Christopher Draper

Implementation of *Making K-means even faster*

**1. Introduction**

The k-means algorithm is a commonly used clustering algorithm for data points in d-dimensional space. It takes n data points and attempts to group them into k clusters based on minimizing the within-cluster variance. The k-means algorithm is a very useful algorithm, used everywhere from machine learning, image compression, density estimation, and many other fields. Anything that can be represented as a point in Euclidean space can have the k-means algorithm applied to it. However k-mean clustering has two major problems associated with it. First is that the theoretical worst case run time is super-polynomial [2]. While this case is uncommon, most basic versions of k-means algorithms are very slow and can boarder on super-polynomial time. Second is that most k-means algorithms guarantee a solution but not an optimal solution. Providing an optimal solution is an NP-hard problem while providing any solution can be significantly faster. The paper I implemented was *Making K-means faster* by Greg Hamerly [1] which attempts to solve the first issue by providing a novel way of reducing the time needed to run a k-means algorithm. For this paper and project I will be implementing Hamerly’s k-means clustering algorithm (Hamerly’s algorithm) for two dimensional data and repeating the tests done using the provided datasets, testing for memory usage and run time as done in the paper.

**2. Survey of related work**

In a basic over view the k-means algorithm can be broken down into two stages. First the initialization stage where the starting location of each k cluster is chose and then the convergence stage where points are assigned to clusters. These stages can be considered separate problems and any algorithm that solves the initialization stage and easily be attached to any algorithm that solves the convergence stage. While this does not hold true for all k-means algorithms I will be discussing improvements made to both stages and an alternative method to k-means clustering.

The selection of initial cluster locations plays a role in how optimal the final cluster locations are and can play a role in how fast the k-means algorithm runs. Given poor starting initial clustering locations it is possible for k-means to converge onto less than optimal positions or to take more steps to converge into a solution. The simplest but still effective method is the Forgy method of initialization. The Forgy method randomly selects k unique data points from the data set and uses those locations as the starting location for the clusters. Using data points from the data set guarantees that a cluster will start out with at least one data point assigned to it and prevents clusters being assigned to empty space in the data set. Next is the Random Partition method which is simpler but has been found to have a slightly higher quality [3]. This works by assigned all points to a random cluster and then finding the average location of all the points for each cluster and making that the new cluster location. This will place all the clusters in the approximate center of the data set so that they will spread out from there. This process offers more of a guarantee that the clusters will be evenly spread out across the data set as the k-means algorithm runs. Lastly is the k-means++ initialization method which offers improvements over both the Forgy method and the Random Partition method in terms of accuracy and speed up [4]. K-means++ works to evenly place the cluster’s initial location across the data points. It does this by first randomly selecting a data point for the starting cluster. Then it calculates the distance between every data point and the cluster closest to it. This distance is used as a probability for what data point will be selected as the next cluster, with a larger distance meaning a larger probability of being selected. This process repeats the second and third step until all k clusters have been chosen. While this causes a slowdown in the initialization stage overall it causes the k-means algorithm to be overall faster and more accurate [4].

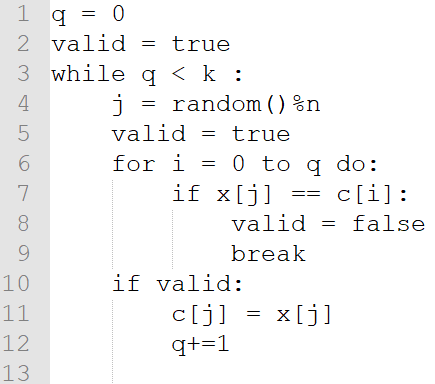
For the convergence stage there are two main algorithms to look at, Lloyd’s algorithm and Elkan’s algorithm. Lloyd’s algorithm is the original k-means algorithm and is what both Elkan’s and Hamerly’s algorithm is based on. Lloyd’s algorithm is sometimes called naïve k-means as it is essentially the brute force approach to clustering. For each point it calculates the distance between that point and every cluster and assigns the point to the closest cluster. After doing this for every point then for each cluster it calculates the average location of all the points assigned to a cluster and sets that to be the new cluster location. It repeats this process until convergence and the clusters stop moving. It is for this reason why k-means is a NP hard problem. As assigning all the points to a cluster takes N\*K and this can run for a long time depending on the initial cluster location and distribution of the data points. Elkan’s algorithm attempts to improve on Lloyd’s algorithm by calculating extra information that would allow it to skip over calculating the distance between a point and every cluster. It calculates three extra pieces of information. First it calculates the upper bound on the distance between a point and its currently assigned cluster. This serves to measure how faraway that cluster can move before the algorithm needs to consider that the point may need to be assigned to a new cluster. Next it also calculates the lower bound between a point and every other cluster. This lower bound says how close a cluster needs to be to a point before it is worth considering assigning the point to that cluster. Lastly the algorithm calculates the distance between each cluster. With these three pieces of information Elkan’s algorithm is able to avoid doing distance calculations when assigning points to clusters. While these three things do take time to update it ultimately saves time over Lloyd’s algorithm as the distance calculation is more costly than these updates, especially when in higher dimensions [5].

Lastly there are the alternative methods to calculating k-means that deviate from Lloyd’s basic algorithm significantly. While there are many others I will only be talking about the kd-tree method as kd-trees are mentioned and tested against in Hamerly’s paper. This method works by first placing all the data points into a kd-tree. Then given a set of initial cluster locations, each node in the kd-tree is assigned a set of candidate clusters. Each of these clusters are a viable cluster that a node or the points associated with a node further down the tree could be assigned to once convergence has been reached. Since the kd-tree is based around the data points and not the clusters the kd-tree structure does not need to be updated each iteration, only the set of viable clusters for each node needs to be updated. This allows the kd-tree method to run faster the Lloyd’s algorithm [6]. However as shown later in the Hamerly paper the kd-tree method fares poorly in higher dimensions [1].

**3. Technical**

For this section I will go over the three main parts of the code I have written, the cluster initialization method, Lloyd’s Algorithm, and Hamerly’s algorithm. To initialize the clusters I chose to make use of the Forgy method. This method of random initialization worked well as it was simple to implement and allowed for easier testing of my algorithms. To search for issues with in the clustering algorithms I could generate a variety of starting positions to look at. If there are significantly more data points than there are clusters, as is the case with the testing done here, Forgy’s method will run in O(2k) time which is faster than the O(n) time the Random Partition method will run in for n and k used in the tests done here. While this does make Forgy’s method slightly slower than kmeans++, which will run in O(log k) [4], Forgy’s method is simpler to implement and can allow for the generation of more sample output as mentioned before.

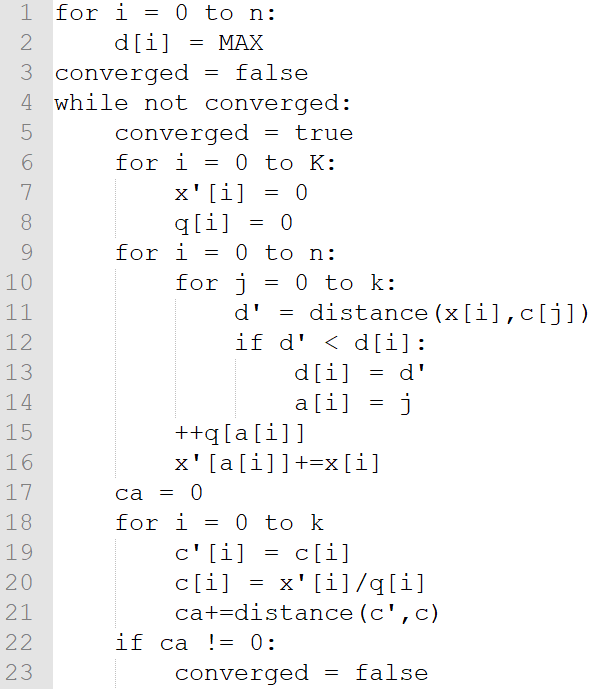
**3.1 Forgy Method**



Pseudo-Code 1 Forgy's Method

For the Forgy method the process of generation initial cluster locations is simple. Starting at q= 0 the algorithm generates a random number between 0 and the number of data points available, n is the number of available data points. Then the algorithm compares the newly generated data point to every previously accepted cluster location. If this new data point does not match any of the starting cluster locations then it is added to the array of cluster locations and q is increased. This process repeats until k, the desired amount of clusters, have been generated. Given a good random number generator the starting cluster locations should be evenly spread across the available data points.

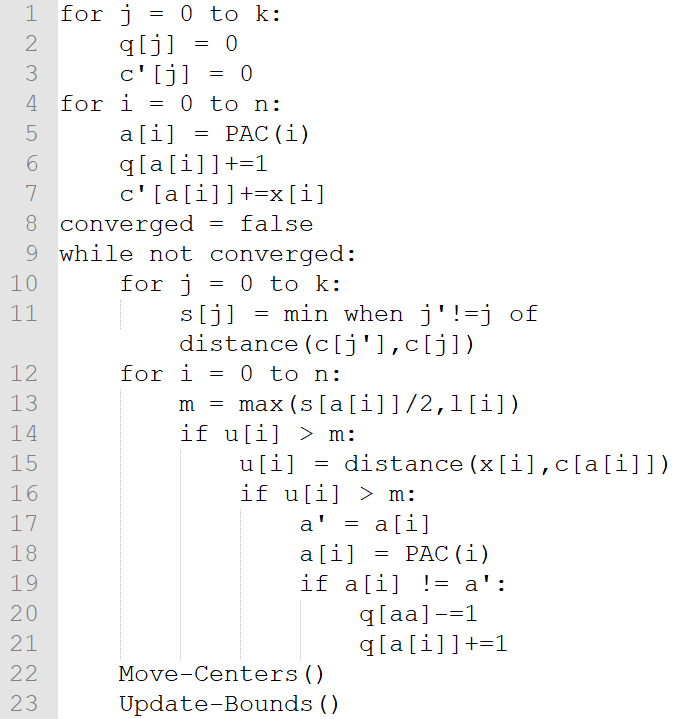
**3.2 Lloyd’s Algorithm**



Pseudo-Code 2Lloyd's Algorithm

For Lloyd’s algorithm the algorithm starts off by setting the initial value between a point and the cluster it is assigned to be the largest possible value. The algorithm then enter the main loop which will only exit once Lloyd’s algorithm has reached convergence. The algorithm will assume convergence has been reached until it is disproven later in the algorithm. At the beginning of the while loop the algorithm sets the distance current weight of the cluster to be 0, this is stored in x’ and is just the sum of all the points assigned to that cluster, and sets the number of points assigned to each cluster to be 0, this is stored in q. As these values increase as points are assigned to a cluster the algorithm can use that information to calculate the new cluster center. Next for every point and every cluster the algorithm calculates the distance between the point and cluster. If the distance is less than the distance between the point and the cluster it is currently assigned to then that point is assigned to that new cluster and has its distance updated. This deals with the initialization case since the distance between a point and its cluster would be the maximum possible value, as any graph that includes the case where the distance between a point and its closest cluster is that maximum value will also have the distance calculations between that point and every other cluster overflow and cause inaccurate results. This calculation also deals with the case were a cluster has moved cluster to a point and now should have that point reassigned to it as now the distance between the new cluster and the point will be smaller than the distance between the point and the old cluster. After the algorithm as assigned a point to its cluster q and x’ will be updated to reflect this. Once this has been done for all points the algorithm moves on to updating the cluster locations and testing if convergence has been reached. For every cluster the old cluster location is stored into c’ from c. Then the new cluster location is found by dividing x’ by q. x’ contains the location of all points assigned to a cluster and q contains the number of points assigned to that cluster so by doing c = x’/q the algorithm is just taking the average location of all the points. Once the average location of the points has been found and stored into c, the algorithm calculations the distance between c’ and c and sums it with the current value of ca. If ca is equal to 0 then that means that for every cluster its previous location and its current location are the same. If this is true then Lloyd’s algorithm has reached convergence and can exit. Once the algorithm is complete the program makes a check if it is in testing mode. If it is then the program checks how long it has been running and how much memory it has used and prints that. If the program is not in testing mode then it prints out the cluster locations, then the data points and what cluster they are assigned too.

**3.3 Hamerly’s Algorithm**

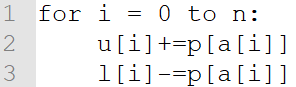


Pseudo-Code 3 Hamerly's Core Algorithm

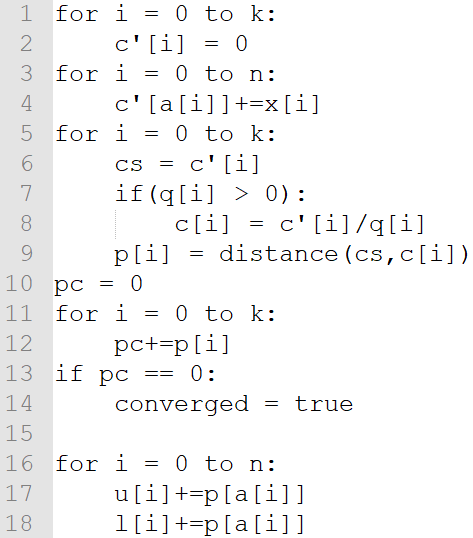
Hamerly’s algorithm is a direct improvement upon Elkan’s algorithm, instead of calculating a lower bound between a point and every cluster as done in Elkan’s algorithm Hamerly instead only chose to calculate the lower between a point and the second closest cluster. Hamerly’s algorithm will now calculate the distance between a point and every cluster if this single lower bound test fails, instead of Elkan’s method of doing multiple tests and only calculating the distances as needed. While this may cause Hamerly’s algorithm to do more distance calculations over all it heavily reduces the amount of calculations needed to be done in between each iteration of assigning points to each cluster. Hamerly shows that his algorithm will skip doing any distance calculations from 95%~80% of the time while Elkan’s will skip doing any distance calculations 56%~20% of the time on real data sets and 55%~0% on synthetic datasets. While this shows Hamerly’s algorithm can be a vast improvement over Elkan’s algorithm, Hamerly does not show the cases for when Elkan’s algorithm only makes one distance calculation, or two, or three, etc. However Hamerly’s algorithm is still an improvement as shown later.

For the pseudo code for Hamerly’s algorithm I have included three functions which I will explain after the core loop. In short PAC(i) returns the cluster number that data point i should be assigned to and updates the lower bound, l[i], and the upper bound, u[i]. Mover-Centers() tests for convergence by moving the centers of the clusters and seeing how far they moved. Update-bounds updates the upper and lower bounds as needed. Lines 1-8 are initialization code, the algorithm sets the number of points assigned to a cluster, q[j], to zero and sets the weight of the cluster, c’[i], to zero as well. Likes 4 to 7 the algorithm brute force assigns points to a cluster, a[i] holds what cluster data point i is assigned too. So q[a[i]]+=1 is increasing the number of data points that cluster a[i] has, and c’[a[i]]+=x[i] is similarly increasing the weight of cluster a[i]. Lines 9 to 23 are the core loop of Hamerly’s algorithm and will run until the algorithm reaches a convergence. Lines 10 and 11 are finding the inner cluster distances, for each cluster it finds the closest cluster to it (not including itself) and stores that into s[j]. Now for every point the algorithm checks if that point might need to be reassigned. First it checks for the larger of two values between s[a[i]]/2 and l[i]. s[a[i]]/2 is the distance between the cluster a[i] and the midpoint to the cluster closest to a[i]. Any point that is with in this distance is guaranteed to be assigned to the cluster a[i] while any point outside this distance may not be assigned to this distance. l[i] by default is the distance between a point and the second closest cluster to it, l[i] will slowly decrease as this cluster moves as the algorithm will assume this cluster is always moving towards the data point i. If a data point ever falls within l[i] then the algorithm now must consider the possibility that the data point could be reassigned to l[i] or even another cluster. Once the algorithm as the max between s[a[i]]/2 and l[i] it will compare it to u[i]. u[i] is the upper bound and by default is the distance between the data point i and the cluster it is assigned too. Much like l[i], u[i] will be increased in Update-Bounds() as the clusters move with the algorithm assuming that a[i] is always moving away from the data point i. If u[i] is the larger value that means it is possible that the cluster locations have changed enough for a[i] to no longer be the closest cluster to the data point i. Next the algorithm recalculates u[i] to be the distance between the data point and its assigned cluster. This step will only occur for points that are on the edge of a cluster’s range. If a data point is right next to a cluster then u[i] will be extremely small and it is highly unlike that l[i] or s[a[i]]/2 will ever be smaller than u[i] so the algorithm can skip doing any distance calculations for that data point. After this the algorithm checks the upper bound again against l[i] or s[a[i]]/2 and if u[i] is still larger than it is highly likely that the point needs to be reassigned and the algorithm will call PAC() to do a distance calculation between the point and every cluster. If the point is assigned to a new cluster than the number of points owned by both the old and new cluster are updated.

Moving the centers in Hamerly’s algorithm works similarly to how it was done in Lloyd’s algorithm. For every cluster I sum up the weights of all the points and take their average to find the new cluster location. The algorithm also stores how far each cluster has moved to use later in the Update-Bounds() function in p[i]. If no cluster has moved then the algorithm as reached convergence and will exit soon. This is also one place where my code as a slight deviation from Hamerly’s pseudo code. In Hamerly’s implementation he updates c’ only when a point has been moved, this would go right after like 22 in the main code in the same if block. As opposed to summing up all the weights again as done in like 1 to 4. I did this as for some reason Hamerly’s method was producing incorrect results, I believe the floating point math was under flowing but I am unsure. This method runs O(n+k) time and in the worst case Hamerly’s will run in the same time, however in the average case Hamerly’s method of calculating the new cluster centers will run about 80% faster.



Pseudo-Code 4 Update-Bounds()



Pseudo-Code 5 Move-Centers()

To update the bounds the algorithm simple changes the upper and lower bound by the amount that the cluster the point has been assigned to has moved. This is also a slightly deviation from Hamerly’s Algorithm. In Hamerly’s algorithm the larger and second largest value for p[i] is found and l[i] is decreased by one of those values instead of p[a[i]]. However I found that Hamerly’s method was producing an incorrect value when doing it by the method as described in Hamerly’s pseudo code.

Finally, in my program I check if it the algorithm is in testing mode or not. If the algorithm is in testing mode it will calculate the memory usage and speed of program and print those results. If the program is not in testing mode it will simply print the results.

**4. Evaluation**

To evaluate my code I will be making use of two data sets that were both used on Hamerly’s evaluations as well. First is a synthetic data set containing 1,250,000 data points, this data set is a uniform hypercube distribution in two dimensions. Since this is a random uniform data set this should be the slower of the two tests to run as k-means algorithms with pruning methods tend to be less effective on data that is not clustered. Second is a data set that Hamerly got from the Elkan’s algorithm paper and is a non-synthetic data set containing 100,000 data points, this data set is a 10 by 10 grid of Gaussian clusters. For my own use while testing the functionality and correctness of the code I have also included a simple python program that generates 1000 random data points in two dimensions. I have also created an implementation of Lloyd’s algorithm to compare the results from Hamerly’s algorithm against and to compare their speed and memory usage. Lastly I have also created a simple python program that displays the output from both my implementation of Hamerly’s and Lloyd’s algorithm. This program displays up to 1000 data points and color codes them based on what cluster they have been assigned to, up to a max of 10 clusters.

Both the implementation of Hamerly’s algorithm and Lloyd’s algorithm are single-threaded C++ codes. They were compiled with g++ version 4.8.5 and c++ version 98. They were compiled and run on the FSU linprog server.

To measure the performance of my programs I am measuring them by the two statistics also measured in Hamerly’s paper. First I measure the user CPU time using the getrusage() function. Second I measure the memory footprint using /proc/[PID]/statm. When measuring these things I avoid printing out the results, while this had no effect on the memory footprint I found that it had significant effect on the CPU time and represented the majority of the programs’ run time in some cases. Since I am testing the run time of Hamerly’s algorithm and not the speed of printing to terminal I decided to leave it out when testing for CPU time.

For my tests I will run my programs for a cluster size of 3, 20, 100, and 500 just as done in the Hamerly paper. When testing I shall be using a common seed value for the srand function for use when determining the starting cluster locations so that each run of the program will use the same starting location for the clusters. This is done as the starting location of clusters can impact the speed at which the program runs. For each data set, cluster size, and program I will run the test three separate times and present the results as the average of those three times. The full results will be available in the attached GitHub.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | Average total user CPU Seconds | | | |
| Uniform Random | | K=3 | K=20 | K=100 | K=500 |
| N = 12500000  D = 2 | Hamerly | 2.29759 | 10.9967 | 17.0942 | 44.3246 |
| Lloyd | 2.3549 | 13.9456 | 27.5589 | 358.453 |
| Birch | |  |  |  |  |
| N= 100000  D=2 | Hamerly | 0.173740 | 0.437830 | 0.861558 | 2.04486 |
| Lloyd | 0.295347 | 0.487605 | 2.71865 | 9.98092 |
| Table 1: This table shows the time, in user CPU seconds, that it took for both algorithms to run as gathered by the getrusage() function. The first data set is the synthetically generated dataset and the other is the dataset used by Elkan. In all cases we find that Hamerly’s algorithm was faster than Lloyd’s Algorithm. This speed is slight at low k values but is very significant at high k values. As expected both algorithms preform worse for the random synthetic data set however Lloyd’s algorithm deals with the random data significantly worse than Hamerly’s algorithm when compared to the non-synthetic dataset. | | | | | |
|  |  | Average memory usage in Kb | | | |
| Uniform Random | | K=3 | K=20 | K=100 | K=500 |
| N = 12500000  D = 2 | Hamerly | 11307 | 11320 | 11301.67 | 11309 |
| Lloyd | 8862.33 | 8855 | 8864.33 | 8877.33 |
| Birch | |  |  |  |  |
| N= 100000  D=2 | Hamerly | 1191 | 1191 | 1190.67 | 1196.67 |
| Lloyd | 993 | 992.33 | 1012.67 | 1004.67 |
| Table 2: This table shows the memory usage, in kb, as determined by the resident set size found in /proc/[pid]/statm . As in table 1 the first data set is the larger synthetic dataset and the other is a smaller data set used by Elkan. In all cases Hamerly’s algorithm makes use of more memory than Lloyd’s algorithm. This is to be expected as Hamerly’s algorithm stores more information about the state set such as the upper and lower bounds. Both algorithms use slightly more memory at higher k values however they both use significantly more memory at a higher n value. | | | | | |

**5. Conclusion**

While overall my implementation of Hamerly’s algorithm and Lloyd’s algorithm ran significantly faster, due to the strength of my testing platform, the efficiency of Hamerly’s algorithm over Lloyd’s algorithm is obvious. At the lowest end Hamerly’s algorithm was slightly faster than Lloyd’s algorithm however once larger values of k are used Hamerly’s algorithm was up to 8 times faster than Lloyd’s algorithm. All while Hamerly’s algorithm only used about 20% more memory than Lloyd’s algorithm. This low memory usage is part of the reason why Hamerly’s algorithm is so efficient, as it has a low I/O cost much like Lloyd’s algorithm. Over all Hamerly’s algorithm does less calculations by avoiding the inner point to all cluster distance calculations, and what extra calculations Hamerly’s algorithm does introduce they are far less costly than a distance calculation and over all save time. While my implementation of Hamerly’s algorithm may be slightly slower than the official version it still shows just how fast Hamerly’s algorithm is. Hamerly’s algorithm is a fast and relatively easy to implement k-means algorithm. It is a resounding improvement over Lloyd’s algorithm and Hamerly’s algorithm should be used in most cases when possible.

**6. Submission**

All code, data sets, and supporting files can be found here:

https://github.com/chdraper16/kmeans-5725

**7. Sources**

[1] Greg Hamerly. 2010. Making k-means even faster. *Proceedings of the 2010 SIAM International Conference on Data Mining* (2010). DOI:http://dx.doi.org/10.1137/1.9781611972801.12

[2] David Arthur and Sergei Vassilvitskii. 2006. How slow is the k-means method? *Proceedings of the twenty-second annual symposium on Computational geometry - SCG 06* (2006). DOI:http://dx.doi.org/10.1145/1137856.1137880

[3] Greg Hamerly and Charles Elkan. 2002. Alternatives to the k-means algorithm that find better clusterings. *Proceedings of the eleventh international conference on Information and knowledge management - CIKM 02* (2002). DOI:http://dx.doi.org/10.1145/584887.584890

[4] David Arthur and Sergei Vassilvitskii. 2006. k-Means: The Advantages of Careful Seeding. *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2006* (2006).

[5] Charle Elkan. 2006. Using the Triangle Inequality to Accelerate k-means. *Proceedings of the Twentieth International Conference on Machine Learning* (2006). https://www.aaai.org/Papers/ICML/2003/ICML03-022.pdf

[6] T. Kanungo, D.m. Mount, N.s. Netanyahu, C.d. Piatko, R. Silverman, and A.y. Wu. 2002. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, 7 (2002), 881–892. DOI:http://dx.doi.org/10.1109/tpami.2002.1017616