雲端運算程式設計期末報告

1. **介紹**

在現今各種聚類分析的演算法中，k-means 演算法有著易於開發、易於收斂等特性，但其缺點也是顯而易見的：計算時間久，以及不易到達最佳解。缺點[5]如下：

1. 使用k-means時，必須先知道分群數才可進行運算。
2. k-means的運作必須依賴初始的群集中心點。
3. k-means在分群後所得到的解答可能陷入局部解
4. 無法妥善處理重疊的資料點

因此自k-means被提出[1]以來，亦有無數的方案來改進其缺憾，本次的期末報告即是選擇一種改進方案，將其實作在spark上以進行加速運算。

作為易於開發的演算法，k-means的實際步驟相當的簡易：

1. 初始化
2. while 未達成終止條件
3. 重新分群
4. 重置中心
5. 計算終止條件
6. end while

關於初始化的方法，分為Forgy和Random Partition。Forgy法將每個點隨機指定到一個群中，再來計算其平均中心；而Random Partition則是隨機指定中心點，再來依此進行分群。由於本次的評估依據是以SSE作為好壞的依據，根據[2]的研究，我們選擇了Random Partition作為初始化的方法。

在取得中心點後，我們利用歐式距離[3]作為分群的依據，將每個資料點重新指定至最近之中心點，以達k-means的目標：盡量讓SSE最小化。

在得到新的分群結果後，以該群的點之形心為新的中心，計算其SSE，以作為計算終止條件的依據，我們在此SSE的變化率小於10e-5 (=10e-3 %)為終止條件。

1. **改善方法**

由於前述k-means的相關缺憾，我們選擇了[4]作為想要實作出來的改進方案，作者基於基因演算法的精神，修改了部分演算法的步驟，進而提出了Genetic k-means Algorithm(GKA)。由於基因演算法有以下特性[5]，使得在解決最佳化的問題上，可優先選擇基因演算法：

1. 基因演算法是以編碼後的染色體基因作為運算依據，而不是參數本身。
2. 基因演算法在搜尋空間中以多個參考點來找尋最佳近似解，可避免落入局部最佳解(local optimum)的問題。
3. 基因演算法是使用機率規則方式引導搜尋方向，而非使用明確的規則，因此較能符合各種不同類型的最佳化問題。

GKA演算法如下：

input:

Mutation Probability Pm

Population size N

Maxinum number of generation MAX\_GEN

1. init：初始化族群
2. geno = MAX\_GEN #最大世代數
3. while geno > 0
4. 計算適應值(fitness values)
5. selection：依照適應值選擇一個染色體做為母體
6. mutation：從母體產生下一世代並進行變異
7. k-means：將變異結果進行一個步驟的k-means
8. 與當前最好結果比較，若更好則取代
9. geno = geno - 1

init：將所有資料隨機指定到一個群，產生N筆結果，做為第0世代

fitness values：將所有資料計算其SSE後，得到N筆結果，並進行計算其平均SSEavg與標準差SSESD後，得到其適應值：

fitness values = SSEavg - SSE­i=1,2,…,N + cm × SSESD (若<0則設為0)

selection：依照剛剛算出來的適應值，進行輪盤式挑選，做為下一個世代的母體。

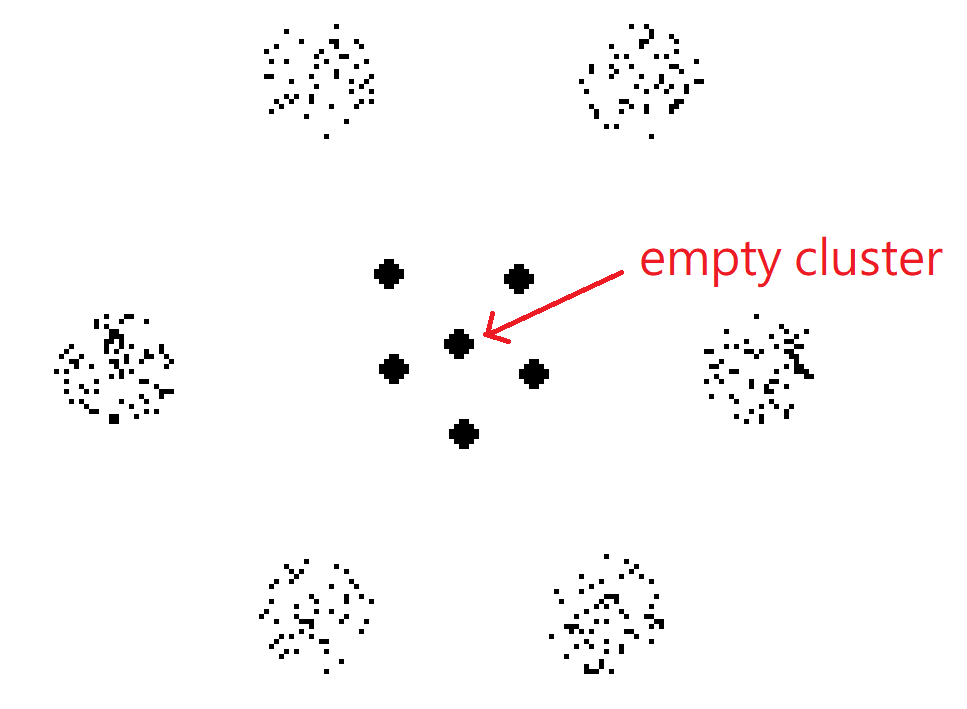
mutation：將母體做為初始值，並以機率Pm挑選一些點進行變異，使其改變所屬之群。改變的機率p則與該點與所有中心之距離相關：pi=cm × dmax – dj，在此仍然以輪盤式進行挑選。

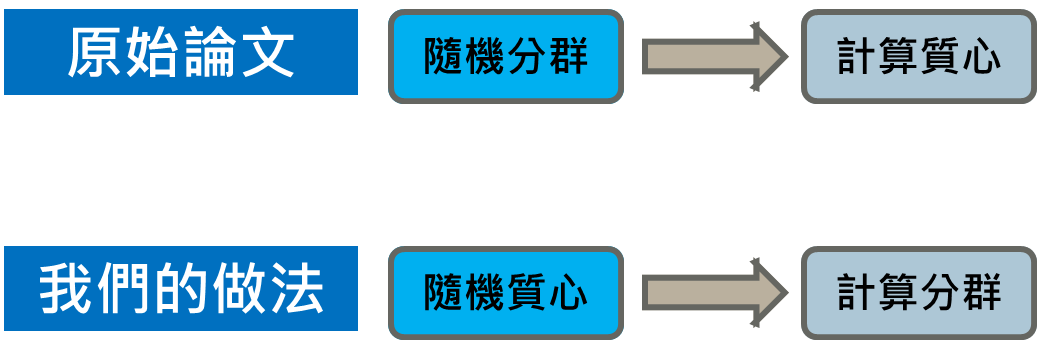
k-means：這裡只做一個步驟的k-means，也就是重置中心、重新分群即重新開始下一個世代的挑選。

1. **實作方法**
   1. Initial階段：

在實作的過程中，我們發現在初始化的時候，若用作者提供的初始化方式(Forgy)，相當容易出現空群，因此在一開始，我們在隨機分群後，先對該結果做一個步驟的*k*-means，再來做後續的流程，但即使如此，結果仍然不是很好。

我們認為原因應該是在樣本數巨大時，其初始的中心點會相當接近整體資料集的形心，導致在內部的中心不容易被分到資料點(如圖一)。最後我們將其改為Random Partition，在所有資料點中選擇數點做為其形心，再依此進行分群。

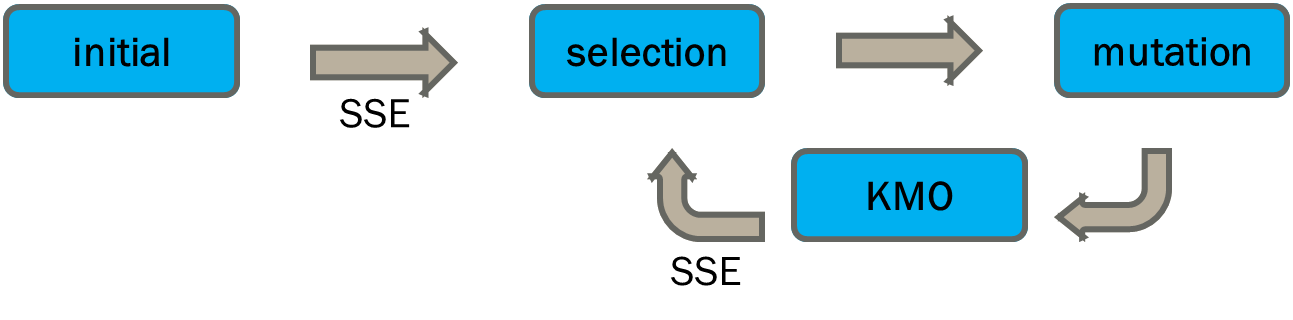
圖一：Forgy之初始點問題

 在這篇論文中，作者並沒有提到當*k*-means這個步驟若出現空群時的處理方式，這部份我們的做法是隨機挑一個資料點指定至該群，並再度作一次重置中心與重新分群，直到沒有群為空為止。

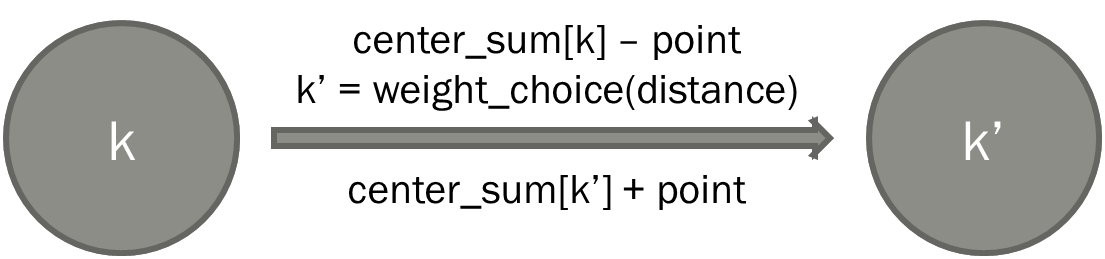
圖二：在初始化階段中，與原始論文之差異

* 1. seletion階段：

在此階段中，需要用到的SSE來計算各個染色體的適應值(fitness value)，然而在初始化(initial)與KMO(k-means operator)階段中，就會計算到SSE，因此這部分直接使用前一個階段的結果，而不需要再重新計算。

圖三：selection階段中，SSE的來源

* 1. mutation階段：

單機版：在論文中，每次的變異都需重新計算質心，然而這樣作法的結果，就是計算時間會與資料筆數的平方成正比O(n2)。因此我們作了點修正：在計算質心時，記錄的是總和而非平均，結果是在變異時，只需將該點從原本的中心移除，加入新的質心即可，如此一來便能節省大量的計算時間。

圖四：mutation階段中，質心的計算方式

Spark版：在論文的原始做法中，質心的位置會隨著每次的變異而改變，使得每個迴圈之間的資料產生了相依性，而不容易進行平行化。因此我們修改了這個步驟，使得計算質心只在一開始執行。

* 1. KMO(k-means operator)階段：

在mutation階段變異完成後，KMO對變異結果計算質心，並且依照新的質心位置進行重新分群，來達成一個步驟的KMO。

當重新分群後，偶爾會發生有空群的狀況，對於空群的處理，我們隨機挑選一個資料點作為該空群之中心，並重新進行KMO階段

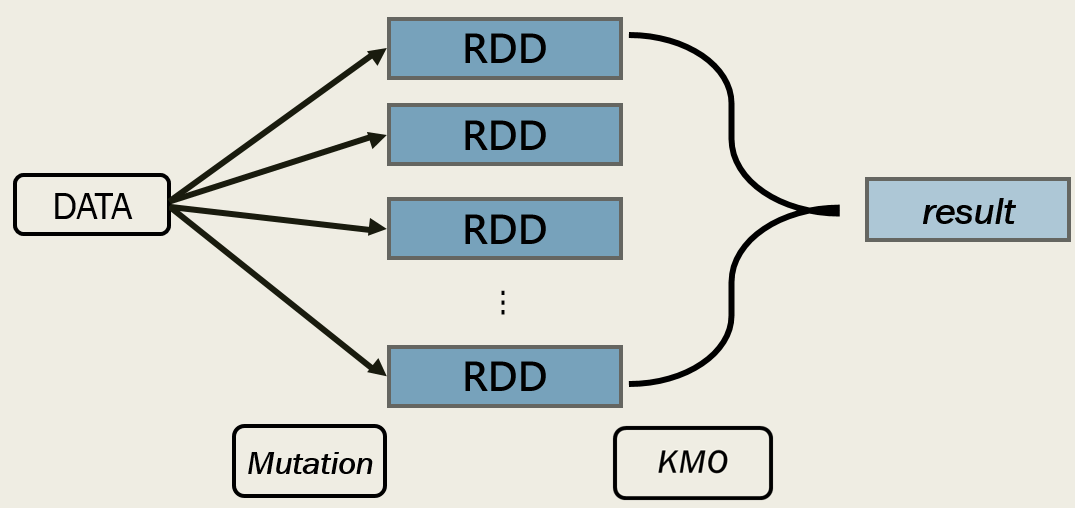
1. **平行處理與加速運算**
   1. 單機版

使用numba.jit進行加速，他可以用修飾函數JIT編譯python函數，並返回一個可在Python中調用程式碼的包裝對象。

* 1. Spark版

使用Spark進行多節點加速，使用MAP和REDUCE針對各個不同的RDD做運算來達到平行處理的效果。

在原本的做法，將每個result各自當成一個RDD，並在mutation與KMO階段進行平行運算，最後再將結果collect，然而在實作過程中發現，這樣會導致記憶體的不足。因此改變了平行化的方法：將原始的DATA做成RDD，經過mutation和KMO後合併成一個result，用此方法即可避免記憶體不足的問題。



1. **結果與比較**
   1. 運作環境

|  |  |  |  |
| --- | --- | --- | --- |
|  | CPU | memory | OS |
| 主要結果 | I7-4770(3.40GHz 8CPUs) | 8 GB | Ubuntu 16.04 |
| 筆記型電腦/對照 | I5-3230M(3.20GHz 4CPUs) | 12 GB |
| 伺服器/對照  (master) | Intel Xeon CPU E5-1620v3(3.50GHz and 4 cores) | 16 GB |
| 伺服器/對照  (node1) | Intel Xeon E5-2620v4(2.1GHz and 8 cores)(SLI) | 78 GB |
| 伺服器/對照  (node2) | 48 GB |

* 1. 使用參數

世代數Generation：10

族群數Population size：10

變異機率Pm：0.05

變異常數Cm：1

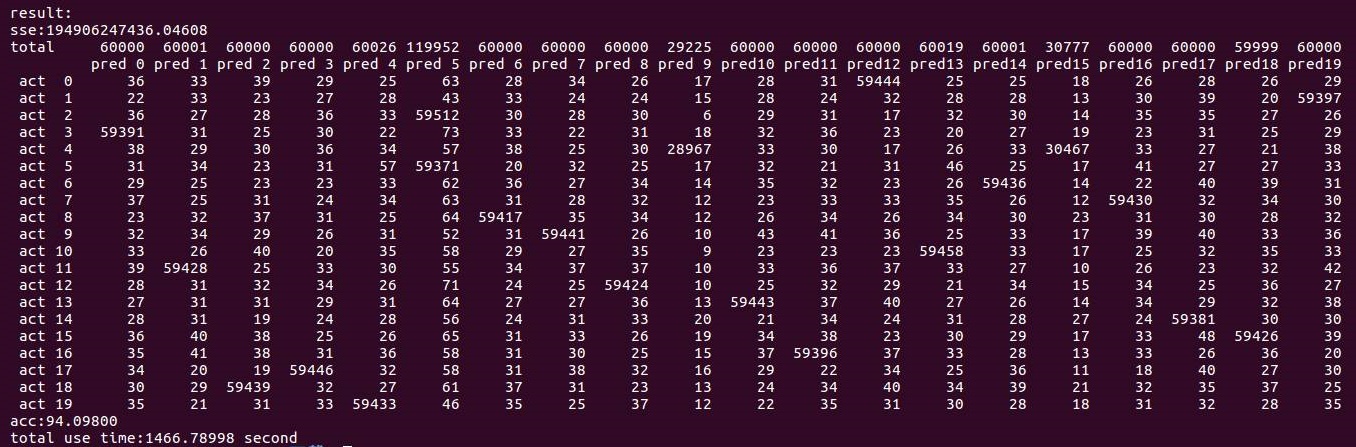
變異標準差C：2

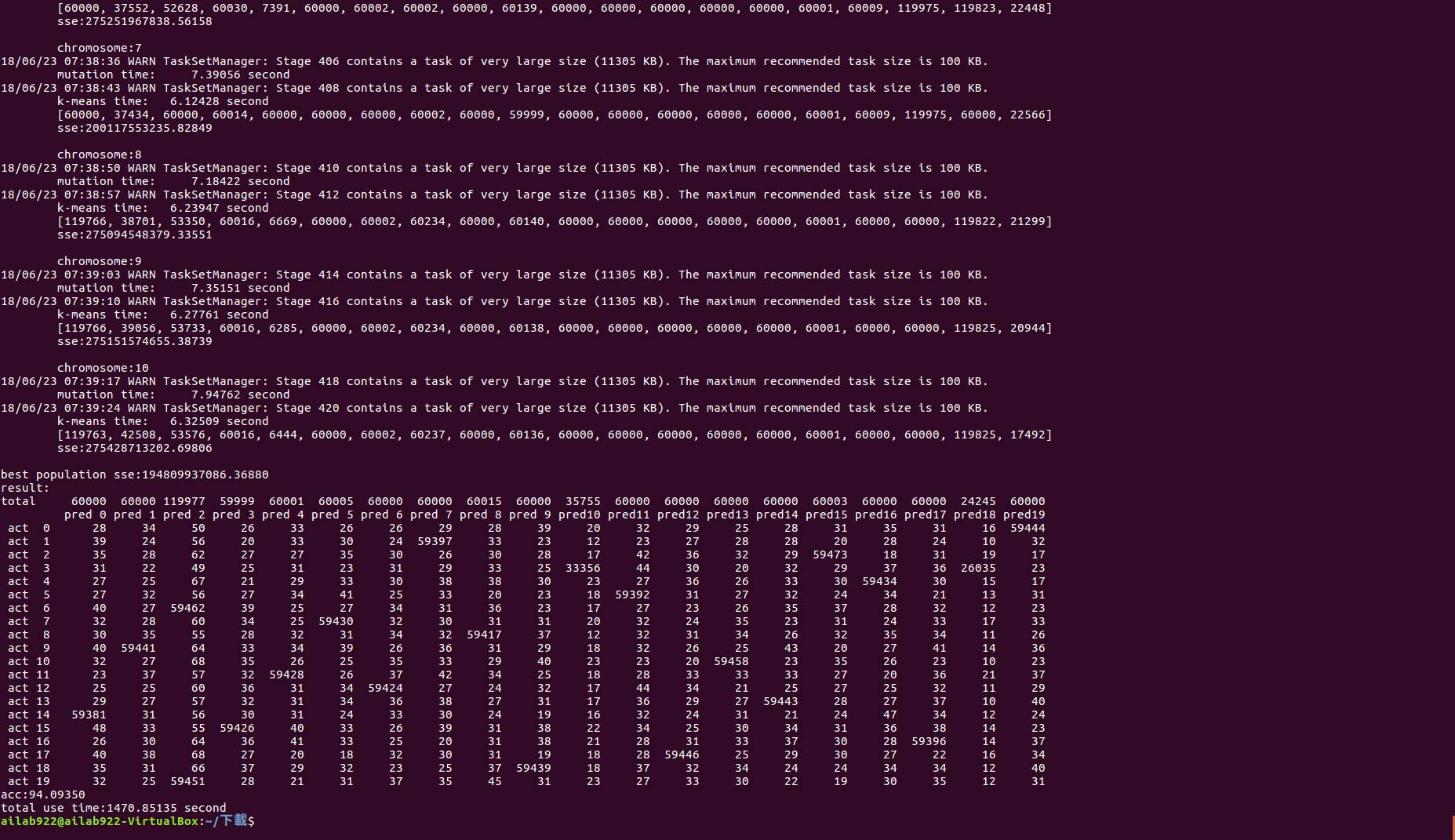
* 1. 運作結果(單機版 / spark版)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | **k-means** | **Bisecting**  **k-means** | **GKA** |
| **單機版** | **時間(sec)** | 775(3239\*1) | 90 | 1466(13512\*1) |
| **SSE** | 215648934874 | 126405183046 | 194906247436 |
| **ACC(%)** | 90.31925% | 98.50875% | 94.09800% |
| **SPARK** | **時間(sec)** | 202 | 215(3674\*2) | 1470 |
| **SSE** | 188525641091 | 148235346487 | 194809937086 |
| **ACC(%)** | 96.55717% | 98.46821% | 94.09350 % |

\*1：括號內的數字代表沒有使用numba.jit加速的時間

\*2：括號內的數字代表改進前的執行時間

(單機版)

(SPARK版)

1. **討論**
   1. **單機版/SPARK GKA實作方式的差異**

在原始的論文中，每一次的mutation皆需要重新計算質心，單機版本的質心是以加減之運算來得到；SPARK版則是在mutation階段皆不去計算質心，而是選擇K-means階段進行計算。

* 1. **GKA較適合於分群數較多的資料集**

在IRIS中，由於使用一次性的k-means(k-means operator)容易在下個世代的mutation階段把正確的分群結果變異掉，故在準確率上較易受到影響。

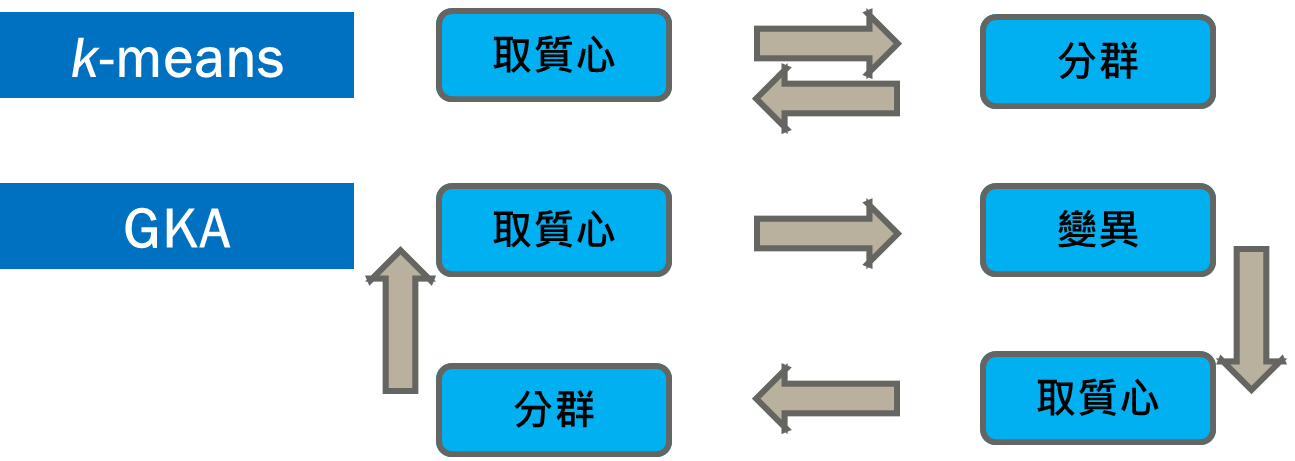
* 1. **計算時間的問題**

在實作的過程中，每次染色體的計算(mutation+k-means)皆不超過30秒，然而以實體機器運作的結果往往超過240秒，使得在10個世代與10個染色體的條件中，計算時間總和超過25000秒(約七小時)，因此商借了伺服器來執行相同的程式則只需要約半小時左右，就結果而言，計算時間可能在於資料的讀取時間與節點之間的溝通時間影響最大。

* 1. **GKA的最佳解問題**

由於selection階段只會取一個染色體進行演化，使得GKA的做法不易達到全域最佳解，但能做到算是不錯的解，然而區域及全域最佳解皆很難達成。因為在初始化時，不一定能夠挑到全域最佳解的分支，以及前述之原因，即使經過一次次的演化，也很難移動到全域最佳解的範圍。至於區域最佳解的部分，由於在演化時，一次的變異只會搭配一次的KMO，使得結果不易收斂至區域的最佳解。

* 1. ***k*-means與GKA的差異**

在*k*-means中，做的就是取質心、分群這兩個步驟的循環。而GKA則是將兩次循環合併成一次循環，並改變了其中一個步驟，使得循環變成：取質心、變異、取質心、分群四個步驟的循環。

1. **結論**
   1. 在我們的實作結果中，單機版和SPARK版時間相當接近，原因在於使用numba.jit進行加速。在單機版中，整體的運算速度大約可提升高達10倍，而在SPARK版中，其加速結果僅僅約莫2倍左右，使得其表現旗鼓相當。若在不使用numba.jit進行加速的情況下，SPARK的加速程度是顯而易見的。
   2. 實作的過程中我們發現，同樣的一份程式碼，在不同機器上，其結果也大不相同，在筆記型電腦中，其運算時間高達25000秒；而在主要結果中，則只需要5700多秒，因此借用伺服器等級的電腦來執行，結果僅需約2000秒即可。
   3. 對於時間而言，GKA運行的時間遠遠超過一般k-means與Bisecting k-means。
   4. 以SSE之變化程度來看，一般k-means會有十分不穩定的變化；Bisecting k-means則較有穩定的表現；而GKA介於其中。
   5. 準確率的部分，k-means徘迴於80%~90%之間，而GKA則是在90%~97%左右，至於Bisecting k-means幾乎落在97%~99%。
   6. 在GKA的運算中，若經JIT，mutation跟k-means的時間比例約是1(7s)：1(7s)左右；沒有JIT加速時，mutation跟k-means的時間比例約為1(13s)：7(90s)。而selection的計算時間幾乎可以忽略不計。
2. **參考資料**

[1] MacQueen, J. B. Some Methods for classification and Analysis of Multivariate Observations. Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. University of California Press: 281–297. 1967 [2009-04-07].

[2] Hamerly, G. and Elkan, C. Alternatives to the k-means algorithm that find better clusterings (PDF). Proceedings of the eleventh international conference on Information and knowledge management (CIKM). 2002

[3]<https://zh.wikipedia.org/wiki/%E6%AC%A7%E5%87%A0%E9%87%8C%E5%BE%97%E8%B7%9D%E7%A6%BB>

[4] K. Krishna, M. Murty, " Genetic K-means algorithm ", IEEE Trans. Syst. Man Cybern. B Cybern., vol. 29, no. 3, pp. 433-439, Jun. 1999.

[5] 葉承銓(2002)。《應用適應性基因演算法於資料分群的問題》。樹德科技大學資訊管理系研究所論文。高雄市。