Travail Pratique

Algorithemes pour l'optimisation et l'analyse des mégadonnées

Math-60607

Équipe 8

Rapport

Algorithme de Kmeans

Par	*Numéro d'identification*
Mbolatina Randriamifidy	11235474
Julien Deslongchamps	11250404
Mustapha Bouhsen	11321500

Travail présenté à
Monsieur
Gilles Caporossi

2 décembre 2022

HEC MONTRÉAL

Table des Matières

Introduction **2**

Génération des données **3**

La fonction Kmeans **4**
Étapes préliminaires . . . 4
Échelle des données/Standardisation des variables . . . 4
La distance . . . 4
Les centroides et les classes . . . 4
Complexité de l'algorithme K-means . . . 5
Parallélisation . . . 5
Application de K-means . . . 7
Amélioration de K-means . . . 7
Test K-means amélioré . . . 8

Processus VNS adapté à Kmeans **9**
Execution de k-means pour tester une selection de nombres de clusters . . . 9
Execution de 100 iterations de VNS . . . 10
Visualisation des clusters issus de VNS . . . 11
Visualisation des clusters pour chacune des variables . . . 11
Visualisation des positions des centroides pour chaque variable . . . 11
Evolution du silhouette_score à chaque itération VNS . . . 14
Temps d'execution à chaque itération VNS: . . . 14

Conclusion **15**
Limites rencontrées . . . 15
Pistes d'améliorations/alternatives possibles . . . 15
Mot de fin . . . 16

Introduction

Dans le cadre du cours MATH 60607 – Algorithme pour l'optimisation et analyse des mégadonnées, nous avons réalisé le travail de programmer un algorithme des k-moyennes du début jusqu'à la fin et de le personnaliser. Dans cette brève section, il est question de résumer le cœur de ce rapport, incluant l'algorithme de base et les modifications et ajouts qui y ont été apportées.

En premier lieu, il faut décrire l'algorithme classique des k-moyennes et notre motivation derrière le choix de cet algorithme. L'algorithme a pour but de regrouper des observations dans des groupes distincts dont le nombre a été déterminé à l'avance. Ainsi, après avoir fixé des centroïdes (points de base) aléatoires, il faut assigner chaque observation au centroïde ayant la distance euclidienne la plus proche, puis recalculer le centroïde, et ensuite réassigner les observations. Ces 2 dernières étapes sont réalisées jusqu'à ce qu'un critère d'arrêt soit rencontré, comme un nombre d'itérations maximum ou un seuil de variation minimum atteint.

L'algorithme a un champ d'application dans plusieurs domaines, comme l'identification de segments marketing et de gammes de produits, ou bien le regroupement de corpus textuels et la détection d'anomalies, pour en nommer quelques-uns. Vu le potentiel énorme perçu, nous avons choisi d'en faire notre sujet pour ce projet. Il est important de mentionner d'emblée que le projet a été réalisé dans le langage de programmation Python Après avoir programmé l'algorithme de base, nous avons personnalisé celui-ci en essayant d'y rajouter certaines fonctionnalités et améliorations potentiels. Dans cette optique, nous avons rajouté dans notre algorithme la standardisation des données, chose souvent faite en pratique.

Ensuite, nous avons essayé 2 améliorations majeures à l'algorithme :

- 1- La parallélisation de l'algorithme afin d'essayer de le rendre plus rapide.
- 2- L'ajout d'une métaheuristique afin de rendre celui-ci plus performant.

Dans ce rapport, nous rentrons en profondeur dans ces points et présentons notre algorithme, son équivalent dans le langage de programmation Python ainsi que sa performance avec l'implantation des modifications et améliorations essayées.

Génération des données

Dans le but d'obtenir de meilleures illustrations, nous allons générer `100,000` observations à l'aide de la loi `Normale` pour seulement deux variables, `X` et `y`. Le but est de pouvoir visualiser les résultats à l'aide d'un graphique à 2 axes.

Tout d'abord, afin d'illustrer, nous allons générer les coordonnées de trois points qui vont représenter les centres de nos observations. Les coordonnées de ces points seront $(\mu_{i,x}, \mu_{i,y})$

Pour les coordonnées des `X`:

$$\mu_{1,x} \sim Normal(\mu = 50, \sigma = 100)$$
$$\mu_{2,x} \sim Normal(\mu = 50, \sigma = 100)$$
$$\mu_{3,x} \sim Normal(\mu = 50, \sigma = 100)$$

Pour les coordonnées des `y`:

$$\mu_{1,y} \sim Normal(\mu = 50, \sigma = 100)$$
$$\mu_{2,y} \sim Normal(\mu = 50, \sigma = 100)$$
$$\mu_{3,y} \sim Normal(\mu = 50, \sigma = 100)$$

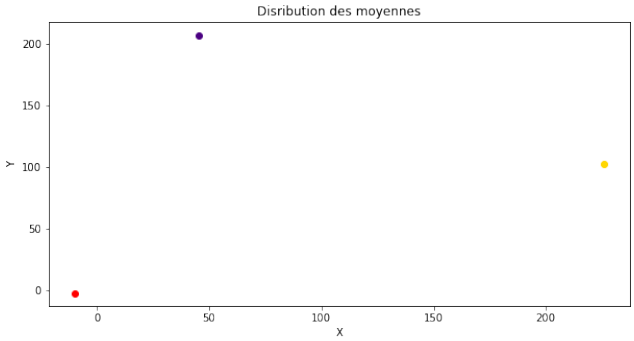


Figure 1: Les centres

Ensuite, nous allons utiliser ces points comme moyenne de loi `Normale` que nous allons utliser pour générer les observations.

Pour les coordonnées des `X`:

$$X_1 \sim Normal(\mu_{1,x}, \sigma = 10)$$
$$X_2 \sim Normal(\mu_{2,x}, \sigma = 20)$$
$$X_3 \sim Normal(\mu_{3,x}\sigma = 30)$$

On fera la même chose pour les coordonnées des `y`

$$y_1 \sim Normal(\mu_{1,y}, \sigma = 10)$$
$$y_2 \sim Normal(\mu_{2,y}, \sigma = 20)$$
$$y_3 \sim Normal(\mu_{3,y}\sigma = 30)$$

On obtiens à la fin ce nuage de points

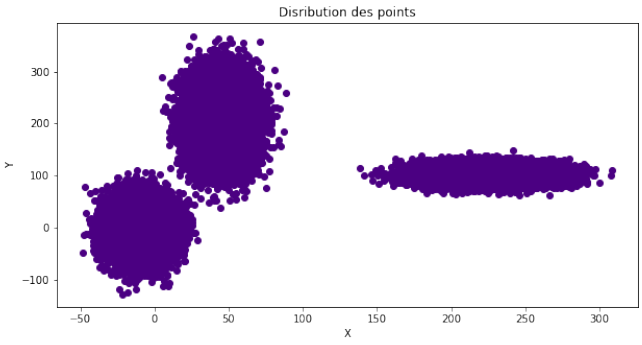


Figure 2: Distribution des points

La fonction Kmeans

Le k-moyenne est un algorithme dont le but est de classer des observations dans des groupes distincts dont le nombre est déterminé à l'avance. Il peut être utilisé dans plusieurs applications d'affaires, comme établir plusieurs segments de marché ou de gammes de produits.

Étapes préliminaires

Échelle des données/Standardisation des variables

Nous allons mettre nos variables sur la même échelle pour mieux calculer les distances. Nous allons utiliser la fonction suivante:

$$x_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

La distance

Pour classer un jeu de données à l'aide de l'algorithme de `Kmeans`, il faut que toutes les variables soient numériques pour pouvoir calculer les distances. Il existe plusieurs façons pour calculer la distance entre deux observations. Dans notre cas (pour le jeu de données que nous avons généré), nous allons utiliser la distance euclidienne, qui est la distance par défaut de l'algorithme des k-moyennes.

$$dist(x_i, x_{i'}) = \sqrt{\sum_{j}^{p} (x_{i,j} - x_{i',j})^2}$$

Les centroides et les classes

La première étape consiste à choisir le nombre de groupes, soit k. Ensuite, dans la phase d'initialisation, k observations sont choisies au hasard et servent de centroïdes, c'est-à-dire de point de référence pour chaque groupe. Par la suite, pour chaque observation, il faut calculer la distance euclidienne avec chaque centroïde, puis assigner l'observation au groupe dont la distance avec le centroïde est la plus faible.

La deuxième étape est une boucle et consiste à définir de nouveaux centroïdes en calculant la moyenne des variables pour chaque groupe, puis à réassigner les observations au groupe la distance euclidienne avec le centroïde est la plus faible. On répète la boucle jusqu'à ce qu'un critère d'arrêt soit respecté. Dans notre cas, il y aura un nombre maximum d'itérations possibles et une proportion minimale d'observations changeant de groupes. Ces conditions seront abordées plus loin dans ce rapport.

Complexité de l'algorithme K-means

En supposant que le jeu de données est de $n \times p$ dimentions, on aura :

- n le nombre d'observations
- p nombre de variables
- k nombre de classes que nous avons choisi
- i le nombre d'itérations maximale

La complexité de l'algorithme `kmeans` est $\mathbb{O}(n \times k \times p \times i)$

Le $n \times k$ est justifiée par le fait qu'on calcule la distance entre chaque n_i observation et chaque `k` centroïde.

Ensuite, on suppose qu'on va calculer la distance `p` fois, le nombre de variables, donc on aura $n \times k \times p$

Enfin, on suppose que l'algorithme roule au pire des cas, qui est `i` fois, ce qui justifie $\mathbb{O}(n \times k \times p \times i)$

Parallélisation

Afin d'améliorer le temps de calcul de notre algorithme, nous allons utliser la parallélisation. L'idée consiste à diviser notre jeu de donnée en fonction du nombre de processeurs que nous souhaitons utiliser.

Dans chaque processeur, nous allons appliquer la fonction qui assigne la classe appropriée pour la x_i observation en se basant sur la distance minimale entre le point et les centroïdes. Ensuite, nous allons regrouper toutes les classes dans un seul tableau.

Le schéma de la Figure 3 illustre notre idée.

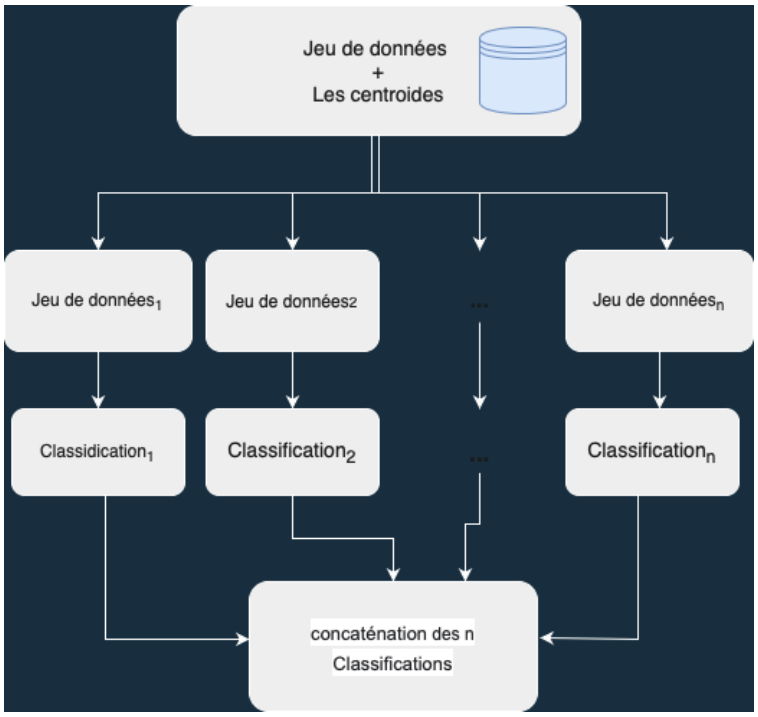


Figure 3: diagramme de parallélisation

Nous allons utiliser un objet de la classe `Pool` de la librairie `multiprocessing`, avec laquelle on serait capable d'utliser plusieurs processeurs pour faire nos calculs, en lui passant le nombre de processeurs à utiliser en argument.

la méthode `map` va recevoir les sous-ensembles de données que nous allons créer comme argument et la fonction qui assigne les classes aux observations. Cependant, la méthode `map` prend un seul argument et notre fonction en prend plusieurs. Pour pallier à ce problème, nous allons utiliser la méthode `partial` de la librairie `functools`. Cette dernière permet de créer un objet fonction auquel nous allons lui assigner tous les paramètres, sauf les sous-jeux de données. Ensuite, on va utiliser cet objet comme fonction dans `map` avec les sous-ensembles de données comme argument.

Pour avoir des bons résultats avec la parallélisation, nos tentatives nous ont démontré qu'il faut au moins `100,000` observations dans son jeu de données afin d'augmenter la vitesse, sinon c'est la fonction sans parallélisation qui est plus rapide.

Voici un graphique dans la Figure 4 qui illustre le temps de calcul en secondes en fonction du nombre de processeurs.

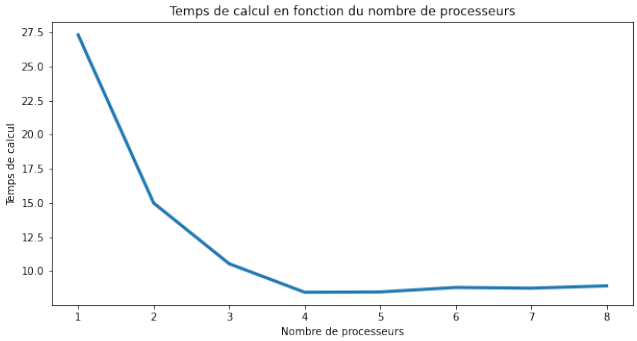


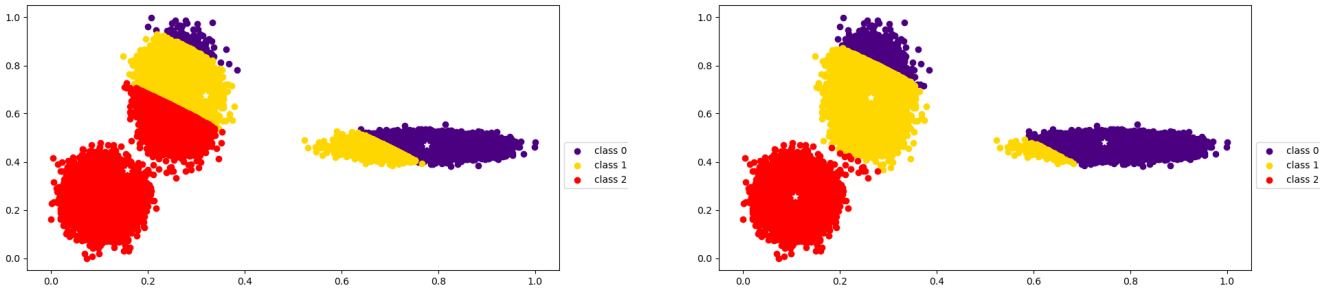
Figure 4: temps de calculs

En observant le graphique, on remarque que le temps de calul commence a se stabiliser à partir de `4` processeurs. Cela est normal, car on utlise le nombre totale de nos processeurs qui est de `4`. En effet, la fonction `CPU` affiche `8` processeurs, mais en réalité nous avons que `4`, car notre ordianteur utlise de `Hyper-Threading`.

Application de K-means

Après avoir rouler l'algorithme sur notre jeu de données, nous avons obtenus ces résultats. Les graphiques suivants dans la Figure 5 représentent la classification pour la première et la dernière itération en ordre.

Les étoiles représentent les centroïdes.



À l'oeil nu, on peut voir que notre algorithme ne classifie pas nos observations d'une manière optimale.

La première étape, qui est de choisir les centroïdes aléatoirement, semble être cruciale pour l'efficacité de notre algorithme de classification. Il nous faut donc trouver un moyen pour choisir les meilleurs centroïdes possibles.

Amélioration de K-means

Pour palier au problème d'assignation inadéquate de notre algorithme `Kmeans`, nous allons apporter quelques modifications.

La première étape reste la même, c'est-à-dire choisir aléatoirement `k` observations qui vont nous servir de centroïdes. Cependant, nous allons rechoisir les centroïdes en se basant sur le choix initial.

L'idée est de calculer la moyenne des centroïdes, ensuite de calculer la distance entre chaque observation et la moyenne des centroides.

$$\mu_{centre} = \frac{1}{K}\sum_{i}^{K} \bar{x_i}$$

$$dist(x_i, \mu_{centre}) = \sqrt{\sum_{j}^{p}(x_{i,j} - \mu_{centre,j})^2}$$

Nous allons utliser ces distances pour calculer le poids de chaque observations dans notre jeu de données.

$$poids(x_i) = \frac{dist(x_i, \mu_{centre})}{\sum_i^n dist(x_i, \mu_{centre})}$$

Les observations extrêmes auront plus de poids. Donc, on aura plus de chance de sélectionner ces observations.

Ensuite, on roule notre algorithme, avec la même condition d'arrêt qu'avant, c'est-à-dire jusqu'à ce qu'il n'y ait plus de changement dans la classfication, plusieur fois, au moins `5` fois pour avoir des bons résultats.

À chaque itération, on enregistre la classification obtenue ainsi que la moyenne des variances de chaque groupe.

$$var_{groupe_i} = \frac{1}{\#K_i} \sum_{j \in i} (dist(x_j, \bar{x_i})$$

$$mean_{var} = \frac{1}{K} \sum_{i}^{K} var_{groupe_i}$$

- où $\#K_i$ est le nombre d'observation dans la classe `i`

Pour chaque moyenne calculée, nous avons une classification.

La dernière étape est de choisir la classification qui correspond à la moyenne des variances minimales.

Test K-means amélioré

Après avoir rouler notre algorithme `10` fois, nous avons obtenus les résultats suivants dans la Figure 5:

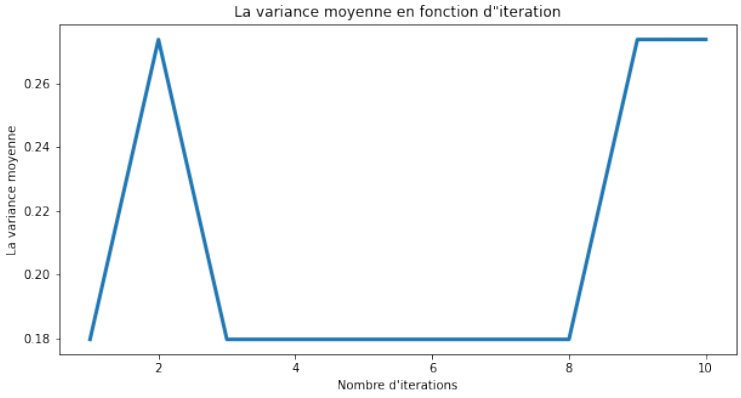
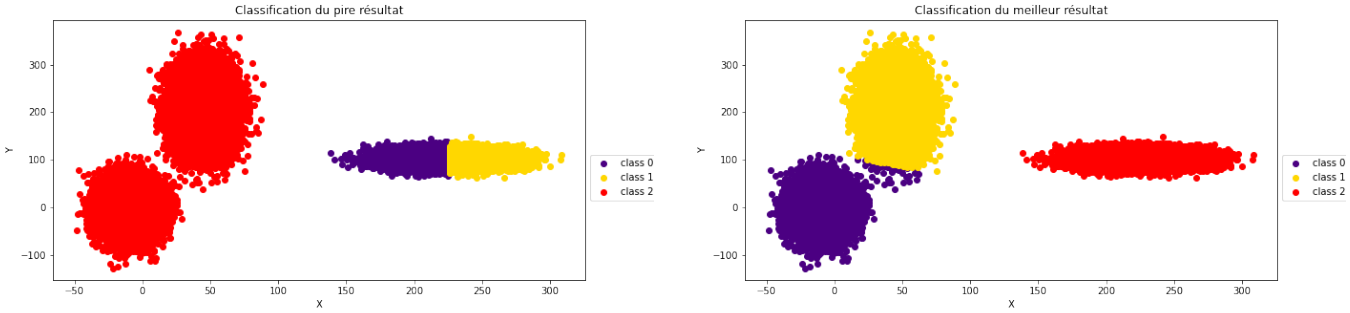


Figure 5: diagramme de parallélisation

Sur les 10 itérations que nous avons faites, il semble avoir beaucoup de valeurs de la moyenne des variances qui se ressemble. On peut choisir notre meilleure classification à partir de toutes les itérations sauf la : 2, 9 et 10.

Afin de comparer les résultats, voici une représentation de la meilleure classification (itération 6) et la classification du pire résultat.



On remarque qu'il y a une grande différence entre la classification selon la pire et la meilleure moyenne des variances.

On a aussi la confirmation que notre amélioration de l'algorithme de `Kmeans` fonctionne très bien si on le compare avec la version initiale.

Processus VNS adapté à Kmeans

L'adaptation de la recherche à voisinages variables (RVV ou VNS) est empruntée de l'algorithme PERTMERGE décrit dans le manuel de cours VNS (fichier VNS p.24-25). En résumé, le "k-means" proprement dit correspond à l'étape de recherche locale dans laquelle on cherche à améliorer la solution courante.Par la suite, on choisit aléatoirement un "cluster" dont un certain nombre d'éléments seront échangés avec ceux d'un autre "cluster" voisin. Cette seconde étape constitue une perturbation aléatoire de deux "clusters" visant à s'extraire d'un optimum local. Une fois l'échange d'éléments effectué, un "k-means" restreint est appliqué aux deux "clusters" concernés pour mettre à jour leurs centroides et réassigner les classes à leurs élements.

Voici les principales étapes de cet algorithme:

- À partir des clusters initiaux, selectionner aléatoirement un cluster à perturber
- Sélectionner aléatoirement un élément du cluster à perturber
- Calculer la distance entre cet élément pris aléatoirement avec tous les centroïdes des autres clusters
- Considérer le centroïde le plus proche comme étant le cluster voisin
- Sélectionner aléatoirement un pourcentage défini d'éléments dans chacun des deux clusters voisins et échanger ces éléments
- Effectuer un k-means classique et restreint entre ces deux clusters cibles
- Répéter k-fois les mêmes étapes antérieures jusqu'à ce que k-max soit atteint

Execution de k-means pour tester une selection de nombres de clusters

Concernant la mise en oeuvre de notre modèle k-means, les conditions d'arrêt sont définies:

- soit par la stabilité des centroïdes qui est caractérisée par un écart de distance inférieur à "epsilon" entre les anciens centroïdes et les nouveaux centroïdes correspondants
- soit par un nombre maximal d'itérations qui a été fixé à 300 dans notre expérience

D'abord, comme cette partie du travail s'éxecute sans le processus de parallélisation, un échantillon de `1000` observations a été prelevé aléatoirement des observations initiales. Puis, nous avons vérifié que cet échantillon garde toujours non seulement la même distribution des observations mais également l'absence de corrélation entre les deux variables.

Enfin, nous avons testé `19` modèles `k-means` en variant les nombres de "clusters" de `2` à `20` dans le but de sélectionner les meilleurs nombres de "clusters". À cet effet,la métrique "silhouette" sert à évaluer chacun de ces modèles en se rappelant que la "silhouette" d'un modèle performant se trouve entre `0.5` et l'unité. Cette approche nous permet donc d'identifier les nombres de groupes qui sont les plus intéressants pour la suite de la recherche à voisinages variables.

À titre de précision, nous calculons le coefficient silhouette "s" de chaque élément d'un groupe avec la formule suivante:

```
s=(b-a)/max(a,b)
```

avec:

a:la distance moyenne entre un élément et les autres éléments appartenant au meme groupe

b: la distance moyenne entre un élément d'un groupe et tous les éléments du groupe voisin le plus proche

À partir de tous les coefficients silhouette de l'ensemble des observations, on établit une moyenne qui est considéré comme le score silhouette du modèle.

Globalement, les modèles avec des nombres de "clusters" inférieurs à `6` sont les plus intéressants. Les performances des modèles se dégradent rapidement pour de grands nombres de clusters. À cette étape de l'analyse, nous soulignons surtout que le modèle à `3` "clusters" est celui qui émerge vraiment du lot avec

un score de `0.82` et que les `19` modèles ne présentent aucun élement mal classé, leur silhouette score étant toujours positif.

Nombre de cluster	silouette
2	0.54
3	0.82
4	0.63
5	0.53
6	0.45
...	...
20	0.35

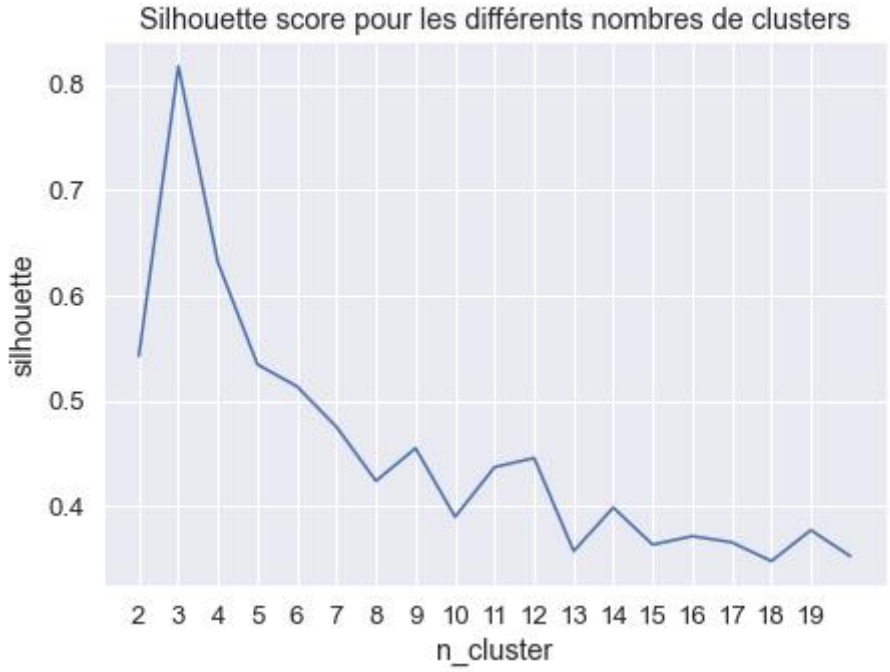


Figure 6: Silhouettes des modèles avec différents nombres de groupes

Execution de 100 iterations de VNS

À partir des `19` modèles, on applique l'algorithme de recherche à voisinages variables à chacun des `5` meilleurs modèles correspondant aux nombres de clusters de `2` à `6`. Pour chaque modèle, une séquence de processus de perturbation (PERTMERGE) suivie d'une recherche locale restreinte `k-means` entre deux clusters voisins est répétée `100` fois.

Une question pourrait se poser quant au choix du nombre d'itération `VNS`. Faute de recommandation, nous avons adopté un nombre plus ou moins grand de `100` itérations. Et l'approche par sélection de la meilleure solution a été intégrée dans la mise en oeuvre de l'algorithme `VNS`.

À l'issue du processus, nous constatons que seul le modèle à `2` clusters a réussi à améliorer son score en passant de `0.54` à `0.63`. Les autres modèles semblent indiquer une certaine stabilité de leurs centroides et clusters.

Visualisation des clusters issus de VNS

Après la RVV, un meilleur score "silhouette" de `0.82` est obtenu avec le modèle à `trois` "clusters" sans que cette performance ne soit meilleure que celle du `k-means` classique. Les perturbations apportées à ce modèle n'ont donc pas modifié les clusters initiaux.

En clair, ce modèle distingue nettement les trois groupes d'observations, ce qui est cohérent aux résultats attendus d'autant plus que les clusters sont plus éloignés des uns des autres avec notre échantillon. L'amélioration du modèle à `2` clusters vient du fait que les observations à grandes valeurs de la variable `X`("col1") ont été isolé du reste.

Par contre, les autres modèles avec `4`, `5` et `6` clusters restent insensibles aux perturbations. Ils tendent surtout à fractionner les clusters ayant des valeurs de la variable `y` ("col2") élevées en laissant intact le groupe à valeurs plus ou moins faibles.

Ces nouvelles fragmentations créent beaucoup de proximité entre les élements des clusters voisins, ce qui a pour tendance de faire baisser leur score "silhouette". Enfin, nous remarquons aussi le basculement d'une observation intermédiaire d'un groupe à un autre lorsque le nombre de cluster croît (voir la figure 7 montrant le passage de `4` à `5` clusters)

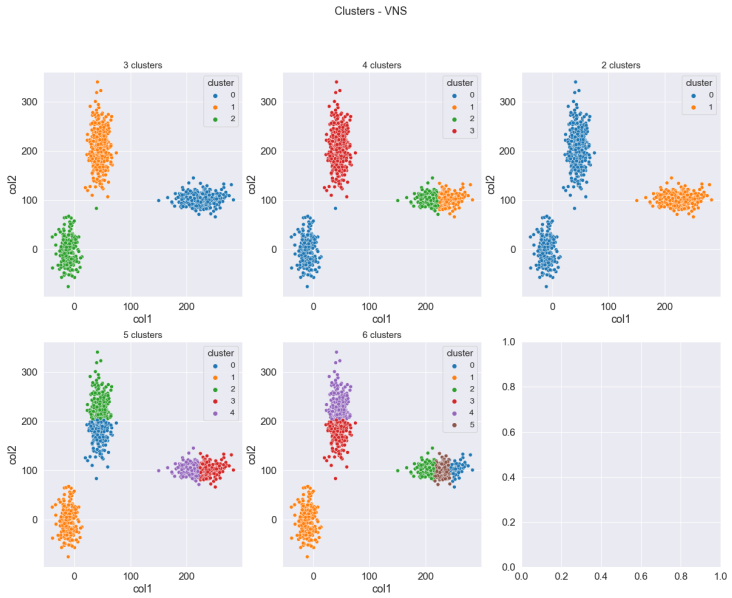


Figure 7: Représentation des groupes après VNS

Visualisation des clusters pour chacune des variables

Cette analyse a pour objectif de situer les observations de chaque cluster par rapport aux deux variables dans la figure 8. Il en ressort que les observations ont des intervalles de valeurs bien distincts pour les différents groupes des modèles à `2` et `3` clusters. Ce qui n'est pas le cas pour les modèles à `4`, `5` et `6` clusters dans lesquels il y a des groupes dont les valeurs d'au moins une des deux variables sont mélangées.

Visualisation des positions des centroides pour chaque variable

En cohérence avec l'analyse des clusters par rapport aux variables, nous observons dans **figure 9** aussi que certains centroides ont les memes coordonnées par rapport à une variable lorsque le nombre de clusters est supérieur à 3. Ce qui conforte l'idée d'une subdivision d'un groupe à chaque augmentation du nombre de

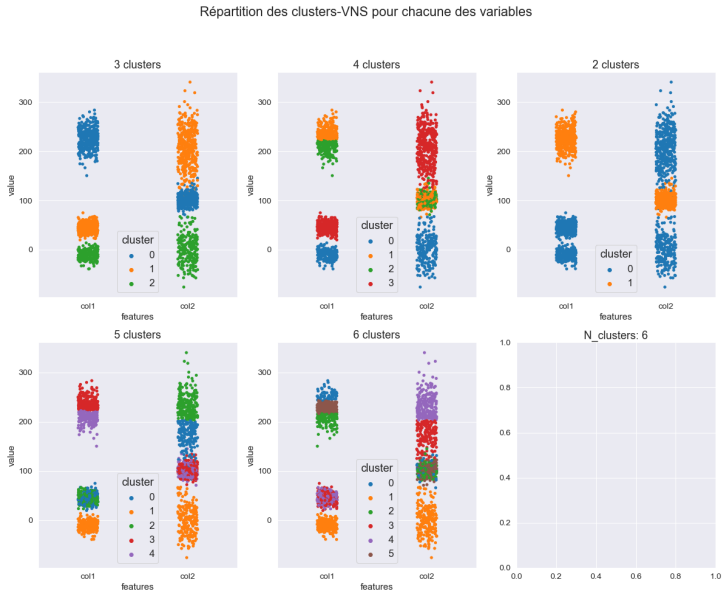


Figure 8: Répartition des groupes en fonction de leurs nombres et les variables

clusters. Les centroïdes des modèles à 2 à 3 clusters sont quant à eux bien séparés les uns des autres. En particulier, un des centroïdes du modèle à 3 clusters correspond aux plus faibles valeurs des variables `X` (“col1”) et `y` (“col2”).

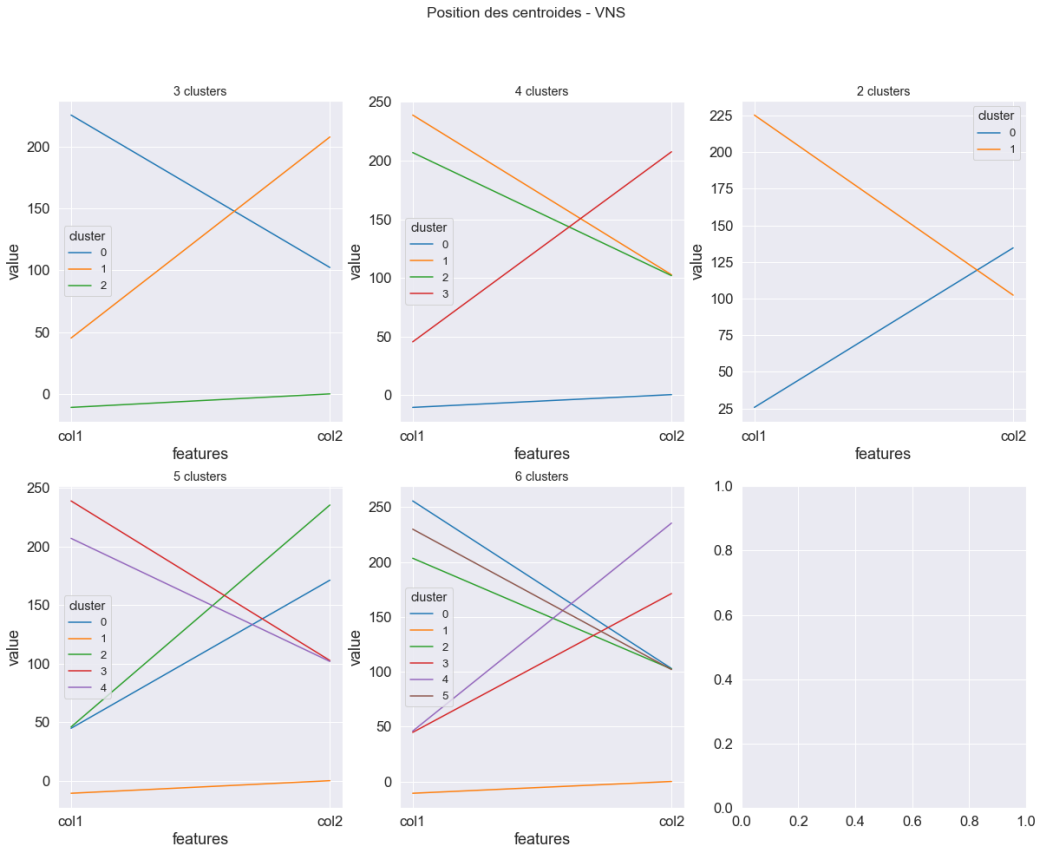


Figure 9: Positions des centroides par rapport aux variables

Evolution du silhouette_score à chaque itération VNS

À l'issue de 100 itérations VNS,les scores silhouettes des modèles à `3`, `4`, `5` et `6` clusters demeurent inchangés. Ces modèles sont donc typiquement robustes pour notre jeu de données, c'est-à-dire l'échantillon.

Mais compte tenu d'une des remarques précédentes, la multiplication des observations intermédiaires entre les clusters, ou autrement dit le rapprochement des frontières des clusters, risque de perturber assez significativement les clusters et de modifier les scores des modèles.

Le modèle à `2` clusters fait l'objet d'une fluctuation (**figure 10**) de son score silhouette. Ce qui sous-entend une forte variation des clusters dans certaines itérations. Néanmoins, le modèle avec `2` clusters le plus performant et le plus fréquent est celui avec le score de `0.63`. On peut donc conclure que, hormis pour ce modèle, le`VNS` n'est donc pas toujours nécessaire pour l'analyse de regoupement `k-means` pourvu que les clusters sont suffisamment éloignés les uns des autres. Mais le `VNS` a quand meme le mérite d'infirmer ou de confirmer la stabilité des modèles créés.

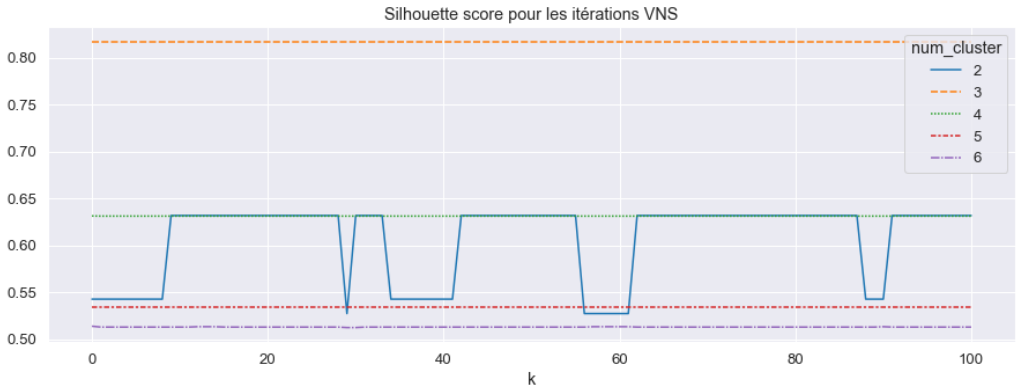


Figure 10: Silhouettes des modèles en fonction des itérations VNS

Temps d'execution à chaque itération VNS:

La création des clusters initiaux avec k-means tout court est plus dispendieuse en temps que le processus de recherche à voisinages variables combinant une recherche locale `k-means` restreint et une perturbation. Ce constat renforce l'idée que le k-means est pénalisant computationnellement.

Et la restriction appliquée lors du processus VNS, semble avoir relativement accéléré l'obtention de meilleures solutions plutôt que d'adopter un `k-means` en mode "multi-start" par exemple. A part le modèle à `2` clusters dont le temps d'execution du processus VNS peut dépasser les `20` secondes pour chaque itération, ces temps tournent autour de `5` seconde pour les autres modèles (**figure 11**). Nous remarquons également que les temps d'execution sont très peu fluctuants pour le modèle à `3` clusters.

En outre, bien que globalement, les temps computationnels k-means augmentent avec le nombre de clusters défini, il apparaît également que la difficulté d'apprentissage ajoute un temps supplémentaire. Ceci est assez évident en comparant, d'une part, les modèles à `2` et `3` clusters, et d'autre part, les modèles à `5` et `6` clusters.

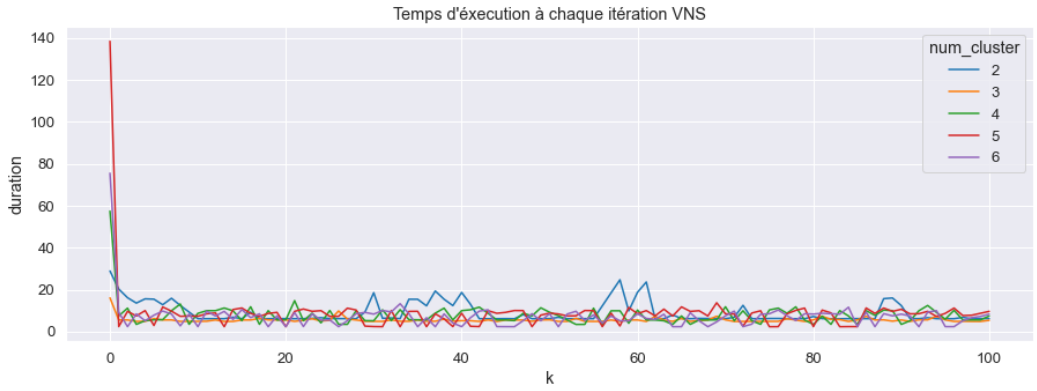


Figure 11: Temps d'éxécution en seconde

Conclusion

Limites rencontrées

Somme toute, ce projet constitue pour nous un apprentissage significatif et a comporté son lot de défi, malgré la réussite d'implantation des algorithmes proposés. En effet, côté compétences, chacun des membres de notre équipe n'avait des bases solides en programmation pure et la pente d'apprentissage a été abrupte. De plus, certaines implémentations furent relativement fastudieuses, comme avec la métaheuristique VNS proposée. Aussi, pour notre métaheuristique VNS, nous avons constaté que le temps computationnel a été très pénalisant, celui-ci étant très élevé au-delà de quelques milliers d'observations. Notre jeu de données en comportant plus de 100 000, nous avons été dans l'obligation de sous-échantillonner et n'en utiliser que 1000. Il faut toutefois justifier que nous avons faits en sorte que les formes de nuages de points lorsqu'on représente nos données sur un graphique restent inchangées. Tout de même, la comparaison de performance entre nos 2 améliorations proposées reste difficile en partie à cause de ce sous-échantillonnage et aussi car nos mesures de performance diffèrent entre nos 2 améliorations. Puis, que ce soit avec l'algorithme initial ou optimisé, choisir le nombre d'itérations a été relativment difficile, pour ne pas dire pénalisant.

Pistes d'améliorations/alternatives possibles

Le projet que nous avons effectué peut totalement être reproduit et nous encourageons toute personne voulant essayer de programmer cet algorithme à le faire. Nous allons quand même proposer quelques pistes d'améliorations possibles afin d'outiller ceux qui marcheront dans nos pas. Premièrement, il serait intéressant d'essayer une autre métaheuristique, tel que PERTBASE, PERTSPLIT, ou même tout algorithme génétique. En effet, bien que leur fonctionnement soit un peu différent du VNS, nous pensons qu'essayer ces changements permettrait une bonne comparaison avec notre algorithme et il n'est pas exclu que ces autres métaheuristiques soient plus efficaces dans le cas des k-moyennes, pouvant potentiellement palier aux difficultés que nous avons rencontrés.

Après, nous recommandons d'essayer des alternatives dans la phase d'initialisation, à l'étape de fixer les centroïdes initiaux, car nous n'avons pas obtenu les meilleurs résultats avec la sélection aléatoire. À la place, nous orientons le lecteur vers des méthodes comme le kmeans++, qui est une méthode probabiliste implantée notamment dans la librairie d'apprentissage machine « sci-kit learn ».

Troisièmement, il existe une méthode pour choisir le nombre optimal de clusters selon une autre méthode de segmentation, la méthode hiérarchique. Un bon exemple d'application est dans l'analyse des mégadonnées. L'algorithme k-means est plus rapide, mais il recquiert de fixer un nombre k de clusters, alors que la méthode hiérarchique apprend le nombre de clusters optimaux, mais est généralement moins rapide sur des mégadonnées (plusieurs millions d'observations). On pourrait alors sous-échantillonner tout en gardant la distribution des données, appliquer la méthode hiérarchique, puis appliquer le nombre de clusters trouvé ainsi que leurs

centroïdes à l'algorithme kmeans.

Puis, nous pensons qu'il serait judicieux d'essayer une autre mesure de performance que le score de silhouette, afin de comparer les résultats obtenus avec plusieurs métriques, puis de choisir celle dont le résultat répond le mieux aux besoins précis qui ont mené à l'exécution de l'algorithme, comme un besoin d'affaires par exemple.

Finalement, pour ceux désirant s'épargner la programmation de l'étape de parallélisation, il faut savoir que celle-ci peut être faite sur databricks par l'entremise de spark. Nous redirigeons donc le lecteur dans ces eaux.

Mot de fin

Pour finir, nous avons construit dans le langage Python un algorithme des k-moyennes, d'abord initial, puis modifié. La modification a été la parallélisation de l'assignation des clusters aux observations. Par la suite, 2 optimisations ont été essayées, soit le recalcul des centroïdes par poids puis l'implantation de l'algorithme VNS.Outre nos solutions proposées, il existe un éventail d'alternatives à notre travail qui méritent d'être explorées et nous espérons que notre travail pourrait servir de base pour d'autres.