Read me - Programme kPLM

Claire Brécheteau

25 juillet 2021

Résumé

La k-PLM est une méthode de type partitionnement de données. On dispose d'un nuage de points dans $\mathbb{R}^d$. Notre but est de donner à chaque point une étiquette. Les points de même étiquette sont considérés comme appartenant à un même groupe.

Dans notre cas, notre méthode est du même type que l'algorithme des k-means. C'est-à-dire qu'il y a k groupes (donc k étiquettes différentes : de 1 à k). À chaque groupe est associé un centre. L'étiquette de chaque point correspond au groupe dont le centre est le plus proche du point, pour une certaine distance. Les centres sont régulièrement mis à jour, en fonction des points présents dans le groupe associé.

1 Les fichiers dans include

Le fichier `kPLM_algo.h` est le fichier appelé par le fichier principal `main.cpp`, il fait lui-même appel aux 4 fichiers d'extension `.h`, contenant les classes, situés dans le dossier `kPLM`.

Pour compiler le fichier `main.cpp` :

```
g++ main.cpp -o ./a.out -I ../../../../../../../../../usr/include/eigen3/
```

Pour exécuter le fichier `a.out` :

```
./a.out ../data/data_symb.csv
```

L'affichage est le suivant :

```
The parameters of the algorithm are :
method = 0
k = 50
q = 20
n_signal = 1000
n_starts = 10
epochs = 20
d_intrinsic = 2
sigma2_min = 0.01
sigma2_max = 0.01
lambda_min = 0.01
lambda_max = 100
normalized_det = 0
The number of points for the algorithm is :1500

The first five points of the algorithm are :
1.51817, -1.20943
0.748019, -0.645651
-2.91832, -0.950895
1.64785, 1.11451
2.16077, 0.829922

Algorithm with non increasing intrinsic dimension :
6 5 4 3 2 1 0 6 5 4 3 2 1 0 6 5 4 3 2 1 0 6 5 4 3 2 1 0 6 5 4 3 2 1 0 6
5 4 3 2 1 0 6 5 4 3 2 1 0 6 5 4 3 2 1 0
-6237.67
-6278.32
-6280.87
-6280.48
-6280.48
-6280.48
-6280.48
```

1. Les paramètres affichés sont écrits dans le fichier `main.cpp`, mais il est tout à fait possible de modifier le fichier de sorte à ce qu'ils ne soient pas connus à la compilation, et donnés par l'utilisateur à l'exécution.

   ATTENTION : la dimension des données d doit être connue à la compilation !
2. Les deux premières coordonnées des cinq premiers points des données sont ensuite affichées
3. L'algorithme kPLM est lancé, pour une dimension intrinsèque variant entre 6 et 0...colorred Bizarre vu que nos données sont de dimension 2 ! Revoir

Les fonctions font appel à la librairie `Eigen`.

1.1 Point.h

Un élément de classe Point est un point du nuage de points à partitionner.

Il contient : les coordonnées du point, son étiquette, la distance au centre le plus proche, et l'étiquette et la distance au centre optimal pour la meilleure tentative, car l'algorithme de partitionnement est potentiellement répété plusieurs fois, on garde la tentative pour laquelle la somme des coûts sur tous les points du nuage est minimale.

Attention, les attributs privés sont notés publics pour l'instant, à modifier, et modifier le reste en conséquence

1.1.1 Classes amies

1. `Centroid<d>;`
2. `kPLM<d>;`

1.1.2 Attributs privés

1. `Matrix<double,1,d> X;` Coordonnées du point
2. `size_t cluster;` Numéro de la cellule (ou du centre) à laquelle (auquel) ce point est associé. Initialement `cluster = 0`. Sa valeur est modifiée dans les méthodes des classes amies.
3. `double minDist;` Distance au centre le plus proche pour le coût souhaité *(dans le cas de la k-PLM : norme de mahalanobis au carré entre le point et le paramètre* `m` *(du centre) (pour la matrice l'inverse de* `Sigma_inv` *du centre) +* `v` *(du centre))*. Infini par défaut.
4. `size_t opt_cluster;` Numéro de la cellule pour la meilleure tentative : celle pour laquelle la somme des coûts `minDist` de tous les échantillons est la plus faible.
5. `double opt_minDist;` Coût pour la meilleure tentative.

1.1.3 Constructeurs

Les arguments des constructeurs sont : vide, `const double (&tab)[d]` ou `const Matrix<double,1,d> & tab`.

— Les attributs `cluster` et `opt_cluster` sont mis à 0.
— Les attributs `minDist` et `opt_minDist` sont mis à `__DBL_MAX__`.
— L'attribut `X` est soit la matrice avec que des 0, ou la matrice `tab`, en fonction du constructeur appelé.

1.1.4 Attributs publics

1. `get_X` Pour récupérer les coordonnées `X` du point qui sont privées.

1.1.5 Fonctions amies

Les deux fonctions suivantes ont la même fonction : calculer la moyenne et la matrice de covariance d'un sous-ensemble d'un nuage de points. L'une utilise un vecteur de Points, l'autre une matrice qui contient les coordonnées des points du nuage.

Pour l'instant, je ne sais pas encore s'il est plus rapide d'utiliser l'une ou l'autre, tout dépend de comment sera codée la méthode de partitionnement.

1. `std::pair<Matrix<double,1,d>, Matrix<double,d,d» calcul_mean_cov<d>(const std::vector<Point<d» & points, const std::vector<size_t> & indices);` Calcule la moyenne et la matrice de covariance des points `points` dont l'indice est dans `indices`.
2. `std::pair<Matrix<double,1,d>, Matrix<double,d,d» calcul_mean_cov<d>(const Matrix<double,Dynamic,d> & block_points, const std::vector<size_t> & indices);` Calcule la moyenne et la matrice de covariance du nuage de points `block_points` dont l'indice est dans `indices`.

1.1.6 Autres fonctions

1. `vector<Point<d» readcsv(char* adresse)` lit des points dans un document et créer le vecteur de points correspondant.

1.2 Sigma_inverted.h

Un élément de classe `Sigma_inverted` est l'inverse d'une matrice carrée symétrique réelle positive de taille $d \times d$, où d est la dimension des points.

Dans notre algorithme de partitionnement, on associera à chaque centre, une matrice, donc un élément de type `Sigma_inverted`. Cette matrice sera utilisée pour calculer la distance. On utilisera des normes de Mahalanobis (qui dépendent donc de la matrice) plutôt que des normes Euclidiennes, comme c'est le cas des k-means.

1.2.1 Classes amies

1. `Centroid<d>;`
2. `kPLM<d>;`

1.2.2 Attributs privés

1. `Matrix<double,d,1> Eigenvalues_inverted_initial;` contient les inverses des valeurs propres, les valeurs propres sont en ordre croissant. Ce sont les valeurs propres de la matrice calculée à partir des points dans la cellule et les plus proches voisins du centre pour la distance tordue.
2. `Matrix<double,d,1> Eigenvalues_inverted_to_use;` ce sont les valeurs propres calculées à partir de `Eigenvalues_inverted_initial`, modifiées en fonction de la méthode souhaitée (déterminant constant égal à 1, ou valeurs propres tronquées, ou certaines valeurs propres égales...). Ce sont ces valeurs propres qui sont utilisées dans la suite de l'algorithme.
3. `Matrix<double,d,d> Eigenvectors;` Les vecteurs propres de la matrice, calculés à partir des points dans la cellule et les plus proches voisins du centre pour la distance tordue.

1.2.3 Constructeurs

Il existe un constructeur qui copie les attributs d'un objet de type `Sigma_inverted`. Sinon,

— `Eigenvalues_inverted_initial` et `Eigenvalues_inverted_to_use` sont des vecteurs avec que des 1, ou quand c'est précisé, sont égaux à `Eigenvalues_inverted`.

— `Eigenvectors` vaut la matrice identité, sauf quand `Eigenvectors` est donné.

1.2.4 Attributs publics

1.

1.3 Centroid.h

Les k centres de l'algorithme de partitionnement sont k éléments de type `Centroid`.

1.3.1 Classes amies

1. `kPLM<d>;`

1.3.2 Attributs privés

1. `Matrix<double,1,d> X;` : Coordonnées du centre. (Dans l'algorithme de la k-PLM (similaire à k-means), il s'agit de la moyenne des points de la cellule de Voronoï tordue par le critère) auquel le centre est associé.
2. `std::vector<size_t> Voronoi_points_indices;` Indices des points de la cellule de Voronoï tordue. Important pour mettre à jour `X` et la matrice `Sigma_inv`.
3. `Sigma_inverted<d> Sigma_inv;` Inverse de la matrice associée au centre.
4. `Matrix<double,1,d> m;` Moyenne des `q` plus proches voisins de `X` pour la norme de Mahalanobis associée à la matrice inverse de `Sigma_inv`, dans un nuage de points `block` (mis à jour avec la fonction `update_m_v`).
5. `double v;` Variance (pour la norme de Mahalanobis associée à la matrice inverse de `Sigma_inv`) des `q` plus proches voisins de `X` pour la norme de Mahalanobis associée à la matrice inverse de `Sigma_inv` - log(det(`Sigma_inv`))
6. `bool active;` Indique si la cellule de Voronoï associée au centre est non vide. `true` par défaut. Le centre n'a plus d'intérêt lorsque active vaut `false`.

1.3.3 Constructeurs

1.3.4 Attributs publics

1.

1.4 kPLM.h

Il s'agit de l'algorithme de partitionnement de données.

1.4.1 Attributs privés

1. `size_t method;`
   - `method` = 0 : on ne modifie pas les valeurs propres
   - `method` = 1 : on tronque les valeurs propres
   - `method` = 2 : on tronque les valeurs propres avec les plus petites valeurs égales à une valeur propre donnée.
   - `method` = 3 : on renormalise les matrices pour qu'elles soient de déterminant égal à 1.
2. `std::vector<Point<d>>` points ;
3. `Matrix<double,Dynamic,d>` block ;
4. `std::vector<Centroid<d>>` initial_centroids ;
5. `size_t k;` Nombre de centres initial.
6. `size_t q;` Nombre de plus proches voisins.
7. `size_t n_signal;` Nombre de points du nuage de points considérés comme du signal, les autres sont considérés comme des données aberrantes.
8. `size_t n_starts;` Nombre d'initialisations différentes de l'algorithme, en gros, nombre de fois où on lance l'algorithme.
9. `size_t epochs;` Nombre d'itérations de l'algorithme.
10. `size_t d_intrinsic;` Entier compris entre 0 et d. Pour chacune des k matrices, les $d - d\_intrinsic$ plus petites valeurs propres doivent être égales. (Cette valeur peut différer d'une matrice à l'autre)
11. `double sigma2_min;` Les plus petites valeurs propres sont supérieures à cette valeur.
12. `double sigma2_max;` Les $d - d\_intrinsic$ plus petites valeurs propres sont inférieures à cette valeur.
13. `double lambda_min;` Les $d\_intrinsic$ plus grandes valeurs propres sont supérieures à cette valeur.
14. `double lambda_max;` Les $d\_intrinsic$ plus grandes valeurs propres sont inférieures à cette valeur.
15. `bool normalized_det;` Vaut 1 si le déterminant des matrices doit être égal à 1, 0 si aucune normalisation sur les matrices doit être effectuée.
16. `std::vector<Centroid<d>> optimal_centroids;`
17. `std::vector<size_t> optimal_labels;`
18. `std::vector<double> optimal_costs;`
19. `double optimal_cost;`
20. `bool optimisation_done;`

1.4.2 Constructeurs

1.4.3 Attributs publics

1.

2 Les fichiers dans test

Pour chacun des fichiers .cpp suivants, le fichier exécutable correspondant est du même nom avec l'extension .out, et la sortie a été enregistrée dans le fichier de même nom avec l'extension _output.

Par exemple, pour lancer un des fichier .cpp, pour recréer le fichier .out, on utilise la commande :

```
g++ basic_centroid.cpp -o ./basic_centroid.out -I ../../../../../../../../usr/include/eigen3/
```

Attention, cela nécessite d'avoir installé le package `eigen3` dans le dossier `usr/include` et le nombre de `../` dépend de l'emplacement des fichiers.

Pour avoir la sortie telle que celle enregistrée dans le document d'extension _output, on utilise la commande :

```
./basic_centroid.out ../data/data_dim6.csv
```

2.1 basic_centroid.cpp

2.2 basic_centroid_trunc_eigenvalues.cpp

2.3 compute_PLM_values.cpp

2.4 mahalanobis_methods.cpp

2.5 read_points_compute_mean_cov.cpp

2.6 transform_covariance_matrix_compute_Mahalanobis.cpp

2.7 transform_covariance_matrix_compute_Mahalanobis_nodata.cpp

3 Les fichiers dans doc

Le fichier `intro_kPLM.h` est juste une copie d'un fichier de documentation de code dans Gudhi, pour avoir une idée vague de comment commenter le code. A remplir plus tard.

4 Les fichiers dans examples

Aucun pour l'instant.

5 Les fichiers dans build

Aucun.

6 Les fichiers dans data

Ce sont des fichiers .csv de données simulées.

6.0.1 data.csv

60 points dans $\mathbb{R}^2$:
Trois petits groupes dont deux collés, avec quelques points aberrants.

6.0.2 data_symb.csv

1500 points dans $\mathbb{R}^2$:
Les 1000 premiers sont générés sur le symbole infini, avec du bruit gaussien dans $\mathbb{R}^2$, les 500 autres sont générés uniformément dans un rectangle contenant le symbole infini.

6.0.3 data_dim6.csv

550 points dans $\mathbb{R}^6$:
Les 500 premiers sont générés sur le symbole infini dans $\mathbb{R}^2 \times \{0\}^4$, avec un bruit gaussien dans $\mathbb{R}^6$, les 50 autres sont générés uniformément dans un hypercube contenant le symbole infini.