MACHINE LEARNING

(en Python), con ejemplos

Versión 0.2

11-Septiembre-2016

Ricardo Moya García, PhD

Índice

1.- MACHINE LEARNING 4

¿Qué es el Machine Learning? 4

Aprendizaje del sistema: Regresión y Clasificación 5

Overfitting y Underfitting 7

Tipos de aprendizaje 10

Métodos de evaluación de un sistema 10

Técnicas del Machine Learning 11

2.- CLUSTERING 13

K-means 15

Implementación del K-means 19

K-means con scikit-learn 28

Expectation-maximization (EM) 35

Implementación del Expectation-maximization 41

Expectation-maximization (Gaussian mixture models) con scikit-learn 51

Selección del número óptimo de Clusters 59

Método del codo (Elbow Method) 60

Dendrogramas 63

Gap 66

BIBLIOGRAFÍA 70

GLOSARIO DE TÉRMINOS 71

1.- Machine Learning

¿Qué es el Machine Learning?

El **Machine Learning** (ML) o **Aprendizaje Autónomo** es una rama de la **Inteligencia Artificial** (IA) que tiene como objetivo crear sistemas capaces de aprender por ellos mismos a partir de un conjunto de datos (data set), sin ser programados de forma explícita.

Para que estos sistemas puedan aprender por ellos mismos, se utilizan una serie de *técnicas y algoritmos* capaces de crear modelos predictivos, patrones de comportamiento, etc. Aunque no existe en la bibliografía actual un listado concreto y acotado de aquellas técnicas y algoritmos que se enmarcan dentro de la rama del ML (aunque hay algunas técnicas que claramente son propias de dicha área), si que podemos decir que en el área del ML encaja todo proceso de resolución de problemas, basados más o menos explícitamente en una aplicación rigurosa de la teoría de la decisión estadística; por tanto, es muy normal que el área del ML se solape con el área de la estadística. En ML; a diferencia de la estadística, se centra en el estudio de la complejidad computacional de los problemas, ya que gran parte de estos son de la clase NP-completo (o NP-hard) y por tanto el reto del ML está en diseñar soluciones factibles para este tipo de problemas.

Veamos a continuación dos definiciones de ML:

Arthur Samuel (1959): *El Machine Learning es un campo de estudio que da a las computadoras la capacidad de aprender sin ser programadas de forma explicita.*

Tom Michell (1998): *Un programa se dice que aprende de una experiencia '**E**' con respecto a alguna tarea '**T**' y alguna medida de rendimiento '**R**', si su rendimiento en '**T**' medida por '**R**', mejora con la experiencia '**E**'.*

Aprendizaje del sistema: Regresión y Clasificación

Como se ha comentado en la definición de ML, este área debe de crear sistemas que tienen que ser capaces de aprender por ellos mismos sin ser programados de forma explicita, con la finalidad de predecir hechos futuros, realizar recomendaciones, clasificaciones de elementos, eventos, tags, etc. Por tanto; después de una fase de aprendizaje, tendremos un "sistema experto" que dada una determinada entrada nos proporcionara una salida (predicción, recomendación, clasificación, etc.), como resultado de haber aplicado una función de regresión o clasificación (que debe de aprender el sistema) sobre los datos de entrada.

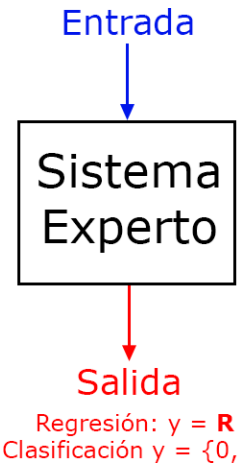

Ilustración 1: Esquema de un sistema experto

Dos ejemplos de sistemas expertos creados tras aplicar alguna/s técnica/s de ML, serían los siguientes: uno, un sistema experto en predicción de quinielas (*clasificación*), que pasándole el nombre del equipo local y visitante, devuelve como resultado una de las tres opciones de la quiniela (1, X, 2); y otro un sistema experto en el cálculo de calorías quemadas (*regresión*) al hacer carrera continua (running), en el que pasándole como entrada el peso de la persona, el tiempo de carrera y la velocidad, devuelva como resultado el número de calorías quemadas ($0 \leq$ calorías $< \infty$).

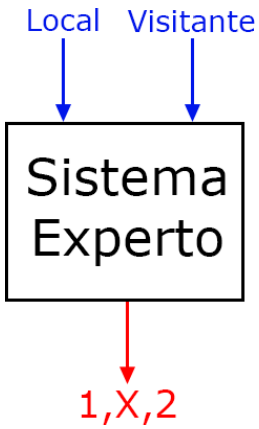

Ilustración 2: Esquema de un sistema experto en predicción de quinielas

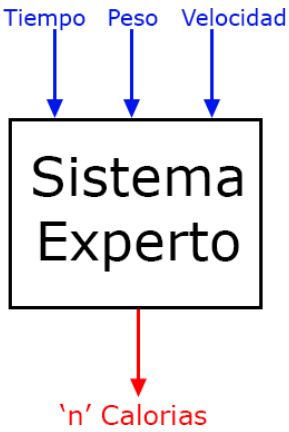

Ilustración 3: Esquema de un sistema experto en predicción de calorías quemadas

Como hemos visto, estos sistemas tienen dos formas de proporcionar un resultado: uno; la ***clasificación***, que devuelve como salida un conjunto finito de resultados; generalmente pequeño, (y={0,1}, y={1,X,2}, y={si,no}) y otro; la ***regresión***, que devuelve como salida un valor arbitrario (un número real, un vector de números reales, cadenas de símbolos, etc.).

Una vez explicado lo que es la regresión y la clasificación, veamos a continuación una definición más formal de las mismas:

Regresión: *Tanto los datos de entrada como los de salida, pertenecen a dominios (X,Y) arbitrarios. Un ejemplo seria $X = Y = \mathbb{R}$, siendo el modelo resultante o la hipótesis una función $f: \mathbb{R} \to \mathbb{R}$*

Clasificación: *Los datos de entrada X son arbitrarios y la salida Y es un conjunto finito y generalmente pequeño de N elementos $Y = \{1,2, \dots, N\}$*

Visto el tipo de sistemas expertos que queremos conseguir tras aplicar alguna/s técnica/s de ML, tenemos que ver *como aprenden estos sistemas* para obtener esa función de clasificación o regresión, que en adelante la denominaremos "hipótesis".

Para que los sistemas aprendan, se ha de tener un ***conjunto de datos*** (o data set) ***de aprendizaje o entrenamiento*** (datos de entrada $x \in X$ y de salida $y \in Y$) que son utilizados para obtener una ***hipótesis*** (modelo o función $f : X \to Y$) que generalice esos datos adecuadamente. Cuando se habla de "generalizar", se habla de ***predecir*** la salida a partir de nuevos datos de entrada (***datos de test***) distintos a los datos de entrenamiento.

Este proceso de aprendizaje para la obtención de una hipótesis, lo podemos ver esquematizado en la siguiente imagen. En primer lugar contamos con un conjunto de datos que los utilizaremos para entrenar o enseñar al sistema. Aplicando alguna de las técnicas de ML, obtendremos una hipótesis que a priori debería ser la mejor función que se ajusta y generaliza los datos de entrenamiento. Para entender este esquema, supongamos que estamos en el caso de un aprendizaje supervisado (que explicamos en el punto de Tipos de Aprendizaje), en el que para cada entrada de los datos de entrenamiento, conocemos cual va a ser su salida. Por tanto; para obtener la hipótesis en este tipo de aprendizaje, tenemos que obtener una función que se ajuste a esos datos de entrenamiento; o dicho de otra manera, que minimize el "***error empírico***" que es el error medido tras aplicar la hipótesis a los datos de entrenamiento. Ciertamente lo que interesa es que la hipótesis obtenida tenga el menor error posible con los datos de test, pero se asume que los datos de entrenamiento son una muestra lo suficientemente representativa como para que el error cometido con los datos de test sea similar al error empírico.

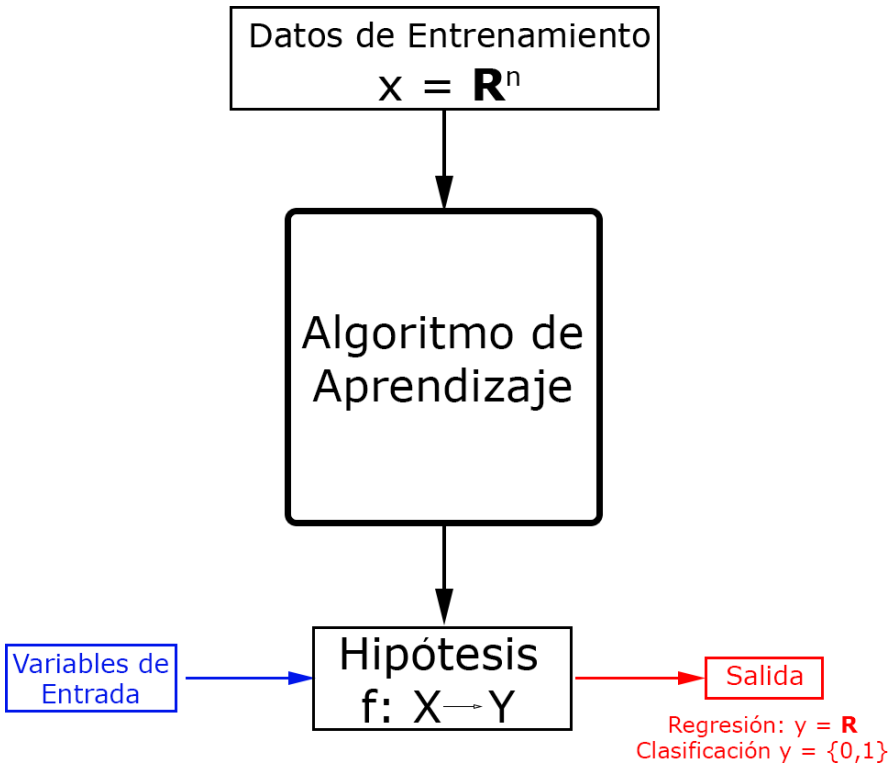

Ilustración 4: Esquema de aprendizaje para la obtención de la hipótesis

En resumen el objetivo de estas técnicas es encontrar aquella función o modelo (hipótesis); dentro de todo el conjunto de funciones o modelos, que mejor se ajusta y generaliza los datos de entrenamiento para aplicarlo a los datos de test y así obtener una predicción, recomendación, clasificación, etc.

Overfitting y Underfitting

Uno de los puntos más interesantes en el ML, es que se haya conseguido un aprendizaje correcto con los datos de entrenamiento, obteniendo una hipótesis capaz de generalizar correctamente los nuevos datos de entrada. Por el contrario, el obtener una hipótesis como resultado de un ***sobreajuste (overfitting)*** o ***sobregeneralización (underfitting)*** de los datos de entrenamiento, hará que la salida proporcionada con nuevos datos de entrada tenga (muy probablemente) un error muy elevado, lo que quiere decir que la predicción no será correcta.

Para ver el significado más en detalle de overfitting y underfitting, supongamos el caso de nuestro sistema experto para la predicción de quinielas: *Sabemos que el FC Barcelona es uno de los equipos más potentes del mundo y obviamente de la liga Española. Por el contrario hay equipos más modestos como puede ser el caso del Rayo Vallecano al que le encajan muchos goles y pierde más partidos de los que gana; pero casualmente en los datos de entrenamiento, tenemos tres resultados en los que el Rayo Vallecano ha ganado al FC Barcelona en su estadio. Por otro lado en la mayoría de los datos de*

entrenamiento nos encontramos que los partidos de fútbol jugados por el FC Barcelona se saldan con victoria; y a parte, otro dato relevante, es que el 60% de los partidos de los datos de entrenamiento tiene como resultado la victoria del equipo local (resultado = 1). Con estos datos ***¿Cuál sería la predicción del resultado del encuentro "FC Barcelona-Rayo Vallecano"?****.*

Aunque esto sería un resumen de los datos que hay en el conjunto de datos de entrenamiento, una persona (sea o no experta en fútbol) y viendo los datos de entrenamiento, podría predecir que el resultado de este partido sería un ***"1" (gana el FC Barcelona)*** ya que el FC Barcelona es: uno de los equipos más potentes, gana la mayoría de los partidos, se enfrenta a un equipo que pierde más partidos de los que gana y además juega en su estadio (60% de los partidos los ganan los equipo locales). Aunque los algoritmos de aprendizaje en ML no tienen la capacidad de hacer estos razonamientos (ya que estos se hacen de forma matemática) si que se espera que el resultado sea coherente y similar al que haría un humano. Pero, *¿Que resultado arrojaría el sistema si se ha producido overfitting o underfitting?:*

1. Overfitting ("FC Barcelona-Rayo Vallecano = **Victoria del Rayo Vallecano**"): Como en el conjunto de datos de entrenamiento se ha dado la casualidad de que *solo hay 3 datos de enfrentamientos entre estos equipos y son de victorias de Rayo Vallecano*; ajustándome a los datos de entrenamiento, digo que gana el Rayo Vallecano porque es la información de la que se dispone.

2. Underfitting ("FC Barcelona-Rayo Vallecano = **Victoria del FC Barcelona**"): Al haber sobregeneralización la hipótesis arroja como resultado que va a ganar el equipo local ya que el 60% de los resultados son victorias de los locales; por tanto, en este caso acertaría el resultado.

3. Underfitting ("Rayo Vallecano-FC Barcelona = **Victoria del Rayo Vallecano**"): Por analogía con el caso anterior; al ser el Rayo Vallecano el equipo local, ganaría el Rayo Vallecano.

Lo que se ha pretendido al explicar este ejemplo tan largo, es que la hipótesis obtenida no tiene porque predecir correctamente el 100% de los casos, pero si que se ha de exigir; en cierta medida, que generalice correctamente aunque esto suponga cometer errores puntuales. Dicho de otra manera, hay que ser capaz de encontrar una hipótesis en la que haya un equilibrio entre el sobreajuste y la sobregeneralización.

Veamos a continuación gráficamente el concepto de overfitting y underfitting para los problemas de clasificación y regresión:

Obtención de un clasificador que aproxime $f: \mathbb{R}^2 \rightarrow \{\cdot, x\}$:

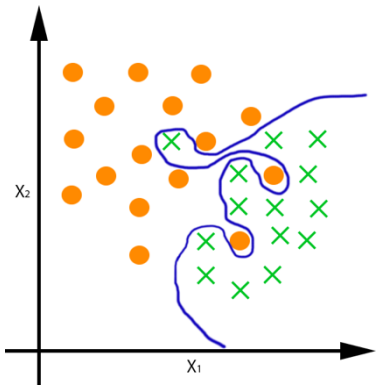

Ilustración 5: Overfitting en un ejemplo de clasificación

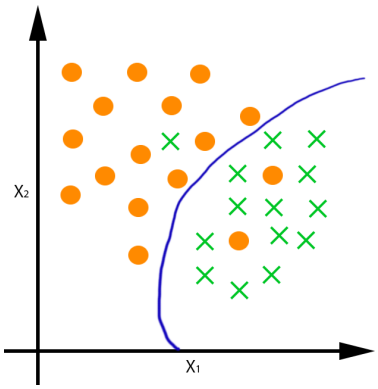

Ilustración 6: Clasificación correcta en un ejemplo de clasificación

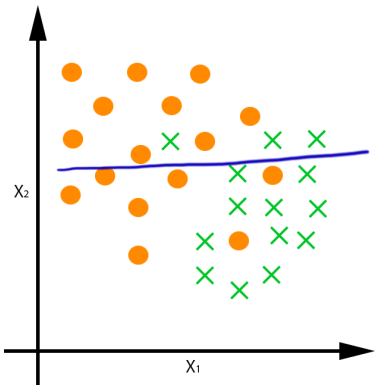

Ilustración 7: Underfitting en un ejemplo de clasificación

Modelo de regresión que aproxime $f: \mathbb{R} \rightarrow \mathbb{R}$:

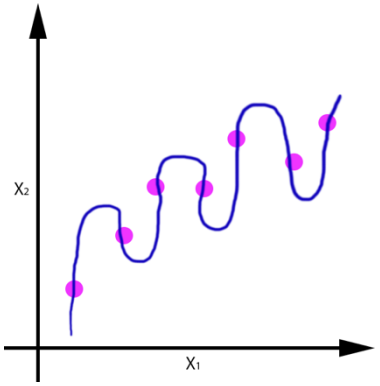

Ilustración 8: Overfitting en un ejemplo de regresión

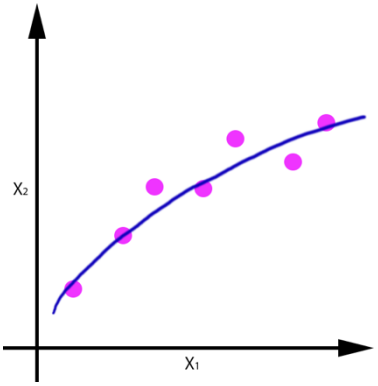

Ilustración 9: Ajuste correcto en un ejemplo de regresión

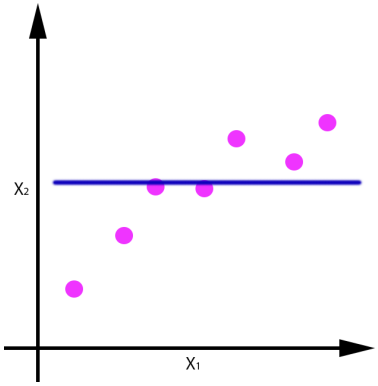

Ilustración 10: Undefitting en un ejemplo de regresión

Veamos por ejemplo para el caso de la regresión, el error que se cometería si nos llegase un nuevo dato de entrada (punto rojo), para cada uno de las funciones obtenidas:

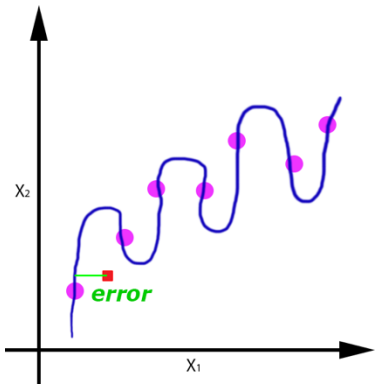

Ilustración 11: Error Overfitting

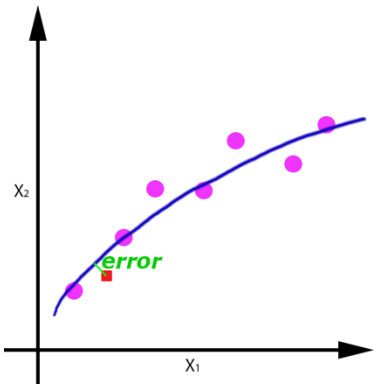

Ilustración 12: Error de un ajuste correcto

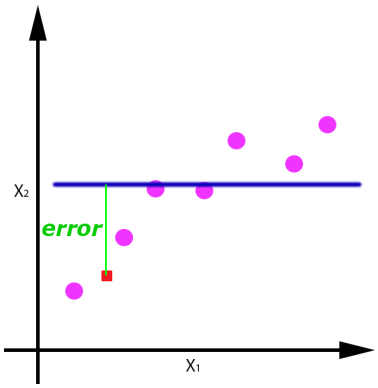

Ilustración 13: Error Underfitting

Se puede apreciar como el error cometido con un dato de test, es mayor en el caso en el que se produce overfitting y underfitting, y es menor en el caso en el que se ha generalizado de forma correcta, aunque evidentemente tiene un pequeño error.

Tipos de aprendizaje

Dependiendo de cómo sean los datos de los que dispongamos para entrenar al sistema, podemos aplicar un tipo de aprendizaje u otro. A continuación se enumeran y explican los tipos de aprendizaje más comunes:

1.- Aprendizaje supervisado: Tipo de aprendizaje en el que se tiene la información completa de los datos de entrenamiento; es decir, los datos de entrada y la salida de los mismos. Es el tipo de aprendizaje que mejores resultado ofrece ya que es el que más información tiene.

2.- Aprendizaje no supervisado: Tipo de aprendizaje en el que únicamente se disponen de los datos de entrada y tiene como objetivo el obtener información sobre la estructura del dominio de salida.

3.- Aprendizaje semi-supervisado: Es un tipo de aprendizaje híbrido entre el aprendizaje supervisado y no supervisado.

4.- Aprendizaje Adaptativo: Tipo de aprendizaje en el que se parte de un modelo previo cuyos parámetros se modifican o adaptan usando los nuevos datos de entrenamiento.

5.- Aprendizaje on-line: En este tipo de aprendizaje no hay una distinción concreta entre la fase de test y de entrenamiento. El sistema aprende (normalmente desde cero) mediante el propio proceso de predicción en el que hay una supervisión humana que consiste en validar o corregir cada salida en función de la entrada.

6.- Aprendizaje por refuerzo: Tipo de aprendizaje híbrido entre el aprendizaje on-line y aprendizaje semi-supervisado en el que la supervisión es incompleta; normalmente una información del tipo {si,no}, {0,1}, {premio,castigo}. Es un tipo de aprendizaje que se basa en el "*argumentum ad baculum*", utilizado normalmente en la educación de los animales.

Métodos de evaluación de un sistema

Para evaluar las hipótesis obtenidas tras la aplicación de alguna de las técnicas de ML, es necesario disponer de un conjunto de datos (etiquetados o no) para generar la mejor de las hipótesis posibles y minimizar el error empírico. Dado un conjunto de datos, podemos

enumerar los siguientes métodos de evaluación en función de cómo se dividen los datos de entrenamiento y de test:

1.- Resustitución: Es un método muy optimista en el que todos los datos disponibles se utilizan como datos de test y de entrenamiento.

2.- Partición (Hold Out) : Este método divide los datos en dos subconjuntos: uno de entrenamiento y uno de test. El problema que tiene este método es que se desaprovechan los datos de test para la obtención de la hipótesis.

3.- Validación cruzada (Cross Validation) : Este método divide los datos aleatoriamente en 'N' bloques. Cada bloque se utiliza como test para un sistema entrenado por el resto de bloques. El inconveniente de este método es que reduce el número de datos de entrenamiento cuando el número de datos de cada bloque es grande.

4.- Exclusión individual (Leaving one out) : Este método utiliza cada dato individual como dato único de test de un sistema entrenado con todos los datos excepto el de test. Es similar al método de la validación cruzada, pero en este caso el coste computacional es muy grande por la cantidad de fases de aprendizaje que se deben de realizar.

Técnicas del Machine Learning

Resulta verdaderamente complejo hacer una jerarquización o clasificación de las técnicas de ML en función de los problemas que pueden resolver; ya que por ejemplo, hay técnicas muy concretas como el *K-means* o el *Expecation-Maximization* (EM) que se utilizan claramente para problemas de *Clustering* y sin embargo técnicas como la de *Support Vector Machine* (SVM) o incluso las *Redes Neuronales* pueden encajar tanto en problemas de *Regresión* como de *Clasificación*, aunque sean más propias de la clasificación que de la regresión. Independientemente de que cada una de las técnicas pueda ser clasificada de una manera o de otra, lo que si es obvio es que los profesionales del ML deben de preocuparse en aprender correctamente cuantas más técnicas mejor y de esta forma tener una formación lo suficientemente alta como para saber abordar y solucionar los problemas utilizando la/s técnica/s que mejor se puedan adaptar.

A continuación mostramos una clasificación de las técnicas y problemas del ML, utilizando en parte la clasificación propuesta por la Wikipedia Inglesa.

Problemas	Aprendizaje Supervisado	Clustering
• Classification • Clustering • Regression • Anomaly detection • Association rules • Reinforcement learning • Structured prediction • Feature engineering • Feature learning • Online learning • Semi-supervised learning • Unsupervised learning • Learning to rank • Grammar induction	• Decision trees • Ensembles (Bagging, Boosting, Random forest) • k-NN • Linear regression • Naive Bayes • Neural networks • Logistic regression • Perceptron • Relevance vector machine (RVM) • Support vector machine (SVM)	• BIRCH • Hierarchical • k-means • Expectation-maximization (EM) • DBSCAN • OPTICS • Mean-shift

Reducción de la dimensionalidad	Modelos Probabilísticos	Detección de anomalías
• Factor analysis • Canonical correlation (CCA) • Independent component analysis (ICA) • Linear discriminant analysis (LDA) • Non-negative matrix factorization (NMF) • Principal component analysis (PCA) • t-distributed stochastic neighbor embedding (t-SNE)	• Modelos gáficos probabilísticos (PGM) • Redes Bayesianas • Conditional random field (CRF) • Modelo oculto de Márkov (HMM)	• k-nearest neighbors (k-NN) • Local outlier factor

Redes Neuronales	Sistemas de Recomendación	Teorías
• Deep learning • Multilayer perceptron • Recurrent neural network (RNN) • Restricted Boltzmann machine • Self-organizing map (SOM) • Convolutional neural network	• k-nearest neighbors (k-NN) • Singular Value Descomposition (SVD) • Latent Semantic Indexing (LSI) • Probabilistic Latent Semantic Indexing (PLSI) • Latent Dirichlet Allocation (LDA)	• Bias-variance (trade off) dilema • Computational learning theory • Empirical risk minimization • Occam learning • PAC learning • Statistical learning • VC theory

2.- Clustering

El **Clustering** es una tarea que consiste en agrupar un conjunto de objetos (no etiquetados) en subconjuntos de objetos llamados **Clusters**. Cada **Cluster** está formado por una colección de objetos que son similares (o se consideran similares) entre sí, pero que son distintos respecto a los objetos de otros Clusters.

Cluster: *Conjunto de objetos que son similares entre sí.*

Clustering: *Tarea de dividir un conjunto de objetos en subconjuntos de objetos (Clusters) similares entre sí.*

En el campo del ML, el ***Clustering*** se enmarca dentro del ***aprendizaje no supervisado***; es decir, que para esta técnica solo disponemos de un conjunto de datos de entrada, sobre los que debemos obtener información sobre la estructura del dominio de salida, que es una información de la cual no se dispone.

Es importante no confundir el Clustering con los problemas de Clasificación. Las técnicas de Clasificación se enmarcan dentro del aprendizaje supervisado porque para cada dato tenemos información sobre sus variables de entrada y de salida; es decir, cada dato u objeto esta etiquetado. Sin embargo para aquellos casos en los que no disponemos de la salida de cada dato y queramos agrupar estos objetos en grupos similares, debemos de aplicar alguna de las técnicas de Clustering para saber la procedencia de estos datos.

Supongamos el siguiente caso de ejemplo: *Tenemos un data set (DS1) con el color de pelo de un conjunto de personas. Como entrada tenemos el color en RGB del pelo de cada persona y como salida un conjunto finito de etiquetas {Moreno, Rubio, Castaño, Canoso}. Por otro lado tenemos otro data set (DS2) con el color del pelo en RBG de otro conjunto de personas, pero en este caso no tenemos como salida una etiqueta que nos diga si el individuo es Moreno, Rubio, Castaño o Canoso. Para ambos casos queremos generar*

un modelo que nos permita etiquetar a nuevas personas en Morenos, Rubios, etc. ***¿Cómo abordaríamos este problema para el DS1 y DS2?****.*

- **DS1:** El primer caso se trata de un **problema de clasificación**, ya que para cada individuo tenemos su color de pelo en RGB y su clasificación correspondiente {Moreno, Rubio, Castaño, Canoso}. Para este caso debemos de obtener una función (o hipótesis) en base a los datos de entrenamiento para que a cada nueva entrada (color del pelo en RGB) lo etiquete de forma correcta:

$$f(x = color) = \{Moreno, Rubio, Castaño, Canoso\}$$

- **DS2:** El segundo caso se trata de un ejemplo claro de **Clusterización**, ya que tenemos que obtener información sobre su estructura o dominio de salida en base a los datos del color de pelo que tenemos en el data set. Para ello debemos de utilizar alguna de las técnicas de Clustering e indicarle que nos agrupe todos los datos en 4 Clusters. Una vez hecha esta Clusterización podremos identificar que individuos son Morenos, Rubios, Castaños o Canosos viendo solamente el valor medio (o Centroide) de cada Cluster. Un ejemplo de Clusterización de este data set visto en 2D sería el siguiente:

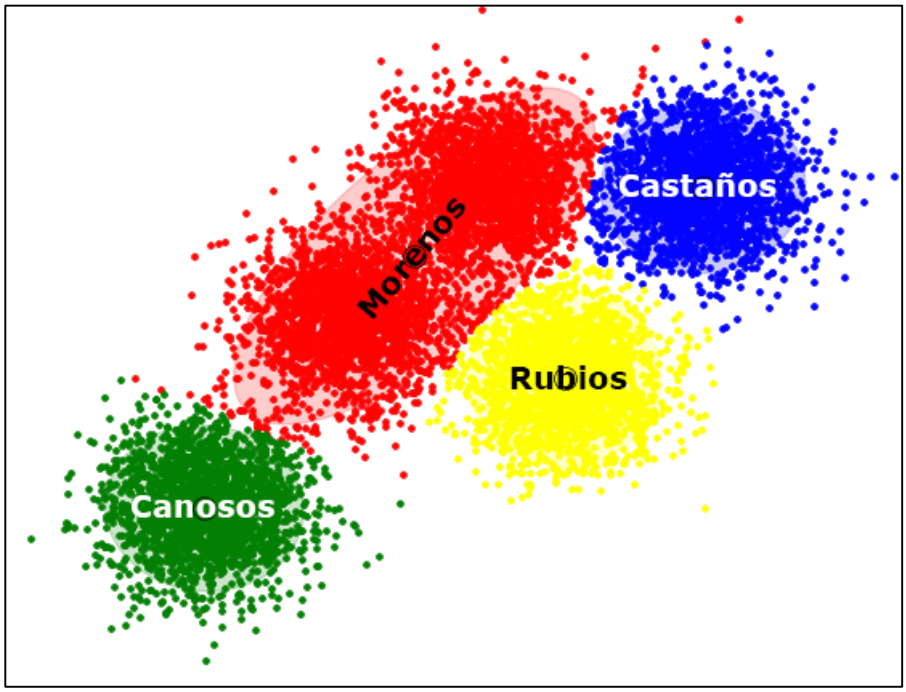

Ilustración 14: Ejemplo de Clustering

En este ejemplo podemos apreciar como nos divide los individuos en 4 grupos, en función de su color de pelo y como podemos ser capaces posteriormente de identificar y etiquetar a cada grupo de personas.

Este ejemplo propuesto, es un ejemplo muy sencillo y didáctico para ver la diferencia entre las tareas de Clasificación y de Clustering; para dejar muy claro, que la primera se enmarca dentro del aprendizaje supervisado y la segunda en el aprendizaje no supervisado. Para los casos prácticos en los que hay que aplicar técnicas de Clustering no solemos tener información a priori sobre los datos a analizar ni información sobre los modelos que han generado estos datos. En el ejemplo propuesto hemos asumido que solo tendríamos personas con estos cuatro tipos de pelo y no sabemos si habría personas con el pelo teñido de color azul, verde, etc. lo que supondría haber realizado una tarea de Clustering poco acertada ya que posiblemente agruparía en Clusters diferentes a personas consideradas Morenas o asumiría que las personas Castañas y Rubias pertenecerían al mismo Cluster.

En resumen, las técnicas de Clustering son apropiadas para los casos de aprendizaje no supervisado en los que se quiere agrupar y tener conocimiento a un alto nivel de cómo se han generado los datos y como están organizados, sin tener conocimiento a priori de estos.

K-means

El K-means es un método de Clustering que separa 'K' grupos de objetos (Clusters) de similar varianza, minimizando un concepto conocido como ***inercia***, que es la suma de las distancias al cuadrado de cada objeto del Cluster a un punto '$\boldsymbol{\mu}$' conocido como Centroide (punto medio de todos los objetos del Cluster).

$$Inercia = \sum_{i=0}^{N} \|x_i - \mu\|^2$$

El algoritmo de los K-means tienen como objetivo elegir 'K' centroides que reduzcan al mínimo la inercia:

$$\underset{C}{\operatorname{argmin}} \sum_{j=1}^{k} \sum_{x_i \in C_j} \left\|x_i - \mu_j\right\|^2$$

El funcionamiento de este algoritmo comienza ***eligiendo un centroide para cada uno de los 'K' Clusters***. El método de elección de estos centroides puede ser cualquiera; siendo los dos más comunes, el inicializarlo de forma aleatoria o el de elegir 'K' objetos del data set, bien sea de forma aleatoria o haciendo un pre-procesamiento de los datos. Lo recomendable sería la segunda opción e inicializar estos Clusters con objetos del data set. Una vez inicializados los centroides, el algoritmo continua ***alternando los dos siguientes pasos*** (asignación y actualización) de forma iterativa hasta que los centroides converjan:

- **Asignación:** Se asigna cada objeto al Cluster más cercano, aplicando alguna medida de distancia (como por ejemplo la distancia euclídea) entre el objeto y el centroide del Cluster.

$$d_e(X,\mu) = \sqrt{\sum_{i=1}^{n}(x_i - \mu_i)^2}$$

- **Actualización:** Calcula los nuevos centroides, haciendo la media de los objetos que forman el Cluster.

$$\mu = \frac{1}{N}\sum_{j=1}^{N} x_j$$

Se considera que el algoritmo finaliza cuando los centroide convergen o cuando la diferencia entre el valor de los centroides de una iteración a otra es inferior a un determinado umbral.

Veamos a continuación un ejemplo de ejecución del K-means para 3 Clusters, dado un data set en el que cada objeto esta representado por un punto en un espacio de dos dimensiones:

1. **Inicialización de los Clusters:** Para ello cogemos al azar 3 puntos del data set y los asignamos a un Cluster. Como cada Cluster solo tiene un punto, será ese punto el centroide del Cluster:

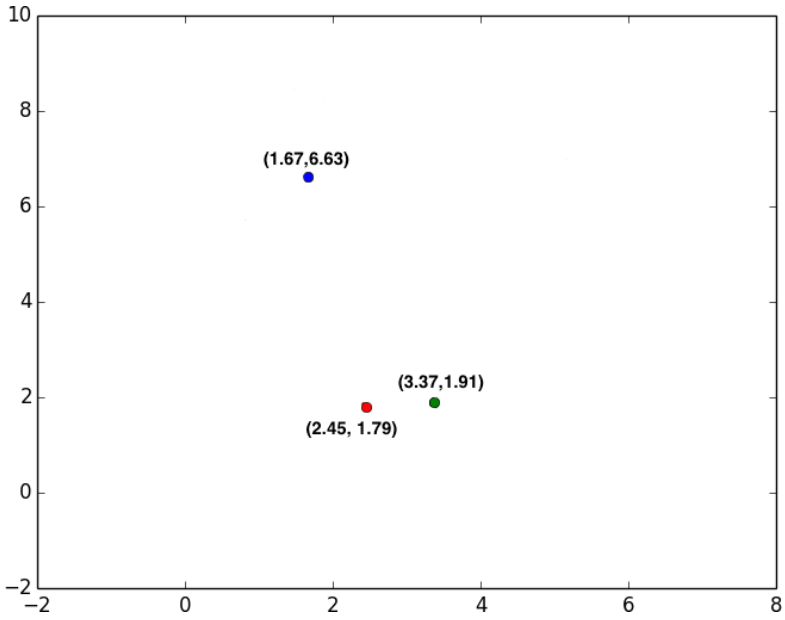

Ilustración 15: Inicialización de Clusters para el K-means

2. **Primera asignación y actualización:** Tras haber elegido al azar 3 Clusters, se asigna cada punto al Cluster más cercano. Una vez que están asignados todos los puntos, se calcula un nuevo centroide siendo este el valor medio de todos los puntos.

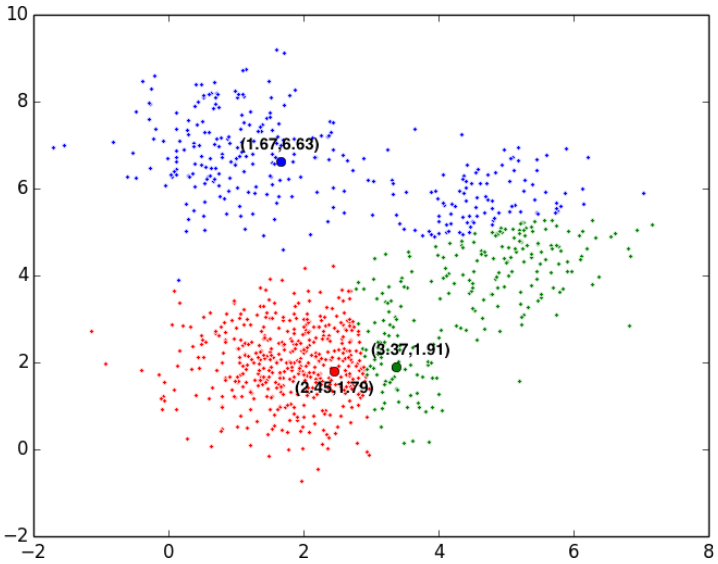

Ilustración 16: Asignación 1 de puntos a los Clusters

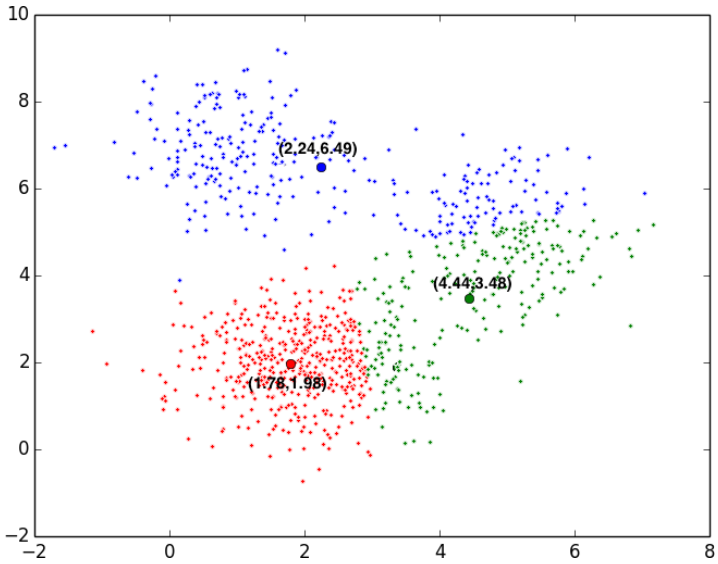

Ilustración 17: Cálculo 1 de nuevos centroides

3. **Segunda asignación y actualización:** Con los nuevos centroides, volvemos a calcular para cada punto cual es el centroide más cercano y asignamos ese punto al centroide. Una vez asignados los puntos a los Clusters, volvemos a calcular los centroides.

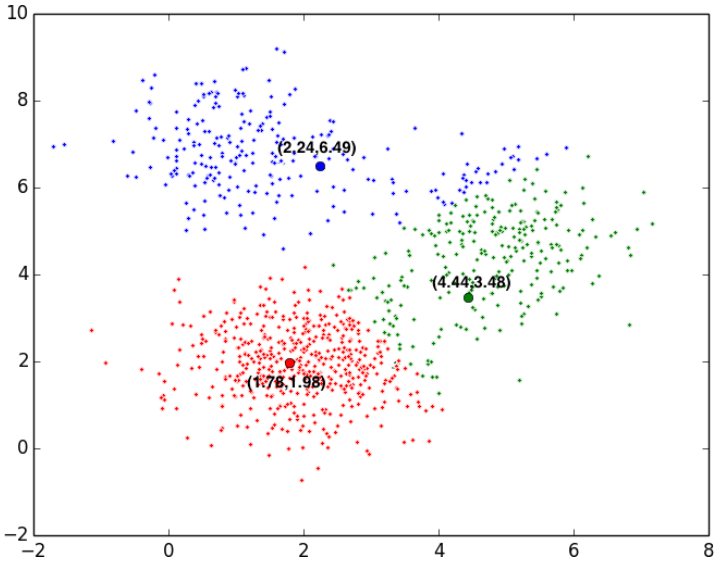

Ilustración 18: Asignación 2 de puntos a los Clusters

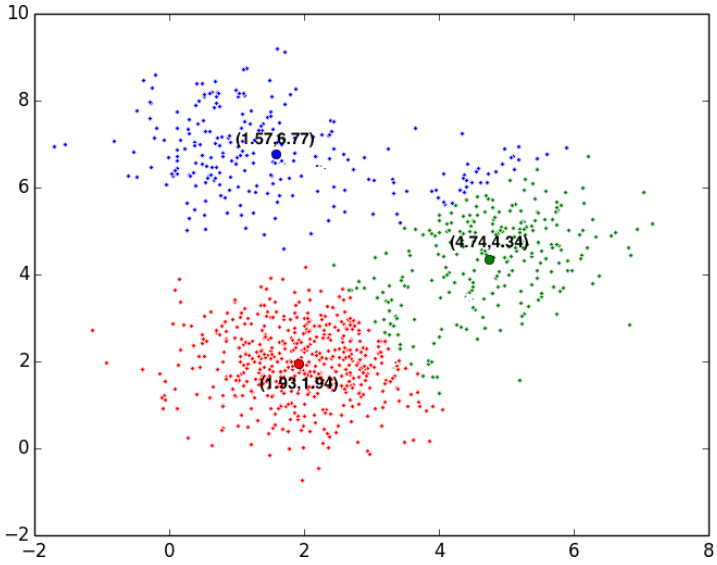

Ilustración 19: Cálculo 2 de nuevos centroides

4. **Converge y resultado final:** Estos pasos de asignación y actualización se repiten hasta que los centroides de los Clusters converjan; es decir, hasta que el valor de los centroides de la última iteración de actualización coincida con el valor de los centroides de la iteración anterior de actualización:

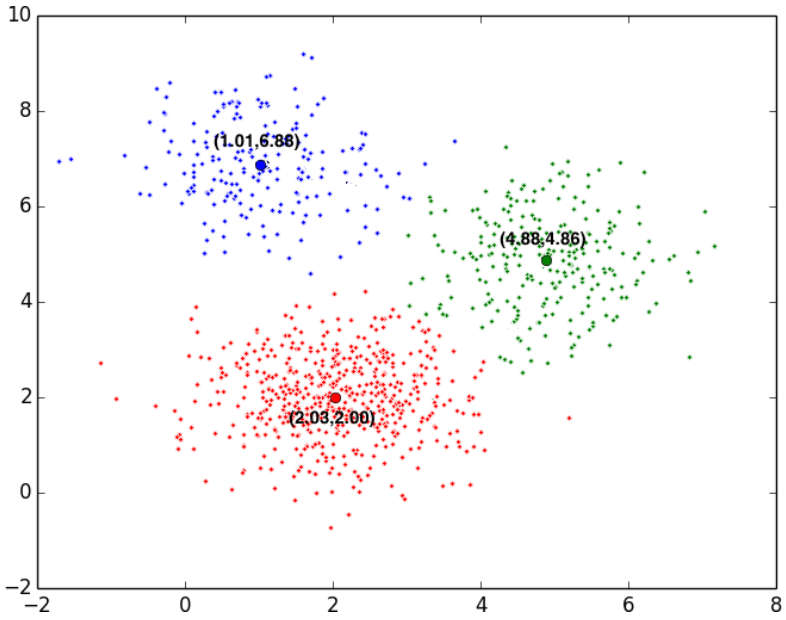

Ilustración 20: Resultado final del Clustering

Definido el funcionamiento del algoritmo paso por paso y con un ejemplo, pasamos a mostrarlo en pseudocódigo:

```
K = num_clusters
1.- Inicializar K Clusters con sus centroides μ1,…,μk de forma aleatoria
2.- while not converge:
   for i in range(dataset):
        C_k := argmin‖x_i − μ_k‖^2
   for j in range(K):
        μ_j := (1/N) Σ_{i=1}^{N} x_i
```

En los dos siguientes puntos: *Implementación del K-means* y *K-means con scikit-learn*, se va a mostrar la implementación del K-means y el uso de la librería ***scikit-learn*** para la resolución de un problema de Clustering con el algoritmo del K-means respectivamente, cuyo código se puede obtener en el siguiente repositorio:

https://github.com/RicardoMoya/KMeans_Python

El código que se encuentra en este repositorio hace uso de las librerías de *numpy*, *matplotlib*, *scipy* y *scikit-learn*. Para descargar e instalar (o actualizar a la última versión

con la opción -U) estas librerías; con el sistema de gestión de paquetes *pip*, se deben ejecutar los siguiente comandos:

```
pip install –U numpy
pip install –U matplotlib
pip install –U scipy
pip install –U scikit–learn
```

Implementación del K-means

En este apartado se va a mostrar la implementación del K-means (desde un punto de vista didáctico) en el que los objetos van a estar representados por un punto en dos dimensiones {x,y}. La implementación realizada en este ejemplo permite que los objetos puedan estar representados con más dimensiones, pero sin perder la perspectiva didáctica de este ejemplo los mostramos en dos dimensiones para que podamos visualizar los resultado en un plano y así entender el correcto funcionamiento de este método.

Para implementar el K-means, vamos a tener objetos que serán representados como puntos en 2D, por tanto implementaremos una clase Punto (*Point.py*) para representar los objetos. Por otro lado se implementa una clase Cluster (*Cluster.py*) para representar a los Clusters y que estará compuesta por un conjunto de objetos de la clase Punto. Por último tendremos un script (*KMeans.py*) en el que estará implementado del K-means con sus dos pasos de asignación y actualización para 'k' Clusters. A continuación se muestra a un alto nivel de abstracción el diagrama de clases de la implementación propuesta del K-means:

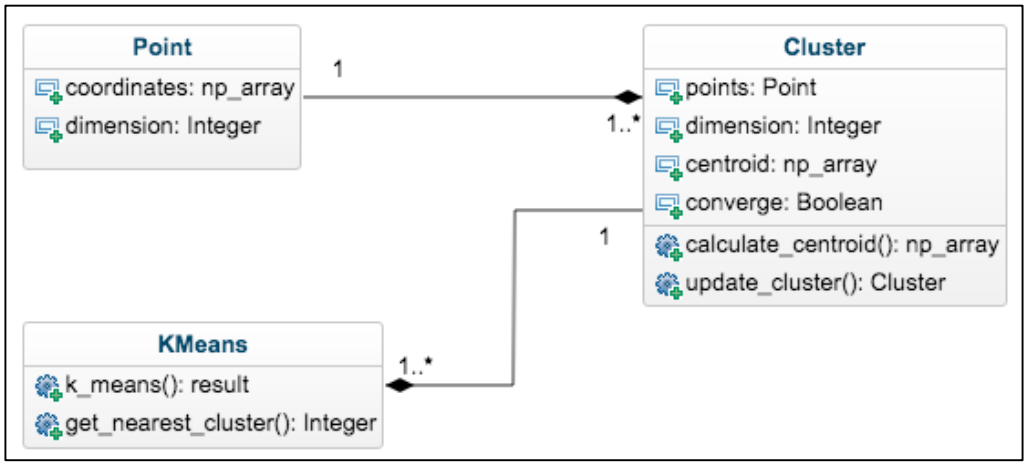

Ilustración 21: Diagrama de Clases del K-Means

Los data sets a utilizar en este ejemplo (que se encuentran dentro de la carpeta *dataSet*) van a ser ficheros de texto en los que en cada línea van a estar las coordenadas

de cada punto {x,y} separados por el separador "::". A continuación mostramos un ejemplo de estos ficheros:

```
1.857611652::2.114033851
2.34822574::1.58264984
1.998326848::4.118143019
1.714362835::2.468639613
1.656134484::1.909955747
```

Esto quiere decir que el primer punto va a estar posicionado en las coordenadas {1.85,2.11} y el segundo punto en las coordenadas {2.34,1.58}.

Para los ejemplos disponemos de 4 data sets con las siguientes características:

Nombre	Nº Puntos	Nº Clusters teóricos	Centroides teóricos
DS_3Clusters_999Points	999	3	{2,2}
			{5,5}
			{1,7}
DS2_3Clusters_999Points	999	3	{2,2}
			{2,4}
			{5,3}
DS_5Clusters_10000Points	10000	5	{0,0}
			{2,3}
			{4,5}
			{5,2}
			{7,5}
DS_7Clusters_100000Points	100000	7	{-1,3}
			{0,0}
			{0,6}
			{2,3}
			{4,5}
			{5,2}
			{7,5}

Visto el diagrama de clases y la estructura de los date sets a utilizar, vamos a pasar a mostrar la implementación de la clases Punto (*Point.py*). A esta clase se le va a pasar en el constructor un "numpy array" con las coordenadas del punto y va a tener como atributos esa coordenada y las dimensiones de la misma, que para el ejemplo que vamos a mostrar será de 2:

```
class Point:

    def __init__(self, coordinates):
        self.coordinates = coordinates
        self.dimension = len(coordinates)

    def __repr__(self):
        return 'Coordinates: ' + str(self.coordinates) + \
               ' -> Dimension: ' + str(self.dimension)
```

Por otro lado vamos a tener la clase Cluster (*Cluster.py*) que se le va a pasar en el constructor una lista de puntos que van a ser los que formen el Cluster. Esta clase va a tener como atributos la lista de puntos del Cluster (*points*), la dimensión de los puntos (*dimension*), el centroide del Cluster (*centroid*) y un atributo (*converge*) que nos va a indicar si el centroide es igual (por tanto converge) que el calculado en el paso de actualización de la iteración anterior. Con la lista de puntos que van a formar el Cluster, hacemos las comprobaciones pertinentes (que el Cluster tenga por lo menos un punto y que todos los puntos sean de la misma dimensión), para no lanzar ninguna excepción. Veamos a continuación el constructor de esta clase:

```
import numpy as np

class Cluster:

    def __init__(self, points):
        if len(points) == 0:
            raise Exception("Cluster cannot have 0 Points")
        else:
            self.points = points
            self.dimension = points[0].dimension

        # Check that all elements of the cluster have the same dimension
        for p in points:
            if p.dimension != self.dimension:
                raise Exception(
                    "Point %s has dimension %d different with %d from the rest "
                    "of points") % (p, len(p), self.dimension)

        # Calculate Centroid
        self.centroid = self.calculate_centroid()
        self.converge = False
```

Como se observa en el atributo centroid, se llama al método *calculate_centroid()* para asignarle el centroide al Cluster en función de la lista de puntos del Cluster. Este método calcula el centroide como el punto medio de todos los puntos que forman el Cluster.

```
    def calculate_centroid(self):

        sum_coordinates = np.zeros(self.dimension)
        for p in self.points:
            for i, x in enumerate(p.coordinates):
                sum_coordinates[i] += x

        return (sum_coordinates / len(self.points)).tolist()
```

NOTA: La implementación de este método puede ser optimizada paralelizando el cálculo del centroide con threads (hilos). Por razones didácticas y por legibilidad y claridad del código no se ha optimizado este método.

Por otro lado implementamos el método *update_cluster(points)*, que es el método encargado de actualizar el estado del Cluster, calculando el nuevo centroide con los nuevos puntos del Cluster tras el paso de asignación y comprobando si convergen los centroides mirando el valor del centroide del paso anterior y del actual. En resumen, con este método actualizamos el estado del Cluster.

```
    def update_cluster(self, points):

        old_centroid = self.centroid
        self.points = points
        self.centroid = self.calculate_centroid()
        self.converge = np.array_equal(old_centroid, self.centroid)
```

Una vez explicadas las Clases Punto y Cluster que necesitamos para estructurar la información, vamos a pasar a explicar la implementación del K-means propiamente dicha. Esta implementación esta en el script *KMeans.py* que mostramos a continuación y que posteriormente vamos a explicar cada fragmento de código relevante:

```
# -*- coding: utf-8 -*-
__author__ = 'RicardoMoya'

import random
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import distance
from Point import Point
from Cluster import Cluster

DATASET1 = "./dataSet/DS_3Clusters_999Points.txt"
DATASET2 = "./dataSet/DS2_3Clusters_999Points.txt"
DATASET3 = "./dataSet/DS_5Clusters_10000Points.txt"
DATASET4 = "./dataSet/DS_7Clusters_100000Points.txt"
NUM_CLUSTERS = 3
ITERATIONS = 1000
COLORS = ['red', 'blue', 'green', 'yellow', 'gray', 'pink', 'violet', 'brown',
          'cyan', 'magenta']


def dataset_to_list_points(dir_dataset):
    """
```

```
    Read a txt file with a set of points and return a list of objects Point
    :param dir_dataset:
    """
    points = list()
    with open(dir_dataset, 'rt') as reader:
        for point in reader:
            points.append(Point(np.asarray(map(float, point.split("::")))))
    return points


def get_nearest_cluster(clusters, point):
    """
    Calculate the nearest cluster
    :param clusters: old clusters
    :param point: point to assign cluster
    :return: index of list cluster
    """
    dist = np.zeros(len(clusters))
    for i, c in enumerate(clusters):
        dist[i] = distance.euclidean(point.coordinates, c.centroid)
    return np.argmin(dist)


def print_clusters_status(it_counter, clusters):
    print '\nITERATION %d' % it_counter
    for i, c in enumerate(clusters):
        print '\tCentroid Cluster %d: %s' % (i + 1, str(c.centroid))


def print_results(clusters):
    print '\n\nFINAL RESULT:'
    for i, c in enumerate(clusters):
        print '\tCluster %d' % (i + 1)
        print '\t\tNumber Points in Cluster %d' % len(c.points)
        print '\t\tCentroid: %s' % str(c.centroid)


def plot_results(clusters):
    plt.plot()
    for i, c in enumerate(clusters):
        # plot points
        x, y = zip(*[p.coordinates for p in c.points])
        plt.plot(x, y, linestyle='None', color=COLORS[i], marker='.')
        # plot centroids
        plt.plot(c.centroid[0], c.centroid[1], 'o', color=COLORS[i],
                 markeredgecolor='k', markersize=10)
    plt.show()


def k_means(dataset, num_clusters, iterations):
    # Read data set
    points = dataset_to_list_points(dataset)

    # Select N points random to initiacize the N Clusters
    initial = random.sample(points, num_clusters)

    # Create N initial Clusters
    clusters = [Cluster([p]) for p in initial]

    # Inicialize list of lists to save the new points of cluster
    new_points_cluster = [[] for i in range(num_clusters)]

    converge = False
    it_counter = 0
```

```
while (not converge) and (it_counter < iterations):
        # Assign points in nearest centroid
        for p in points:
            i_cluster = get_nearest_cluster(clusters, p)
            new_points_cluster[i_cluster].append(p)

        # Set new points in clusters and calculate de new centroids
        for i, c in enumerate(clusters):
            c.update_cluster(new_points_cluster[i])

        # Check that converge all Clusters
        converge = [c.converge for c in clusters].count(False) == 0

        # Increment counter and delete lists of clusters points
        it_counter += 1
        new_points_cluster = [[] for i in range(num_clusters)]

        # Print clusters status
        print_clusters_status(it_counter, clusters)

    # Print final result
    print_results(clusters)

    # Plot Final results
    plot_results(clusters)


if __name__ == '__main__':
    k_means(DATASET1, NUM_CLUSTERS, ITERATIONS)
```

Lo primero que se hace al ejecutar el script es llamar el método *k_means()* al que se le pasa como parámetros el data set que contiene el conjunto de puntos, el número de Clusters que queremos obtener y el número máximo de iteraciones a realizar por si no convergen nunca los centroides. Estos parámetros los tenemos definidos como constantes en el script ya que tenemos disponibles 4 data sets.

```
if __name__ == '__main__':
    k_means(DATASET1, NUM_CLUSTERS, ITERATIONS)
```

El método *k_means()* implementa el algoritmo de los K-means tal y como se indica en el pseudocódigo mostrado anteriormente. En primer lugar leemos de un fichero el data set con el método *dataset_to_list_points(file)* e inicializamos 'N' Clusters (*num_clusteres*) seleccionando de forma aleatoria 'N' puntos del data set. Posteriormente iniciamos los pasos de asignación y actualización de los Clusters hasta que estos converjan o hasta que lleguemos al número máximo de iteraciones (*iterations*). Esto lo hacemos dentro del bucle "While". En cada iteración asignamos cada uno de los puntos del data set al Cluster que tenga el centroide más cercano (Asignación) con el método *get_nearest_cluster()* que calcula la distancia euclídea entre el punto y el centroide y devuelve el índice del Cluster con centroide más cercano, y una vez asignados recalculamos los centroides de los Clusters.

Posteriormente comprobamos la convergencia de los centroides para ver si seguimos iterando o no. En la implementación vemos los métodos *print_clusters_status()*, *print_results()* y *plot_results()* que son métodos auxiliares que utilizamos para mostrar el estado de los Clusters en cada iteración y mostrar y pintar (con la librería *matplotlib*) el resultado final:

```
def k_means(dataset, num_clusters, iterations):

    points = dataset_to_list_points(dataset)

    # INICIALIZACIÓN: Selección aleatoria de N puntos y creación de los Clusters
    initial = random.sample(points, num_clusters)
    clusters = [Cluster([p]) for p in initial]

    # Inicializamos una lista para el paso de asignación de objetos
    new_points_cluster = [[] for i in range(num_clusters)]

    converge = False
    it_counter = 0
    while (not converge) and (it_counter < iterations):
        # ASIGNACION
        for p in points:
            i_cluster = get_nearest_cluster(clusters, p)
            new_points_cluster[i_cluster].append(p)

        # ACTUALIZACIÓN
        for i, c in enumerate(clusters):
            c.update_cluster(new_points_cluster[i])

        # ¿CONVERGE?
        converge = [c.converge for c in clusters].count(False) == 0

        # Incrementamos el contador
        it_counter += 1
        new_points_cluster = [[] for i in range(num_clusters)]

        print_clusters_status(it_counter, clusters)

    print_results(clusters)

    plot_results(clusters)
```

NOTA: La implementación de los pasos de Asignación y Actualización pueden ser optimizados paralelizando los cálculos con threads (hilos). Por razones didácticas y por legibilidad y claridad del código no se ha optimizado esta parte.

Veamos a continuación un ejemplo de la ejecución de este script en el que se mostrará el estado de cada Cluster en cada una de las iteraciones hasta la obtención del resultado final. Para el ejemplo utilizaremos el data set 1 (*DATASET1*) que tiene 999 y que los agruparemos en 3 Clusters:

1. Elección de 3 puntos al azar para inicializar los Clusters. Como cada Cluster será inicializado con un solo punto, ese será su centroide:

```
CLUSTER 1:
Centroid: [1.537719613, 6.803222336]
Dimension: 2
Puntos: [ 1.53771961  6.80322234]

CLUSTER 2:
Centroid: [4.56416743, 5.372954912]
Dimension: 2
Puntos: [ 4.56416743  5.37295491]

CLUSTER 3:
Centroid: [2.580421023, 2.887390198]
Dimension: 2
Puntos: [ 2.58042102  2.8873902 ]
```

2. Iniciamos las iteraciones. Primera asignación y actualización de centroides, quedando estos de la siguiente manera:

```
ITERATION 1
      Centroid Cluster 1: [1.0075506731128203, 6.89332954671282]
      Centroid Cluster 2: [4.965693151866954, 4.989117886197427]
      Centroid Cluster 3: [2.097612910089318, 2.0566417310823106]
```

3. Iteración 2:

```
ITERATION 2
      Centroid Cluster 1: [1.0110313303520406, 6.881641713040817]
      Centroid Cluster 2: [4.902010815637452, 4.87296486188048]
      Centroid Cluster 3: [2.0337844236032625, 2.0092213794438396]
```

4. Iteración 3:

```
ITERATION 3
      Centroid Cluster 1: [1.0110313303520406, 6.881641713040817]
      Centroid Cluster 2: [4.888163414869566, 4.864725364043481]
      Centroid Cluster 3: [2.0297243138036376, 2.002597935785454]
```

5. Iteración 4: Convergen los centroides, teniendo los mismos valores que en la iteración 3.

```
ITERATION 4
      Centroid Cluster 1: [1.0110313303520406, 6.881641713040817]
      Centroid Cluster 2: [4.888163414869566, 4.864725364043481]
      Centroid Cluster 3: [2.0297243138036376, 2.002597935785454]
```

6. Resultado final y representación:

```
FINAL RESULT:
      Cluster 1
            Number Points in Cluster 196
            Centroid: [1.0110313303520406, 6.881641713040817]
```

```
    Cluster 2
            Number Points in Cluster 253
            Centroid: [4.888163414869566, 4.864725364043481]
    Cluster 3
            Number Points in Cluster 550
            Centroid: [2.0297243138036376, 2.002597935785454]
```

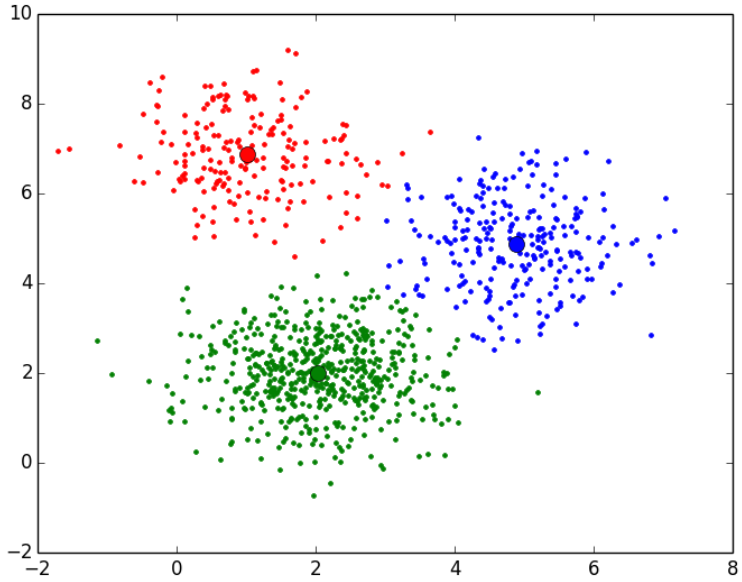

Ilustración 22: Resultado final para el Data Set 1 con 999 puntos y 3 Clusters

Para el resto de Data Sets tenemos los siguientes resultados pintados en 2 dimensiones:

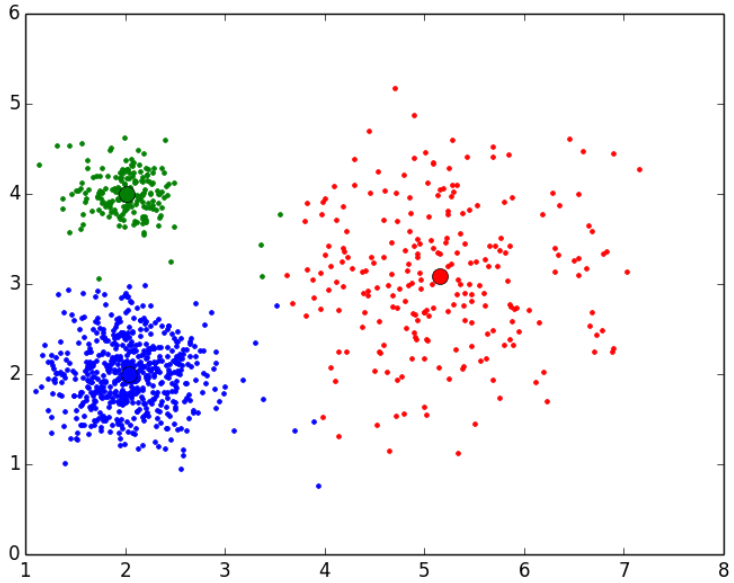

Ilustración 23: Resultado final para el Data Set 2 con 999 puntos y 3 Clusters

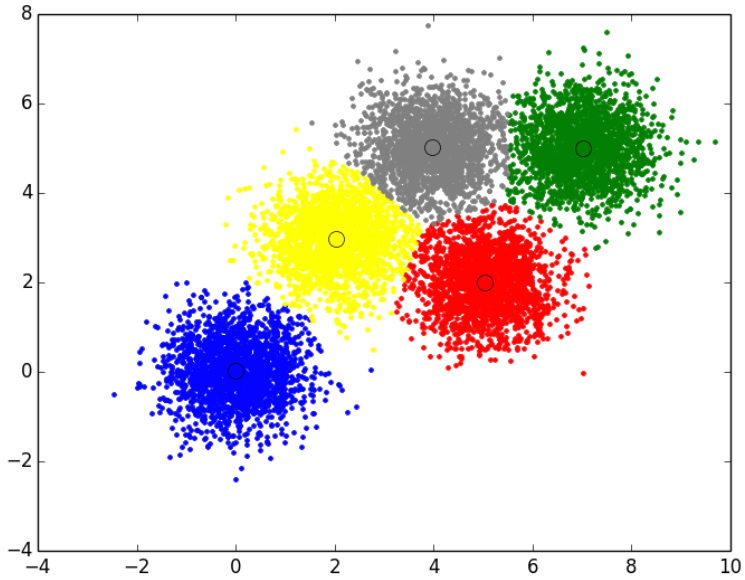

Ilustración 24: Resultado final para el Data Set 3 con 10000 puntos y 5 Clusters

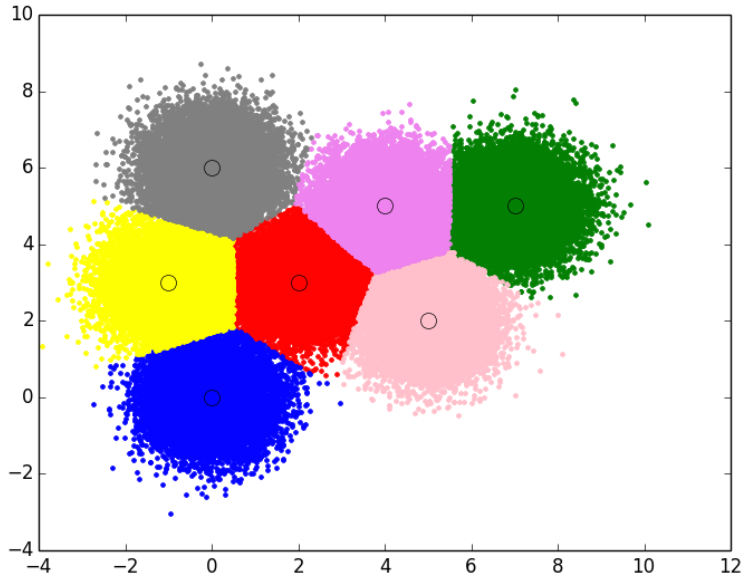

Ilustración 25: Resultado final para el Data Set 4 con 100000 puntos y 7 Clusters

K-means con scikit-learn

En la librería de "*scikit-learn*" esta implementado el K-means de forma bastante optimizada, pudiendo ser ejecutado de diferentes formas en función de los parámetros que se le pase.

Dado que las librerías en general y esta de Machine Learning en particular evolucionan y son modificadas en el tiempo, no vamos a entrar a explicar cada uno de los parámetros que se le puede pasar al constructor de la clase *sklearn.cluster.KMeans()* (para eso ya está la documentación de scikit-learn) pero si que vamos a mostrar aquellos parámetros que se consideran importantes para cualquier implementación del K-means.

En la versión 0.17 de la librería (última versión estable hasta la fecha de realización de este libro) el constructor de la clase KMeans presenta los siguientes parámetros (si no se especifican tiene unos valores definidos por defecto):

```
class sklearn.cluster.KMeans(n_clusters=8, init='k-means++', n_init=10,
max_iter=300, tol=0.0001,precompute_distances='auto', verbose=0, random_state=None,
copy_x=True, n_jobs=1)
```

Como parámetros necesarios e importantes a especificar para que la ejecución del K-means sea correcta según el data set utilizado, debemos de definir correctamente los siguientes dos parámetros:

1. **n_clusters:** Indicaremos el número de Clusters que queremos obtener para agrupar los objetos del data set.
2. **max_iter:** número máximo de ejecuciones de los pasos de asignación y actualización a realizar en el caso de que no converjan los centroides

De forma opcional podemos definir una serie de parámetros con los valores que consideremos, siendo los más prácticos (según la opinión del autor) los siguientes:

1. **init:** indicamos la forma de inicializar los Clusters. Anteriormente se propusieron diferentes formas de inicializar estos Clusters bien sea de forma aleatoria o cogiendo al azar 'k' objetos del data set. En este caso se proponen dos forma: una la opción 'random' que inicializa los Clusters de forma aleatoria y otra la opción 'k-means++' que hace un pequeño pre-procesamiento de los datos del data set para inicializar los Clusters con unos centroides que sean lo suficientemente buenos para que el algoritmo converja rápidamente.
2. **tol:** Con este valor indicamos el umbral, tolerancia o margen de error para la convergencia de los centroides de los Clusters; es decir, que el valor de los centroides no tiene porque ser iguales de una iteración a otra pero si que su diferencia se inferior a la indicada en este parámetro.
3. **n_jobs:** Le indicamos el número de hilos (threads) a utilizar.

Una vez construido el objeto de la clase KMeans, debemos de llamar a los métodos pertinentes para que nos agrupe los objetos del data set en los diferentes Clusters. En este caso y para el ejemplo que a continuación mostramos, vamos a llamar el método *fit(x)* al que pasamos como parámetro una lista de puntos (cada punto en un numpy array) y nos devuelve los ***centroides de los Clusters*** (en el atributo *cluster_centers_*), una lista alineada con la lista de puntos que le pasamos en el que nos dice a que ***Cluster pertenece cada punto*** (en el atributo *labels_*) y la ***inercia*** (en el atributo *inertia_*).

A continuación mostramos el script completo en el que mostramos los resultados del K-means para uno de los 4 data sets:

```
# -*- coding: utf-8 -*-
__author__ = 'RicardoMoya'

import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Constant
DATASET1 = "./dataSet/DS_3Clusters_999Points.txt"
DATASET2 = "./dataSet/DS2_3Clusters_999Points.txt"
DATASET3 = "./dataSet/DS_5Clusters_10000Points.txt"
DATASET4 = "./dataSet/DS_7Clusters_100000Points.txt"
NUM_CLUSTERS = 3
MAX_ITERATIONS = 10
INITIALIZE_CLUSTERS = ['k-means++', 'random']
CONVERGENCE_TOLERANCE = 0.001
NUM_THREADS = 8
COLORS = ['red', 'blue', 'green', 'yellow', 'gray', 'pink', 'violet', 'brown',
          'cyan', 'magenta']


def dataset_to_list_points(dir_dataset):
    """
    Read a txt file with a set of points and return a list of objects Point
    :param dir_dataset: path file
    """
    points = list()
    with open(dir_dataset, 'rt') as reader:
        for point in reader:
            points.append(np.asarray(map(float, point.split("::"))))
    return points


def print_results(centroids, num_cluster_points):
    print '\n\nFINAL RESULT:'
    for i, c in enumerate(centroids):
        print '\tCluster %d' % (i + 1)
        print '\t\tNumber Points in Cluster %d' % num_cluster_points.count(i)
        print '\t\tCentroid: %s' % str(centroids[i])


def plot_results(centroids, num_cluster_points, points):
    plt.plot()
    for nc in range(len(centroids)):
        # plot points
        points_in_cluster = [boolP == nc for boolP in num_cluster_points]
        for i, p in enumerate(points_in_cluster):
            if bool(p):
                plt.plot(points[i][0], points[i][1], linestyle='None',
                         color=COLORS[nc], marker='.')
        # plot centroids
        centroid = centroids[nc]
        plt.plot(centroid[0], centroid[1], 'o', markerfacecolor=COLORS[nc],
                 markeredgecolor='k', markersize=10)
    plt.show()


def k_means(dataset, num_clusters, max_iterations, init_cluster, tolerance,
            num_threads):
    # Read data set
```

```
    points = dataset_to_list_points(dataset)

    # Object KMeans
    kmeans = KMeans(n_clusters=num_clusters, max_iter=max_iterations,
                    init=init_cluster, tol=tolerance, n_jobs=num_threads)

    # Calculate Kmeans
    kmeans.fit(points)

    # Obtain centroids and number Cluster of each point
    centroids = kmeans.cluster_centers_
    num_cluster_points = kmeans.labels_.tolist()

    # Print final result
    print_results(centroids, num_cluster_points)

    # Plot Final results
    plot_results(centroids, num_cluster_points, points)


if __name__ == '__main__':
    k_means(DATASET1, NUM_CLUSTERS, MAX_ITERATIONS, INITIALIZE_CLUSTERS[0],
            CONVERGENCE_TOLERANCE, NUM_THREADS)
```

A continuación pasamos a explicar detalladamente como hemos realizado el script. Todo el trabajo se realiza en el método *k_means(dataset, num_clusters, max_iterations, init_cluster, tolerance, num_threads)* al que se le pasan como parámetros del data set, el número de Clusters, el número máximo de iteracciones, el método de inicialización de los Clusters (random o k-means++), el umbral de convergencia y el número de threads para la ejecución; en resumen, todos los parámetros explicados anteriormente. A este método le vamos a llamar desde el 'main', siendo los parámetros que se le pasan constantes definidas al principio del script:

```
# Constant
DATASET1 = "./dataSet/DS_3Clusters_999Points.txt"
NUM_CLUSTERS = 3
MAX_ITERATIONS = 100
INITIALIZE_CLUSTERS = ['k-means++', 'random']
CONVERGENCE_TOLERANCE = 0.001
NUM_THREADS = 8

--------------------------------------------------------------------------------

if __name__ == '__main__':
    k_means(DATASET1, NUM_CLUSTERS, MAX_ITERATIONS, INITIALIZE_CLUSTERS[0],
            CONVERGENCE_TOLERANCE, NUM_THREADS)
```

Dentro del método k_means() empezamos leyendo los datos (puntos) del data set del que le indicamos, metiendo esos puntos en una lista de numpy arrays:

```
def k_means(dataset, num_clusters, max_iterations, init_cluster, tolerance,
            num_threads):
    # Read data set
    points = dataset_to_list_points(dataset)
```

Inicializamos el objeto de la Clase KMeans (de scikit-learn), pasándole como parámetros el número de Clusters (*n_clusters*), número máximo de iteraciones (*max_iter*), forma de inicializar los Clusters (*init*), el umbral de tolerancia (*tol*) y el número de threads o ejecuciones en paralelo (*n_jobs*):

```
kmeans = KMeans(n_clusters=num_clusters, max_iter=max_iterations,
                init=init_cluster, tol=tolerance, n_jobs=num_threads)
```

Llamamos al método *fit(x)* al que le pasamos la lista de puntos (numpy array), para que calcule los centroides de los Clusters, el Cluster al que pertenece cada punto y la inercia:

```
kmeans.fit(points)
```

El valor de los centroides y el Cluster al que pertenece cada punto lo tenemos en los atributos *cluster_centers_* y *labels_* respectivamente, siendo el atributo *cluster_centers_* un numpy array de numpy arrays y *labels_* una lista de enteros que indica el número del Cluster al que pertenece cada punto del date set:

```
centroids = kmeans.cluster_centers_
num_cluster_points = kmeans.labels_.tolist()
```

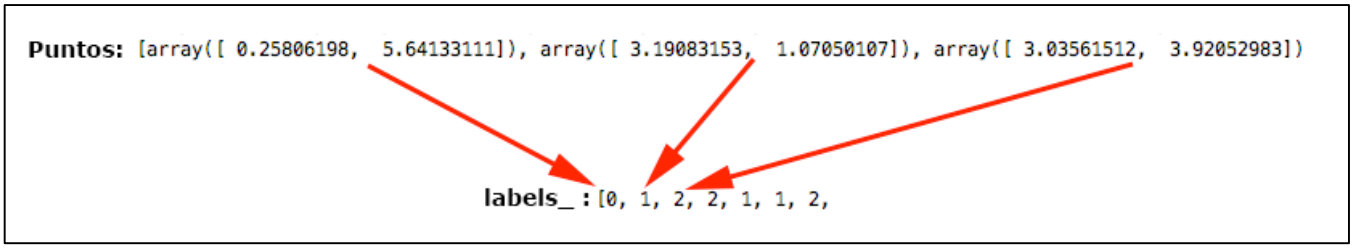

Ilustración 26: Relación de objetos con el Cluster al que pertenece

Por último implementamos dos métodos para imprimir por pantalla los resultados obtenidos (centroides, y número de puntos de cada Cluster) y para pintar (con la librería *matplotlib*), los centroides y los puntos asignados a cada Cluster:

```
print_results(centroids, num_cluster_points)
plot_results(centroids, num_cluster_points, points)
```

Aplicando este script para los 4 data sets propuestos, tenemos los siguientes resultados:

- **Data Set 1:** 3 Clusters y 999 puntos:

```
FINAL RESULT:
      Cluster 1
              Number Points in Cluster 550
              Centroid: [ 2.02972431  2.00259794]
      Cluster 2
              Number Points in Cluster 196
              Centroid: [ 1.01103133  6.88164171]
      Cluster 3
              Number Points in Cluster 253
              Centroid: [ 4.88816341  4.86472536]
```

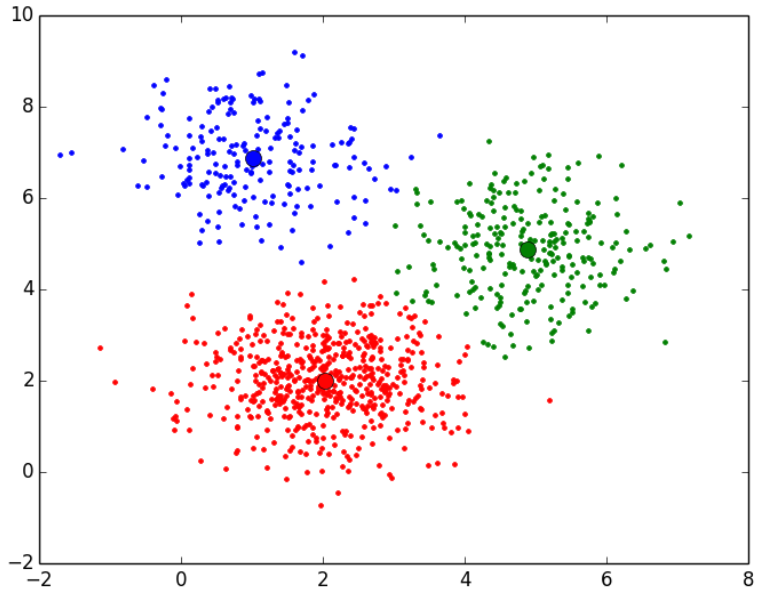

Ilustración 27: Resultado final para el Data Set 1 con 999 puntos y 3 Clusters (scikit-learn)

- **Data Set 2:** 3 Clusters y 999 puntos:

```
FINAL RESULT:
      Cluster 1
              Number Points in Cluster 191
              Centroid: [ 2.02022231  4.00032196]
      Cluster 2
              Number Points in Cluster 575
              Centroid: [ 2.04429024  1.99783017]
      Cluster 3
              Number Points in Cluster 233
              Centroid: [ 5.15350073  3.09247426]
```

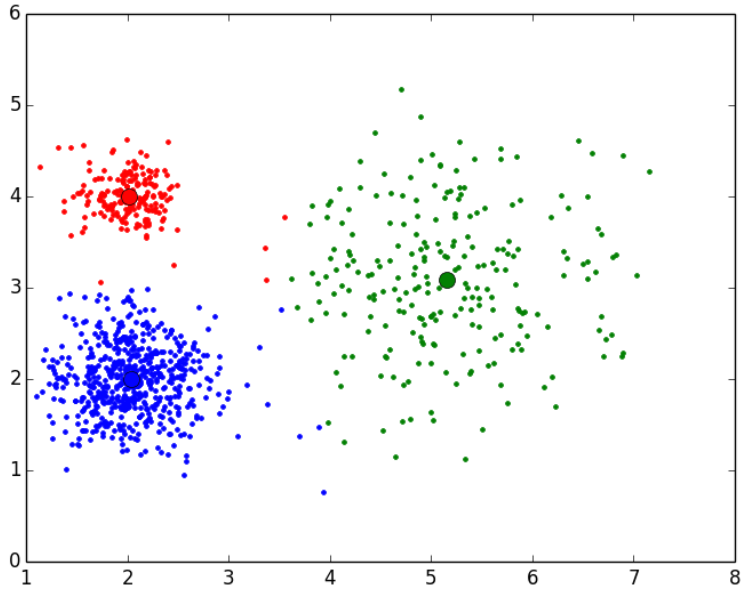

Ilustración 28: Resultado final para el Data Set 2 con 999 puntos y 3 Clusters (scikit-learn)

- **Data Set 3:** 5 Clusters y 10000 puntos:

```
FINAL RESULT:
      Cluster 1
              Number Points in Cluster 1977
              Centroid: [ 3.96715056  5.01607862]
      Cluster 2
              Number Points in Cluster 1999
              Centroid: [ 0.00084638  0.0212526 ]
      Cluster 3
              Number Points in Cluster 2003
              Centroid: [ 7.01565609  5.00501468]
      Cluster 4
              Number Points in Cluster 2009
              Centroid: [ 5.03315337  2.01020813]
      Cluster 5
              Number Points in Cluster 2012
              Centroid: [ 2.01509102  2.96339142]
```

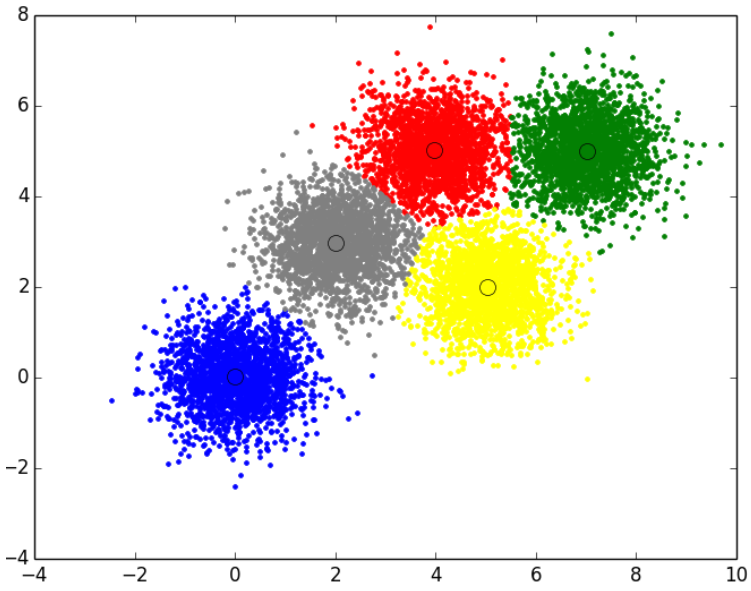

Ilustración 29: Resultado final para el Data Set 3 con 10000 puntos y 5 Clusters (scikit-learn)

- **Data Set 4:** 7 Clusters y 100000 puntos:

```
FINAL RESULT:
      Cluster 1
              Number Points in Cluster 14235
              Centroid: [ 3.99180237  5.01440659]
      Cluster 2
              Number Points in Cluster 14338
              Centroid: [ 2.0156146   2.99755542]
      Cluster 3
              Number Points in Cluster 14356
              Centroid: [ -5.19034931e-03   5.99360853e+00]
      Cluster 4
              Number Points in Cluster 14285
              Centroid: [ 0.00899975 -0.01445316]
      Cluster 5
              Number Points in Cluster 14273
              Centroid: [ 5.01486554  1.99725272]
      Cluster 6
              Number Points in Cluster 14227
              Centroid: [-1.00989699  2.99240803]
      Cluster 7
              Number Points in Cluster 14286
              Centroid: [ 7.00982285  5.00231668]
```

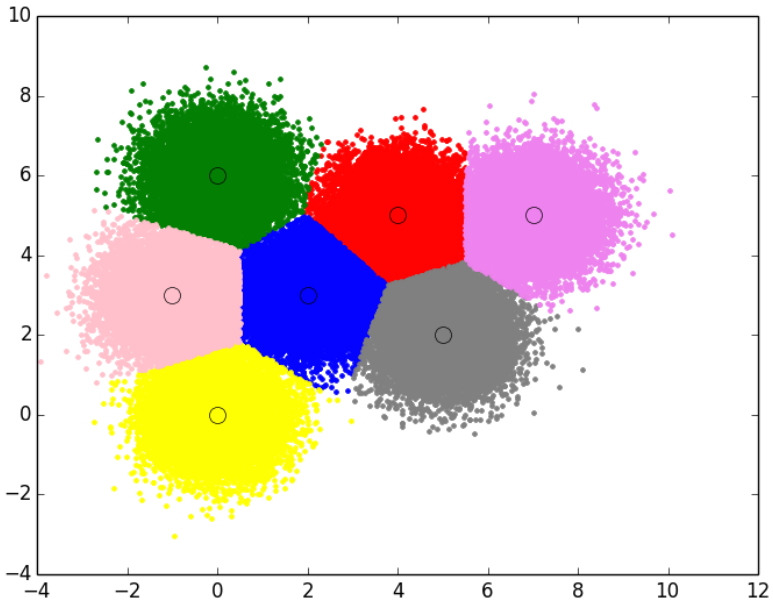

Ilustración 30: Resultado final para el Data Set 3 con 100000 puntos y 7 Clusters (scikit-learn)

Expectation-maximization (EM)

El ***Expectation-maximization*** (EM) es un método estadístico de Clustering similar al K-means, pero con un enfoque probabilístico. Este método asume que todos los objetos (del data set) han sido generados a partir de 'k' distribuciones de probabilidad de las cuales desconocemos a priori sus parámetros.

Para entender que queremos resolver con el EM, supongamos el siguiente caso en el que sabemos que dos grupos de objetos (Clusters) se generan de acuerdo a dos distribuciones normales (gaussianas) $p(x|\mu,\sigma)$ definidas por su media (μ) y su desviación típica (σ):

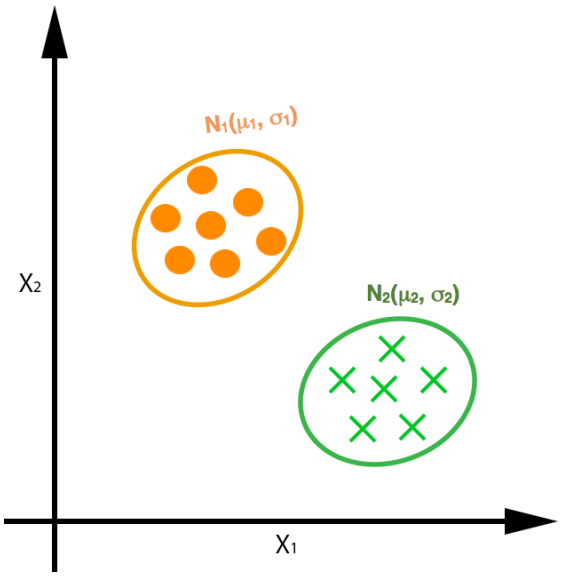

Ilustración 31: Clusters generados por 2 distribuciones normales conocidas

En este ejemplo conocemos los parámetros de las distribuciones normales (la media y desviación típica) y por tanto; dado un punto cualquiera, vamos a saber a que Cluster pertenece o a que distribución de probabilidad pertenece (o tiene más probabilidades de pertenecer).

Planteemos ahora el problema al revés, en el que tenemos un conjunto de objetos de los cuales desconocemos los parámetros de las distribuciones de probabilidad que los generan (en el caso de una distribución normal su media y desviación típica); o dicho de otra manera, el data set no nos aporta la información suficiente para saber como han sido creados los modelos probabilísticos asumidos. Esta falta de información (o información ausente) en los datos de entrenamiento se denomina "datos perdidos" o "*variables latentes*".

El EM tiene por tanto como finalidad el encontrar; a partir de los datos de entrenamiento (data set), los parámetros de las distribuciones de probabilidad que asumimos que han generado estos datos; o dicho de una manera un poco más formal, *encontrar estimadores de máxima verosimilitud de los parámetros de los modelos estadísticos que dependen de las variables latentes*.

Aunque el EM es un método que puede ser aplicado sobre cualquier distribución de probabilidad (no tiene porque ser solo sobre distribuciones normales debido a que no sabemos la naturaleza de los datos de los que disponemos), vamos a pasar a continuación

a explicar en detalle los cálculos y pasos a dar para calcular los parámetros de 'k' distribuciones normales, asumiendo que los objetos de nuestro data set se han generado de acuerdo a 'k' distribuciones normales $p(x|\mu,\sigma)$; por tanto, podemos agrupar estos datos en 'k' Clusters.

A igual que el K-means, debemos de definir previamente cuantos Clusters vamos a tener para agrupar los objetos. Supongamos que elegimos 3 Clusters (K=3). Dado un objeto de nuestro data set, asumimos que este sigue una de las tres distribuciones normales; por tanto, tendrá una determinada probabilidad 'π' de pertenecer a cada uno de los Clusters. A priori decimos, que tendrá un 33.3% de pertenecer al Cluster 1, un 33.3 % de pertenecer al Cluster 2 y un 33.4% de pertenecer al Cluster 3. Una vez que tengamos más conocimiento de los objetos que hay en cada Cluster, esta probabilidad por Cluster cambiará; por ejemplo, si tenemos 100 objetos de los cuales 50 están en el Cluster 1, 35 están en el Cluster 2 y 15 están en el Cluster 3; decimos que dado un punto, este tendrá un 50% de pertenecer al Cluster 1, un 35% de pertenecer al Cluster 2 y un 15% de pertenecer al Cluster 3.

Por otro lado; para cada una de las distribuciones de probabilidad, tenemos que calcular sus parámetros; y en este caso asumiendo que se siguen distribuciones normales, debemos de calcular la media y la desviación típica de cada una de ellas en función de los objetos de data set. Una vez vayamos ajustando estos valores; según se va ejecutando el algoritmo del EM, cada uno de los objetos del data set tendrá una probabilidad determinada de pertenecer a cada unos de los Clusters; por tanto (y de forma similar al K-means con las distancias), cada objeto pertenecerá a aquel Cluster cuya probabilidad sea mayor (el producto de $\pi_i \cdot p_i(x|\mu_i,\sigma_i)$).

A continuación se muestra en representación "*Plate Notation*" el EM para distribuciones normales (gaussianas):

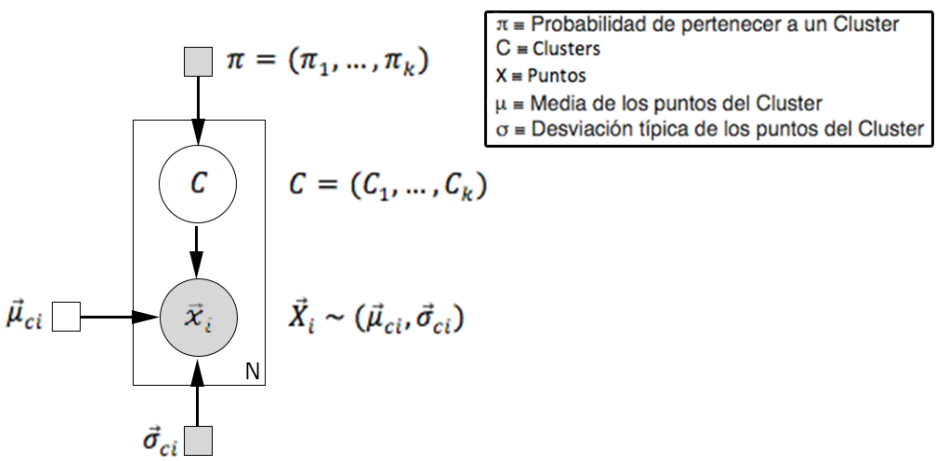

Ilustración 32: Representación gráfica (en Plate Notation) del EM, para una distribución normal (gaussiana)

En la imagen anterior definimos los círculos como distribuciones de probabilidad y los cuadrados como escalares o vectores, siendo los de fondo gris valores o distribuciones de probabilidad conocidas y las de fondo blanco desconocidas; y que por tanto hay que calcular. Lo que viene a representar la imagen anterior es que tenemos 'N' objetos en el que cada unos de ello 'x_i' esta en una distribución de probabilidad con media 'μ_i' desconocida (y que por tanto hay que calcular) y desviación típica 'σ_i' conocida. Cada uno de los objetos puede pertenecer con una probabilidad 'π_i' a cada uno de los 'k' Clusters 'C', asignando finalmente un objeto concreto del data set al Cluster cuya probabilidad definida como el producto de 'π_i' por $p_i(x|\mu_i,\sigma_i)$ sea la mayor de todas.

El problema a resolver es por tanto el calcular la probabilidad de pertenencia de un objeto a cada uno de los Cluster. Para ello tenemos lo siguiente:

$$P(X, C|\pi, \vec{\mu}, \vec{\sigma}) = P(X|C, \vec{\mu}, \vec{\sigma}) \cdot P(C|\pi)$$

Lo que nos interesa saber es la probabilidad de pertenencia de un objeto a cada uno de los Clusters:

$$P(X|C, \vec{\mu}, \vec{\sigma})$$

Tenemos por otro lado que $P(C|\pi)$ es un valor constante ($P(c = 1|\pi_1)$, $P(c = 2|\pi_2)$, …, $P(c = k|\pi_k)$), cosa que nos ayuda a la hora de realizar el cálculo de probabilidades quedando de la siguiente manera:

$$P(X|\pi, \vec{\mu}, \vec{\sigma}) = \sum_{k=1}^{K} P(X, c = k|\pi, \vec{\mu}, \vec{\sigma}) = \sum_{k=1}^{K} \pi_k \cdot P_k(X|, \vec{\mu}, \vec{\sigma})$$

Aplicando la función de densidad de una distribución normal, tenemos que la suma de todas las probabilidades de pertenencia de un punto a los Clusters es:

$$P(X|\pi, \vec{\mu}, \vec{\sigma}) = \sum_{k=1}^{K} \pi_k \cdot \frac{1}{\sigma_k\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{x-\mu_k}{\sigma_k}\right)^2}$$

Podemos simplificar esta probabilidad ya que $\frac{1}{\sqrt{2\pi}}$ es una constante, quedando:

$$P(X|\pi, \vec{\mu}, \vec{\sigma}) = \sum_{k=1}^{K} \pi_k \cdot \frac{e^{-\frac{1}{2}\left(\frac{x-\mu_k}{\sigma_k}\right)^2}}{\sigma_k}$$

Esto último lo podemos hacer ya que lo que nos interesa es asignar el objeto al Cluster con mayor probabilidad; es decir:

$$argmax \sum_{k=1}^{K} \pi_k \cdot \frac{e^{-\frac{1}{2}\left(\frac{x-\mu_k}{\sigma_k}\right)^2}}{\sigma_k}$$

Una vez definido el cálculo de la probabilidad de pertenencia de un objeto a un Cluster, vamos a explicar el funcionamiento del EM, que como se podrá apreciar va a tener un comportamiento muy similar al K-means, teniendo dos pasos: el paso de esperanza (expectation) y el paso de maximización (maximization) que es similar a los pasos de asignación y actualización del K-means respectivamente.

El funcionamiento del EM comienza ***inicializando los parámetros de las distribuciones de probabilidad y probabilidades de pertenencia a los Clusters***. Esto lo podemos hacer de muchas manera, proponiendo por ejemplo inicializar la probabilidad de pertenencia a un Cluster 'π' como 1/k y para las distribuciones normales inicializar con desviación típica a '1' y la media inicializándola con el valor de un punto del data set. Una vez inicializados los parámetros, el algoritmo continua alternando los dos siguientes pasos (esperanza y maximización) de forma iterativa hasta que las medias de las distribuciones normales converjan:

- **Esperanza (Expectation):** Con los parámetros conocidos de las distribuciones normales y la probabilidad 'π' de pertenencia a un Cluster, asignar cada objeto al Cluster con el mayor valor de probabilidad de pertenencia:

$$argmax \sum_{k=1}^{K} \pi_k \cdot P_k(X|, \vec{\mu}, \vec{\sigma})$$

- **Maximización (Maximization):** Calcular con los objetos asignados a cada Cluster el valor de los parámetros (π_k, μ_k y σ_k):

$$\pi_k = \frac{Objetos\ en\ el\ Cluster}{Total\ Objetos}$$

$$\mu_k = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i \qquad y \qquad \sigma_k = \sqrt{\frac{1}{n-1} \cdot \sum_{n=1}^{n} (x_i - \mu_k)^2}$$

Definido el funcionamiento del algoritmo paso por paso, pasamos a mostrarlo en pseudocódigo:

```
K = num_clusters

1.- Inicialización de parámetros (πi, μi y σi):
   1.1.- π = [1/k for i in range(K)]
   1.2.- μ = [random(object) for i in range(K)]
   1.3.- σ = [1 for i in range(K)]

2.- while not converge:

   for i in range(dataset):

            Ck := argmax πk e^(-1/2((x-μk)/σk)^2) / σk

   for j in range(K):

            πj := Objetos en el Cluster / Total Objetos

            μj := 1/N Σ_{i=1}^{N} xi

            σj := sqrt( 1/(n-1) · Σ_{n=1}^{n} (xi - μj)^2 )
```

En los dos siguientes puntos: Implementación del Expectation-maximization y Expectation-maximization (Gaussian mixture models) con scikit-learn, se va a mostrar la implementación del EM asumiendo que los datos son generados en base a distribuciones normales y el uso de la librería scikit-learn para la resolución de un problema de Clustering con el algoritmo del Gaussian mixture models (mezcla de modelos gaussianos) respectivamente, cuyo código se puede obtener en el siguiente repositorio:

https://github.com/RicardoMoya/ExpectationMaximization_Python

El código que se encuentra en este repositorio hace uso de las librerías de *numpy*, *matplotlib*, *scipy* y *scikit-learn*. Para descargar e instalar (o actualizar a la última versión con la opción -U) estas librerías; con el sistema de gestión de paquetes *pip*, se deben ejecutar los siguiente comandos:

```
pip install -U numpy
pip install -U matplotlib
pip install -U scipy
pip install -U scikit-learn
```

Implementación del Expectation-maximization

En este apartado se va a mostrar la implementación del EM (desde un punto de vista didáctico) en el que los objetos van a estar representados por un punto en dos dimensiones {x,y}. La implementación realizada en este ejemplo permite que los objetos puedan estar representados con más dimensiones, pero sin perder la perspectiva didáctica de este ejemplo los mostramos en dos dimensiones para que podamos visualizar los resultado en un plano y así entender el correcto funcionamiento de este método.

Para implementar el EM, vamos a tener objetos que serán representados como puntos en 2D, por tanto implementaremos una clase Punto (*Point.py*) para representar los objetos. Por otro lado se implementa una clase Cluster (*Cluster.py*) para representar a los Clusters y que estará compuesta por un conjunto de objetos de la clase Punto. Por último tendremos un script (*EM.py*) en el que estará implementado del EM para distribuciones normales con sus dos pasos de esperanza y maximización para 'k' Clusters. A continuación se muestra a un alto nivel de abstracción el diagrama de clases de la implementación propuesta del EM:

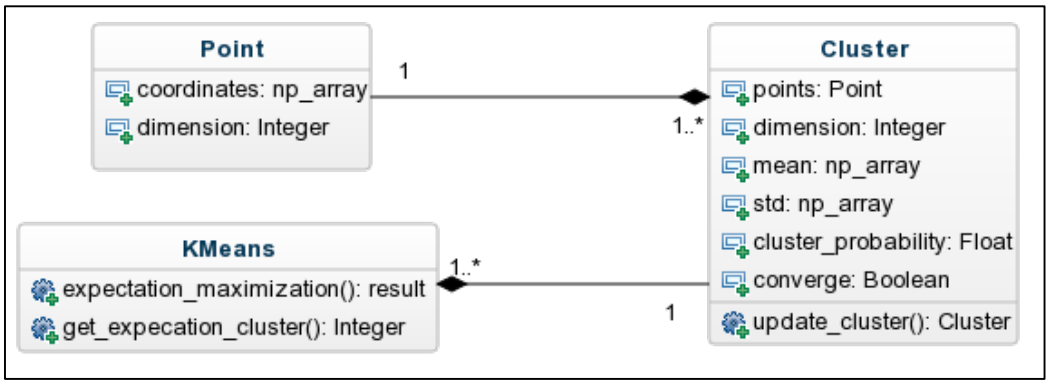

Ilustración 33: Diagrama de Clases del EM

Los data sets a utilizar en este ejemplo (que se encuentran dentro de la carpeta *dataSet*) van a ser ficheros de texto en los que en cada línea van a estar las coordenadas de cada punto {x,y} separados por el separador "::". A continuación mostramos un ejemplo de estos ficheros:

```
1.857611652::2.114033851
2.34822574::1.58264984
1.998326848::4.118143019
1.714362835::2.468639613
1.656134484::1.909955747
```

Esto quiere decir que el primer punto va a estar posicionado en las coordenadas {1.85,2.11} y el segundo punto en las coordenadas {2.34,1.58}.

Para los ejemplos disponemos de 4 data sets con las siguientes características:

Nombre	Nº Puntos	Nº Clusters teóricos	Centroides teóricos
DS_3Clusters_999Points	999	3	{2,2}
			{5,5}
			{1,7}
DS2_3Clusters_999Points	999	3	{2,2}
			{2,4}
			{5,3}
DS_5Clusters_10000Points	10000	5	{0,0}
			{2,3}
			{4,5}
			{5,2}
			{7,5}
DS_7Clusters_100000Points	100000	7	{-1,3}
			{0,0}
			{0,6}
			{2,3}
			{4,5}
			{5,2}
			{7,5}

Visto el diagrama de clases y la estructura de los date sets a utilizar, vamos a pasar a mostrar la implementación de la clases Punto (*Point.py*). A esta clase se le va a pasar en el constructor un "numpy array" con las coordenadas del punto y va a tener como atributos esa coordenada y las dimensiones de la misma, que para el ejemplo que vamos a mostrar será de 2:

```
class Point:

    def __init__(self, coordinates):
        self.coordinates = coordinates
        self.dimension = len(coordinates)

    def __repr__(self):
        return 'Coordinates: ' + str(self.coordinates) + \
               '\n\t -> Dimension: ' + str(self.dimension)
```

Por otro lado vamos a tener la clase Cluster (*Cluster.py*) que se le va a pasar en el constructor una lista de puntos que van a ser los que formen el Cluster y el número de

elementos del Cluster que lo forman. Esta clase va a tener como atributos la lista de puntos del Cluster (*points*), la dimensión de los puntos (*dimension*), la media de los puntos del Cluster (*mean*), la desviación típica de los puntos del Cluster (*std*), la probabilidad 'π' de pertenencia al Cluster (*cluster_probability*) y un atributo (*converge*) que nos va a indicar si la media es igual (por tanto converge) que el calculado en el paso de maximización de la iteración anterior. Veamos a continuación el constructor de esta clase:

```
import numpy as np

class Cluster:

    def __init__(self, points, total_points):
        if len(points) == 0:
            raise Exception("Cluster cannot have 0 Points")
        else:
            self.points = points
            self.dimension = points[0].dimension

        # Check that all elements of the cluster have the same dimension
        for p in points:
            if p.dimension != self.dimension:
                raise Exception(
                    "Point %s has dimension %d different with %d from the rest "
                    "of points") % (p, len(p), self.dimension)

        # Calculate mean, std and probability
        points_coordinates = [p.coordinates for p in self.points]
        self.mean = np.mean(points_coordinates, axis=0)
        self.std = np.array([1.0, 1.0])
        self.cluster_probability = len(self.points) / float(total_points)
        self.converge = False
```

Por otro lado implementamos el método *update_cluster(points)*, que es el método encargado de actualizar los parámetros del modelo probabilístico que define el Cluster, calculando la media y la desviación típica de la distribución normal y la probabilidad 'π' de pertenencia al Cluster; en definitiva, este método realiza el paso de maximización. Este método también comprueba si convergen las medias de la distribución de una iteración a otra. En resumen, con este método actualizamos los parámetros del modelo probabilístico que define al Cluster.

```
    def update_cluster(self, points, total_points):

        old_mean = self.mean
        self.points = points
        points_coordinates = [p.coordinates for p in self.points]
        self.mean = np.mean(points_coordinates, axis=0)
        self.std = np.std(points_coordinates, axis=0, ddof=1)
        self.cluster_probability = len(points) / float(total_points)
        self.converge = np.array_equal(old_mean, self.mean)
```

Una vez explicadas las Clases Punto y Cluster que necesitamos para estructurar la información, vamos a pasar a explicar la implementación del EM propiamente dicha. Esta implementación esta en el script *EM.py* que mostramos a continuación y que posteriormente vamos a explicar cada fragmento de código relevante:

```
# -*- coding: utf-8 -*-
__author__ = 'RicardoMoya'

import random
import math
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
from Point import Point
from Cluster import Cluster

DATASET1 = "./dataSet/DS_3Clusters_999Points.txt"
DATASET2 = "./dataSet/DS2_3Clusters_999Points.txt"
DATASET3 = "./dataSet/DS_5Clusters_10000Points.txt"
DATASET4 = "./dataSet/DS_7Clusters_100000Points.txt"
NUM_CLUSTERS = 3
ITERATIONS = 1000
COLORS = ['red', 'blue', 'green', 'yellow', 'gray', 'pink', 'violet', 'brown',
          'cyan', 'magenta']


def dataset_to_list_points(dir_dataset):
    """
    Read a txt file with a set of points and return a list of objects Point
    :param dir_dataset: path file
    """
    points = list()
    with open(dir_dataset, 'rt') as reader:
        for point in reader:
            points.append(Point(np.asarray(map(float, point.split("::")))))
    return points


def get_probability_cluster(point, cluster):
    """
    Calculate the probability that the point belongs to the Cluster
    :param point:
    :param cluster:
    :return: probability =
    prob * SUM(e ^ (-1/2 * ((x(i) - mean)^2 / std(i)^2 )) / std(i))
    """
    mean = cluster.mean
    std = cluster.std
    prob = 1.0
    for i in range(point.dimension):
        prob *= (math.exp(-0.5 * (
            math.pow((point.coordinates[i] - mean[i]), 2) /
            math.pow(std[i], 2))) / std[i])

    return cluster.cluster_probability * prob


def get_expecation_cluster(clusters, point):
    """
    Returns the Cluster that has the highest probability of belonging to it
    :param clusters:
```

```
    :param point:
    :return: argmax (probability clusters)
    """
    expectation = np.zeros(len(clusters))
    for i, c in enumerate(clusters):
        expectation[i] = get_probability_cluster(point, c)

    return np.argmax(expectation)


def print_clusters_status(it_counter, clusters):
    print '\nITERATION %d' % it_counter
    for i, c in enumerate(clusters):
        print '\tCluster %d: Probability = %s; Mean = %s; Std = %s;' % (
            i + 1, str(c.cluster_probability), str(c.mean), str(c.std))


def print_results(clusters):
    print '\n\nFINAL RESULT:'
    for i, c in enumerate(clusters):
        print '\tCluster %d' % (i + 1)
        print '\t\tNumber Points in Cluster: %d' % len(c.points)
        print '\t\tProbability: %s' % str(c.cluster_probability)
        print '\t\tMean: %s' % str(c.mean)
        print '\t\tStandard Desviation: %s' % str(c.std)


def plot_ellipse(center, points, alpha, color):
    """
    Plot the Ellipse that defines the area of Cluster
    :param center:
    :param points: points of cluster
    :param alpha:
    :param color:
    :return: Ellipse
    """

    # Matrix Covariance
    cov = np.cov(points, rowvar=False)

    # eigenvalues and eigenvector of matrix covariance
    eigenvalues, eigenvector = np.linalg.eigh(cov)
    order = eigenvalues.argsort()[::-1]
    eigenvector = eigenvector[:, order]

    # Calculate Angle of ellipse
    angle = np.degrees(np.arctan2(*eigenvector[:, 0][::-1]))

    # Calculate with, height
    width, height = 4 * np.sqrt(eigenvalues[order])

    # Ellipse Object
    ellipse = Ellipse(xy=center, width=width, height=height, angle=angle,
                      alpha=alpha, color=color)

    ax = plt.gca()
    ax.add_artist(ellipse)

    return ellipse


def plot_results(clusters):
    plt.plot()
    for i, c in enumerate(clusters):
        # plot points
```

```
        x, y = zip(*[p.coordinates for p in c.points])
        plt.plot(x, y, linestyle='None', color=COLORS[i], marker='.')
        # plot centroids
        plt.plot(c.mean[0], c.mean[1], 'o', color=COLORS[i],
                 markeredgecolor='k', markersize=10)
        # plot area
        plot_ellipse(c.mean, [p.coordinates for p in c.points], 0.2, COLORS[i])

    plt.show()


def expectation_maximization(dataset, num_clusters, iterations):
    # Read data set
    points = dataset_to_list_points(dataset)

    # Select N points random to initiacize the N Clusters
    initial = random.sample(points, num_clusters)

    # Create N initial Clusters
    clusters = [Cluster([p], len(initial)) for p in initial]

    # Inicialize list of lists to save the new points of cluster
    new_points_cluster = [[] for i in range(num_clusters)]

    converge = False
    it_counter = 0
    while (not converge) and (it_counter < iterations):
        # Expectation Step
        for p in points:
            i_cluster = get_expecation_cluster(clusters, p)
            new_points_cluster[i_cluster].append(p)

        # Maximization Step
        for i, c in enumerate(clusters):
            c.update_cluster(new_points_cluster[i], len(points))

        # Check that converge all Clusters
        converge = [c.converge for c in clusters].count(False) == 0

        # Increment counter and delete lists of clusters points
        it_counter += 1
        new_points_cluster = [[] for i in range(num_clusters)]

        # Print clusters status
        print_clusters_status(it_counter, clusters)

    # Print final result
    print_results(clusters)

    # Plot Final results
    plot_results(clusters)


if __name__ == '__main__':
    expectation_maximization(DATASET1, NUM_CLUSTERS, ITERATIONS)
```

Lo primero que se hace al ejecutar el script es llamar el método *expectation_maximization()* al que se le pasa como parámetros el data set que contiene el conjunto de puntos, el número de Clusters que queremos obtener y el número máximo de iteraciones a realizar por si no convergen nunca las medias de las distribuciones de

probabilidad. Estos parámetros los tenemos definidos como constantes en el script ya que tenemos disponibles 4 data sets.

```
if __name__ == '__main__':
    expectation_maximization(DATASET1, NUM_CLUSTERS, ITERATIONS)
```

El método *expectation_maximization()* implementa el EM tal y como se indica en el pseudocódigo mostrado anteriormente. En primer lugar leemos de un fichero el data set con el método *dataset_to_list_points(file)* e inicializamos las 'N' distribuciones que definen los Clusters (*num_clusteres*) con la media seleccionando de forma aleatoria 'N' puntos del data set. Posteriormente iniciamos los pasos de esperanza y maximización hasta que estos converjan o hasta que lleguemos al número máximo de iteraciones (*iterations*). Esto lo hacemos dentro del bucle "While". En cada iteración asignamos cada uno de los puntos del data set a la distribución de probabilidad que mayor probabilidad de pertenencia tenga (el producto de $\pi_i \cdot p_i(x|\mu_i,\sigma_i)$) con el método *get_expectation_cluster()* que devuelve el índice de la distribución (o el Cluster), y una vez asignados recalculamos los parámetros de las distribuciones de probabilidad. Posteriormente comprobamos la convergencia para ver si seguimos iterando o no. En la implementación vemos los métodos *print_clusters_status()*, *print_results()* y *plot_results()* que son métodos auxiliares que utilizamos para mostrar los parámetros de las distribuciones de probabilidad que definen los Clusters en cada iteración y mostrar y pintar (con la librería *matplotlib*) el resultado final:

```
def expectation_maximization(dataset, num_clusters, iterations):

    points = dataset_to_list_points(dataset)

    # INICIALIZACIÓN: Selección aleatoria de N puntos para inicializar la media de
    # las de las distribuciones normales que definen un Cluster
    initial = random.sample(points, num_clusters)

    # Create N initial Clusters
    clusters = [Cluster([p], len(initial)) for p in initial]

    # Inicializamos una lista para el paso de esperanza (Expectation)
    new_points_cluster = [[] for i in range(num_clusters)]

    converge = False
    it_counter = 0
    while (not converge) and (it_counter < iterations):
        # Esperanza
        for p in points:
            i_cluster = get_expecation_cluster(clusters, p)
            new_points_cluster[i_cluster].append(p)

        # Maximización
        for i, c in enumerate(clusters):
            c.update_cluster(new_points_cluster[i], len(points))

        # ¿CONVERGE?
```

```
        converge = [c.converge for c in clusters].count(False) == 0

        # Incrementamos el contador
        it_counter += 1
        new_points_cluster = [[] for i in range(num_clusters)]

        print_clusters_status(it_counter, clusters)

    print_results(clusters)

    plot_results(clusters)
```

NOTA: La implementación de los pasos de Esperanza y Maximización pueden ser optimizados paralelizando los cálculos con threads (hilos). Por razones didácticas y por legibilidad y claridad del código no se ha optimizado esta parte.

Veamos a continuación un ejemplo de la ejecución de este script en el que se mostrará el estado de cada Cluster en cada una de las iteraciones hasta la obtención del resultado final. Para el ejemplo utilizaremos el data set 1 (*DATASET1*) que tiene 999 y que los agruparemos en 3 Clusters:

1. Elección de 3 puntos al azar para inicializar las medias de las distribuciones normales. Por otro las inicializamos la desviación típica a '1' y la probabilidad de pertenencia a un Cluster como 1/3:

```
Cluster 1: Probability = 0.33; Mean = [ 1.77  6.21]; Std = [ 1.  1.];
Cluster 2: Probability = 0.33; Mean = [ 4.97  6.91]; Std = [ 1.  1.];
Cluster 3: Probability = 0.33; Mean = [ 2.39  1.49]; Std = [ 1.  1.];
```

2. Iniciamos las iteraciones: Iteración 1

```
Cluster 1: Probability = 0.21; Mean = [ 1.15  6.69]; Std = [ 1.01  1.01];
Cluster 2: Probability = 0.21; Mean = [ 5.05  5.09]; Std = [ 0.72  0.83];
Cluster 3: Probability = 0.58; Mean = [ 2.17  2.06]; Std = [ 1.04  0.87];
```

3. Iteración 2:

```
Cluster 1: Probability = 0.20; Mean = [ 1.05  6.86]; Std = [ 0.92  0.91];
Cluster 2: Probability = 0.23; Mean = [ 4.98  4.95]; Std = [ 0.75  0.89];
Cluster 3: Probability = 0.57; Mean = [ 2.08  2.04]; Std = [ 0.95  0.87];
```

4. Iteración 3:

```
Cluster 1: Probability = 0.20; Mean = [ 1.01  6.88]; Std = [ 0.87  0.90];
Cluster 2: Probability = 0.24; Mean = [ 4.93  4.92]; Std = [ 0.78  0.93];
Cluster 3: Probability = 0.56; Mean = [ 2.06  2.02]; Std = [ 0.92  0.86];
```

5. Iteración 4:

```
Cluster 1: Probability = 0.19; Mean = [ 0.99  6.88]; Std = [ 0.86  0.90];
Cluster 2: Probability = 0.25; Mean = [ 4.91  4.92]; Std = [ 0.79  0.94];
Cluster 3: Probability = 0.56; Mean = [ 2.05  2.02]; Std = [ 0.92  0.85];
```

6. Iteración 5:

```
Cluster 1: Probability = 0.19; Mean = [ 0.99  6.88]; Std = [ 0.86  0.90];
Cluster 2: Probability = 0.25; Mean = [ 4.91  4.92]; Std = [ 0.80  0.94];
Cluster 3: Probability = 0.56; Mean = [ 2.05  2.01]; Std = [ 0.92  0.85];
```

7. Iteración 6 y resultado final:

```
FINAL RESULT:
    Cluster 1
            Number Points in Cluster: 195
            Probability: 0.195195195195
            Mean: [ 0.99752164  6.87917477]
            Standard Desviation: [ 0.85525541  0.90396413]
    Cluster 2
            Number Points in Cluster: 248
            Probability: 0.248248248248
            Mean: [ 4.90529473  4.91753593]
            Standard Desviation: [ 0.79991193  0.94110496]
    Cluster 3
            Number Points in Cluster: 556
            Probability: 0.556556556557
            Mean: [ 2.05069432  2.01442116]
            Standard Desviation: [ 0.92130407  0.85040552]
```

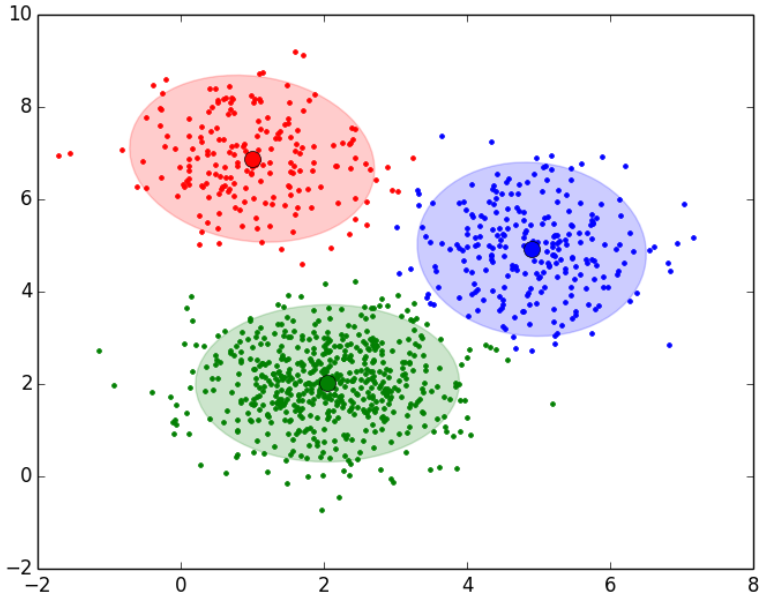

Ilustración 34: Resultado final del EM para el Data Set 1 con 999 puntos y 3 Clusters

Para el resto de Data Sets tenemos los siguientes resultados pintados en 2 dimensiones:

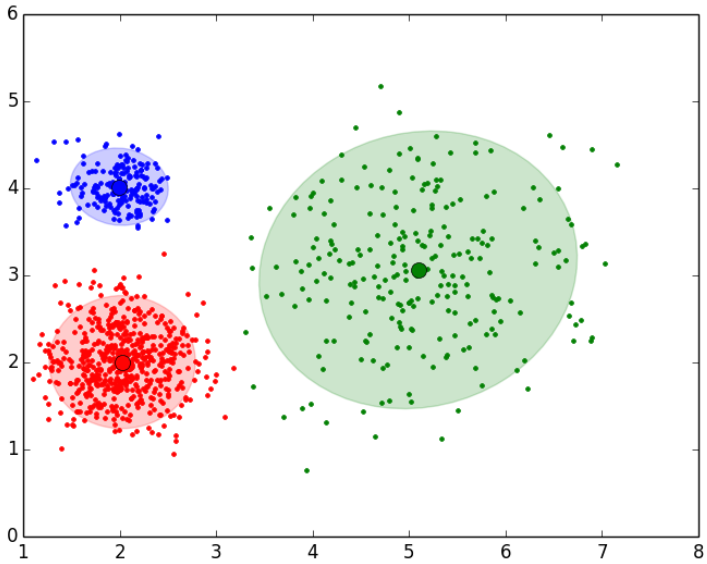

Ilustración 35: Resultado final del EM para el Data Set 2 con 999 puntos y 3 Clusters

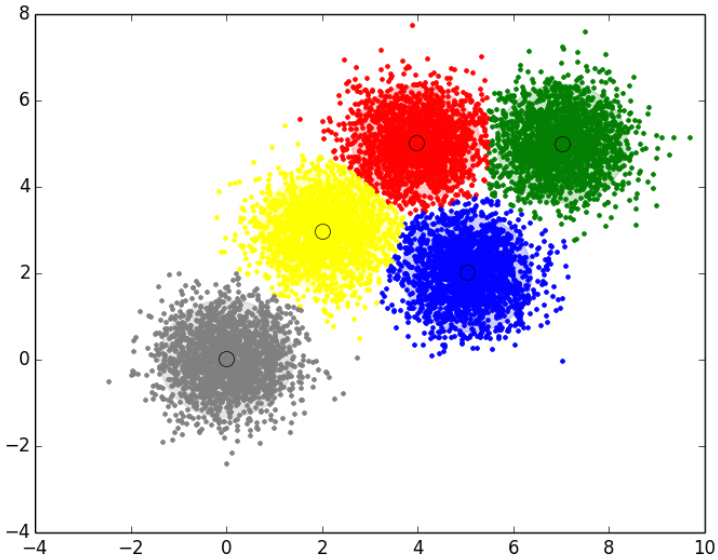

Ilustración 36: Resultado final del EM para el Data Set 3 con 10000 puntos y 5 Clusters

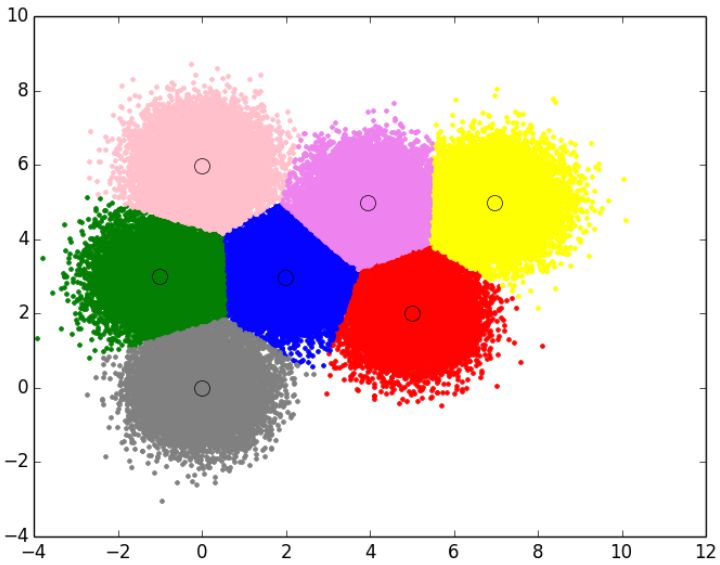

Ilustración 37: Resultado final del EM para el Data Set 4 con 100000 puntos y 7 Clusters

Expectation-maximization (Gaussian mixture models) con scikit-learn

En la librería de "*scikit-learn*" esta implementado el Gaussian mixture models (GMM) que es el EM asumiendo distribuciones normales de probabilidad (o gaussianas); es decir, que los datos a Clusterizar se asume que han sido generados por distribuciones gaussianas, de ahí el nombre de "mezcla de modelos gaussianos". Esta implementación del GMM, esta implementada de forma bastante optimizada, pudiendo ser ejecutado de diferentes formas en función de los parámetros que se le pase.

Dado que las librerías en general y esta de Machine Learning en particular evolucionan y son modificadas en el tiempo, no vamos a entrar a explicar cada uno de los parámetros que se le puede pasar al constructor de la clase *sklearn.cluster.GMM()* (para eso ya está la documentación de scikit-learn) pero si que vamos a mostrar aquellos parámetros que se consideran importantes para cualquier implementación del EM (o en esta caso del GMM).

En la versión 0.17 de la librería (última versión estable hasta la fecha de realización de este libro) el constructor de la clase GMM presenta los siguientes parámetros (si no se especifican tiene unos valores definidos por defecto):

```
class sklearn.mixture.GMM(n_components=1, covariance_type='diag', random_state=None
, thresh=None,tol=0.001, min_covar=0.001, n_iter=100, n_init=1, params='wmc', init_
params='wmc', verbose=0)
```

Como parámetros necesarios e importantes a especificar para que la ejecución del GMM sea correcta según el data set utilizado, debemos de definir correctamente los siguientes dos parámetros:

1. **n_components:** Indicaremos el número de distribuciones normales (o Clusters) que queremos obtener para agrupar los objetos del data set.
2. **n_iter:** número máximo de ejecuciones de los pasos de esperanza y maximización a realizar en el caso de que no converjan las medias de las distribuciones normales.

De forma opcional podemos definir una serie de parámetros con los valores que consideremos, siendo los más prácticos (según la opinión del autor) los siguientes:

1. **tol:** Con este valor indicamos el umbral, tolerancia o margen de error para la convergencia.
2. **params:** Le indicamos los parámetros que debe de actualizar en el paso de maximización.

Una vez construido el objeto de la clase GMM, debemos de llamar a los métodos pertinentes para que nos calcule los parámetros anteriormente indicados. En este caso y para el ejemplo que a continuación mostramos, vamos a llamar el método *fit(x)* al que pasamos como parámetro una lista de puntos (cada punto en un numpy array) y nos devuelve la probabilidad 'π' de ***pertenencia a las distribuciones normales*** (en el atributo *weights_*) y la media de las distribuciones normales (en el atributo *means_*). Por otro lado llamando al método *predict(x)*; al que pasamos como parámetro una lista con los puntos del data set, nos calcula una lista alineada con la lista de puntos que le pasamos en el que nos dice a que ***distribución normal o Cluster pertenece cada punto.***

A continuación mostramos el script completo en el que mostramos los resultados del GMM para uno de los 4 data sets:

```
# -*- coding: utf-8 -*-
__author__ = 'RicardoMoya'

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
from sklearn.mixture import GMM

# Constant
DATASET1 = "./dataSet/DS_3Clusters_999Points.txt"
DATASET2 = "./dataSet/DS2_3Clusters_999Points.txt"
DATASET3 = "./dataSet/DS_5Clusters_10000Points.txt"
DATASET4 = "./dataSet/DS_7Clusters_100000Points.txt"
NUM_CLUSTERS = 3
MAX_ITERATIONS = 10
CONVERGENCE_TOLERANCE = 0.001
COLORS = ['red', 'blue', 'green', 'yellow', 'gray', 'pink', 'violet', 'brown',
          'cyan', 'magenta']


def dataset_to_list_points(dir_dataset):
    """
    Read a txt file with a set of points and return a list of objects Point
    :param dir_dataset:
    """
    points = list()
    with open(dir_dataset, 'rt') as reader:
        for point in reader:
            points.append(np.asarray(map(float, point.split("::"))))
    return points


def print_results(means_clusters, probability_clusters, label_cluster_points):
    print '\n\nFINAL RESULT:'
    for i, c in enumerate(means_clusters):
        print '\tCluster %d' % (i + 1)
        print '\t\tNumber Points in Cluster %d' % label_cluster_points.count(i)
        print '\t\tCentroid: %s' % str(means_clusters[i])
        print '\t\tProbability: %02f%%' % (probability_clusters[i] * 100)
```

```
def plot_ellipse(center, covariance, alpha, color):

    # eigenvalues and eigenvector of matrix covariance
    eigenvalues, eigenvector = np.linalg.eigh(covariance)
    order = eigenvalues.argsort()[::-1]
    eigenvector = eigenvector[:, order]

    # Calculate Angle of ellipse
    angle = np.degrees(np.arctan2(*eigenvector[:, 0][::-1]))

    # Calculate with, height
    width, height = 4 * np.sqrt(eigenvalues[order])

    # Ellipse Object
    ellipse = Ellipse(xy=center, width=width, height=height, angle=angle,
                      alpha=alpha, color=color)

    ax = plt.gca()
    ax.add_artist(ellipse)

    return ellipse


def plot_results(points, means_clusters, label_cluster_points,
                 covars_matrix_clusters):
    plt.plot()
    for nc in range(len(means_clusters)):
        # Plot points in cluster
        points_cluster = list()
        for i, p in enumerate(label_cluster_points):
            if p == nc:
                plt.plot(points[i][0], points[i][1], linestyle='None',
                         color=COLORS[nc], marker='.')
                points_cluster.append(points[i])
        # Plot mean
        mean = means_clusters[nc]
        plt.plot(mean[0], mean[1], 'o', markerfacecolor=COLORS[nc],
                 markeredgecolor='k', markersize=10)

        # Plot Ellipse
        plot_ellipse(mean, covars_matrix_clusters[nc], 0.2, COLORS[nc])

    plt.show()


def expectation_maximization(dataset, num_clusters, tolerance, max_iterations):
    # Read data set
    points = dataset_to_list_points(dataset)

    # Object GMM
    gmm = GMM(n_components=num_clusters, covariance_type='full', tol=tolerance,
              n_init=max_iterations, params='wmc')

    # Estimate Model (params='wmc'). Calculate, w=weights, m=mean, c=covars
    gmm.fit(points)

    # Predict Cluster of each point
    label_cluster_points = gmm.predict(points)

    means_clusters = gmm.means_
    probability_clusters = gmm.weights_
    covars_matrix_clusters = gmm.covars_
```

```
    # Print final result
    print_results(means_clusters, probability_clusters,
                  label_cluster_points.tolist())

    # Plot Final results
    plot_results(points, means_clusters, label_cluster_points,
                 covars_matrix_clusters)


if __name__ == '__main__':
    expectation_maximization(DATASET1, NUM_CLUSTERS, CONVERGENCE_TOLERANCE,
                             MAX_ITERATIONS)
```

A continuación pasamos a explicar detalladamente como hemos realizado el script. Todo el trabajo se realiza en el método *expectation_maximization(dataset, num_clusters, tolerance, max_iterations)* al que se le pasan como parámetros del data set, el número de Clusters, el umbral de convergencia y el número máximo de iteraciones. A este método le vamos a llamar desde el 'main', siendo los parámetros que se le pasan constantes definidas al principio del script:

```
# Constant
DATASET1 = "./dataSet/DS_3Clusters_999Points.txt"
DATASET2 = "./dataSet/DS2_3Clusters_999Points.txt"
DATASET3 = "./dataSet/DS_5Clusters_10000Points.txt"
DATASET4 = "./dataSet/DS_7Clusters_100000Points.txt"
NUM_CLUSTERS = 3
MAX_ITERATIONS = 10
CONVERGENCE_TOLERANCE = 0.001
COLORS = ['red', 'blue', 'green', 'yellow', 'gray', 'pink', 'violet', 'brown',
          'cyan', 'magenta']

--------------------------------------------------------------------------------

if __name__ == '__main__':
    expectation_maximization(DATASET1, NUM_CLUSTERS, CONVERGENCE_TOLERANCE,
                             MAX_ITERATIONS)
```

Dentro del método *expectation_maximization()* empezamos leyendo los datos (puntos) del data set del que le indicamos, metiendo esos puntos en una lista de numpy arrays:

```
def expectation_maximization(dataset, num_clusters, tolerance, max_iterations):
    # Read data set
    points = dataset_to_list_points(dataset)
```

Inicializamos el objeto de la Clase GMM (de scikit-learn), pasándole como parámetros el número de Clusters (*n_components*), el umbral de tolerancia (*tol*) y número máximo de iteraciones (*n_init*) y los parámetros a actualizar en el paso de maximización (*params*):

```
gmm = GMM(n_components=num_clusters, covariance_type='full', tol=tolerance,
          n_init=max_iterations, params='wmc')
```

Llamamos al método *fit(x)* y al método *predict(x)*; al que les pasamos la lista de puntos (numpy array), para que calculen los parámetros de las distribuciones y el Cluster al que pertenece cada punto:

```
# Estimate Model (params='wmc'). Calculate, w=weights, m=mean, c=covars
gmm.fit(points)

# Predict Cluster of each point
label_cluster_points = gmm.predict(points)
```

Obtenemos los valores de la media de las distribuciones de los Clusters (*means_clusters*), la probabilidad de pertenencia a cada Cluster (*probability_clusters*) y la matriz de covarianzas (*covars_matrix_clusters*) que utilizaremos para pintar un área de dominio del Cluster:

```
means_clusters = gmm.means_
probability_clusters = gmm.weights_
covars_matrix_clusters = gmm.covars_
```

Por último implementamos dos métodos para imprimir por pantalla los resultados obtenidos (punto medio, número de puntos de cada Cluster y la probabilidad de pertenencia) y para pintar (con la librería *matplotlib*), los puntos medios y los puntos asignados a cada Cluster:

```
print_results(means_clusters, probability_clusters, label_cluster_points.tolist())
plot_results(points, means_clusters, label_cluster_points, covars_matrix_clusters)
```

Aplicando este script para los 4 data sets propuestos, tenemos los siguientes resultados:

- **Data Set 1:** 3 Clusters y 999 puntos:

```
FINAL RESULT:
      Cluster 1
              Number Points in Cluster 552
              Centroid: [ 2.03233538  2.00853155]
              Probability: 55.061812%
      Cluster 2
              Number Points in Cluster 253
              Centroid: [ 4.85698668  4.87166594]
              Probability: 25.632025%
      Cluster 3
              Number Points in Cluster 194
              Centroid: [ 0.98302746  6.88925282]
              Probability: 19.306162%
```

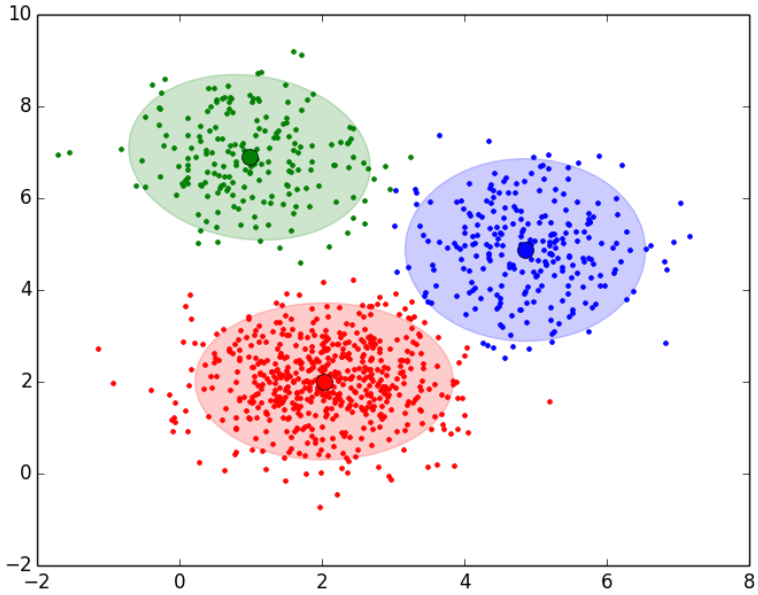

Ilustración 38: Resultado final del GMM para el Data Set 1 con 999 puntos y 3 Clusters (scikit-learn)

- **Data Set 2:** 3 Clusters y 999 puntos:

```
FINAL RESULT:
      Cluster 1
              Number Points in Cluster 186
              Centroid: [ 1.99735203  4.01744321]
              Probability: 18.639301%
      Cluster 2
              Number Points in Cluster 242
              Centroid: [ 5.08393907  3.06038361]
              Probability: 24.320184%
      Cluster 3
              Number Points in Cluster 571
              Centroid: [ 2.02688227  2.00363213]
              Probability: 57.040515%
```

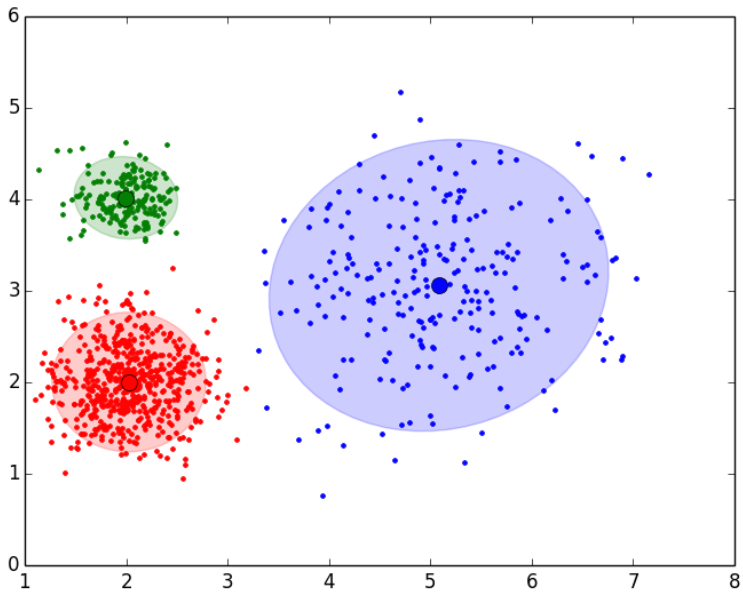

Ilustración 39: Resultado final del GMM para el Data Set 2 con 999 puntos y 3 Clusters (scikit-learn)

- **Data Set 3:** 5 Clusters y 10000 puntos:

```
FINAL RESULT:
      Cluster 1
              Number Points in Cluster 1971
              Centroid: [ 7.02380409  4.97477739]
              Probability: 19.503728%
      Cluster 2
              Number Points in Cluster 1935
              Centroid: [-0.03006596 -0.03848878]
              Probability: 18.591360%
      Cluster 3
              Number Points in Cluster 2074
              Centroid: [ 3.5474937   4.55700106]
              Probability: 19.963076%
      Cluster 4
              Number Points in Cluster 2012
              Centroid: [ 5.0348546   2.00442168]
              Probability: 19.923560%
      Cluster 5
              Number Points in Cluster 2008
              Centroid: [ 2.41381883  3.29885127]
              Probability: 22.018275%
```

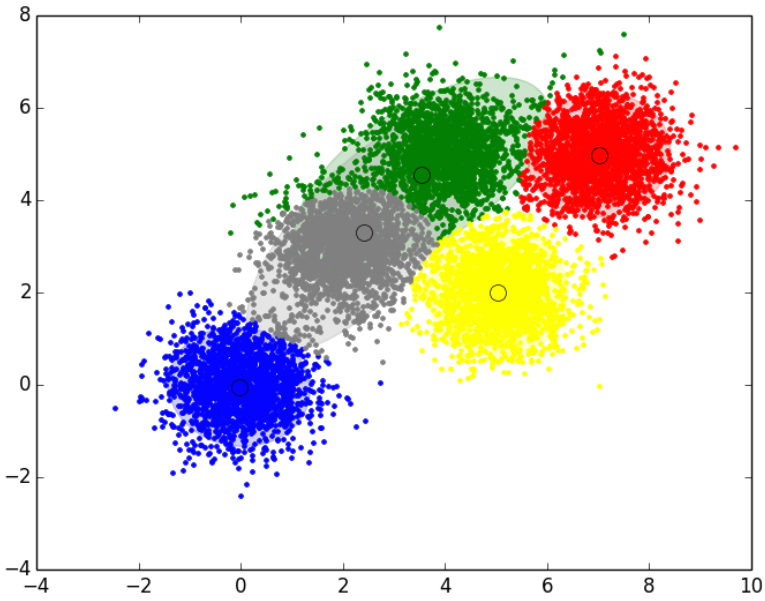

Ilustración 40: Resultado final del GMM para el Data Set 3 con 10000 puntos y 5 Clusters (scikit-learn)

- **Data Set 4:** 7 Clusters y 100000 puntos:

```
FINAL RESULT:
      Cluster 1
              Number Points in Cluster 14270
              Centroid: [ 5.00790293  2.00510173]
              Probability: 14.267490%
      Cluster 2
              Number Points in Cluster 14219
              Centroid: [-0.99805653  2.99389472]
              Probability: 14.217667%
      Cluster 3
              Number Points in Cluster 14407
              Centroid: [ 6.97099609  4.99667353]
              Probability: 14.545643%
      Cluster 4
              Number Points in Cluster 14090
              Centroid: [ 1.99643225  2.98306353]
              Probability: 13.934259%
      Cluster 5
              Number Points in Cluster 14329
              Centroid: [ 0.0083095  -0.00197396]
              Probability: 14.331673%
      Cluster 6
              Number Points in Cluster 14366
              Centroid: [ -4.37927760e-03   5.98768206e+00]
              Probability: 14.355278%
      Cluster 7
              Number Points in Cluster 14319
              Centroid: [ 3.940767    4.97634401]
              Probability: 14.347991%
```

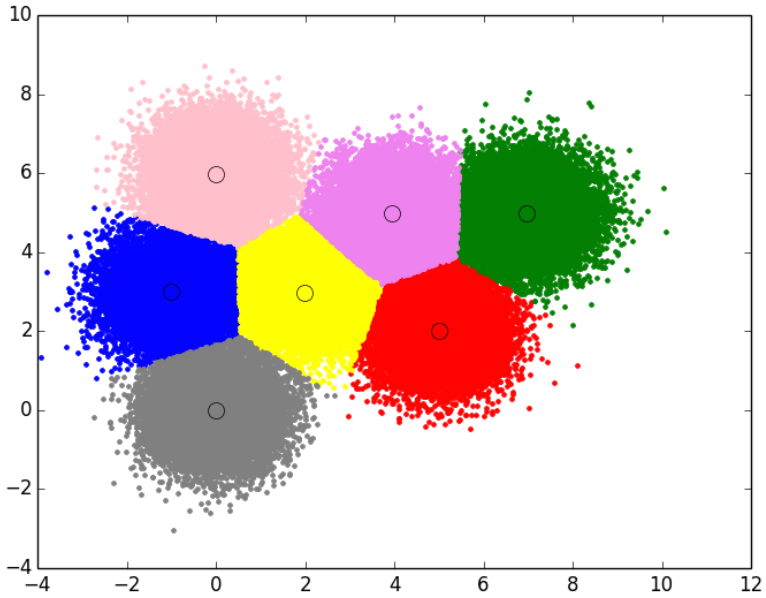

Ilustración 41: Resultado final del GMM para el Data Set 3 con 100000 puntos y 7 Clusters (scikit-learn)

Selección del número óptimo de Clusters

Uno de los problemas que nos encontramos a la hora de aplicar alguno de los métodos de Clustering (K-means o EM) es la elección del número de Clusters. No existe un criterio objetivo ni ampliamente válido para la elección de un número óptimo de Clusters; pero tenemos que tener en cuenta, que una mala elección de los mismos puede dar lugar a realizar agrupaciones de datos muy heterogéneos (pocos Clusters); o datos, que siendo muy similares unos a otros los agrupemos en Clusters diferentes (muchos Clusters).

Aunque no exista un criterio objetivo para la selección del número de Clusters, si que se han implementado diferentes métodos que nos ayudan a elegir un número apropiado de Clusters para agrupar los datos; como son, el método del codo (elbow method), el criterio de Calinsky, el Affinity Propagation (AP), el Gap (también con su versión estadística), Dendrogramas, etc. Dada la complejidad de alguno de estos métodos, vamos a explicar aquellos que son más sencillos y que nos dan; para la mayoría de los casos, unos resultados que nos permiten tomar la decisión de cuál será el número optimo de Clusters para el conjunto de datos. Estos métodos serán el método del codo, los Dendrogramas y el Gap.

La implementación de los métodos mencionado se encuentran implementados en el siguiente repositorio:

https://github.com/RicardoMoya/OptimalNumClusters

Para poder ejecutar estos scripts es necesario tener instaladas las librerías de numpy, matplotlib, scipy y scikit-learn. Para descargar e instalar (o actualizar a la última versión con la opción -U) estas librerías; con el sistema de gestión de paquetes *pip*, se deben ejecutar los siguiente comandos:

```
pip install -U numpy
pip install -U matplotlib
pip install -U scipy
pip install -U scikit-learn
```

Método del codo (Elbow Method)

Este método utiliza los valores de la inercia obtenidos tras aplicar el K-means a diferente número de Clusters (desde 1 a N Clusters), siendo la inercia la suma de las distancias al cuadrado de cada objeto del Cluster a su centroide:

$$Inercia = \sum_{i=0}^{N} \|x_i - \mu\|^2$$

Una vez obtenidos los valores de la inercia tras aplicar el K-means de 1 a N Clusters, representamos en una gráfica lineal la inercia respecto del número de Clusters. En esta gráfica se debería de apreciar un cambio brusco en la evolución de la inercia, teniendo la línea representada una forma similar a la de un brazo y su codo. El punto en el que se observa ese cambio brusco en la inercia nos dirá el número óptimo de Clusters a seleccionar para ese data set; o dicho de otra manera: el punto que representaría al codo del brazo será el número óptimo de Clusters para ese data set.

El script que se muestra a continuación, calcula los valores de la inercia tras aplicar el K-means de 1 a 20 Clusters (para uno de los 3 data sets que tenemos) y los pinta en una gráfica lineal (número de Clusters respecto a la inercia) para poder apreciar "el codo" y por tanto determinar el número optimo de Clusters para el data set:

```
# -*- coding: utf-8 -*-
__author__ = 'RicardoMoya'

import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Constant
DATASET1 = "./dataSet/DS_3Clusters_999Points.txt"
```

```
DATASET2 = "./dataSet/DS2_3Clusters_999Points.txt"
DATASET3 = "./dataSet/DS_5Clusters_10000Points.txt"
LOOPS = 20
MAX_ITERATIONS = 10
INITIALIZE_CLUSTERS = 'k-means++'
CONVERGENCE_TOLERANCE = 0.001
NUM_THREADS = 8


def dataset_to_list_points(dir_dataset):
    """
    Read a txt file with a set of points and return a list of objects Point
    :param dir_dataset:
    """
    points = list()
    with open(dir_dataset, 'rt') as reader:
        for point in reader:
            points.append(np.asarray(map(float, point.split("::"))))
    return points


def plot_results(inertials):
    x, y = zip(*[inertia for inertia in inertials])
    plt.plot(x, y, 'ro-', markersize=8, lw=2)
    plt.grid(True)
    plt.xlabel('Num Clusters')
    plt.ylabel('Inertia')
    plt.show()


def select_clusters(dataset, loops, max_iterations, init_cluster, tolerance,
                    num_threads):
    # Read data set
    points = dataset_to_list_points(dataset)

    inertia_clusters = list()

    for i in range(1, loops + 1, 1):
        # Object KMeans
        kmeans = KMeans(n_clusters=i, max_iter=max_iterations,
                        init=init_cluster, tol=tolerance, n_jobs=num_threads)

        # Calculate Kmeans
        kmeans.fit(points)

        # Obtain inertia
        inertia_clusters.append([i, kmeans.inertia_])

    plot_results(inertia_clusters)


if __name__ == '__main__':
    select_clusters(DATASET1, LOOPS, MAX_ITERATIONS, INITIALIZE_CLUSTERS,
                    CONVERGENCE_TOLERANCE, NUM_THREADS)
```

A continuación se muestran los resultados obtenidos para cada uno de los tres data sets. El script solo devuelve la gráfica lineal. La representación de los Clusters que se muestra al lado de la gráfica lineal se ha obtenido con el scripts que implementa el EM y se muestran para poder apreciar que el número de Clusters que nos indica el método del codo es coherente:

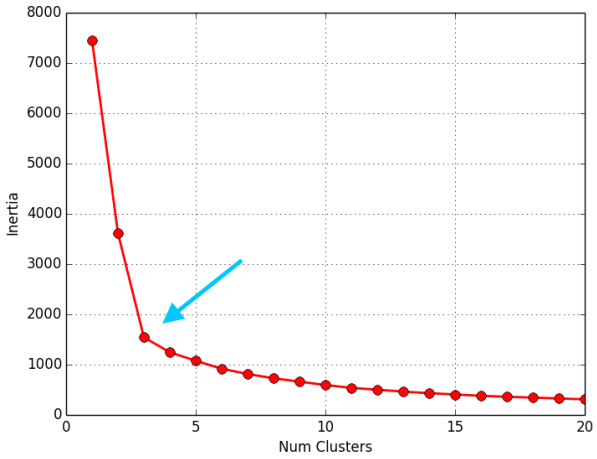

Ilustración 42: Numero de Clusters vs Inercia. Data set 1.

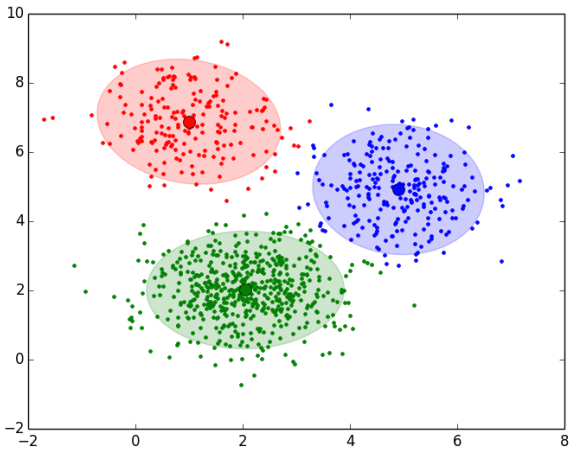

Ilustración 43: Resultado del data set 1 para 3 Clusters.

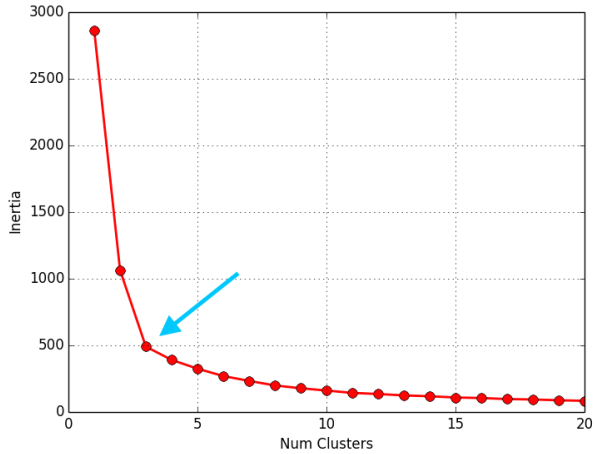

Ilustración 44: Numero de Clusters vs Inercia. Data set 2.

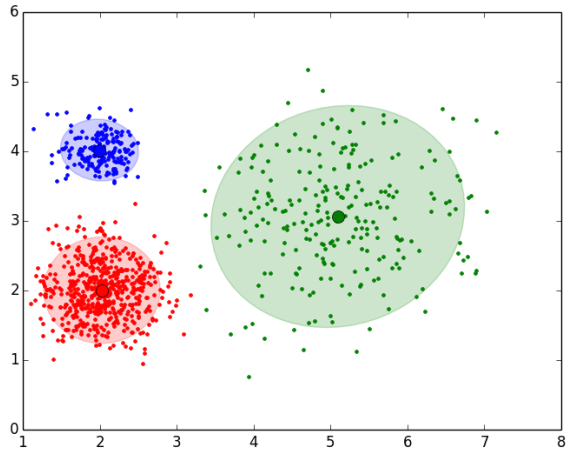

Ilustración 45: Resultado del data set 2 para 3 Clusters.

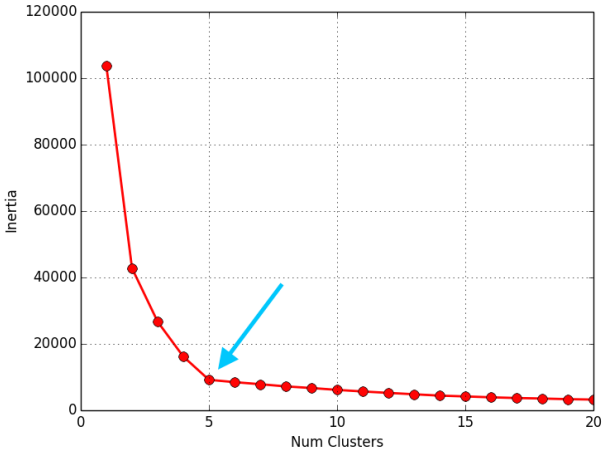

Ilustración 46: Numero de Clusters vs Inercia. Data set 3.

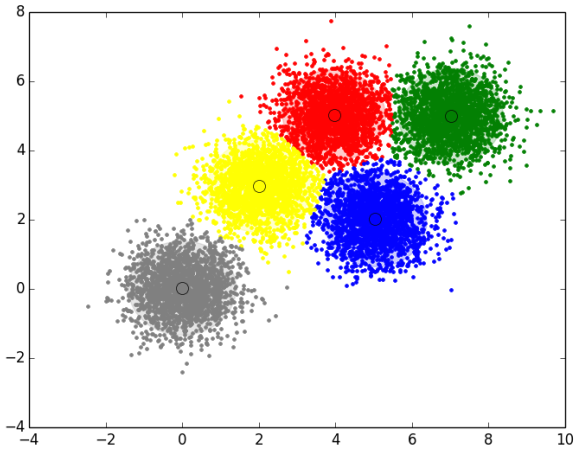

Ilustración 47: Resultado del data set 3 para 5 Clusters.

Como se puede apreciar en los resultados obtenidos, el método del codo devuelve unos valores muy coherentes y se ajusta a los resultados esperados. Es posible que al aplicar este método para otro conjunto de datos no se aprecie "el codo" o incluso se observen dos o más codos (o cambios bruscos en la evolución de la inercia). En ese caso

habría que estudiar más en detalle o con otras técnicas el número optimo de Clusters a seleccionar. Dada la finalidad didáctica de estos ejemplos, se aprecia muy bien "el codo" en la evolución de la inercia, pero en la realidad no siempre se observa este comportamiento tan claro.

Dendrogramas

Un dendrograma es un tipo de representación gráfica en forma de árbol que organiza y agrupa los datos en subcategorías según su similitud; dada por alguna medida de distancia. Los objetos similares se representan en el dendrograma por medio de un enlace cuya posición está determinada por el nivel de similitud entre los objetos o grupos de objetos. Dadas estas características, hace que los dendrogramas sean un tipo de diagrama muy útil para estudiar las agrupaciones de objetos; es decir, para estudiar los Clusters que pueden darse en un data set.

Veamos a continuación un sencillo ejemplo, en el que tenemos 9 objetos representados en un plano:

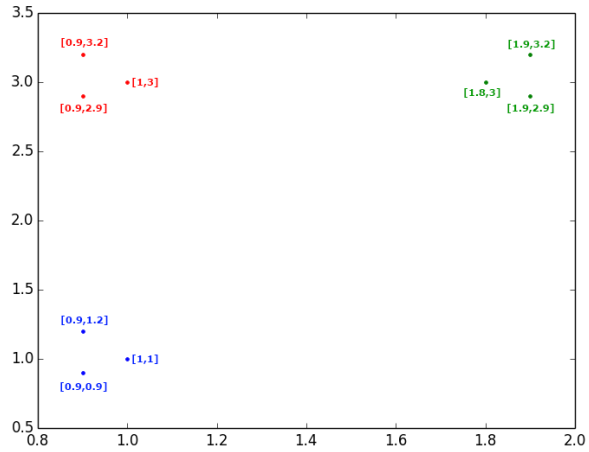

Ilustración 48: Ejemplo de formación de un dendrograma I

Se puede observar claramente que estos 9 objetos los podemos agrupar en 3 Clusters, pero comencemos estudiando las similitudes y distancias entre objetos. En primer lugar fijémonos en los objetos de color azul y veamos las distancias (euclideas en este caso) entre esos objetos. Vemos que los dos puntos más cercanos entre sí son el punto {1,1} y el punto {0.9,0.9}, por lo que los uniríamos en el dendrograma con un enlace. Si agrupamos estos dos puntos y calculamos su centroide, obtenemos el punto {0.95,0.95} y este nuevo punto tiene como punto más cercano de entre todos los que hay en el plano al punto {0.9,1.2}; por tanto, uniríamos con otro enlace al primer grupo formado, con el punto {0.9,1.2}. En resumen lo que se hace es calcular la distancia entre todos los puntos,

cogemos la menor distancia, agrupamos esos dos puntos, y volvemos a calcular la distancia entre todos los puntos o entre todos los grupos ya formados y puntos.

Siguiendo estos pasos, el dendrograma resultante para estudiar la relación entre los 9 puntos antes mostrados sería el siguiente, dando lugar a interpretar que el número optimo de Clusters a seleccionar serian 3:

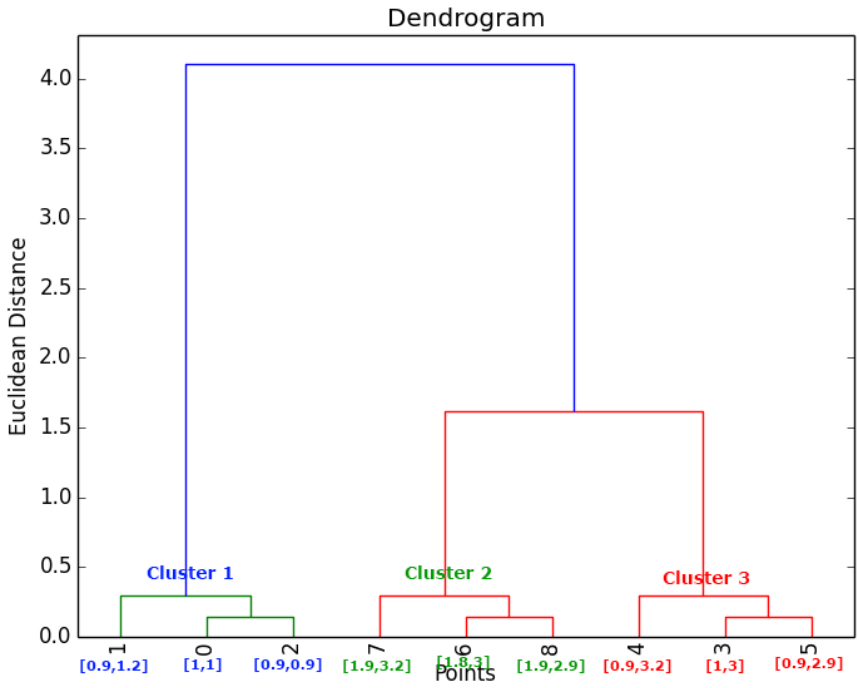

Ilustración 49: Ejemplo de formación de un dendrograma II.

El script que se muestra a continuación calcula con el método *linkage()* las relaciones entre los puntos y grupos más similares, representando estos resultado en un dendrograma al cual hemos limitado el nivel de detalle de las agrupaciones hasta que muestre en su nivel más bajo 12 grupos de objetos:

```
# -*- coding: utf-8 -*-
__author__ = 'RicardoMoya'

import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage

# Constant
DATASET1 = "./dataSet/DS_3Clusters_999Points.txt"
DATASET2 = "./dataSet/DS2_3Clusters_999Points.txt"
DATASET3 = "./dataSet/DS_5Clusters_10000Points.txt"


def dataset_to_list_points(dir_dataset):
    """
    Read a txt file with a set of points and return a list of objects Point
    :param dir_dataset:
    """
```

```
    points = list()
    with open(dir_dataset, 'rt') as reader:
        for point in reader:
            points.append(np.asarray(map(float, point.split("::"))))
    return points


def plot_dendrogram(dataset):
    points = dataset_to_list_points(dataset)

    # Calculate distances between points or groups of points
    Z = linkage(points, metric='euclidean', method='ward')

    plt.title('Dendrogram')
    plt.xlabel('Points')
    plt.ylabel('Euclidean Distance')

    # Generate Dendrogram
    dendrogram(
        Z,
        truncate_mode='lastp',
        p=12,
        leaf_rotation=90.,
        leaf_font_size=12.,
        show_contracted=True
    )

    plt.show()


if __name__ == '__main__':
    plot_dendrogram(DATASET1)
```

A continuación se muestran los resultados obtenidos para cada uno de los tres data sets. El script solo devuelve el dendrograma. La representación de los Clusters que se muestra al lado del dendrograma se ha obtenido con el scripts que implementa el EM y se muestran para poder apreciar que el número de Clusters que nos indica el dendrograma.

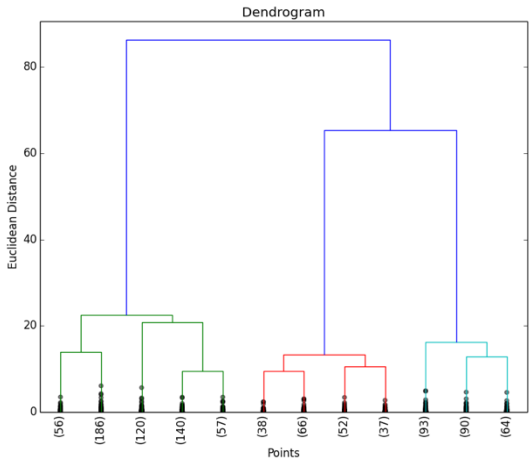

Ilustración 50: Dendrograma resultante para el data set 1.

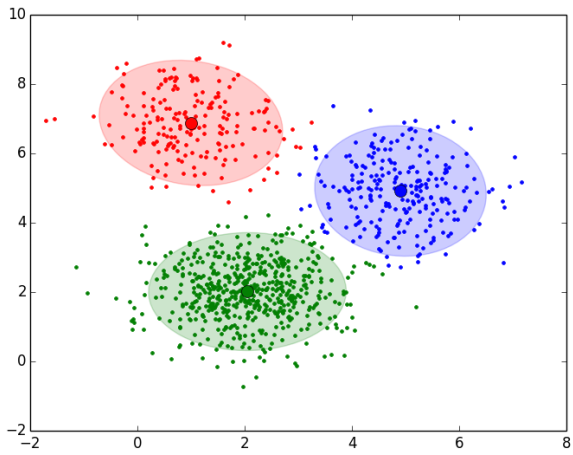

Ilustración 51: Resultado del data set 1 para 3 Clusters.

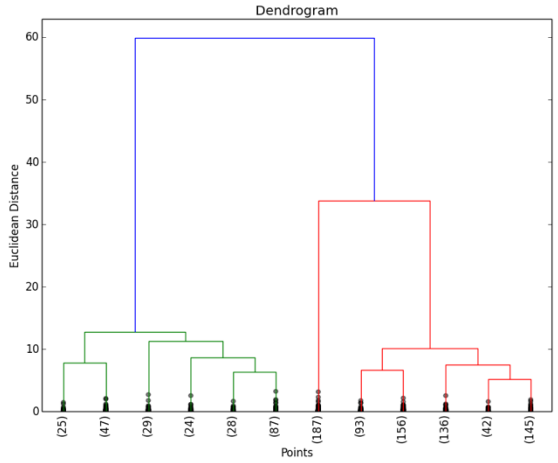

Ilustración 52: Dendrograma resultante para el data set 2.

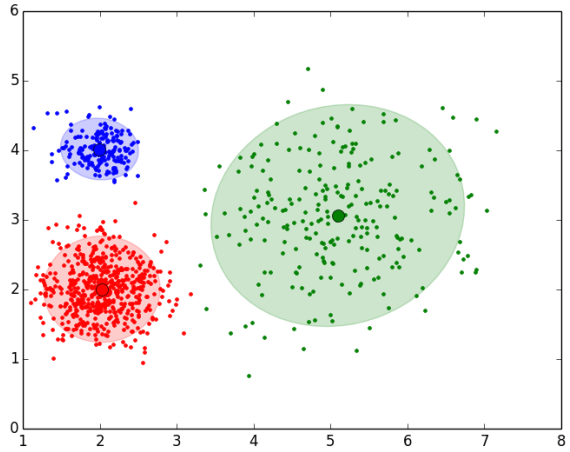

Ilustración 53: Resultado del data set 2 para 3 Clusters.

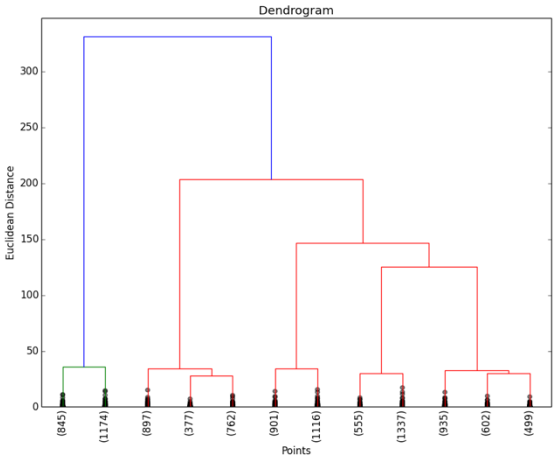

Ilustración 54: Dendrograma resultante para el data set 3.

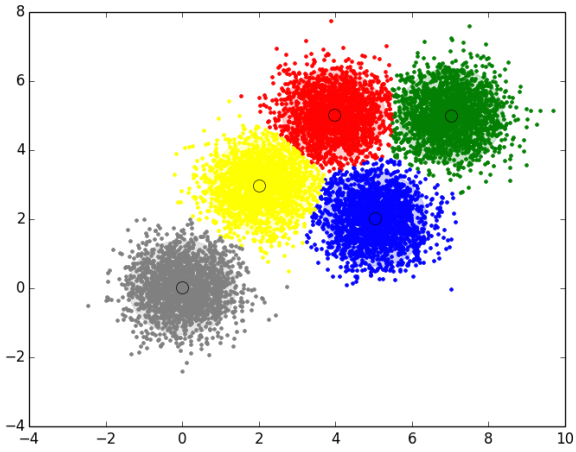

Ilustración 55: Resultado del data set 3 para 5 Clusters.

Como puede observarse en los resultados obtenidos, los dendrogramas muestran las agrupaciones (Clusters) de objetos esperados. Al igual que en la aplicación del método del codo, es posible que no se puedan apreciar claramente las agrupaciones de objetos, por lo que habría que estudiar con otras técnicas el número óptimo de Clusters a seleccionar.

Gap

El último de los métodos que vamos a mostrar es el Gap (brecha); similar al método del codo, cuya finalidad es la de encontrar la mayor diferencia o distancia que hay entre los diferentes grupos de objetos que vamos formando para representarlos en un dendrograma. Para ello vamos cogiendo las distancias que hay de cada uno de los enlaces que forman el dendrograma y vemos cual es la mayor diferencia que hay entre cada uno de estos enlaces.

El script que se muestra a continuación calcula estas diferencias (para alguno de los 3 data sets de los que disponemos) y las representa en una gráfica lineal, siendo el punto máximo, el número de Clusters optimo para ese data set.

```
# -*- coding: utf-8 -*-
__author__ = 'RicardoMoya'

import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage

# Constant
DATASET1 = "./dataSet/DS_3Clusters_999Points.txt"
DATASET2 = "./dataSet/DS2_3Clusters_999Points.txt"
DATASET3 = "./dataSet/DS_5Clusters_10000Points.txt"


def dataset_to_list_points(dir_dataset):
    """
    Read a txt file with a set of points and return a list of objects Point
    :param dir_dataset:
    """
    points = list()
    with open(dir_dataset, 'rt') as reader:
        for point in reader:
            points.append(np.asarray(map(float, point.split("::"))))
    return points


def plot_gap(dataset):
    points = dataset_to_list_points(dataset)

    # Calculate distances between points or groups of points
    Z = linkage(points, metric='euclidean', method='ward')

    # Obtain the last 10 distances between points
    last = Z[-10:, 2]
    num_clustres = np.arange(1, len(last) + 1)

    # Calculate Gap
    gap = np.diff(last, n=2)  # second derivative
    plt.plot(num_clustres[:-2] + 1, gap[::-1], 'ro-', markersize=8, lw=2)
    plt.show()


if __name__ == '__main__':
    plot_gap(DATASET1)
```

A continuación se muestran los resultados obtenidos para cada uno de los tres data sets. El script solo devuelve la gráfica lineal. La representación de los Clusters que se muestra al lado de la gráfica lineal se ha obtenido con el scripts que implementa el EM y se muestran para poder apreciar que el número de Clusters que nos indica la gráfica.

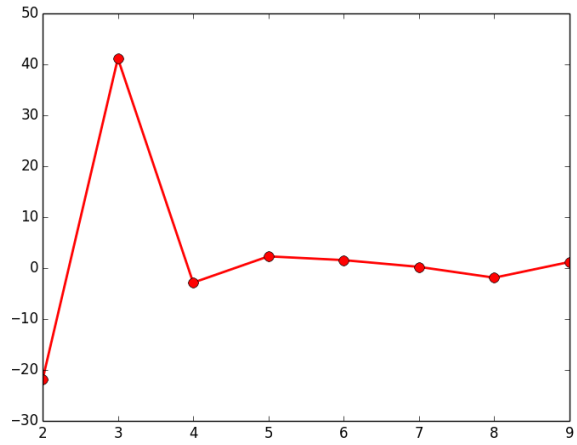

Ilustración 56: GAP resultante para el data set 1.

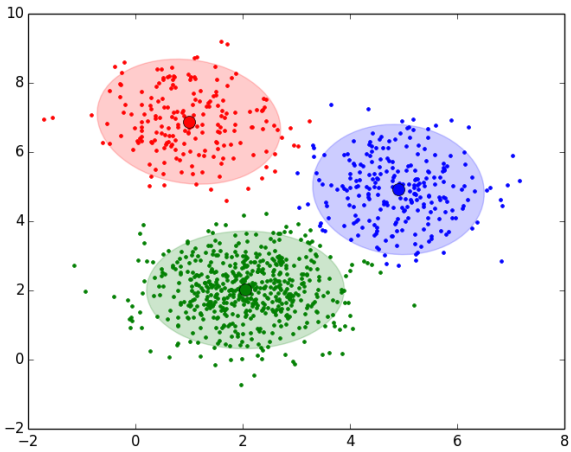

Ilustración 57: Resultado del data set 1 para 3 Clusters.

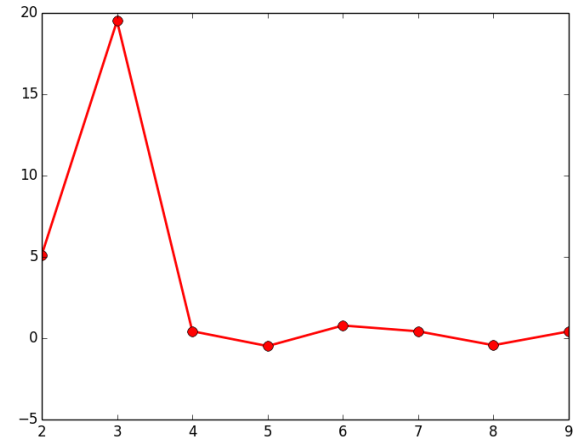

Ilustración 58: GAP resultante para el data set 2.

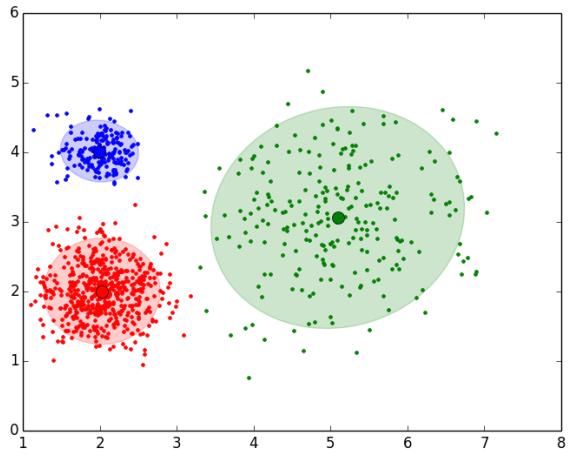

Ilustración 59: Resultado del data set 2 para 3 Clusters.

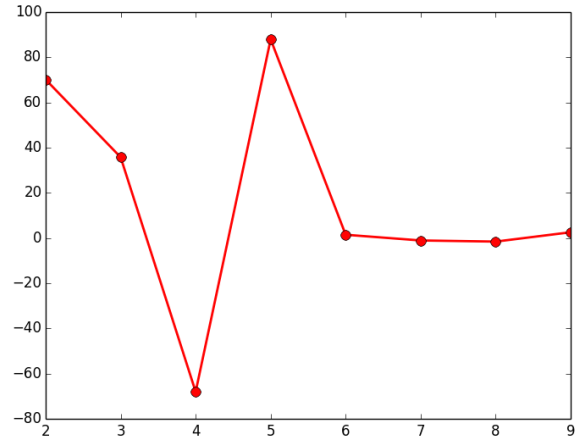

Ilustración 60: GAP resultante para el data set 3.

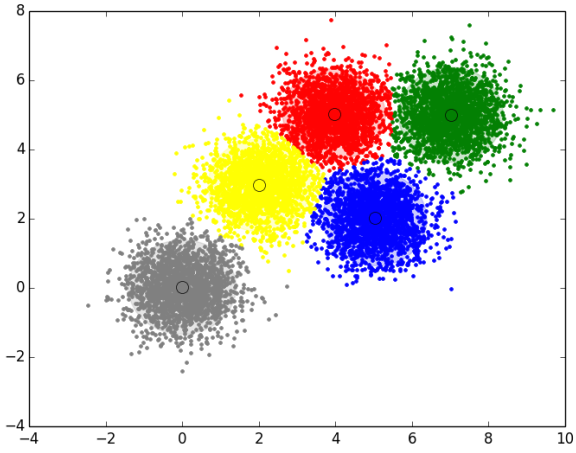

Ilustración 61: Resultado del data set 3 para 5 Clusters.

Como puede observarse en los resultados obtenidos estudiando el gap, devuelve el número de agrupaciones (Clusters) esperados. Al igual que en el resto de métodos mostrados anteriormente, es posible que no se puedan apreciar claramente las

agrupaciones de objetos, por lo que habría que estudiar con otras técnicas el número óptimo de Clusters a seleccionar.

Bibliografía

- Borman.S. The Expectation Maximization Algorithm A short tutorial. *2009.*

- Cluster analysis in R: determine the optimal number of Clusters, http://stackoverflow.com/questions/15376075/cluster-analysis-in-r-determine-the-optimal-number-of-clusters, *Octubre 2014*

- Do.C y Batzoglou.S. What is the expectation maximization algorithm?. *Nature Biotechnology, 2008.*

- Wikipedia Inglesa, https://en.wikipedia.org/wiki/Machine_learning, *Enero 2016*

- Scikit-learn web, http://scikit-learn.org/stable/user_guide.html, *Enero 2016*

Glosario de términos

A

Aprendizaje adaptativo, 10

Aprendizaje Autónomo, 4

Aprendizaje no supervisado, 10

Aprendizaje on-line, 10

Aprendizaje por refuerzo, 10

Aprendizaje semi-supervisado, 10

Aprendizaje supervisado, 10

C

Clasificación, 5, 6

Cluster, *13*

Clustering, *13*

Cross Validation, 11

D

Data set, 4

Dendrogramas, 63

E

Elbow Method, 60

Error empírico, 6

Exclusión individual, 11

Expectation-maximization, 35

G

Gap, 66

H

Hipótesis, 6

Hold Out, 11

I

Inercia, 15

Inteligencia Artificial, 4

K

K-means, 15

L

Leaving one out, 11

M

Machine Learning, 4

Método del codo, 60

N

NP-completo, 4

O

Overfitting, 7

P

Partición, 11

Plate Notation, 37

R

Regresión, 5, 6

Resustitución, 11

S

Sistema experto, 5

U

Underfitting, 7

V

Validación cruzada, 11

variables latentes, 36