

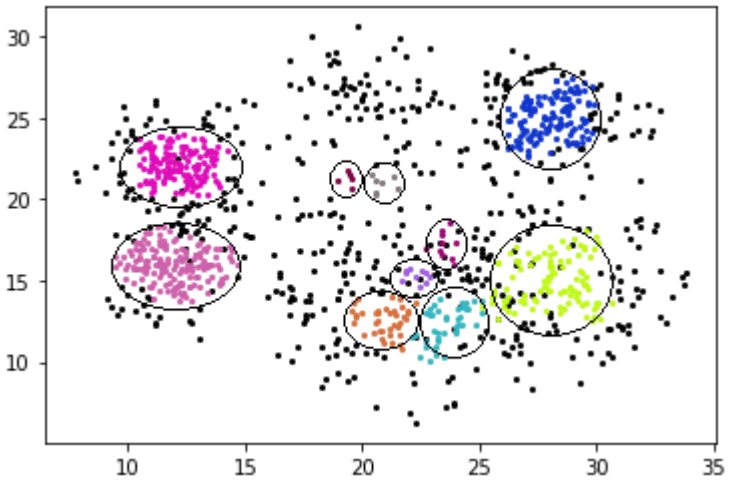


Figura 4.34: Resultado de aplicar las pruebas 1 y 2 juntas.

En este caso se han detectado 10 clusters. Al haber empleado la función “n_prep” de la prueba 1, cada cluster encontrado estará constituido por una misma especie de planta. Además, al emplear la función “w_card” con los pesos de la prueba 2, se habrá establecido, en función de la dispersión en la que crece cada planta, la facilidad con la que cada una creará un cluster. De esta forma es posible detectar el ruido que es particular para cada especie de planta. Es decir, si el brezo es una especie que suele crecer en grandes grupos y las palmeras una especie que crece de manera más dispersa, entonces lo que se considera ruido para un brezo no lo será para una palmera. Por lo que se podrá obtener un resultado más ajustado a las características particulares de cada especie de planta.

4.4 Dataset real.

El presente dataset sobre el que se aplicará el algoritmo GDBSCAN, ha sido extraído de la página oficial de INGEMMET, el Instituto Geológico, Minero y Metalúrgico, el cual está adscrito al Ministerio de Energías y Minas de Perú. Su labor es la investigación de la geología básica, los recursos del subsuelo, los riesgos geológicos y el geoambiente.

El dataset obtenido consta de información sobre la ubicación de depósitos y ocurrencias de minerales industriales a lo largo de todo el territorio peruano. En total, se dispone de 3102 registros y 8 columnas. Las cuales se muestran a continuación.

CODIGO	NOMBRE	HOJA	ZONA	LONGITUD	LATITUD	NOMBRE_COM	TIPO
188119.0	Veintitres de Febrero	36-x	19.0	-69.863321	-17.797566	Sílice	Intrusivo
188121.0	Virgen de la Puerta Tacna	36-x	19.0	-69.942264	-17.557697	Sílice	Intrusivo
188349.0	Cantera Cerro Pelado	36-x	19.0	-69.882234	-17.806520	Mármol	Sedimentario Clástico
188380.0	Enrique Pelado	36-x	19.0	-69.886907	-17.797461	Mármol	Sedimentario Clástico
189791.0	Consorcio Minera 20 de Setiembre	36-x	19.0	-69.938114	-17.666172	Sílice	Intrusivo
...	...	...	...	...	...	...	...

Figura 4.35: Columnas del dataset real.

Atributo	Descripción
CÓDIGO	Código único que identifica cada yacimiento mineral.
NOMBRE	Nombre de la zona donde se encuentra el yacimiento.
HOJA	Identificador de la hoja que contiene información sobre el yacimiento.
ZONA	Identificador de la zona del yacimiento.
LONGITUD	Longitud en la que se ubica el yacimiento
LATITUD	Latitud en la que se ubica el yacimiento
NOMBRE_COM	Nombre del mineral que está presente en el yacimiento.
TIPO	Indica el tipo de mineral del yacimiento.

Tabla 4.2: Explicación de los atributos del dataset real.

En su origen los datos se encontraban en formato DBF, por lo que para trabajar de manera más cómoda se convirtieron al formato CSV. Antes de proceder a trabajar con los datos se comprobó si existían valores nulos en las columnas con el fin de valorar si sería necesario aplicar alguna técnica de imputación o interpolación de datos.

```
import seaborn as sns

#Valores nulos
sns.heatmap(dataset.isnull(), cbar=False);
print(dataset.isnull().sum())
```

Figura 4.36: Código para visualizar valores nulos.

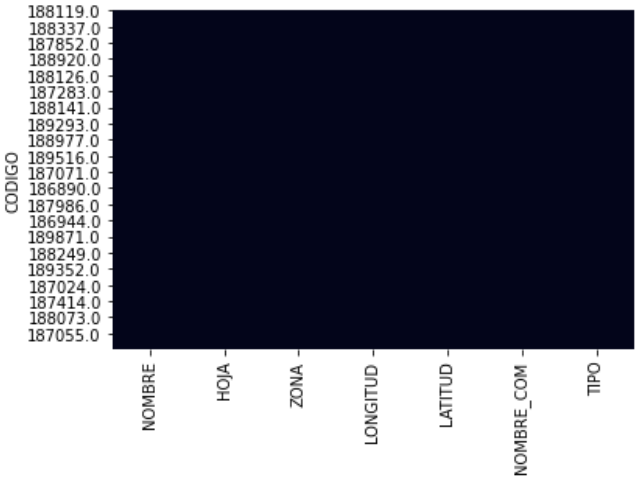


Figura 4.37: Visualización de valores nulos.

```
NOMBRE        0
HOJA          0
ZONA          0
LONGITUD      0
LATITUD       0
NOMBRE_COM    0
TIPO          0
dtype: int64
```

Figura 4.38: Cantidad de valores nulos.

Como se puede observar en las imágenes anteriores, no existen valores nulos en el dataset, por lo que se procederá a trabajar con los datos.

Primero se leerá el archivo de datos en formato CSV y a continuación se visualizará usando como atributos espaciales la longitud y la latitud de cada yacimiento. En cuanto a los atributos no espaciales, este ejercicio se centrará en el uso de los atributos “NOMBRE_COM” y “TIPO”.

```
dataset=pd.read_csv('/content/Rocas_mineral_industrial.csv', index_col=0)

points_csv = []
for i in range(len(dataset)):
  no_spatial = {"Nombre_componente":dataset.iloc[i,5], "Tipo":dataset.iloc[i,6]}
  points_csv.append(Point(dataset.iloc[i,3],dataset.iloc[i,4],no_spatial))

fig, ax = plt.subplots(figsize = (8,7))

for point in points_csv:
  ax.plot(point.x, point.y, marker="o", color="black", markersize=2)

im = plt.imread("/content/peru.png")
BBox = ((dataset.LONGITUD.min(),   dataset.LONGITUD.max(),
         dataset.LATITUD.min(), dataset.LATITUD.max()))

ax.set_xlim(BBox[0],BBox[1])
ax.set_ylim(BBox[2],BBox[3])

ax.imshow(im, zorder=0, extent = BBox, aspect= 'equal')
```

Figura 4.39: Código para visualizar el dataset real.

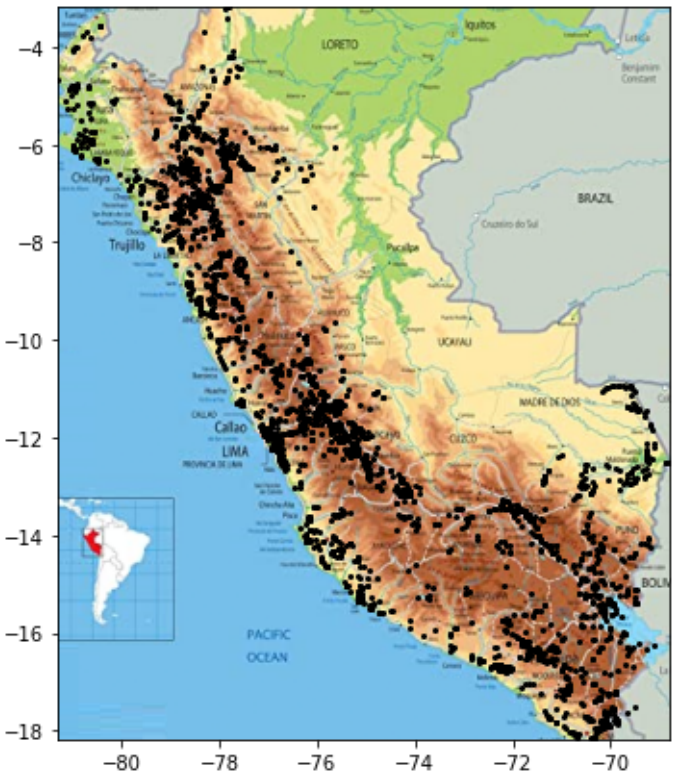


Figura 4.40: Visualización del dataset real.

En la imágen anterior, cada punto representa un depósito de minerales diferente. Dichos puntos han sido dibujados sobre el mapa físico de Perú [24] con el fin de tener una referencia gráfica de la ubicación de los yacimientos a lo largo del país. Como se puede observar, la mayoría de los puntos se localizan en las zonas montañosas.

Antes de aplicar los algoritmos GDBSCAN y DBSCAN, se deberán estimar los parámetros "Eps" y "Mincard" ya que se está empleando un conjunto de datos diferente al de los apartados anteriores. Para ello se reutilizará el código del apartado anterior y se obtendrá el gráfico de K-distancia, usando para "K" el valor 3 (K = 2 x dimensión - 1 = 2 x 2 -1 = 3). El gráfico obtenido es el siguiente.

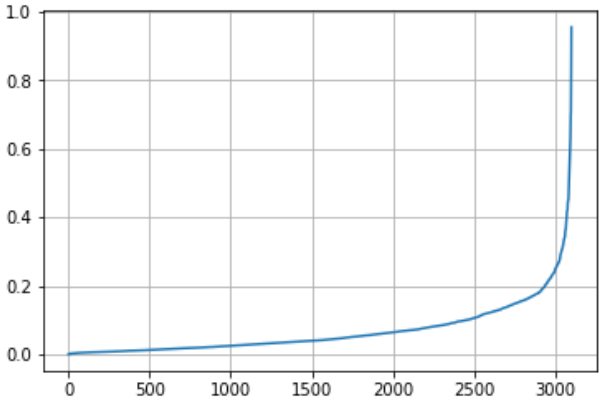


Figura 4.41: Diagrama K-distancia del dataset real.

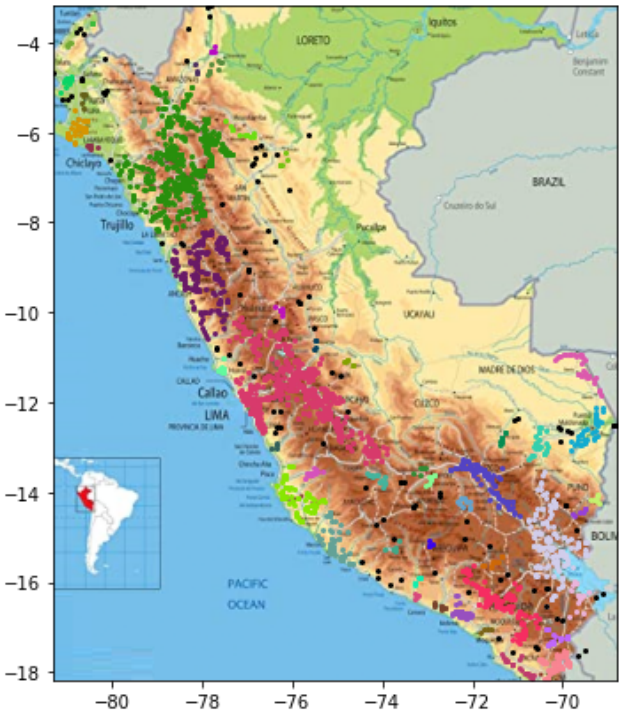


Figura 4.42: DBSCAN dataset real (Mincard = 4).

Según se puede apreciar en el gráfico el valor adecuado para "Eps" es 0.2, ya que ese es el punto de máxima curvatura de la curva. En cuanto a "Mindcard" se probará con 4 (Mindcard = K +1 = 3 + 1 = 4). Si se aplica DBSCAN sobre el dataset con estos valores para los parámetros se obtiene el resultado que se puede apreciar en la imágen de la izquierda. Se han detectado un total de 48 clusters. Como se puede observar, los clusters más importantes se ubican en la zona montañosa, sobre todo en la zona central y al norte.

Antes de aplicar GDBSCAN, se muestran los diferentes valores que toman los atributos "NOMBRE_COM" y "TIPO" a lo largo de los 3102 registros.

```
# Valores diferentes de TIPO
dataset["TIPO"].unique()
```

```
array(['Intrusivo', 'Sedimentario Clástico', 'Lava', 'Metamórfica',
       'No Clasificado'], dtype=object)
```

Figura 4.43: Valores del atributo "TIPO".

```
# Valores diferentes de NOMBRE_COM
dataset["NOMBRE_COM"].unique()
```

```
array(['Sílice', 'Mármol', 'Yeso', 'Áridos', 'Baritina', 'Tufo',
       'Arcilla común', 'Arcilla', 'Sillar', 'Sal', 'Caliza',
       'Piedra Laja', 'Travertino', 'Coquina, Dolomía, Calcita, Caliza',
       'Granito', 'Arena silícea', 'Carbón', 'Bentonita', 'Diatomitas',
       'Azufre', 'Pómez', 'Boratos', 'Conchuelas', 'Puzolana', 'Cuarcita',
       'Pizarra', 'Onix', 'Ocre', 'Caolín', 'Pirofilita', 'Talco',
       'Fosfatos', 'Arcilla caolinítica', 'Granito Gris', 'Salmuera',
       'Feldespatos', 'Andesita', 'Arcilla refractaria', 'Manganeso',
       'Fluorita', 'Micas', 'Bauxita'], dtype=object)
```

Figura 4.44: Valores del atributo "NOMBRE_COM".

A continuación, se modificará la función "n_pred" de tal forma que solo se detecten aquellos cluster de yacimientos que son del mismo tipo.

```
#Función que comprueba si un punto (p2) forma parte de la vecindad de otro punto (p1).
def n_pred(p1, p2):
    return all([math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2) <= 0.2,
                p1.attributes["Tipo"] == p2.attributes["Tipo"]])
```

Figura 4.45: Función "n_pred" para yacimientos del mismo tipo.

Al aplicar GDBSCAN se ha obtenido el siguiente resultado.

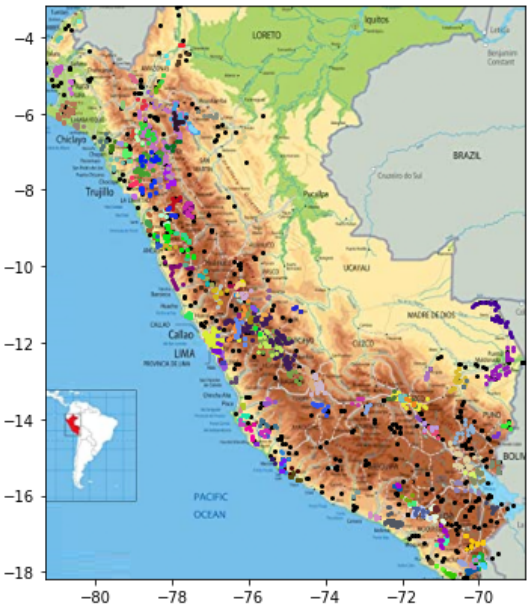


Figura 4.46: Yacimientos del mismo tipo (Eps = 0.2 Mincard = 4).

En total se han detectado 150 clusters diferentes. Como se puede apreciar en la imágen superior existe un gran número de agrupamientos de pequeño tamaño y resulta complejo distinguirlos pues se encuentran muy próximos entre sí. Por ello, con el fin de eliminar los clusters menores y mostrar solo aquellas zonas con mayor densidad, se aumentará el valor de “Mincard” a 7 y se reducirá el valor de “Eps” a 0.12. Obteniendo el siguiente resultado.

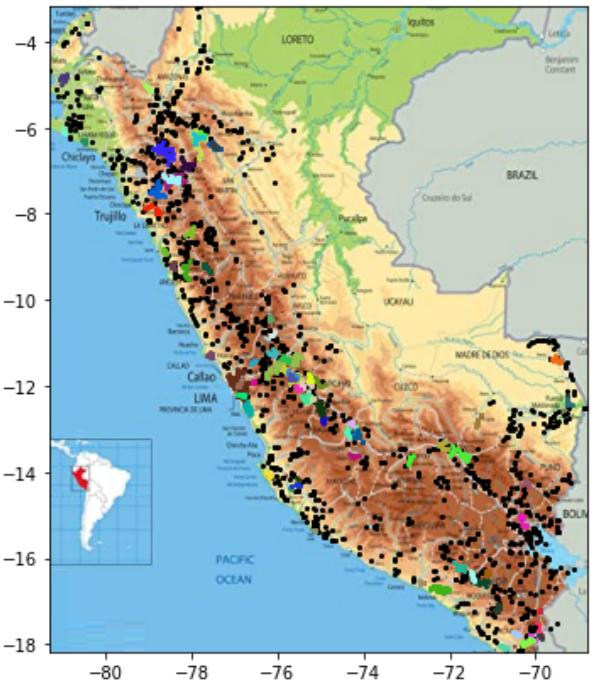


Figura 4.47: Yacimientos del mismo tipo (Eps = 0.12 Mincard = 7).

Con los nuevos parámetros se ha detectado un total de 79 clusters. Los depósitos minerales de un mismo tipo suelen ser originados por procesos o fenómenos similares por lo que es común que se encuentren próximos unos a otros. En la imágen anterior, se puede ver que la gran mayoría de los depósitos de un mismo tipo se ubican en la zona central y al norte del sistema montañoso. Sobre todo en la zona central, donde se encuentran los principales agrupamientos.

A continuación, se repetirá el mismo proceso con la diferencia de que se intentarán detectar clusters de depósitos minerales de un mismo componente. Para ello se empleará la siguiente función “n_pred”.

```
#Función que comprueba si un punto (p2) forma parte de la vecindad de otro punto (p1).
def n_pred(p1, p2):
    return all([math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2) <= 0.12,
                p1.attributes["Nombre_componente"] == p2.attributes["Nombre_componente"]])
```

Figura 4.48: Función “n_pred” para yacimientos del mismo componente.

Al aplicar GDBSCAN, con los mismos valores para los parámetros que en la prueba anterior, se ha obtenido el siguiente resultado.

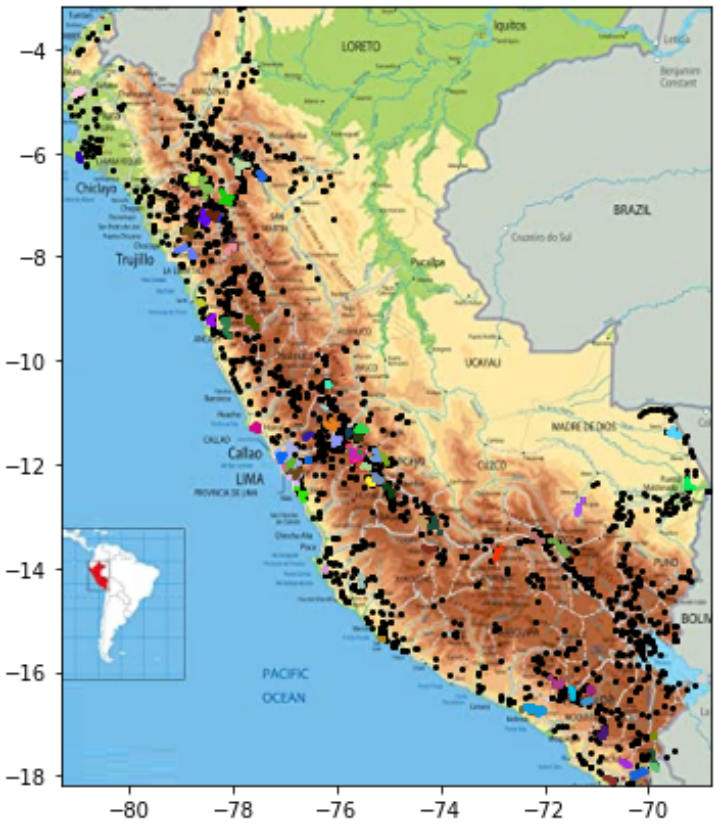


Figura 4.49: Visualización de los yacimientos del mismo componente.

En total se han detectado 76 clusters de depósitos que contienen el mismo mineral. Si nos fijamos, las zonas donde se concentran los clusters coinciden con las vistas en la prueba anterior, lo cual era de esperar, ya que un gran cluster de yacimientos de un mismo tipo muy probablemente contendrá bastantes depósitos de un mismo mineral, ya que su origen limita el que puedan existir otro tipo de minerales. Siguiendo esta línea, también es posible detectar clusters de yacimientos de un mineral concreto. A continuación, se muestran los resultados obtenidos para el caso del sílice, en el cual se han identificado 3 clusters.

```
#Función n_pred para detectar clusters de yacimientos de sílice.
def n_pred(p1, p2):
    return all([math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2) <= 0.12,
                p1.attributes["Nombre_componente"] == "Sílice",
                p2.attributes["Nombre_componente"] == "Sílice"])

def w_card(points):
    return len(points)
```

Figura 4.50: Función “n_pred” para detectar clusters de sílice.

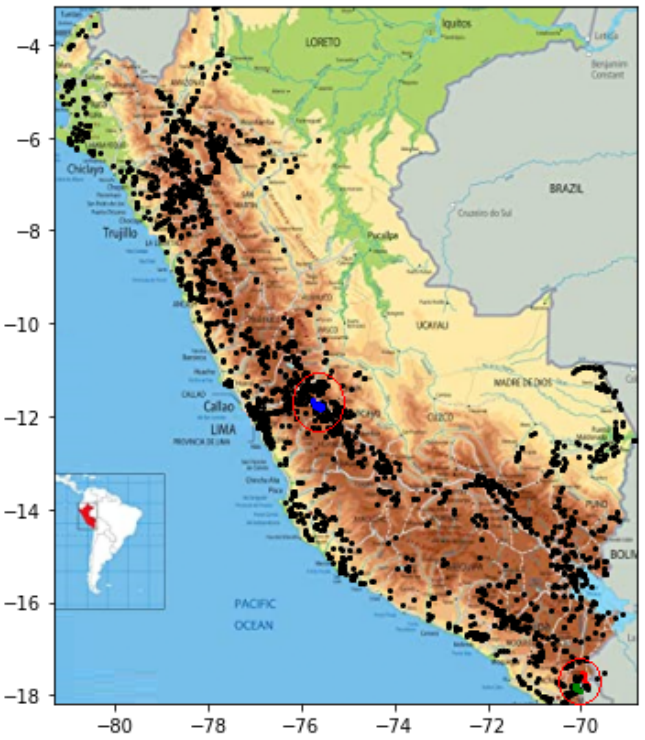


Figura 4.51: Visualización de los clusters de yacimientos de sílice.

Por último, indicar que el contexto de los datos usados dificulta el empleo de pesos en la función “w_card” ya que a diferencia del dataset artificial del área forestal, los yacimientos de un mismo tipo o mineral suelen ubicarse de manera muy próxima, por lo que no es necesario aplicar pesos para el cálculo de la cardinalidad.

4.5 Análisis de rendimiento y funcionamiento.

En este apartado se procederá a realizar diferentes pruebas con el fin de determinar las condiciones en las que el algoritmo presenta un mejor o un peor rendimiento. Para ello se realizarán pruebas con diferentes valores para los parámetros “Eps” y “Mincard”, más allá de los valores estimados mediante el gráfico de K-distancia. También se contemplarán situaciones en las que el dataset tenga muchos o pocos puntos, mucho o poco ruido, muchos o pocos clusters, etc.

Hasta ahora, en los apartados anteriores, se ha trabajado con la herramienta en la nube de Google denominada Google Collaboratory. Dicha herramienta almacena los datos y ejecuta los códigos en un servidor cuyos recursos son compartidos, por lo que probablemente existan momentos en los que dicho servidor estará más saturado, lo que puede llevar a que el tiempo de ejecución del programa se vea afectado y por tanto no ser el real. Por ello, se ha decidido llevar a cabo la ejecución del programa para las pruebas en una máquina local. El equipo en cuestión posee las siguientes características.

Sistema operativo	Windows 8.1
Tipo de sistema	64 bits
Procesador	Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
Memoria RAM	8 GB

Tabla 4.3: Características del equipo.

En primer lugar, se procederá a realizar pruebas con los parámetros “Eps” y “Mincard”. Dichas pruebas se realizarán sobre un dataset que consta de 5000 datos, el cual se visualiza a continuación.

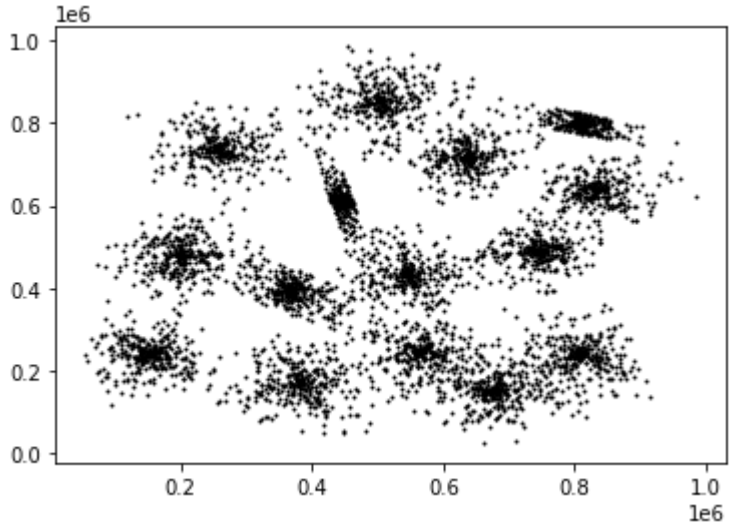


Figura 4.52: Datos para la prueba con “Eps” y “Mincard”.

Primero se procederá a estimar los parámetros “Mincard” y “Eps” con el gráfico de K-distancia.

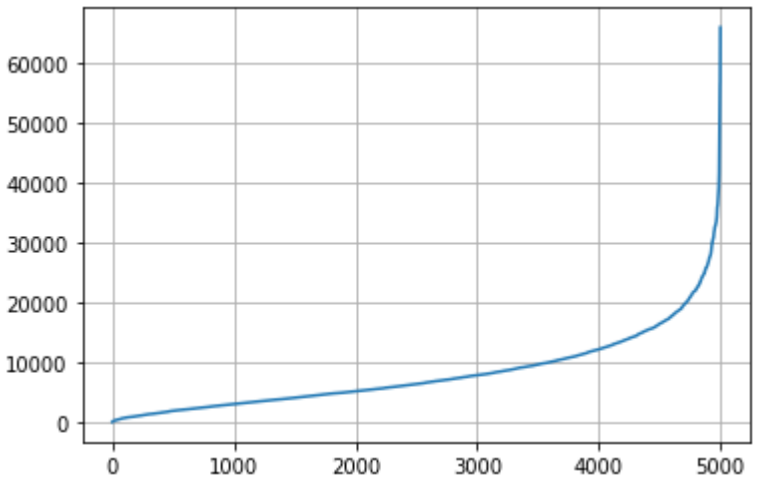


Figura 4.53: Diagrama K-distancia.

Según se puede ver en la gráfica, el valor adecuado para “Eps” estaría en torno a 20000, sin embargo, en el dataset existen clusters que se encuentran muy próximos, como los tres presentes en la esquina inferior derecha, por ello el valor de “Eps” se ha reducido hasta 15000, con el fin de garantizar que se detectan correctamente los 15 clusters principales. En cuanto al valor de “Mincard” se establecerá en 9, ya que en este caso el número de puntos es superior al del ejercicio anterior.

En esta prueba, se ejecutará el algoritmo y se medirá el tiempo que requiere con los parámetros anteriores. Este proceso se repetirá con valores de “Eps” y “Mincard” diferentes, con el fin de comprobar cómo influye en el rendimiento del algoritmo y en el resultado

obtenido, el hecho de usar parámetros que no han sido estimados correctamente. A continuación se pueden ver los resultados, en la descripción de cada imágen se muestra el valor de los parámetros empleados.

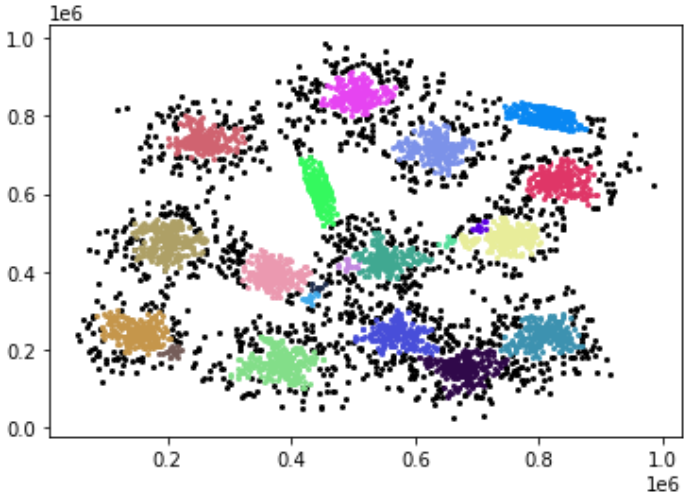


Figura 4.54: Eps = 15000 Mincard = 9 (21 clusters).

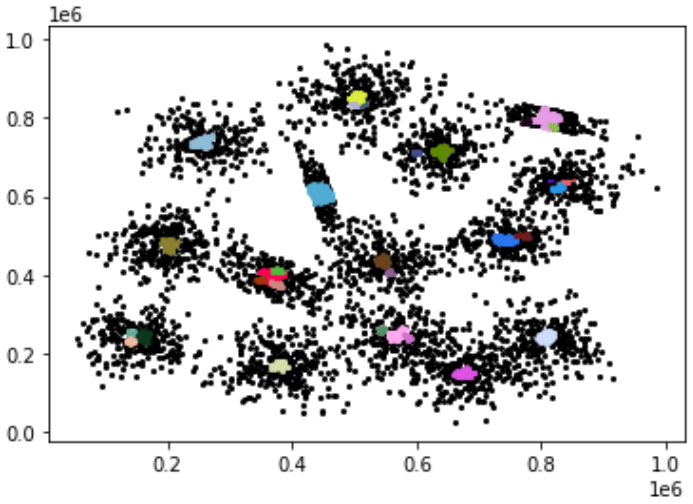


Figura 4.55: Eps = 6000 Mincard = 9 (31 clusters).

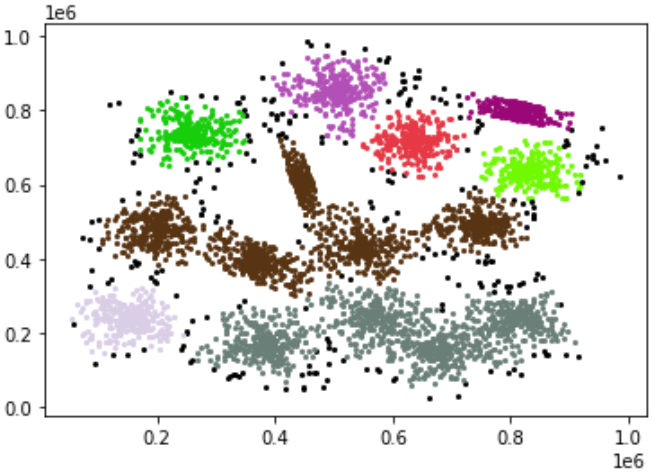


Figura 4.56: Eps = 25000 Mincard = 9 (8 clusters).

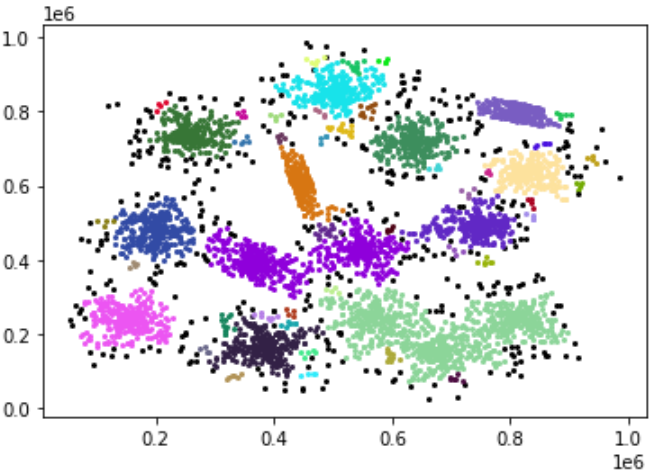


Figura 4.57: Eps = 15000 Mincard = 4 (50 clusters).

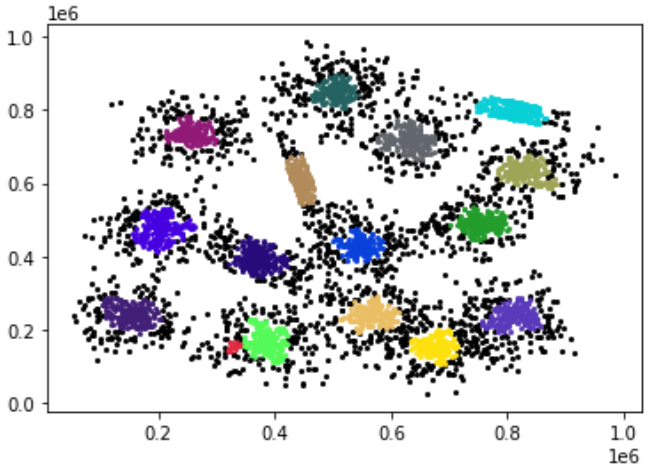


Figura 4.58: Eps = 15000 Mincard = 14 (16 clusters).

A continuación, se muestra una tabla en la que se resumen los resultados obtenidos.

Parámetros	Tiempo (segundos)	Número de clusters	Presencia de ruido	Características
Eps = 15000 Mincard = 9	39,034	21	Normal	Se detectan los 15 cluster principales más alguno secundario.
Eps = 6000 Mincard = 9	39,319	31	Muy elevada	Se detecta sólo la parte más densa de los clusters principales. Aparecen muchos clusters secundarios.
Eps = 25000 Mincard = 9	39,409	8	Poco	No se detectan correctamente los clusters principales.
Eps = 15000 Mincard = 4	39,273	50	Poco	No se detectan correctamente los clusters principales. Existen muchos clusters irrelevantes (de pequeño tamaño).
Eps = 15000 Mincard = 14	39,316	16	Alto	Se detectan los 15 clusters principales.

Tabla 4.4: Resultados de las pruebas con “Eps” y “Mincard”.

Como se puede apreciar en la tabla anterior, en general, el rendimiento ofrecido por el algoritmo, en términos de tiempo, es muy similar en las 6 pruebas. Sin embargo, donde se aprecia una diferencia notable es en los resultados ofrecidos, donde se puede apreciar 3 tipos de resultados. Por ejemplo, en las pruebas 3 y 4 no se han distinguido correctamente los 15 clusters principales, pues existen agrupamientos que están bien definidos y que, sin embargo, han sido unidos en un mismo cluster. Por otro lado, se encuentra la prueba 2, en la que sí se distinguen los clusters principales, aunque el escaso valor de “Eps” ha llevado a que los clusters sean demasiado pequeños, dando lugar a mucho ruido y a que se generen otros clusters de pequeño tamaño en medio de agrupamientos muy densos. Por último, se encuentran las pruebas 1 y 5, las cuales han ofrecido los mejores resultados, pues se han detectado correctamente los clusters principales, la cantidad de ruido detectada es normal y el número de clusters secundarios detectados es aceptable.

A continuación, se realizarán pruebas más concretas con bases de datos de gran tamaño, con el fin de comprobar en qué circunstancias el algoritmo presenta un mayor eficiencia. Primero se realizará una prueba sencilla con datasets con diferentes cantidades de puntos, pero con el mismo número de clusters. Se examinará el tiempo que se ha requerido en cada caso. En cuanto a los parámetros, se estimarán de la misma forma que se ha estado haciendo hasta ahora. A continuación, se puede ver uno de los ejemplos probados, en concreto el del dataset de 10000 puntos. El resto de datasets empleados en esta prueba poseen la misma forma.

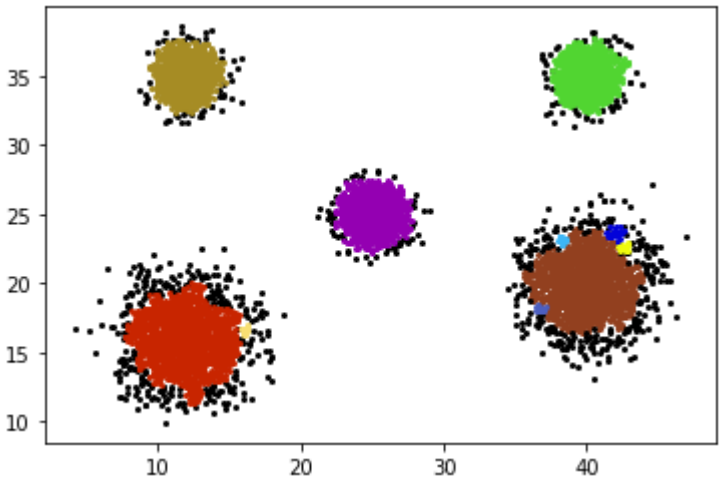


Figura 4.59: Dataset con 10000 puntos.

Cantidad de puntos	Valores de los parámetros	Tiempo (segundos)
5000	Eps = 0,5 Mincard = 5	43,05
10000	Eps = 0,4 Mincard = 10	173,65
15000	Eps = 0,35 Mincard = 15	444,49
30000	Eps = 0,25 Mincard = 30	1967,76

Tabla 4.5: Resultados de la prueba con distintas cantidades de puntos.

Como es obvio, cuanto mayor es el número de puntos del dataset empleado mayor es el tiempo requerido por el algoritmo para llevar a cabo la detección de los clusters. El algoritmo GDBSCAN posee una complejidad de $O(n^2)$ ya que por cada elemento del dataset se necesita revisar, a su vez, la totalidad del dataset para encontrar su vecindad. El hecho de que la complejidad sea $O(n^2)$ implica que el tiempo requerido por el algoritmo para detectar los clusters no experimenta un crecimiento lineal respecto al número de puntos del dataset. Lo cual se puede ver claramente reflejado en los resultados obtenidos. Ya que por ejemplo, en la prueba de 5.000 puntos el algoritmo tardó 43,05 segundos, si el crecimiento fuera lineal se podría esperar que en la prueba de 10.000 puntos el algoritmo tardará en torno a los 86 segundos, sin embargo, el tiempo ha sido de 173,65. Pues para el primer caso se ha tenido que realizar 25.000.000 de accesos a memoria, mientras que en el segundo el número de accesos ha sido de 100.000.000, en concreto se ha requerido el cuádruple. Si fijándonos en la diferencia de accesos necesarios se multiplica 43,05 por 4 se obtendría como resultado 172,2 el cual es una valor mucho más cercano a los 173,65 segundos del ejemplo de los 10.000 puntos.

Como se vió anteriormente, la cantidad de puntos del dataset es el principal aspecto que influye en el tiempo de ejecución del algoritmo. A continuación, se analizarán aspectos como la influencia del ruido o del número de clusters, donde se podrá ver que su efecto sobre el tiempo de ejecución es mínimo.

Para analizar la influencia del ruido se han creado dos datasets con el mismo número de puntos y clusters, pero con la diferencia de que uno tiene más ruido que el otro. A continuación se muestran ambos conjuntos de datos

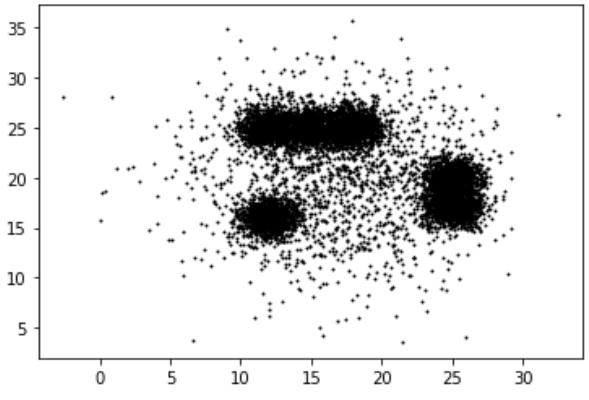


Figura 4.60: Dataset con mucho ruido.

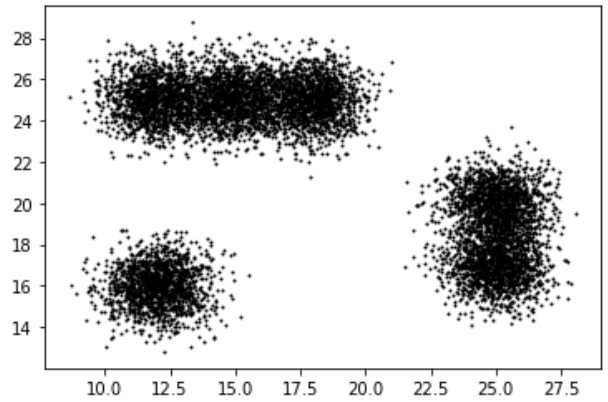


Figura 4.61: Dataset con poco ruido.

El resultado de aplicar el algoritmo es el que se muestra en la siguiente tabla.

Cantidad de puntos	Parámetros	Ruido	Tiempo (segundos)
9000	Eps = 0,25 Mincard = 9	Poco	142,24
9000	Eps = 0,5 Mincard = 9	Mucho	143

Tabla 4.6: Resultados de las pruebas con ruido.

Como se puede ver, la diferencia en tiempo de ejecución es de tan solo 0,76 segundos, esto se debe a lo siguiente. El algoritmo, en su función principal, selecciona un punto del dataset y lo examina a través de la función "_expand_cluster" de manera que si el punto en cuestión reúne las características necesarias generará un nuevo cluster, por lo que por cada cluster detectado en el dataset existe una llamada a dicha función. La función "_expand_cluster" además de detectar clusters también se encarga de identificar el ruido, de manera que por cada punto considerado ruido en el resultado generado por el algoritmo, ha de existir una llamada a esa función. Esto implica que aquellos datasets que posean más ruido requerirán un mayor número de llamadas a "_expand_cluster" que en aquellos datasets en los que el ruido sea escaso.

Para estudiar la influencia del número de clusters, se han creado dos datasets con el mismo número de puntos, pero con distinto número de clusters. Para garantizar que se detectan los clusters esperados, se han ajustado los parámetros para que detecten las zonas más densas, evitando así la aparición de agrupamientos secundarios que afecten al experimento. A continuación se muestran los datasets empleados.

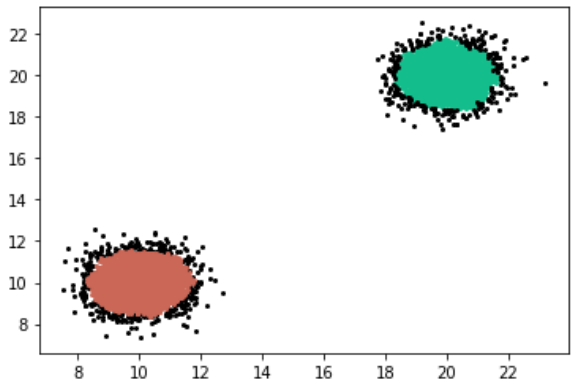


Figura 4.62: Dataset con pocos clusters.

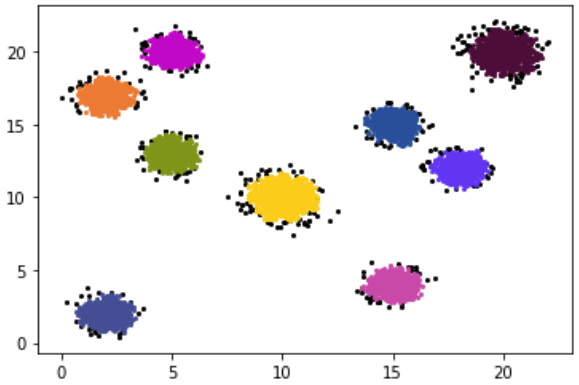


Figura 4.63: Dataset con muchos clusters.

Cantidad de puntos	Parámetros	Número de clusters	Tiempo (segundos)
9000	Eps = 0,19 Mincard = 20	2	141,05
9000	Eps = 0,25 Mincard = 9	9	141,14

Tabla 4.7: Resultados de las pruebas con distinto número de clusters.

Como se puede ver el tiempo de ejecución requerido en ambos casos es muy similar, siendo ligeramente superior en el del dataset con mayor número de clusters. En teoría, el hecho de que existan más clusters implica un mayor número de llamadas a la función "_expand_cluster" desde la función principal del algoritmo GDBSCAN. Sin embargo, su influencia en el tiempo de ejecución es casi inapreciable, sobre todo si se compara con las N^2 consultas a memoria que realiza el algoritmo.

4.6 Mejora del algoritmo.

Como se ha podido observar el algoritmo GDBSCAN posee un funcionamiento bastante sencillo, en el cual tiene especial relevancia el proceso de búsqueda de los vecinos de cada punto. Esta operación se realiza para cada elemento del conjunto de datos por lo que sería de interés agilizar dicho proceso con el fin de lograr una mayor eficiencia.

En la sección en la que se explicó el código, se pudo ver que los datos se almacenan en memoria en crudo, es decir, no se emplea ningún tipo de estructura o mecanismo que agilice el proceso de búsqueda de los vecinos de cada punto en el dataset. Por ello, con el fin de mejorar la eficiencia del algoritmo se ha decidido emplear un árbol KD o "KD tree".

Un "KD tree" es una estructura de datos de particionado del espacio mediante hiperplanos que organiza los puntos en un espacio euclídeo de "K" dimensiones. Los árboles KD son un caso especial de los árboles BSP (Partición binaria del espacio). Un árbol KD emplea sólo planos perpendiculares a los ejes del sistema de coordenadas. Esto difiere de los árboles BSP, donde los planos pueden ser arbitrarios. A continuación se puede ver un ejemplo sencillo de un árbol KD bidimensional.

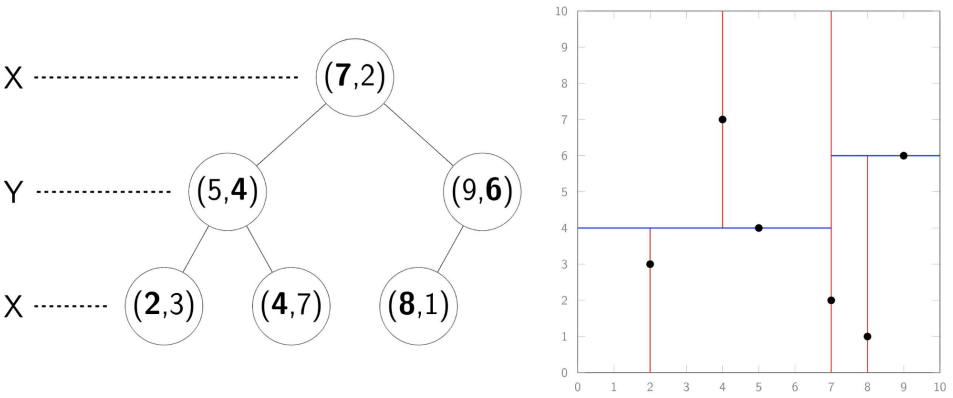


Figura 4.64: Ejemplo de árbol KD bidimensional [25].

En este caso, para la creación del árbol KD se ha empleado el módulo "sklearn.neighbors", el cual contiene la clase "KDTree". A dicha función se le pasará por parámetro un array con las coordenadas de cada punto para obtener el árbol. Una vez creado dicho árbol, se pasará por parámetro a la clase "Points" junto a la lista de puntos. Dicho objeto "Points" se pasa a la función principal de GDBSCAN junto al resto de parámetros ya mencionados en apartados anteriores. A continuación se muestra el código empleado para ello.

```
#Leer datos desde fichero
csv_data=pd.read_csv('/content/10000.csv',index_col=0)

#Se obtiene una lista con las coordenadas de los puntos para kdtree y
#los datos se encapsulan en objetos "Point"
coordinates = []
points = []
for i in range(len(csv_data)):
  coordinates.append([csv_data.iloc[i,0],csv_data.iloc[i,1]])
  points.append(Point(csv_data.iloc[i,0],csv_data.iloc[i,1],{}))

#Se crea el árbol kdtree
kdtree = KDTree(np.array(coordinates))

#Se aplica GDBSCAN
clustered = GDBSCAN(Points(kdtree,points), n_pred, 10, w_card)
print(clustered)
```

Figura 4.65: Creación del árbol KD.

En la clase "Points" se han realizado dos cambios. Por un lado, se ha añadido el atributo "kdtree", y por otro se ha modificado el método "neighborhood" que se encarga de obtener la vecindad de un punto.

```
def __init__(self, kdtree, points):
    self.kdtree = kdtree
    self.points = points
```

Figura 4.66: Se añade el atributo “kdtree”.

```
def neighborhood(self, point, n_pred):
    points_id = kdtree.query_radius([[point.x,point.y]], r=EPS)
    result = []
    for id in points_id[0]:
      result.append(self.points[id])

    return result
```

Figura 4.67: Modificación del método “neighborhood”.

Como se puede ver en la imágen anterior, en el método “neighborhood” se realiza las consultas al árbol KD. Para ello se emplea la función “query_radius”, a la cual se le pasa por parámetro las coordenadas del punto del cual se quiere obtener la vecindad y el radio en el que se desean buscar los vecinos de ese punto, que en esta caso es “Eps”. Dicha función, retorna un array con la ubicación de cada vecino, que será usada para localizar los puntos en cuestión en la lista “points”. Una vez obtenidos los puntos, serán retornados por la función.

El uso del árbol KD ha implicado una importante mejora en términos de eficiencia. A continuación se muestra una tabla con los tiempos de ejecución obtenidos en la prueba con datasets de distintas cantidades de puntos de la fase de análisis del rendimiento, y al lado se han añadido los resultados conseguidos con el algoritmo modificado.

Cantidad de puntos	Tiempo del algoritmo sin modificar (segundos)	Tiempo del algoritmo modificado (segundos)
5000	43,05	0,9
10000	173,65	2,57
15000	444,49	4,68
30000	1967,76	14,15
100000	X	158,49

Tabla 4.8: Comparativa del tiempo de ejecución antiguo con el nuevo.

Como se puede apreciar en la tabla anterior, la mejora en el rendimiento ha sido bastante elevada. Esto se debe a que al aplicar un árbol KD para la búsqueda de los vecinos, la complejidad vista anteriormente de $O(n^2)$ pasa a ser de $O(n \log(n))$, de ahí que los tiempos de ejecución sean tan reducidos.

Capítulo 5 Conclusiones y líneas futuras

5.1 Conclusiones.

Como se ha podido observar los algoritmos GDBSCAN y DBSCAN comparten, en esencia, el mismo funcionamiento, con la diferencia de que GDBSCAN no se limita tan solo al uso de atributos espaciales, sino que también incorpora los atributos no espaciales al proceso de obtención de la vecindad de un punto y al cálculo de la cardinalidad. Ofreciendo con ello un amplio abanico de operaciones. En el presente proyecto se han visto, a través del empleo de un dataset artificial, los principales ejemplos, pero existen multitud de combinaciones dependiendo de la variedad de atributos no espaciales de los que se disponga. También se ha podido comprobar que su uso se puede extender a los conjuntos de datos reales, como se vió con los datos sobre depósitos minerales en Perú. Por otro lado, en el análisis de rendimiento del algoritmo se pudo comprobar que la estimación de los parámetros es fundamental si se quiere obtener un resultado adecuado, y que en lo relativo a las características de los datasets el número de puntos es el factor que más afecta al tiempo de ejecución, seguido por otra característica como es la presencia de ruido. Por último, se pudo comprobar que el uso de un árbol KD para organizar los datos del dataset agiliza notablemente el proceso de búsqueda de la vecindad de cada punto, mejorando con ello el rendimiento del algoritmo.

5.2 Líneas futuras.

El algoritmo en cuestión posee algunas vías adicionales por las que continuar el desarrollo. Primero, se deberán desarrollar heurísticas para determinar los parámetros para GDBSCAN donde wCard es diferente de la función de cardinalidad. En segundo lugar, DBSCAN crea un agrupamiento de un solo nivel, sin embargo, una agrupación jerárquica puede ser más útil, en particular si los parámetros de entrada apropiados no se pueden estimar con precisión. Por lo que se deberá investigar una extensión de GDBSCAN para detectar simultáneamente una jerarquía de agrupaciones.

Capítulo 6 Summary and Conclusions

6.1 Summary.

In this project, the GDBSCAN algorithm and its specific version DBSCAN have been studied. In the first place, its specific concepts, its operation and how to estimate the parameters for its operation have been analyzed. The K-distance graph was used for the estimation.

Next, an artificial dataset was created with the necessary characteristics to be able to show the potential of the GDBSCAN algorithm. Two tests were carried out with that dataset. In the first test, non-spatial attributes were used to calculate the neighborhood, allowing to find plant groups of the same species and of a certain height. In the second test, non-spatial attributes were used as weights to calculate cardinality.

Then, to provide a more realistic approach, a real dataset was sought, in this case related to the mineral deposits of Peru. The techniques mentioned above were applied to it.

On the other hand, the operation and performance of the algorithm was analyzed. In this phase of the project, the effect of a poor choice of parameters was checked. And different characteristics of the datasets were experimented with to find out if they have any influence on the running time of the program.

Finally, with the aim of improving the algorithm performance, an attempt was made to speed up the neighborhood search process by using a KD tree.

6.2 Conclusions.

The GDBSCAN and DBSCAN algorithms have practically the same operation, with the difference that GDBSCAN is not limited only to the use of spatial attributes, but also uses non-spatial attributes to obtain the neighborhood of a point and calculate the cardinality. Allowing a great variety of operations.

In the project, the main examples have been made in an artificial dataset, where it was seen there are many operation combinations depending on the variety of non-spatial attributes available. GDBSCAN has also been used in real dataset, as seen with data on mineral deposits in Peru.

On the other hand, in the performance analysis of the algorithm it was found that the parameters estimation is important to obtain a good result. Regarding the characteristics of the datasets, the number of points is the factor that most affects execution time, followed by another characteristic such as the presence of noise. Finally, the use of a KD tree to organize the data made it possible to speed up the searching process for the neighborhood of each point. Which improved the performance of the algorithm.

Capítulo 7 Presupuesto

En este capítulo se realizará una estimación del presupuesto que se requeriría para llevar a cabo el proyecto en cuestión.

Fase 1: Estudio de los algoritmos.			
Tarea	**Precio €/horas**	**Número horas**	**Precio por tarea**
Estudio del algoritmo DBSCAN.	40 €	5	200 €
Estudio del algoritmo GDBSCAN.	40 €	5	200 €
	Presupuesto de la fase 1 = 400 €		

Tabla 7.1: Presupuesto de la fase de estudio de los algoritmos.

Fase 2: Análisis y mejora.			
Tarea	**Precio €/horas**	**Número horas**	**Precio por tarea**
Análisis del problema y de los datos	40 €	6	240 €
Realización de la tarea de clustering.	60 €	20	1200 €
Análisis de rendimiento y funcionamiento	50 €	5	250 €
Mejora del algoritmo	50 €	5	250 €
	Presupuesto de la fase 2 = 1940 €		

Tabla 7.2: Presupuesto de la fase de análisis y mejora.

Presupuesto total del proyecto = Presupuesto fase 1 + Presupuesto fase 2 = 2340 €

Capítulo 8 Apéndice A.

8.1 Sección 1: Código empleado.

A continuación, se proporciona un enlace al proyecto en Google Collaboratory donde se podrá ver y probar el código python que se ha mostrado a lo largo del presente proyecto.

Enlace a Google Collaboratory.

8.2 Sección 2: Conjuntos de datos empleados.

A continuación, se proporciona un enlace a una carpeta compartida de Google Drive en la que se han depositado los datasets empleados a lo largo del proyecto en formato CSV. También contiene la imágen de Perú usada de fondo en la representación gráfica del dataset real.

Enlace a Google Drive.

Bibliografía

1. ITS El Grullo. *Minería de Datos* [en línea]. Recuperado de https://sites.google.com/site/itsginteligenciadenegocios/home/1-2-componentes-de-la-inteligencia-de-negocios/1-2-1-mineria-de-datos?tmpl=%2Fsystem%2Fapp%2Ftemplates%2Fprint%2F&showPrintDialog=1 (accedido el 20/05/2021).
2. Ribas, Ester. *¿Qué es el Data Mining o minería de datos?* [en línea]. Recuperado de https://www.iebschool.com/blog/data-mining-mineria-datos-big-data/ (accedido el 20/05/2021).
3. Moreno Vegas, J.Marcos. *Extracción de conocimiento en bases de datos*. Recuperado de https://campusdoctoradoyposgrado.ull.es/mod/resource/view.php?id=275253 (accedido el 20/05/2021).
4. *Aprendizaje automático* [en línea]. Recuperado de https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico (accedido el 22/05/2021).
5. *Aprendizaje automático (machine learning)* [en línea]. Recuperado de https://searchdatacenter.techtarget.com/es/definicion/Aprendizaje-automatico-machine-learning (accedido el 22/05/2021).
6. Moreno Vegas, J.Marcos. *Árboles de decisión*. Recuperado de https://campusdoctoradoyposgrado.ull.es/mod/resource/view.php?id=275256 (accedido el 22/05/2021).
7. Moreno Vegas, J.Marcos. *Árboles de regresión*. Recuperado de https://campusdoctoradoyposgrado.ull.es/mod/resource/view.php?id=275269 (accedido el 22/05/2021).
8. *Aprendizaje no supervisado* [en línea]. Recuperado de https://es.wikipedia.org/wiki/Aprendizaje_no_supervisado (accedido el 22/05/2021).
9. Moreno Vegas, J.Marcos. *Clustering basado en prototipos*. Recuperado de https://campusdoctoradoyposgrado.ull.es/mod/resource/view.php?id=275278 (accedido el 22/05/2021).
10. Moreno Vegas, J.Marcos. *Clustering jerárquico*. Recuperado de https://campusdoctoradoyposgrado.ull.es/mod/resource/view.php?id=275280 (accedido el 22/05/2021).
11. *Cómo funciona el clustering basado en densidad* [en línea]. Recuperado de https://pro.arcgis.com/es/pro-app/latest/tool-reference/spatial-statistics/how-density-based-clustering-works.htm (accedido el 24/05/2021).

12. *Aprendizaje no supervisado* [en línea]. Recuperado de https://aprendeia.com/aprendizaje-no-supervisado-machine-learning/ (accedido el 10/06/2021).
13. *Aprendizaje semi-supervisado* [en línea]. Recuperado de https://es.wikipedia.org/wiki/Aprendizaje_semisupervisado (accedido el 10/06/2021).
14. *Aprendizaje semi-supervisado* [en línea]. Recuperado de http://www.predictiva.com.co/blog-predictiva/aprendizaje-semi-supervisado/ (accedido el 10/06/2021).
15. Merino, Marcos. *Conceptos de inteligencia artificial: qué es el aprendizaje por refuerzo* [en línea]. Recuperado de https://www.xataka.com/inteligencia-artificial/conceptos-inteligencia-artificial-que-aprendizaje-refuerzo (accedido el 10/06/2021).
16. *OPTICS algorithm* [en línea]. Recuperado de https://en.wikipedia.org/wiki/OPTICS_algorithm#:~:text=Ordering%20points%20to%20identify%20the,based%20clusters%20in%20spatial%20data.&text=Additionally%2C%20a%20special%20distance%20is,belong%20to%20the%20same%20cluster (accedido el 15/06/2021).
17. *Cómo funciona HDBSCAN* [en línea]. Recuperado de https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html (accedido el 15/06/2021).
18. *Cómo funciona el clustering basado en densidad* [en línea]. Recuperado de https://pro.arcgis.com/es/pro-app/latest/tool-reference/spatial-statistics/how-density-based-clustering-works.htm#:~:text=Autoajuste%20(HDBSCAN)%20es%20el%20m%C3%A9todo,la%20menor%20participaci%C3%B3n%20del%20usuario.&text=En%20Escala%20m%C3%BAltiple%20(OPTICS)%2C,y%20valles%20dentro%20del%20diagrama (accedido el 15/06/2021).
19. C Nwadiugwu, Martin. *Gene-Based Clustering Algorithms: Comparison Between Denclue, Fuzzy-C, and BIRCH* [en línea]. Recuperado de https://journals.sagepub.com/doi/full/10.1177/1177932220909851 (accedido el 15/06/2021).
20. Morales, Eduardo. Jair Escalante, Hugo. *Clustering* [en línea]. Recuperado de https://ccc.inaoep.mx/~emorales/Cursos/NvoAprend/Acetatos/clustering.pdf (accedido el 22/06/2021).
21. *¿Qué es el Deep Learning?* [en línea]. Recuperado de https://www.smartpanel.com/que-es-deep-learning/ (accedido el 22/06/2021).
22. *Introducción al Deep Learning* [en línea]. Recuperado de https://relopezbriega.github.io/blog/2017/06/13/introduccion-al-deep-learning/ (accedido el 22/06/2021).
23. *Clustering basado en densidad* [en línea]. Recuperado de https://elvex.ugr.es/idbis/dm/slides/43%20Clustering%20-%20Density.pdf (accedido el 22/06/2021).
24. *Map of Peru (Physical)* [en línea]. Recuperado de https://www.worldometers.info/maps/peru-map/ (accedido el 25/07/2021).

25. *Árbol K-D, árbol K-D-B, árbol B-K-D* [en línea]. Recuperado de https://programmerclick.com/article/29581187616/ (accedido el 24/08/2021).
26. Jörg Sander, Martin Ester, Hans-Peter Kriegel, Xiaowei Xuh. *Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and its Applications* [en línea]. Recuperado de https://staff.fmi.uvt.ro/~daniela.zaharie/dm2018/en/Projects/Biblio/DBSCAN/DBSCAN_1.pdf (accedido el 03/04/2021).
27. *GDBSCAN* [en línea]. Recuperado de https://github.com/eahlberg/gdbscan/tree/master/gdbscan (accedido el 03/04/2021).
28. *Algoritmo OPTICS* [en línea]. Recuperado de https://en.wikipedia.org/wiki/OPTICS_algorithm#Basic_idea (accedido el 03/09/2021).