Compte rendu : Clustering

Damien de Montis

Le but du devoir est, dans un premier temps de générer un nuage de points via des blobs respectant une loi normale, puis d'y appliquer l'algorithme de K-means afin d'y retrouver différents clusters.

Organisation /

Pour ce faire, j'ai organisé mon projet en différentes classes :

- Point : elle regroupe touts les caractéristiques d'un point. Au delà de ses coordonnées, j'ai choisi d'y ajouter le numéro du cluster auquel il appartient. A part les diverses méthode de constructeurs, affichages ou getters, j'ai choisi d'ajouter la méthode *distance(Point p)* afin d'évaluer le distance entre le point sur lequel s'applique la méthode et le point p.

- Blob : cette classe se caractérise par un vecteur de points et un point centre. Le but de cette classe est de générer un blob de point suivant diverses caractéristiques. En effet on peut par exemple retrouver en paramètres, les bornes du blob. D'autre part, la fonction générant ce blob *make_blob_normal(double ecart, int nb_point)* suit une loi normale afin de générer ces points.

- Kmeans : c'est la classe qui décrit l'algorithme de K-means. Elle est composée d'un vecteur *pts* qui comporte les points sur lesquels s'exécute l'algorithme et un vecteur *centroid* qui comporte les centres, utilisés dans l'algorithme de K-means.

En plus de ces classe j'ai un fichier *Tools.cpp* qui me sert à écrire des fonctions qui ne sont propres à aucune classes. Ici , il y a 2 fonctions de génération de nombre aléatoire, une linéaire et l'autre normalisée.

Pour gérer la compilation, j'ai choisi Cmake tout les fichiers de build et autre makefile sont générés dans le dossier *build.*

Réalisation /

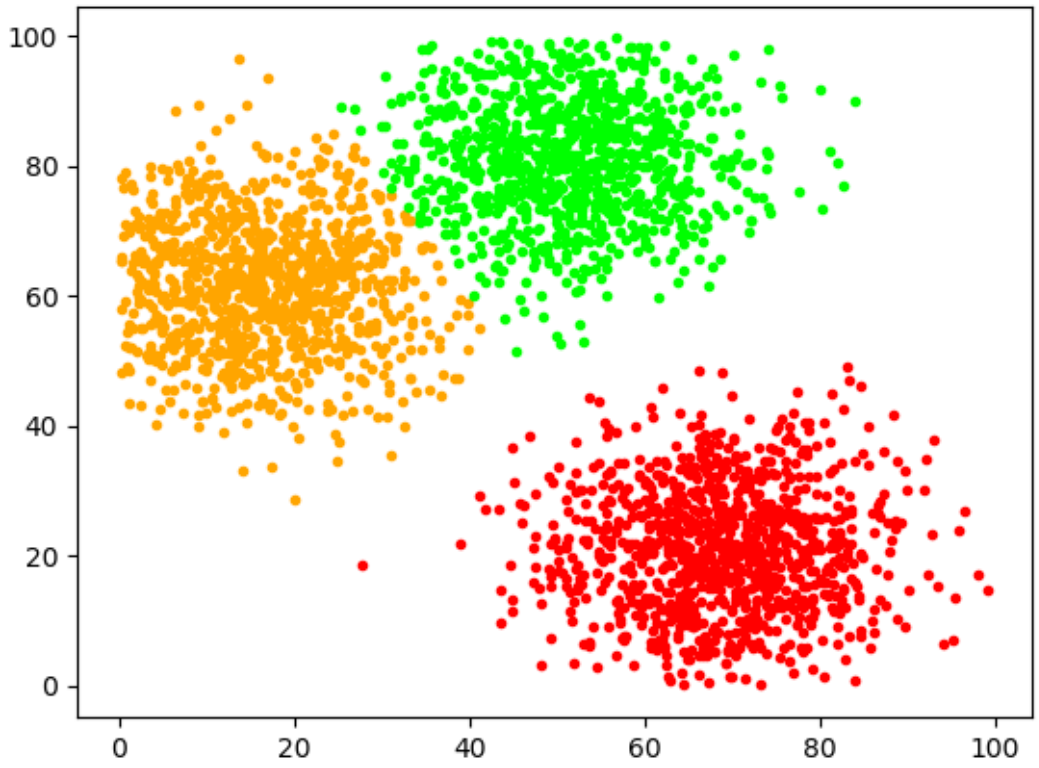
Le premier problème du devoir est de générer des blobs suivant une loi normale. C'est la fonction *void make_blob_normal(double ecart, double nb_point)* qui s'en charge. Celle-ci crée *nb_point* en générant aléatoirement un couple (x,y) via la fonction *rng_normal(double center, double sigma)*. Cette fonction, écrite dans *Tools.cpp*, utilise le générateur aléatoire de la bibliothèque *random*, afin de générer un nombre aléatoire suivant la loi normale centrée sur *center* et d'écart type *sigma*. Dans notre cas on utilisera les coordonnés du centre du blob comme centre et *ecart* donné en argument comme écart type. Une fois un couple trouvé, il suffi de construire le *Point* avec celui-ci, et de l'ajouter au vecteur *pts*. Le reste de la génération du jeu de donnée se fait grossièrement dans le main. On trouve ici un premier axe d'amélioration.

Dans un second temps il est demander d'implémenter l'algorithme de K-means, et de l'executer sur le jeu de données précédemment créé. Cet algorithme consiste à déplacer des centres. J'ai donc choisi de créer un vecteur *centroid* contenant des *Point* qui seront ces centres. Ensuite, cet algorithme nécessite 2 sous-parties. D'abord le clustering. Ceci consiste à simplement évaluer le centre le plus proche de chaque point. Cette fonctionnalité est implémentée dans la méthode *clustering()* dans la classe *Kmeans*. Ensuite, il faut savoir recentrer les centres. C'est la fonction *center_centroids()* dans la classe *Kmeans* qui s'en charge. Celle-ci évalue simplement la moyenne des coordonnées pour chaque cluster, et déplace le centre selon cette moyenne. Ensuite la fonction mère *run(int nb_centers)* dans la classe *Kmeans exécute* l'algorithme. Celle-ci initialise d'abord *nb_centers* centres. Ensuite, il execute *clustering()* et *center_centroids()* en boucle jusqu'à ce que la différence entre les centre de l'itération précédente et les nouveaux centre soit infime (e-4 ici).

Afin d'afficher les résultats j'utilise *matplotlibcpp.* L'affichage se fait grâce à la fonction *draw_pts_vector()* dans la classe *Kmeans.*

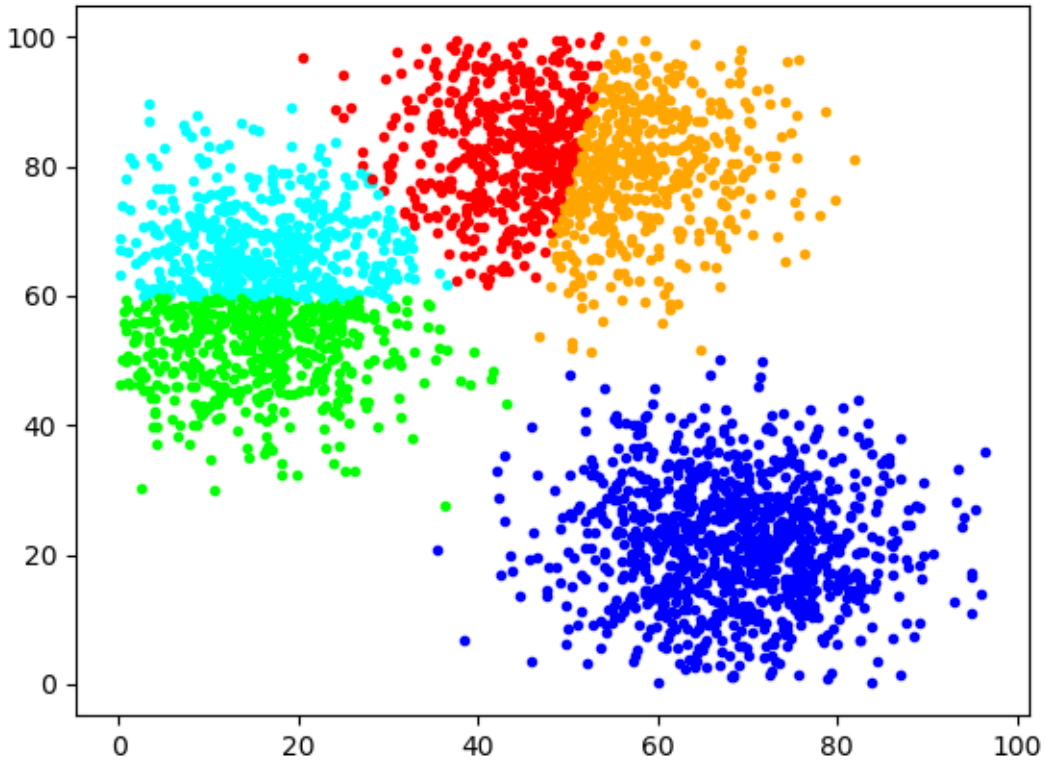
Résultats /

figure 1 :



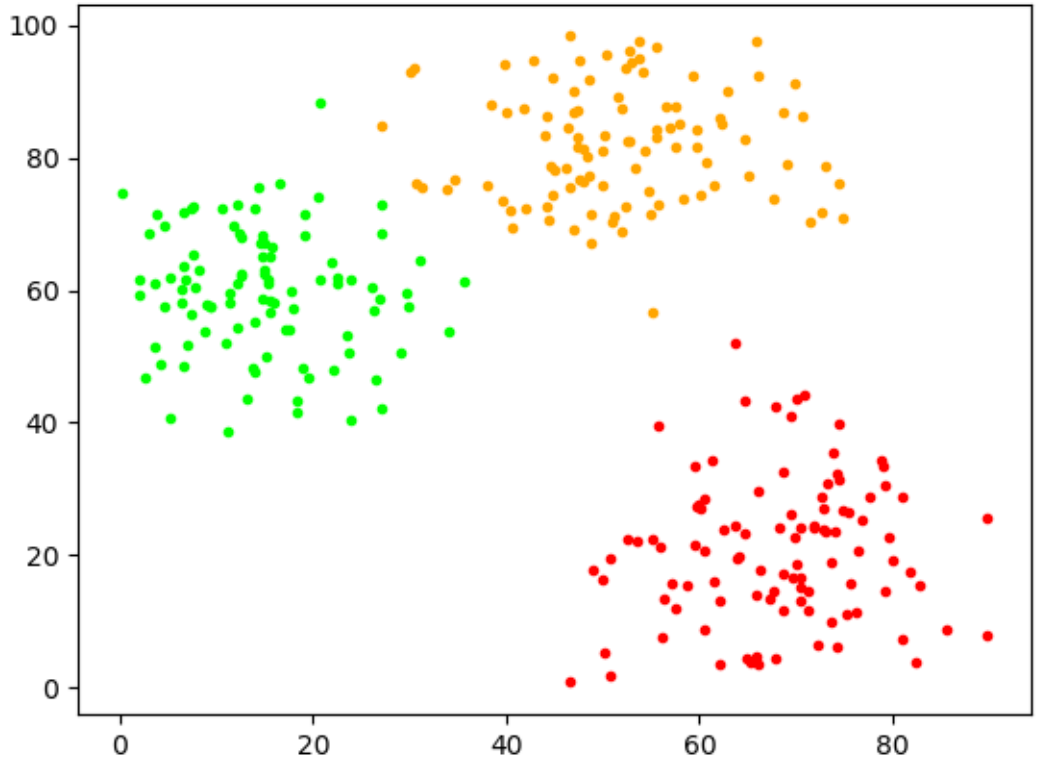
Ci-dessus, on trouve le résultat graphique pour l'exécution de l'algorithme sur un vecteur de points composé de 3 blobs de 1000 points chacun avec 3 centre. On distingue aisément les différents blobs. On a ici le nombre de blobs et de centres identiques. Dans ce cas le temps d'exécution de Kmean est de l'ordre de 10^8 ns.

Figure 2 :



Ici on a les même paramètres que précédemment. On a fait varier le nombre de centres à 5. On remarque 2 blobs coupés en 2. Dans ce cas le temps d'exécution de Kmean est de l'ordre de 10^8 ns.

Figure 3 :



Ici on a fait varier le nombre de points par rapport à la figure 1. On a 100 points par blob. Dans ce cas le temps d'exécution de Kmean est de l'ordre de 10^5 ns.

On remarque que le temps d'exécution varie en fonction du nombre de points traités plutôt qu'en fonction du nombre de centres. En effet, on a beaucoup plus de points à traiter que de centres à gérer. Ce sont donc les boucle parcourant les points qui consomment du temps. Après diverses tests on remarque que la croissance du temps d'exécution par rapport au nombre de points traité est exponentiel.

Problèmes et Axes d’amélioration /

Implémentation :

Tel qu’il est mon programme est fonctionnel mais n’est pas suffisamment générique. En effet, il est difficile d’implémenter un autre algorithme de clustering dans mon programme, étant donné mon architecture. Par ailleurs l’encapsulation n’est pas toujours respectée, dans la classe *Blob* le vecteur de points est publique, ce qui rend le code moins robuste.

Performances :

J’ai ici un gros problème de performance du à l’affichage graphique. J’affiche, le vecteur point par point en créant un vecteur pour chaque point. Ce qui ralentit fortement le temps avant l’affichage. Le majeur problème étant que, pour les exemples précédents. Si chaque blob contient 10^5 points, K-means converge, mais pas l’affichage graphique. J’en déduit que pour un jeu de donné de l’ordre de 10^{15}, l’affichage graphique est impossible ici.