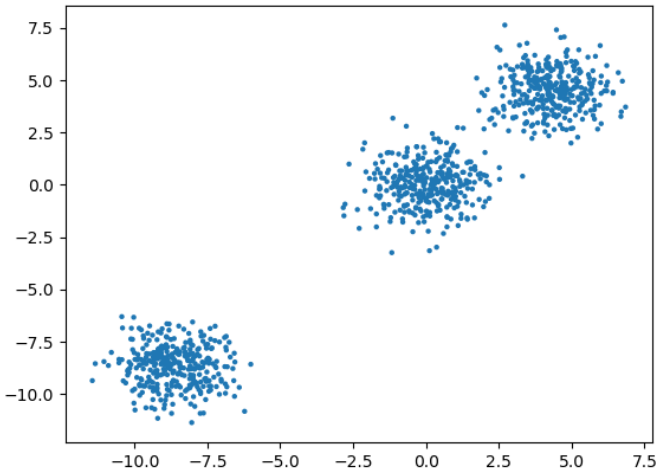
Clustering en `C++`

Prérequis : Pour mener à bien ce travail, il est nécessaire d'avoir installé les outils classiques pour le `C++` (compilateur, `CMake`, etc.) ainsi que la bibliothèque `matplotlib-cpp` permettant l'affichage graphique (via `Python`).

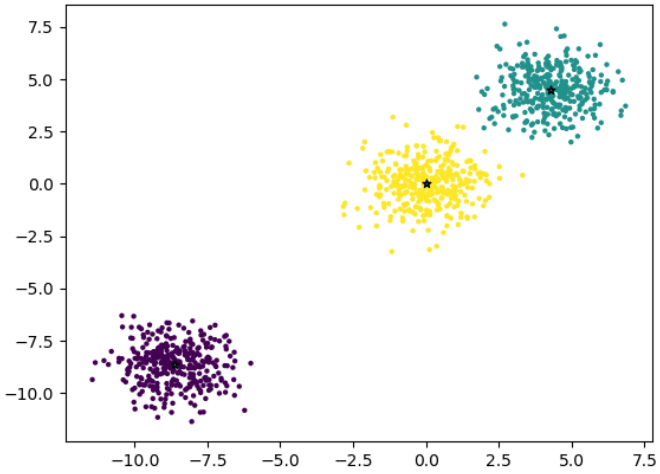
Contexte du travail : On s'intéresse ici à l'implémentation en `C++` de la méthode `k-means`, très populaire en *Machine Learning* pour le partitionnement (ou *clustering*) de données. C'est une méthode simple et très performante. Pour une présentation générale de la méthode, voir par exemple :

- `https://en.wikipedia.org/wiki/K-means_clustering`
- `https://stanford.edu/ cpiech/cs221/handouts/kmeans.html`
- `https://datasciencelab.wordpress.com/tag/lloyds-algorithm`

Pour illustrer, considérons l'ensemble de N points (ou observations) $X = \{x_0, x_1, \ldots, x_n\}$ avec $x_i \in \mathbb{R}^2$ suivant :



L'idée est de determiner K partitions C_k disjointes des données, c'est-à-dire attribuer une *couleur*, typiquement un entier, à chacun des points. Ici, il semble naturel de considérer 3 couleurs pour chaque groupe de données. Un clustering de données valide serait par exemple le suivant :



où les centres μ_k des clusters sont notés par symbole étoile.

L'algorithme `k-means` est une méthode itérative que l'on peut écrire en pseudo code de la manière suivante :

1. Initialisation des centres μ_k
2. Pour tout $i = 1, \ldots, M$:
   (a) Calcul de la distance de chaque point x_i avec les centres μ_k
   (b) Repartition des points x_i dans les clusters par proximité
   (c) Mise des centres μ_k

Travail demandé : Le but de ce travail est d'implémenter en `C++` l'algorithme `k-means` en respectant les consignes suivantes :

- L'implémentation devra utiliser au maximum la librairie standard `stl` et les possibilités du langage offertes par les normes `C++11/14/17`.
- Pour l'affichage des résultats, il conviendra de s'appuyer sur la librairie `matplotlib-cpp`.
- Les données devront être générées de manières aléatoires.
- Le temps d'execution de la méthode devra être mesuré pour le clustering de petits et grands jeux de données.

Quelques piste utiles : Voir `https://en.cppreference.com/w/cpp` pour la documentation officielle du `C++`.

- Ne pas hésiter à utiliser et abuser du mot-clé `auto`.
- En utilisant `<random>`, il est possible de générer des nombres suivant les distributions classiques.
- Pour la génération des données, on peut s'inspirer de la fonction `make_blobs` de la librairie `Python scikit-learn`. Par exemple, vous pouvez choisir aléatoirement K centres puis générer N/K points aléatoirement avec une distribution normale.
- Pour définir la proximité entre 2 points, on peut utiliser la distance euclidienne.

Liens utiles : Voici quelques ressources qui devraient vous inspirer :

- `https://en.cppreference.com/w/cpp/language/templates`
- `https://en.cppreference.com/w/cpp/container/vector`
- `https://en.cppreference.com/w/cpp/utility/tuple`
- `https://en.cppreference.com/w/cpp/language/auto`
- `https://en.cppreference.com/w/cpp/language/structured_binding`
- `https://en.cppreference.com/w/cpp/language/range-for`
- `https://en.cppreference.com/w/cpp/numeric/random`
- `https://en.cppreference.com/w/cpp/chrono`
- `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html`

Bonus : Pour les plus avancés, réfléchissez à la notion de `zip` de séquences pour améliorer l'ergonomie de votre code :

- `https://www.w3schools.com/python/ref_func_zip.asp`
- `https://stackoverflow.com/questions/8511035/sequence-zip-function-for-c11`

Par exemple, on souhaite pouvoir écrire le code suivant :

```
auto X = std::vector<int>{1, 2};
auto Y = std::vector<double>{1., 2.};
for(auto [x,y] : zip(X,Y) {
    std::cout << x << "," << y << std::endl;
}
```