Mean Shift Clustering

Francesco Villi
francesco.villi@stud.unifi.it

Abstract

The mean shift algorithm is a non parametric clustering technique which does not require prior knowledge of the numbers of cluster.
I'm going to compare a parallel and sequential version implemented in C++, measuring the speed during fitting and centroids computation.
I tested both implementations varying number of points and, only for the parallel version, number of threads.

1. Introduction

The purpose of this project is to compare the performance of a parallel and a sequential implementation of the mean shift clustering algorithm. Mean shift clustering is a popular technique for identifying clusters in data sets, and is known for its ability to find clusters of arbitrary shape. The algorithm seeks to find the centers of each cluster in the dataset. The algorithm works by shifting each data point towards the cluster center, until convergence. The process is iterated until the points no longer move significantly. In this way, the algorithm is able to find clusters of arbitrary shape, and does not require the number of clusters to be specified in advance.
The unique hyper-parameter is the bandwidth that conditions the result.
Given n data points $x_i \in R^d$, bandwidth $h > 0$ and kernel function $K_h(x)$; the shift vector of point $x \in R^d$ is:

$$m_{h,K}(x) = \frac{\sum_{i=1}^{n} x_i K_h(\|x - x_i\|^2)}{\sum_{i=1}^{n} K_h(\|x - x_i\|^2)} - x$$

The kernel function $K_h(x)$ is linear:

$$K_h(x) = \begin{cases} 1, & \text{if } x \leq h \\ 0, & \text{otherwise} \end{cases}$$

The algorithm ends when the norm of the highest shift vector is close to zero.

2. Structure

The structure of the code is:

- **MeanShift** defines interface for clustering and it contains:
  - *fit* function that finds all centroids and assigns label at each point related to the closest centroid.
  - *predict* function that returns cluster number for one given point.
- **MeanShiftParallel** and **MeanShiftSequential** classes that implement the algorithms respectively in parallel and sequential version.
- **Point** class that contains the coordinates of single element yielded by the dataset. The main functions are:
  - *division* is an inline function that permits to do a division between point and scalar. This is an inline implementation of *operator* $\backslash$ that increases speed
  - *calc_L2_norm_approx* is an inline static function used to perform the L2 norm approximated between two points because it doesn't compute the square root.
- **ClusterCentroids** class that contains a vector of all centroids, it is necessary in the parallel implementation to define a reduction clause. It is useful for not adding the same centroid twice.

I used inline functions to increase the general performance. To compute the shift vector for each point we have to calculate all distances and get them into kernel function $K_h(x)$ that returns weight related to the distance, so we compute the shift vector. After that we shift all points to the mean, we compute the norm and store the highest because we will stop the iterations when the highest norm is less than the threshold set to $0.1 * bandwidth$. When all points reach the convergence we can search all centroids removing duplicates; to do it we scroll the list of updated points and add the new centroid in a separate array.

The dataset has been created using python script; in details I generated 4 isotropic Gaussian blobs with a standard deviation of 0.6 and an increasing number of points (10.000,25.000,100.000,200.000).

2.1. Sequential version

In this version we have 2 main steps:

1. Compute mean vector for each point considering the region around it; so given a point x, compute the approximated L2 norm between x and all other points and apply kernel function $K_h(x)$ to get the weight. After the computation of all mean points, I update the list of points with new values.

2. After the convergence of all points to the peak, I remove all duplicated points.

In this implementation to find the distance between points there are 2 nested loops where I can compute the distance between all points. In detail the index in the inner loop starts not from zero but from the value of the first index loop, so that I compute only once both distance and kernel. In the outer loop I normalize the shift vector relative to the point i-th. After that I compute the shift vector norm for each point, saving the highest, and then I update the points with the new values. Finally I check if the highest norm is less than threshold and, if it is true, the algorithm can go ahead, otherwise we need another iteration. The last step is about to remove all points duplicated returning a vector that contains only centroids that will represent the clusters, then for each point in the dataset I assign a label related to the closest centroid. The computation of centroids and the label assignment has been done in the same loop.

2.2. Parallel version

In this version I tried to speed up the sequential version making some changes. In particular I removed the dependence between the two loops when I compute all distances and apply kernel function because in this way I can avoid the use of atomic and critical section by favoring parallel for to calculate shift vectors, and also the reduction clause when computing the highest norm. Also the computation of centroids is slightly different so I have to remove the dependence of centroids vector because the update of centroids in sequential version is done in the same loop, so to implement the same algorithm i have to manage it using a critical section but it is not efficient. I chose to use two loops, one for compute all centroids using for directive with a custom reduction clause that uses ClusterCentroids, and the other to assign a label for each point related to the closest centroid.

2.3. Other implementation details

I used a L2 norm approximated to increase the speed:

$$L2_{app}(x, y) = \sum_{i=1}^{dim} (x_i - y_i)^2$$

To maintein correctness during computing of the kernel I raise bandwidth h to the power of 2: Instead of

$$\sqrt{\sum_{i=1}^{dim} (x_i - y_i)^2} \leq h$$

I used

$$\sum_{i=1}^{dim} (x_i - y_i)^2 \leq h^2$$

The inequation is still valid because $h > 0$ and the norm is non-negative.
Another optimization is the use of vectorized operations when update the mean point and the use of inline functions.

3. Environment settings

All tests have been run in a PC with intel core i7-11800H, 16GB of RAM DDR4. The compiler used is gcc version 12.2.0 and the library to parallelize is OpenMP. All test code has been run in relase mode. To generate the points and check the correctness I wrote a script python, you can find it in the gitHub repository. Enabled flags gcc: -O3 -mpc32 -ffast-math -ftree-vectorize -floop-parallelize-all. Due to the speed of the i7-11800H the calculation of centroids is too fast and often the time measured was zero, so I used an old PC with i7-2700H and 8GB DDR3 RAM to measure the calculation time of centroids.

4. Results and Discussion

Nr. points	Parallel	Sequential
10.000	0.078	0.344
25.000	0.49	2.317
100.000	8.025	39.977
200.000	34.436	210.859

Table 1. Mean shift algorithm speed tests (values in seconds)

Nr. points	Parallel	Sequential
10.000	0.001	0.002
25.000	0.002	0.004
100.000	0.003	0.009
200.000	0.015	0.024

Table 2. Centroid speed tests (values in seconds). Tested using intel i7-2700H and 8GB RAM

Nr. points	Nr. threads			
	4	8	12	16
10.000	0.187	0.125	0.090	0.078
25.000	1.162	0.781	0.626	0.49
100.000	21.125	12.30	9.939	8.025
200.000	93.1	51.486	40.94	34.436

Table 3. Speed tests varying number of threads (values in seconds)

In table 1 we can observe as increasing the number of points the performace is better, the speed up is around 5.
The result about computation time of centroids is in table 2. We observe the speed up relative to the centroids calculation is around 2.
The hotspot of this algorithm is the computation of mean vectors and not the computation of centroids that doesn't impact significantly the execution time. Results confirm that the speed is increased in the parallel version; in particular increasing the number of points, while the computations of centroids is not relevant considering the total amount of time but also in this case the parallel version yields better result. In table 3 we can observe the performance increment varying the number of threads keeping fixed the number of points.
Computing L2 norm approximated helps to reach good result saving much time because we don't calculate the sqrt. It is evident that the two versions have a different implementation to use the most efficient way to compute mean shift vectors and centroids, in particular in the parallel version we try to make independent each loop and avoid the use of critical or atomic clauses reducing the synchronization between threads and improving performance.
Instead of in the sequential version we try to reduce the complexity of the algorithm and avoid useless computations e.g. sequential version computes only once the distance between two points and kernel function, while in the parallel version the distance and kernel function is calculated twice.

5. Conclusion

In conclusion, this project has demonstrated that using parallel computing on CPU to perform mean shift clustering results in a significant improvement in performance compared to using CPU in a sequential manner. The parallel implementation was able to process the dataset much faster. This highlights the importance of considering parallel computing when dealing with large datasets and complex algorithms. In future work, it would be interesting to explore the use of even more powerful parallel computing resources, such as GPUs, to further improve the performance of mean shift clustering.