Hamerly's Algorithm

Hamerly's algorithm is an optimization of the standard algorithm and takes advantage of the fact that it is not necessary to recalculate the distance of each point from the centroids each time the centroids are repositioned. Using the **triangular inequality** it is, in fact, possible to identify critical points, that is, points for which it is necessary to recalculate the distance to each centroid to find the closest one. In contrast, for the other points, the class to which they were assigned remains the right one even if the centroid has been moved.

Triangular inequality

Given 3 points A, B, C $\in \mathbb{R}^2$ then

$$d(A,B) \leq d(A,C) + d(B,C) \tag{1}$$

$$d(A,C) \leq d(A,B) + d(C,B) \tag{2}$$

$$d(C,B) \leq d(C,A) + d(B,A) \tag{3}$$

Where $d(\alpha, \beta)$ is the distance between the points α and β. That is, the distance between any two points is less than the sum of the distances between the first point and the third and the second point always with the third one.
This inequality can also be interpreted geometrically: referring to Figure 1, any one side of a triangle is definitely less than or equal to the sum of the other two sides.

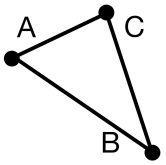

Figure 1: *3 points in space forming a triangle*

Triangular inequality and k-means

For simplicity, let us assume that we have only one point and two centroids (Figure 2).

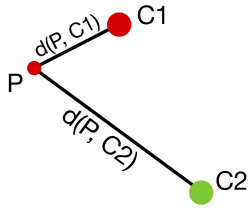

Figure 2: *Example with 2 centroids and one point*
At this point we assume that the centroids shift (Figure 3), because of the other points (which are not represented).

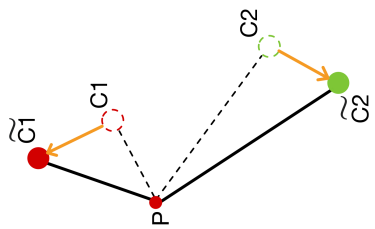

Figure 3: *Example after the centroids have shifted*

Thanks to the triangular inequality, we can state that the distance between P and $\tilde{C_1}$ is:

$$d(P,\tilde{C_1}) \leq d(P,C_1) + d(C_1,\tilde{C_1}) \tag{4}$$

And in the same way:

$$d(P,\tilde{C_2}) \leq d(P,C_2) + d(C_2,\tilde{C_2}) \tag{5}$$

In words, we can say that the distance between point P and one of the centroids is definitely less than the old distance plus the distance traveled by the centroid.
Taking advantage of this result, we can improve the k-means algorithm as described s follow. The first step remains mostly unchanged: it consists of assigning to each point the nearest centroid, and to do this it is necessary to calculate all the distances between the point and the centroids. The difference with the standard algorithm is that in this case, for each point, we have to save the distance (μ) to the nearest centroid and also the distance (l) to the second nearest centroid.
Assuming, for example, that for a point (P), the nearest centroid is C_1 we have that

$$\mu = d(P,C_1) \tag{6}$$

$$l = \min_{C_i \neq C_1} d(P,C_i) \tag{7}$$

After the first step, the centroids are repositioned and again the algorithm is almost identical: eqs 2 and 3 are used to compute the new position of the centroids, but unlike before, the distance (d_i) traveled by the i-th centroid, the distance traveled by the centroid that moved the most (d_1^{max}) and the distance traveled by the second centroid that moved the most (d_2^{max}) are also saved.
At this point, the parameters μ and l are updated for each point, as follows.

$$\tilde{\mu} = \mu + d_i \tag{8}$$

If the closest centroid is also the one that moved the most then.

$$\tilde{l} = l - d_2^{max} \tag{9}$$

Altrimenti

$$\tilde{l} = l - d_1^{max} \tag{10}$$

These updates are justified by the triangular inequation. In fact, we know that after the centroids have moved, the point cannot be farther than $\tilde{mu}$ from the nearest centroid and at the same time cannot be closer than $\tilde{l}$ from the second nearest one.
Thanks to the previous step, it is possible to identify as **critical the points for which** $\tilde{\mathbf{mu}} \geq \tilde{\mathbf{l}}$, that is, the points for which the assigned centroid may not be the nearest
The next step is, then, to iterate for all points but recalculating the distance to the nearest centroid only for the critical ones. Moreover, we can optimize this process even more since, once a critical point has been found, it may be sufficient to recalculate only $\tilde{\mu}$, i.e., the distance to the nearest centroid, and then recheck the condition $\tilde{mu} \geq \tilde{l}$ and only if the condition is still verified then it will be necessary to carry out the full calculation of the distances.
Finally, the centroids are repositioned and the procedure is repeated until the condition of convergence is reached.

Number of iterations

In the case of Hamerly's algorithm, the number of iterations required to complete the classification is:

$$kN_cN_p + mk(\phi N_p N_c + (1-\phi)N_p) \tag{11}$$

Where ϕ is the percentage of critical points to the total number of points.
The improvement over the standard algorithm lies in the fact that only the first time it is necessary to compute all distances, whereas in subsequent iterations, for most points, there is no need to recompute the nearest centroid.

Parallelize the Hamerly's algorithm

The Hamerly's algorithm has three main sub-tasks: assignation of the points to a class, update of the centroids, update the lower and upper bounds of all points.

Assignation of the points to a class

In this phase, each point will be assigned to a centroid which will represent its class. If it is the first iteration, this can be easly parallelizable by simply partitioning the points between the threads. Each thread will then goes through its set of points and find the minimum distance between the point and the centroids and therefore will assign to point to that class. In this first ieration the work load is equally balanced across the threads because the work that has to be done for each point is the same.
This is not true from the second iteration, because in this case the only points for which the minimum distance is calculated are the critical points. Therefore it is not enough to partion the points between the threads because criticals points could be not equally distribuited across the point set and because of this some threads could have more work to do compared to others. A load balancing tequiniche has to be used.
Whatever the iteration, another important task is done during this phase: tracking the points assigned to a class. Suppose for example that we have 3 class and the dimensionality of the data is 2, then one of the private paramenter of the hamerly class will be a 3x2 matrix (called average_per_class) and an array with a length of three (called points_per_class). The first one tracks, for each class (average_per_class[i]) the sum of all coordinates of the points that, during this phase, have been assigned to the i-th class. The points_per_class instead track the number of points assigned to each class. Notice that when a point is assigned to a class (say the i-th class), its coordnate are added respectively added to average_per_class[i][0] and average_per_class[i][1] but if the point was previously assigned to another class, its coordinates have to be removed from the previous class.

Update of the centroids

This is the phase where the centroids are moved. Also this step can be easly parallized by simply partitioning the centroids between the threads which, for every of its centroid will divide the sum of all coordinates of the points assigned to that class by the number of points. These information are saved in the average_per_class matrix and the points_per_class array.

The work load for every centroid is the same, therefore no special cautions are needed when parallelizing it. During this step, we must also save the distances taken by each centroid (in an array called cdistaces), the maximum distance taken from the centroid (in a variable maxdist) and its index (maxdist_index), the second maximum distance taken from the centroids.

Update the lower and upper bounds of all points

During this step, the lower and upper bound of each points are updated using the information saved during the "Update of the centroids". This is where the balancing algorithm is applied and the result of this phase is an array (called criticals) containing the address of all critical points that have been found.