UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Práctica 2. Arquitecturas No Supervisadas (GNG) con Python y Tensorflow.

Asignatura: Sistemas Inteligentes 1

Resumen:

Análisis y Diseño de Arquitecturas Neuronales No Supervisadas (*Growing Neural Gas*; GNG).

Carlos Brito Pérez

Pablo Morales Gómez

Grado en: Ingeniería Informática. 4º Curso

ÍNDICE

Introducción 3
Funcionamiento de la Red 3
Parámetros de la Red 4
Criterios de parada 5
Implementaciones 6
Saver y Loader 6
Plotter 6
Predict 6
Cluster sample 7
Preprocesado de los Datos Propuestos 8

Introducción

El objetivo de esta práctica era familiarizarse con el funcionamiento de la red neuronal no supervisada *Growing Neural Gas*. Debíamos ampliar el código base con diferentes funciones y clases tales como métodos para predecir el comportamiento al introducir un dato o la lógica para la obtención de agrupamientos además de clases para guardar y cargar una red ya entrenada.

Funcionamiento de la Red

Una *Growing Neural Gas* o GNG es un sistema inteligente no supervisado que tiene dos elementos principales que la componen. El primero es una lista llamada **A** donde se almacenan las posiciones de los distintos nodos que la componen. La otra es una lista llamada **N** en la que almacenamos las conexiones entre los nodos y la edad de dichas uniones.

El funcionamiento de una GNG es el siguiente. La red comienza generando 2 nodos aleatorios a y b cuya posición vendrá dada por dos vectores en **A**. En cada iteración escogerá una de las entradas de forma aleatoria (uno de los puntos con los que alimentamos el entrenamiento de la red) que llamaremos **xi**.

De entre todo el conjunto de nodos que exista en ese momento seleccionaremos aquellos dos que se encuentren más próximos a **xi**, reciben el nombre de **s1** y **s2** respectivamente. Ahora aumentaremos la edad de todas las unidades conectadas con **s1**. Cada una de las unidades tiene asociado un valor de error, y en este momento calculamos la diferencia al cuadrado de la distancia entre **s1** y **xi**, y lo acumulamos en su correspondiente variable de error.

A continuación, movemos hacia **xi** la unidad **s1** la distancia especificada en **Epsilon_B**. A su vez, movemos las unidades vecinas de **s1** una distancia **Epsilon_N** hacia **xi**. Si **s1** y **s2** están conectadas entre sí en **N**, la edad de esta conexión se pone a 0, si no fuera así creamos una arista que une a ambos. Después buscamos las conexiones que tengan una edad superior o igual al parámetro **a_max** y las eliminamos. Asimismo, si la poda de estas conexiones deja una unidad sin conexiones, ésta debe ser eliminada.

En el siguiente paso, decidimos si incluir una nueva unidad. Lo hacemos cuando se han procesado un número **eta** de **xi**. Esta se incluirá a medio camino entre la unidad con un mayor error y su vecina con mayor error. El error de ambas unidades será, después de haber creado la

nueva unidad y haberla conectada con sus progenitores, reducida por el parámetro **alfa**. El error de la nueva será igual al de la primera unidad que habíamos escogido.

Finalmente, decrementamos el error de todas las unidades por el parámetro **delta** y si no se cumple el criterio de parada volvemos a comenzar.

Parámetros de la Red

Ahora realizaremos un listado y explicaremos la influencia de los diferentes parámetros que definen el comportamiento de la red:

- **Alfa**: cantidad con la que vamos decrementando el error de las unidades con las que creamos una nueva. Bajamos el error de estas unidades con alfa porque si no a la larga podría pasar que las unidades con mayor error siempre fueran las mismas y solo se crearían nuevos nodos en la misma conexión durante todo el entrenamiento.

- **A_Max**: edad máxima que puede tener una arista, cuando es alcanzada debe ser eliminada. Si es muy pequeña puede pasar que se separen entre sí, nodos que deberían permanecer en el mismo grupo. En cambio, si fuera muy grande nodos que deberían separarse y formar grupos diferentes no se separarían nunca.

- **Delta**: cantidad con la que decrementamos el error de todas las unidades cuando procesamos una xi. El hacer esto tras cada iteración evita que la variable error crezca de forma descontrolada y así podemos mantenerla en un valor manejable.

- **Epsilon_B**: distancia que movemos hacia xi la unidad con mayor error. De alguna forma podríamos ver este parámetro como una suerte de tasa de aprendizaje, porque si es muy grande puede que nos saltemos el punto al que tratamos de aproximarnos y que al final resulte contraproducente ese desplazamiento por acabar más lejos de donde se encontraba al inicio. Al mismo tiempo, si fuera muy pequeño daría unos pasitos demasiado insignificantes y la red evolucionaría a una velocidad muy lenta.

- **Epsilon_N**: distancia que movemos hacia xi las unidades vecinas a la de mayor error. La influencia de este parámetro es similar a la de su homónima B. Además, este valor debería ser menor que el B, puesto que estas variables no son las más próximas a la unidad y por tanto tienen que aproximarse con una mayor cautela, ya que a la larga podría observarse que estos nodos pertenecieran a otro grupo y se acaben separando del nodo que se ve afectado por la epsilon_B.

- **Eta**: cantidad de xi que debemos procesar antes de insertar una nueva unidad. A la hora de establecer el valor de este también debemos encontrar un equilibrio. Por un lado, no nos interesa que sea muy bajo, ya que inundaremos la red con nodos redundantes y no puede ser muy elevado porque es posible que necesitemos más nodos de los que tenemos para poder marcar correctamente los grupos.

Criterios de parada

Hemos estudiado la utilidad de diferentes criterios de parada, entre ellos los 3 más interesantes fueron el número máximo de unidades, el número máximo de grupos y las épocas, como suele ser habitual. Cada uno de ellos tiene su propia utilidad.

Las épocas son el mejor criterio de parada de forma general como suele ser costumbre, ya que es una buena manera de medir cuánto entrenas una red.

El número máximo de unidades sirve para controlar que la red no genere demasiadas unidades siendo una algo así como una medida de contención contra sobre entrenamiento o el uso de malos parámetros tales como las edades máximas.

Por último, el número máximo de grupos es un criterio de parada que, aunque es muy efectivo tiene el requerimiento de saber, de forma al menos aproximada, el número de *clusters*/grupos de antemano.

Por tanto, planteamos que el criterio de parada debería ser mixto. Donde uno de los componentes debería ser las épocas y el otro el número de grupos o de unidades. Este último parámetro debería tener un valor bastante elevado (mayor que el esperado), porque lo usaríamos principalmente para controlar que la red se mantuviera dentro de unos valores aceptables.

Implementaciones

Saver y Loader

La implementación que llevamos a cabo para las clases de *saver* y *loader* consistió en la creación de archivos de tipo JSON para almacenar los datos. Dichos datos fueron los pertenecientes al *self* de la clase de *GrowingNeuralGas* y los dividimos en 4 archivos diferentes, 3 de ellos para los atributos más complejos (*A, N* y *error_)* y una última para el resto, los cuales unimos en una lista para facilitar su guardado.

En el *loader* debemos volver a convertir los datos a tensores y devolverles el tipo, en este caso float32.

Aparte de crear una clase *loader* y una clase *saver* también creamos dos métodos en la clase *GrowingNeuralGas,* llamados *saveGNG* y *loadGNG* los cuales nos sirven como vía de comunicación con las clases, son estos métodos son los encargados de enviar todos los datos necesarios a sus respectivas clases y también son a los que llamaremos cuando queramos hacer uso de las funciones de *save* y *load.*

Plotter

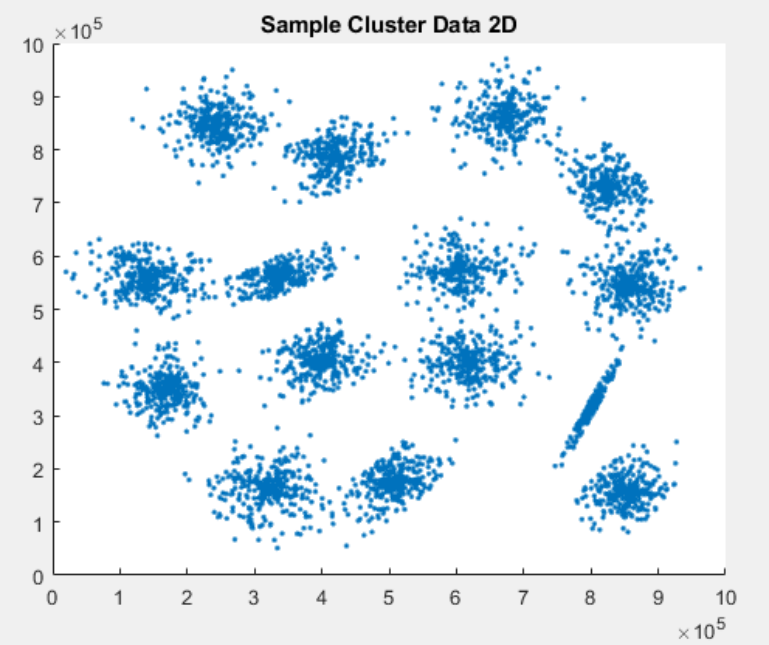
La clase *GrowingNeuralGasPlotter* nos fue proporcionada en el código base pero decidimos modificarla para que se ajustara más a nuestras preferencias y necesidades. Estos cambios consistieron principalmente en la creación de una unión en forma de línea de cada punto con sus vecinos, para esto usamos dos bucles *for* anidados el primer bucle recorre los elementos de N, mientras que el segundo comprueba los vecinos de cada elemento.

Predict

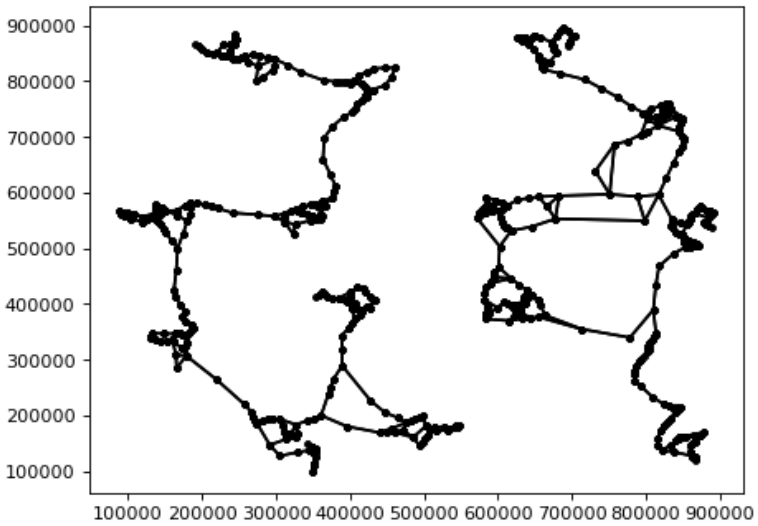
El método *predict* tiene un funcionamiento relativamente sencillo y gran parte de él consiste en llamar a funciones que ya estaban definidas con anterioridad. Primeramente llamamos a *componentesConexas()* de dónde sacamos todos los grupos. Después, tomamos el valor del que queremos conocer su grupo y con *findNearestUnit()* obtenemos la unidad que tiene más cerca. Al averiguar a que grupo pertenece dicha unidad, sabremos en que grupo está incluido el valor del que queremos obtener el resultado.

Cluster sample

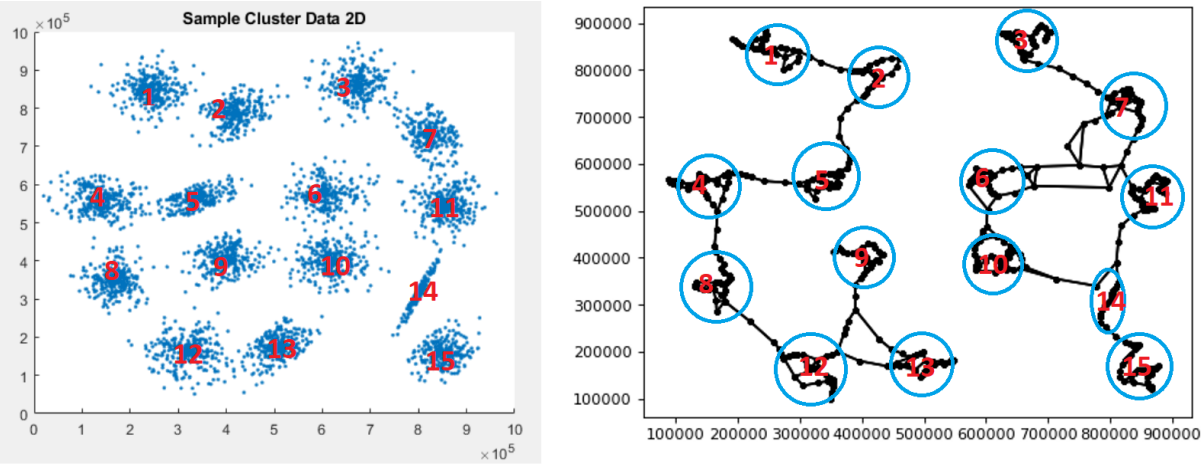
El *dataset Sample_Cluster_Data_2D* lo hemos usado como comprobación del correcto funcionamiento de la red. Esto se debe a que al observar la distribución de datos podemos observar grupos claramente diferenciados lo cual facilita dicha comparativa.



Visualización en MATLAB del *dataset*.



Output de la *Growing Neural Gas*.



Visualización de la comparativa entre el *dataset* y el output de la GNG.

Como podemos observar en la última imagen se pueden distinguir con relativa facilidad los grupos de la GNG. El resultado mostrado fue conseguido al entrenar la red con 1000 datos aleatorios del conjunto. El proceso fue con tan solo 5 épocas y llegando a tener 500 unidades. Debido al tiempo que tarda en entrenar la red con este conjunto de datos y a su función meramente ligada a la comprobación del correcto funcionamiento de la red, decidimos que este nivel de entrenamiento era suficiente demostración.

Preprocesado de los Datos Propuestos

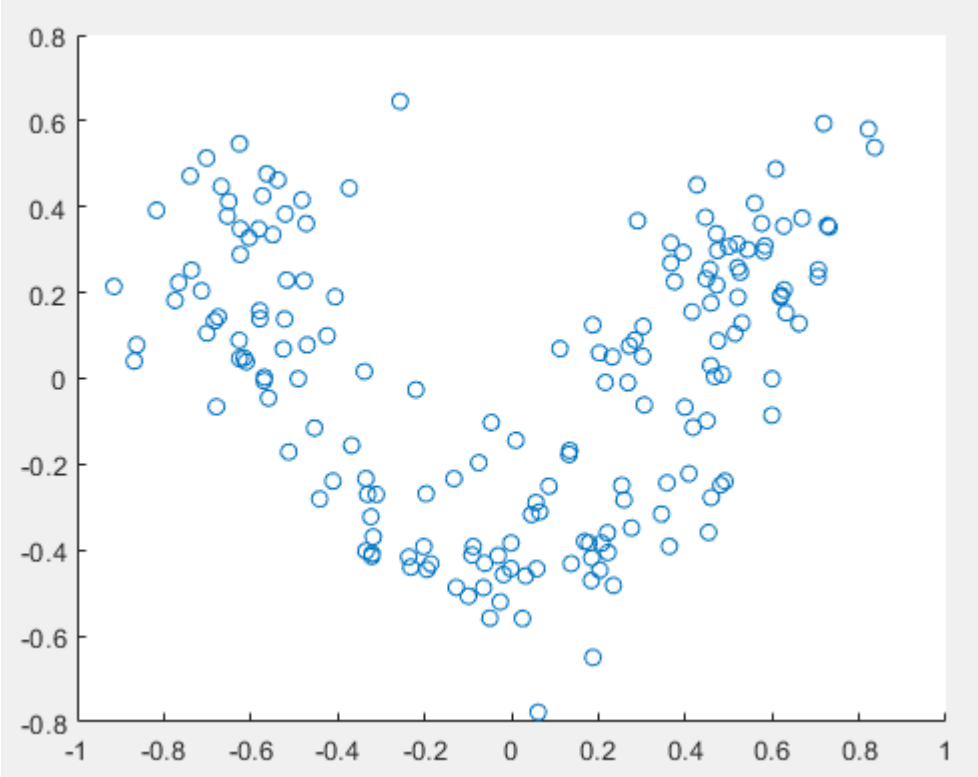
A la hora de hacer un estudio sobre uno de los 3 *dataset* propuestos se optó por escoger el de *Calisificación_Vinos.csv*. Este conjunto de datos ampliamente usado en el ámbito de la ciencia de datos representa una serie de mediciones químicas de unos vinos que se hacen en una misma región italiana por tres viticultores diferentes.

Este *dataset* presenta dos dificultades iniciales con las que tuvimos que lidiar antes de introducirlo en nuestra *Growing Neural Gas*. La primera de ellas es que cada instancia del paquete de datos cuenta con 13 atributos, es decir, 13 dimensiones a la hora de representarlo gráficamente. Por este motivo, se decidió aplicar un algoritmo de reducción de dimensiones. Concretamente utilizamos un PCA con este objetivo.

Al aplicar la PCA nos encontramos con un problema que es que la última característica del *dataset Proline* tiene un rango de valores mucho mayor que el resto de *features* por lo que, aunque el resultado nos asegura que cubre un 99% de varianza del conjunto de datos original con

tan solo dos atributos este resultado está contaminado por la diferencia del rango de esta característica. Por ello, aplicamos previamente un proceso de normalización al *dataset*.

Una vez efectuado este proceso observamos que, aunque con dos variables solo obtenemos en torno a un 60% de la varianza con respecto al conjunto original de datos la representación gráfica de los datos ha mejorado considerablemente y ya podemos distinguir 3 grupos de puntos. Estos se han distribuido trazando una parábola: dos de ellos en los extremos superiores y el tercero en el vértice de la misma. Todo este proceso lo hemos realizado en MATLAB (el código viene adjunto con el resto de los fuentes).



Visualización de los puntos en 2D tras la normalización y la PCA.

Comparación de Resultados

Para realizar un estudio sobre los efectos que tienen los parámetros en los outputs de la GNG, decidimos realizar varios entrenamientos y comparar su estado final. A continuación, mostramos algunos de ellos.

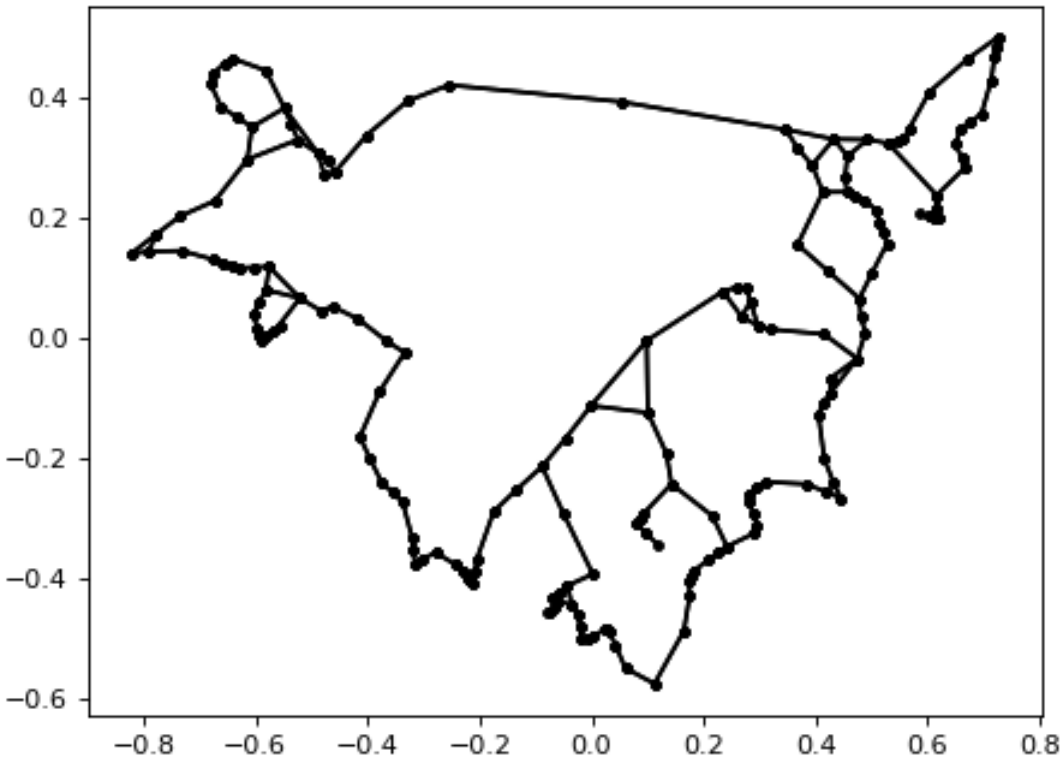


Figura 1. epsilonA_0.1-epsilonN_0.05-Max_15-eta_5-epochs_5-alpha_0.1-delta_0.1

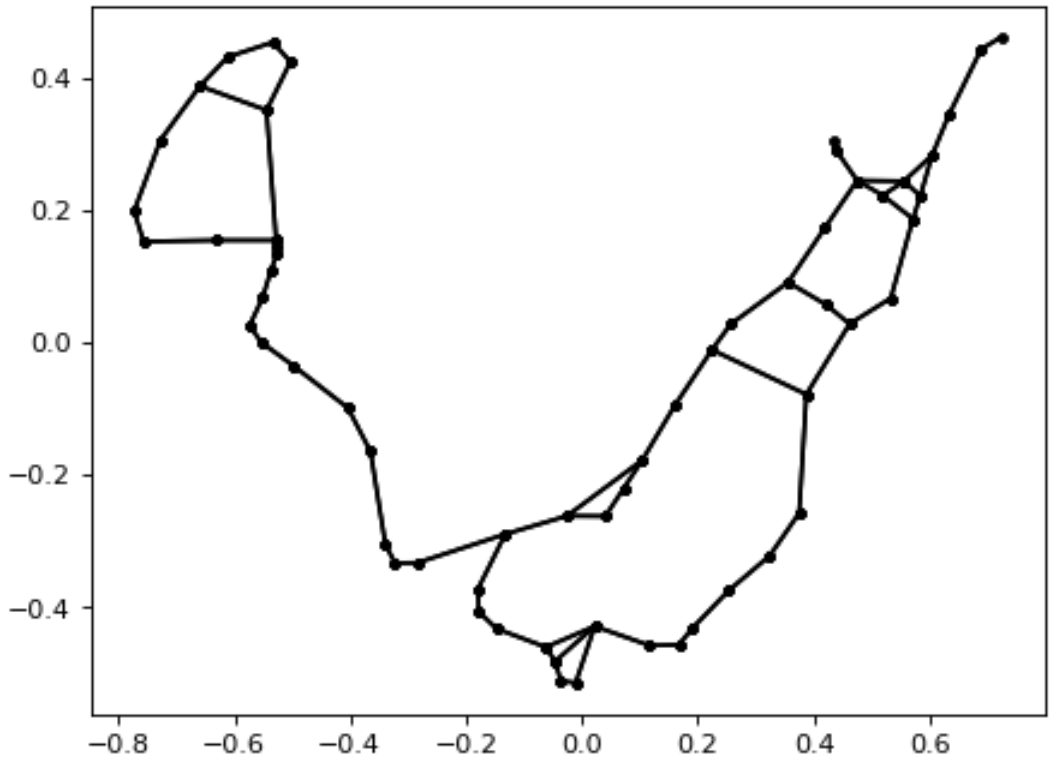


Figura 2. epsilonA_0.1-epsilonN_0.1-Max_10-eta_15-epochs_5-alpha_0.1-delta_0.1.

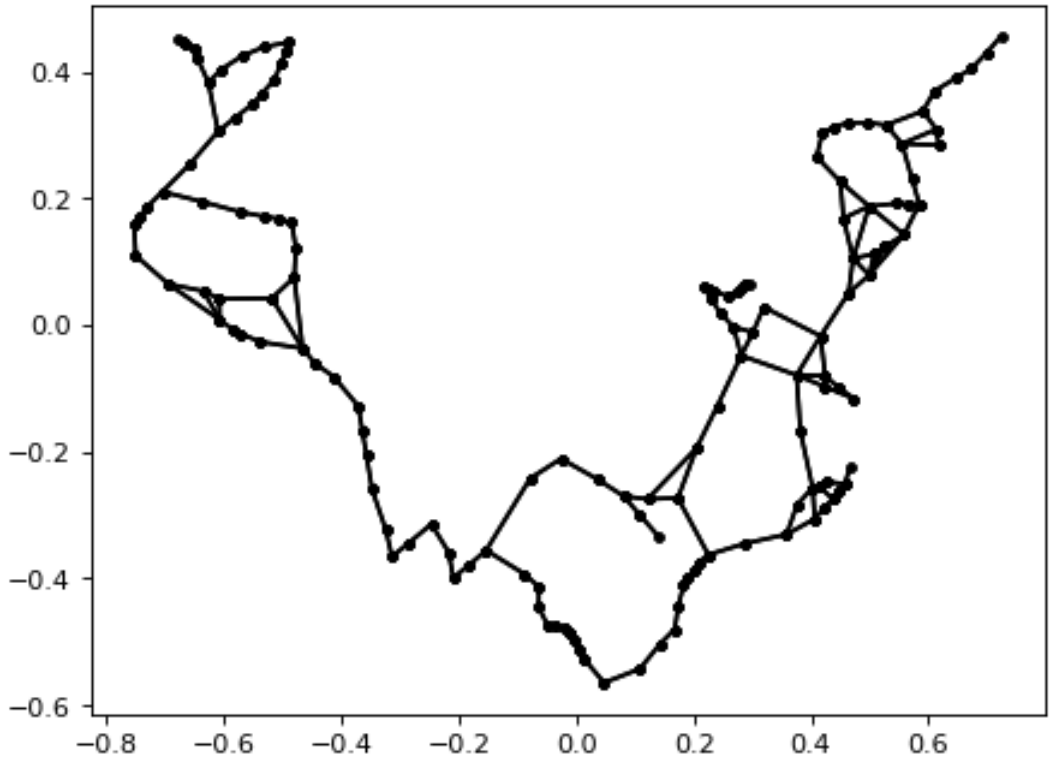


Figura 3. epsilonA_0.1-epsilonN_0.05-Max_10-eta_5-epochs_10-alpha_0.1-delta_0.1.

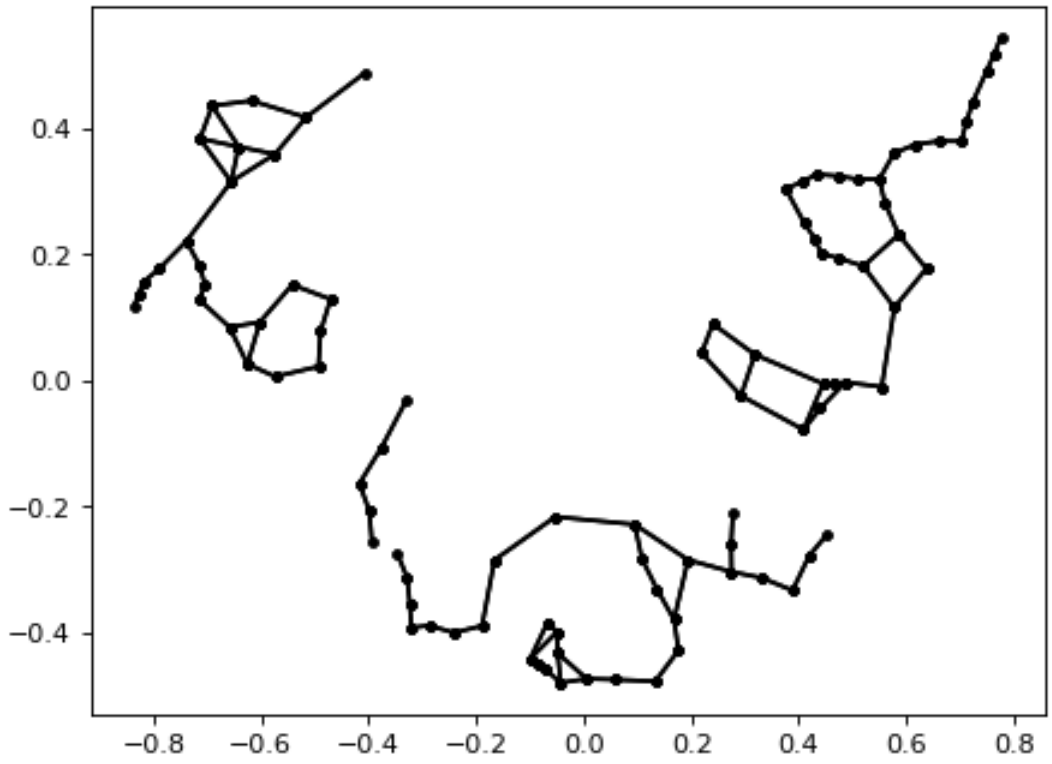


Figura 4 epsilonA_0.1-epsilonN_0.025-Max_10-eta_15-epochs_10-alpha_0.1-delta_0.1.

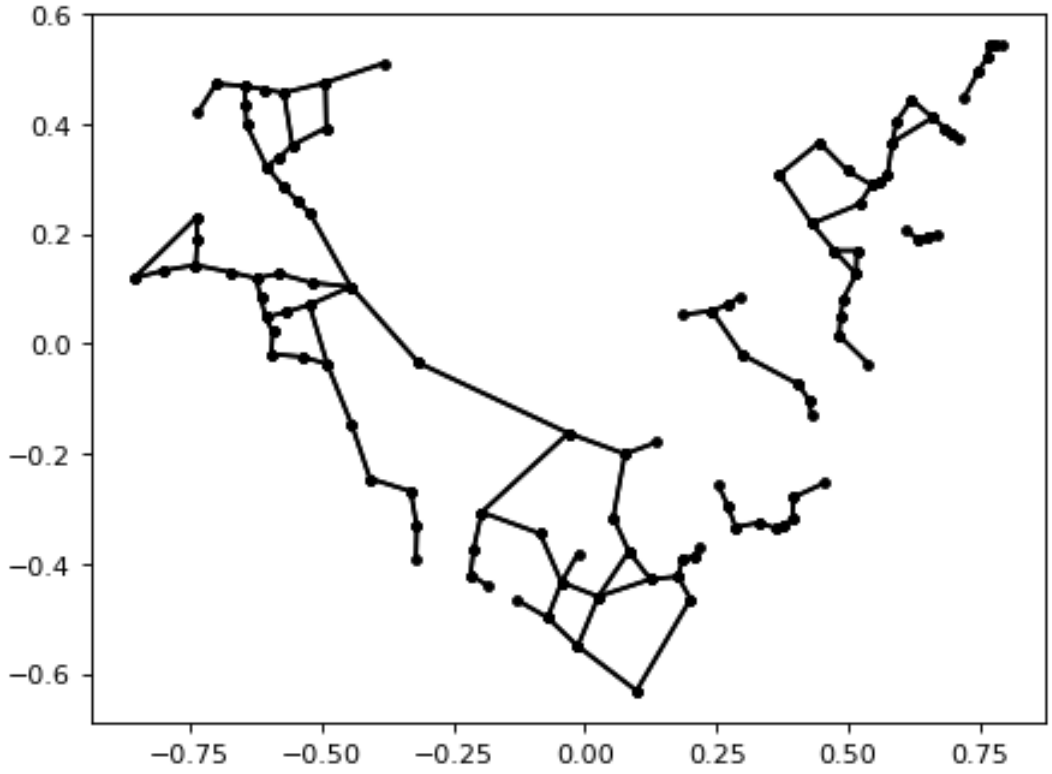


Figura 5. epsilonA_0.2-epsilonN_0.025-Max_5-eta_8-epochs_5-alpha_0.1-delta_0.1

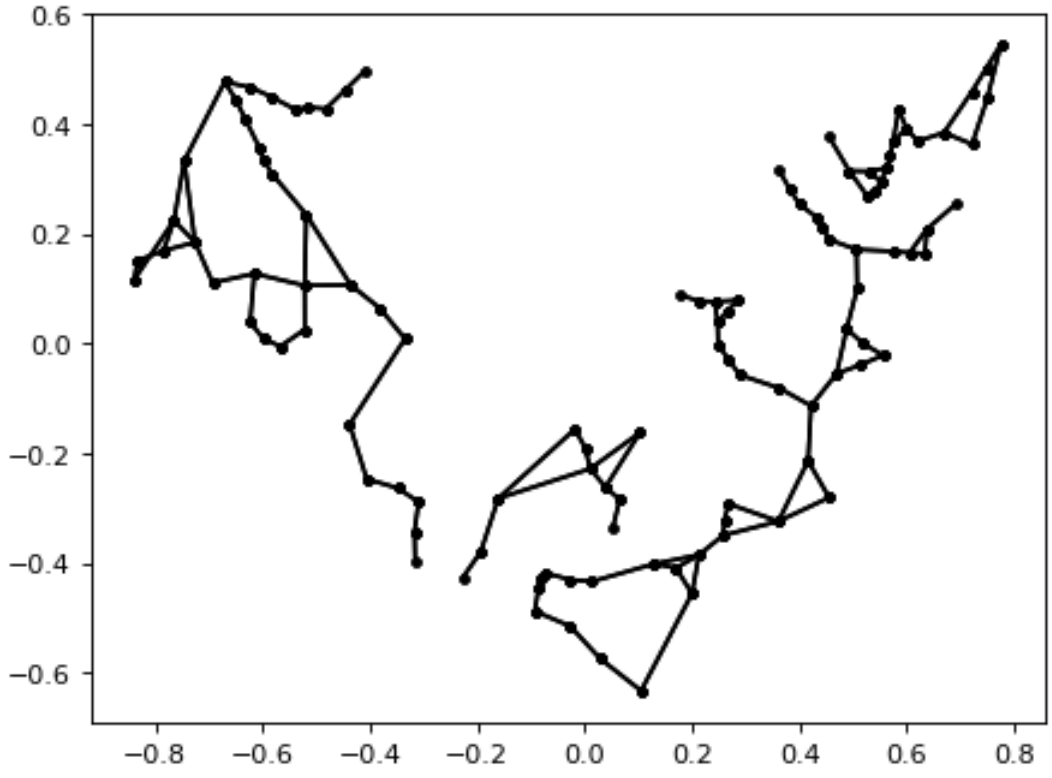


Figura 6. epsilonA_0.2-epsilonN_0.025-Max_8-eta_8-epochs_5-alpha_0.1-delta_0.1

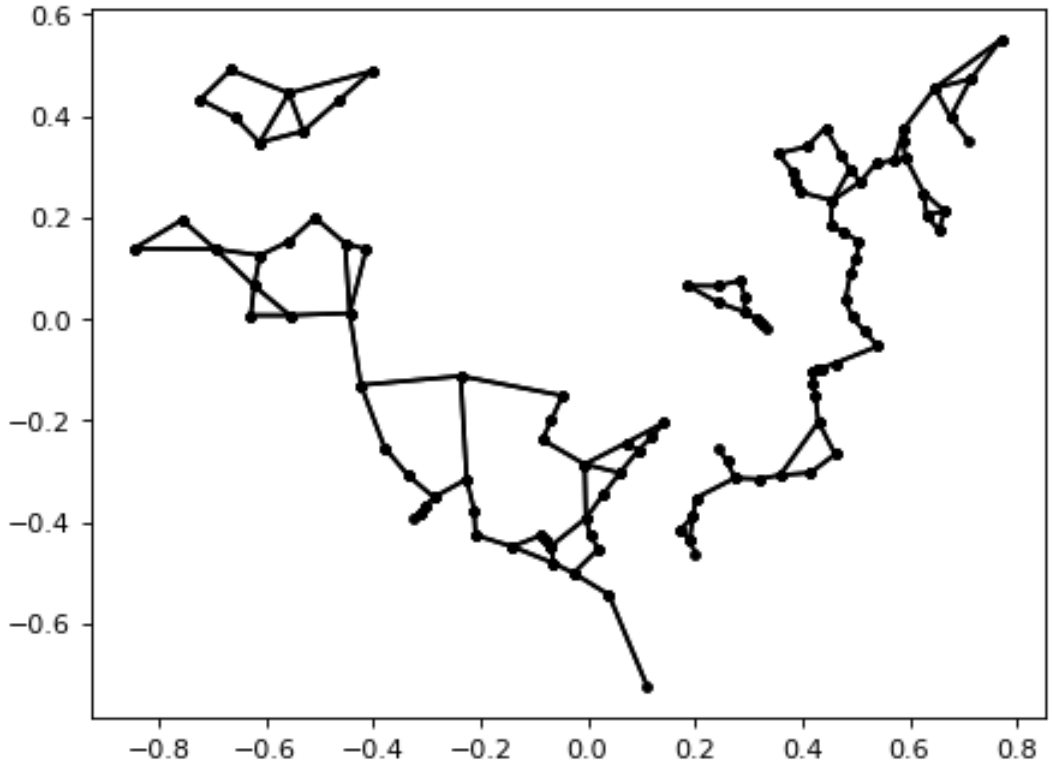


Figura 7. epsilonA_0.2-epsilonN_0.025-Max_8-eta_8-epochs_5-alpha_0.2-delta_0.1

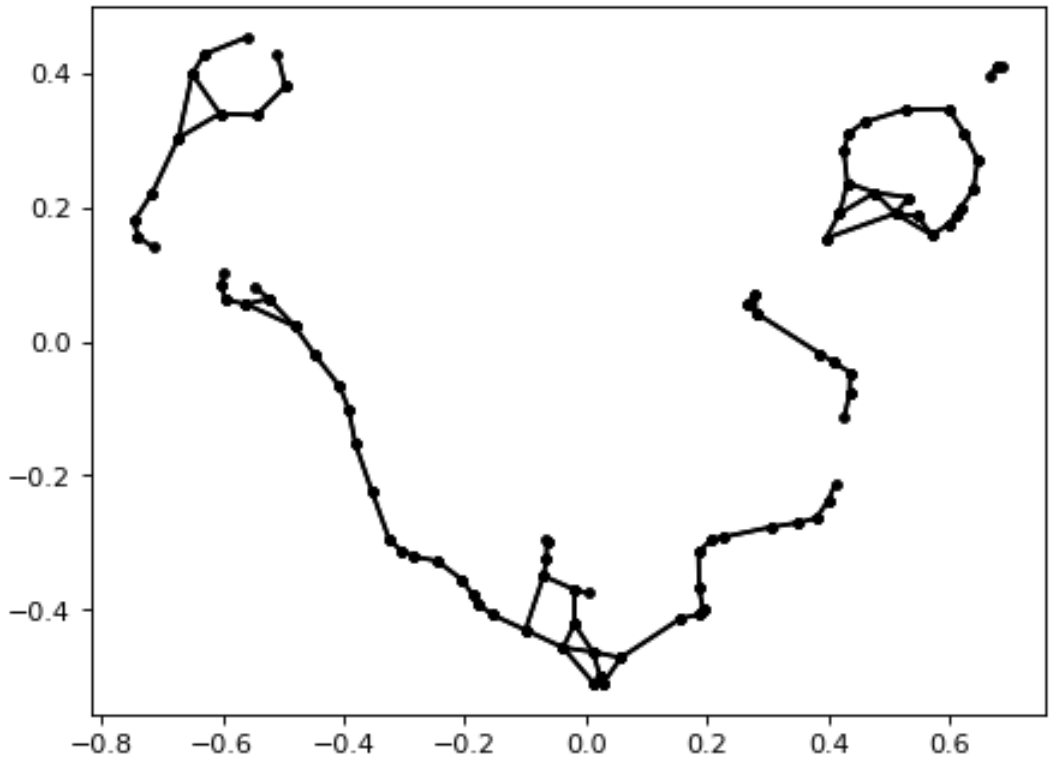


Figura 8. epsilonA_0.05-epsilonN_0.05-Max_5-eta_10-epochs_5-alpha_0.1-delta_0.1

Exceptuando a la primera imagen (Figura 1), todas muestran claramente el patrón en forma de parábola que concuerda con la distribución de datos del dataset. Este fenómeno que tiene lugar en la primera gráfica posiblemente se deba a su elevada edad máxima, lo cual evita que se elimine la conexión de la parte superior, también puede haber afectado el numero bajo de épocas provocando un entrenamiento más superficial.

El resto de las gráficas presentan la distribución parabólica previamente mencionada, pero varían en su habilidad para diferenciar los diferentes grupos, por ejemplo, la (Figura 4) diferencia claramente los 3 grupos de vinos mientras que la (Figura 2) tiene dificultades para separar los dos grupos de la derecha. Al variar los parámetros nos encontramos con los resultados esperados. Por ejemplo, al aumentar *eta* se generaban una menor cantidad de nodos, esto da lugar a situaciones en donde escasean los nodos para una buena representación o viceversa al disminuir *eta,* consideramos que en nuestro caso una *eta* con valor cercano a 10 era el más adecuado.

Por otro lado, probamos a aumentar y disminuir tanto *épsilonA* como *épsilonN* pero los resultados no fueron satisfactorios. Los mejores resultados los obteníamos con los predeterminados de la red, es decir, *épsilonA=0.1 épsilonN =0.05*.

Con respecto a las épocas, nos encontramos que aumentarlas conllevaba, en la gran mayoría de casos, a una mejora significativa. En ningún caso llegamos a una situación de *overfitting* pero esto se puede deber a que no realizamos pruebas con números de épocas realmente grandes ya que el tiempo que tardaba la red en entrenarse aumentaba de manera considerable.

Por último, esta es la representación del *dataset* de *clasificación_Vinos* al pasarlo por la GNG antes de realizarle todo el procesado de datos que mencionamos anteriormente.