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.
The purpose is to compare the performance of parallel and sequential implementation of the mean shift clustering implemented in C++.
I tested both implementations varying number of points and, only for the parallel version, number of threads.

1. Introduction

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 center 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 used for not adding the same centroid twice.

The optimization of a computational process was achieved through the utilization of inline functions. To calculate the shift vector for each data point, the distances were computed and input into the kernel function, $K_h(x)$, which returns a weight based on distance. The shift vector was then computed and all data points were shifted to their mean. The norm was computed and the highest value was stored, the iteration is stopped when the value became less than the defined threshold.
Upon convergence of all data points, centroids were determined by scanning the updated points list and adding unique values to a separate array.

The dataset used was generated using a Python script and consisted of 4 isotropic Gaussian blobs with standard deviation of 0.6 and varying 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 to get the centroid for each cluster.

In this implementation, the calculation of the distance between data points is performed through the utilization of two nested loops. The inner loop have its index starting from the value of the first index of the outer loop, thus reducing the number of computations for both distance and kernel calculation. Then in the outer loop the shift vector has been normalized with respect to the i-th point, and the norm of the shift vector has been computed and saved for each point, with the highest value being recorded. Subsequently, the points are updated with the new values.
A check is performed to determine if the highest norm is less than the defined threshold; if it is, the algorithm continue, otherwise another iteration is required. The final step involve the removal of duplicate points to return a vector containing only the centroids, which represent the clusters. In the same loop, each data point is assigned a label related to the closest centroid. The computation of the centroids and the label assignment are performed simultaneously.

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 and added a private support variable to calculate the mean point; these improvements are useful for execution speed because when I computed all distances and applied kernel function I can avoid the use of atomic and critical section by favoring both *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 removed the dependence of the vector that contains centroids because their update in sequential version is done in the same loop. To implement the same algorithm I should have to manage it using a critical section but it is not efficient, then I chose to use one loop to compute all centroids using *for* directive with a custom reduction clause that uses ClusterCentroids (it permits to calculate the centorids for each chunks and merge them excluding duplicates), and another loop parallelized 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.

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

Table 1. Centroid speed tests (values in seconds). Tested using intel i7-2700H and 8GB RAM

Nr. points	Nr. threads				
	1	4	8	12	16
10.000	0.344	0.187	0.125	0.090	0.078
25.000	2.317	1.162	0.781	0.626	0.49
100.000	39.977	21.125	12.30	9.939	8.025
200.000	210.859	93.1	51.486	40.94	34.436

Table 2. Speed tests varying number of threads (values in seconds)

Nr. points	Speed-up
10.000	4.4
25.000	4.7
100.000	4.9
200.000	6.1

Table 3. Speed up (values in seconds)

4. Results and Discussion

In table 2 we can observe as increasing the number of points the performace is better, the speed up ranges from 4 to 6 as we can see in table 3.
The result about computation time of centroids is in table 1. 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 2 we can observe the performance increment varying the number of threads and points.
Computing L2 norm approximated helps to reach good result saving much time because we don't calculate the square root.
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 between two elements and the kernel function are 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.