Course Project: Mean Shift clustering

Simone Casini
E-mail address
simone.casini3@stud.unifi.it

Giacomo Ravara
E-mail address
giacomo.ravara@stud.unifi.it

Abstract

The mean shift algorithm is a nonparametric clustering technique that assigns the data points to the clusters iteratively by shifting points towards the mode (mode is the highest density of data points in the region). Mean-shift algorithm has applications in the field of image processing and computer vision. The downside to Mean Shift is that it is computationally expensive, but it is embarassingly parallel. This paper shows the results of parallelization, applied to image segmentation.

1. Introduction

The mean shift algorithm is a powerful general nonparametric mode finding procedure. It is based on ideas proposed by Fukunaga and Hostetler [1]. It can be described as a hill-climbing algorithm on the density defined by a finite mixture or a Kernel Density Estimate.The KDE is a non parametric technique to estimate the underlying PDF to the probability distribution which generated the dataset. The KDE idea is exploited in this way: for each iteration, for each point belonging to the dataset, we *shift* the point toward the direction of the nearest peak of the KDE surface (i.e. the mode) following the gradient direction. The mean shift procedure consists of two steps:

- 1. Construction of KDE surface in some feature space;
- 2. The mapping of each point to the maximum (mode) of the density which is closest to it.

Each data point is shifted to the weighted average of the data set.
The unique parameter of the Mean Shift is the bandwidth, which controls the smoothing of the theorical PDF: if a lower bandwidth models a noiser function, this will have more local maxima, i.e. more clusters. So the main advantage of the Mean Shift is that it isn't needed to specify the cluster numbers, or to be initialized, like the more popular K-Means: the only knob is the bandwidth value. Here it is reported the pseudocode about how the single iteration of the Mean Shift works.

Algorithm 1 Single Mean Shift iteration

procedure SHIFTER($Y_t, X, bandwidth$)
  for all $\mathbf{y_i}, i = 1, ..., |Y_t|$ **do**
    $\mathrm{m}_i \leftarrow$ COMPUTESHIFT($\mathbf{y}_i, X, bandwidth$)
    $y_i \leftarrow x_i + m_i$
  return $Y_{t+1} \leftarrow \{y_i, i = 1, ..., |Y_t|\}$

Algorithm 2 Shift Computing Algorithm

procedure COMPUTESHIFT($y_i, X, bandwidth$)
  $weights \leftarrow 0$
  $\mathrm{m}_i \leftarrow 0$
  for all $\mathbf{x_j}, j = 1, ..., |X|$ **do**
    $w_i \leftarrow K(\|y_i - x_j\|)$
    $\mathrm{m}_i \leftarrow m_i + w_j \mathrm{x}_j$
    $weights \leftarrow weights + w_j$
  return $\mathrm{m}_i / weights$ - y_i

Algorithm 3 Sequential MeanShift algorithm

procedure MEANSHIFT($X, bandwidth, , MaxIteration$)
  $iter \leftarrow 0$
  $Y_0 \leftarrow X$
  while $t < MaxIterations$ **do**
    $Y_{t+1} =$ SHIFTER($Y_t, X, bandwidth$)
    SWAP(Y_t,Y_{t+1})
    $t \leftarrow t + 1$
  return Y_{t+1}

Algorithm 1 has $O(n^2)$ complexity cost, where n is the dataset X dimension, because for each x_i, a shift m_i is computed, computing that cost $O(n)$ (**Algorithm 2**). Y is the shifted point set. **Algorithm 1** is simply the pseudo-

code about a single iteration, so the cost of T iterations is $O(Tn^2)$. In the **Algorithm 1**, X represents the original dataset, and Y_{t+1} the shifted point set.

1.1. The Kernel Function

Algorithm 1 is reported in a manner such that highlights the computational cost, however some mathematical details misses about the kernel function $K(r)$ and the shift m_i. The mean shift formula is:

$$\mathbf{m_i} = m(\mathbf{y_i}) = \frac{\sum_{j=1}^{n} K(\|\mathbf{y_i} - \mathbf{x_j}\|)\mathbf{x_j}}{\sum_{j=1}^{n} K(\|\mathbf{y_i} - \mathbf{x_j}\|)} - \mathbf{y_i} \quad (1)$$

$$= \frac{\sum_{j=1}^{n} w_{ij}\mathbf{x_j}}{\sum_{j=1}^{n} w_{ij}} - \mathbf{y_i} = \tilde{\mathbf{m}}_\mathbf{i} - \mathbf{y_i} \quad (2)$$

It's easy to see that the shift is a vector computed by a weighted average between $\mathbf{y_i}$ and each other $\mathbf{x_j}$: the nearest points will have an higher weight, so the points belonging to a cluster (i.e. densest regions) converges to their centroid/local maximum. The most common kernel having this property and able to guarantee the convergence is the **Gaussian kernel**:

$$K(r) = \frac{1}{(\sqrt{2\pi} bandwidth)} e^{-\frac{1}{2}\frac{r^2}{bandwidth}} \quad (3)$$

Watching to the (2), it's sufficient in the **Algorithm 1** assign $\mathbf{y_i} \leftarrow \tilde{\mathbf{m}}_\mathbf{i}$.

2. Mean Shift Segmentation

Segmentation is a process that partitions an image into homogeneous regions. The segmentation algorithm is a straightforward extension of the mean shift algorithm. When all convergence points are found, clusters are built from them. The basins of attraction of the modes are recursively fused until convergence. When the mean shift procedure is applied to every point in the feature space, the points of convergence aggregate in groups that can be merged. These are the detected modes, and the associated data points define their basins of attraction. An image region is defined by all the pixels associated with the same mode in the joint domain. The clusters are separated by the boundaries of the basins, and the value of all the pixels within are set to their average. The process of delineation of the clusters is a natural outcome of the mode seeking process. After convergence, the basin of attraction of a mode, i.e. data points visited by all the mean shift procedures converging to that mode, automatically separate a cluster of arbitrary shape. The number of significant clusters present in the feature space is automatically determined by the number of significant modes detected. In the end, all points are labeled.

3. The implementation

Mean Shift is embarassingly parallel: as we can see in **Algorithm 1** each point $y_i \in Y_t$ can be distinclty processed. Let's consider two arrays Y_t and Y_{t+1}, allocated in memory, plus X the (read-only) array of the original dataset: the first is where the input is read, the second where the output is written. If we assign to each thread an index $i : 1, .., n$, during the execution of ComputeShift, each element of Y_t isn't modified, but only read; each thread have its own local variable m_i (computed through a read-only procedure) and at the end of the execution each thread write in main memory the variable y_i in Y_{t+1}. This approach respects the Bernstein's conditions, because the thread have not to share any data and the points can be processed in any order. How points are actually clustered by Mean Shift there must be a thread barrier at the end of each iteration, because of the iterative nature of the algorithm. Below, is reported the pseudo code of the entire algorithm in the **Algorithm 4**.

4. Parallel version with OpenMP

The mean shift algorithm is a embarrassingly parallel work: each point perform its shifting independently from the other points. This makes it the perfect case for using the OpenMP technology. In fact with a single `pragma` command it was possible to switch from a sequential version to a parallel version.

Algorithm 4 OpenMP Mean Shift Parallel Version

procedure SHIFTER($Y_t, X, bandwidth$)

```
#pragma
for all y_i, i = 1, ..., |Y_t| do
    m_i ← COMPUTESHIFT(y_i, X, bandwidth)
    y_i ← x_i + m_i
return Y_{t+1} ← {y_i, i = 1, ..., |Y_t|}
```

Note that the only difference from the sequential version in 2 is the pragma statement. That statement is placed just before the for loop, in this way there is no need of any critical sections.

4.1. Data structures

The data structures used by the parallel algorithm are the same of the sequential version.

There is a list containing the original points that is shared among the threads. This list is never changed during the computation, so there is not need of any synchronization mechanism.

The other list, that one containing the new positions of the shifted points, is shared among the threads too, but also in this case no synchronizations are necessary because the

parallelized section is the computation of the shifting and after each shifting step there is an implicit barrier at the end of the `for` loop. At each step, each thread computes the new position of a set of points independently from the others.

4.2. Thread scheduling

Not all the points need the same number of steps to reach the final position. In the same dataset some point could converge very quickly while others could perform a larger number of shifting operations. That's why we have to make sure that each thread receives the same amount of workload.

With OpenMP it is possible to change the workload of the `for` loop assigned to each thread. By default, OpenMP uses a *static scheduling*, where the entire `for` loop is divided statically in chunks of equal size. This kind of scheduling is not optimal for this problem, because, as we have noticed before, each point perform a different number of steps depending on how fast it converges to the center of its cluster. So it could be happen that a thread finishes very soon its iterations and then it has to wait the other threads wasting computational resources. The best scheduling strategy for this algorithm is the *dynamic scheduling*, where the iterations are assigned to the threads while the loop is executing. Assigning the workload to the threads in this way should ensure that each thread will never stop its execution waiting for the others. To perform a dynamic scheduling we have to write in the `pragma` statement the clause `schedule(dynamic)`. With that directive an interation is assigned to a thread as soon as the thread has finished the computation of the previously assigned iteration.

Threads	Static	Dynamic
2	124.54s	114.49s
3	87.07s	76.29s
4	63.72s	57.20s
5	52.44s	45.72s
6	44.13s	38.25s

Table 1. Comparison between static and dynamic with 36864 points

5. Java Threads Parallel Version

As mentioned above, the implementation of Mean Shift doesn't lead to race condition, because each thread has only to read the shared original vector and it computes the *shifting* of a set of points independently from the others. So in Java implementation, each thread takes one point and shiftes it, until convergence. This operation is repeated for every point. As said before, with this implementation it could happen that the workload for each thread is not equal. So it can be used a different implementation without a static division of points between threads. With a dynamic version each point is processed by an AtomicThread. This ensures more efficiency. To avoid race condition we used an AtomicInteger variable as the index shared between threads to iterate the original points.

Algorithm 5 Java Mean Shift Parallel Version

procedure MEANSHIFT($Y_t, X, bandwidth$)
  for $t < num\_threads$ **do**
    CREATETHREADMEANSHIFT ($Y_t, X, bandwidth$)
    SHIFTER ($X, bandwidth$)
  return ShiftedPoints

Algorithm 6 Java Mean Shift Parallel Thread

procedure SHIFTER($X, bandwidth$)
  while $TRUE$ **do**
    $i \leftarrow$ GETINDEX
    COMPUTESHIFT($Y_i, X, bandwidth$)
  return ShiftedPoint

Threads	Static	Dynamic
2	135.10s	133.20s
3	93.90s	88.35s
4	77.87s	66.68s
5	64.25s	53.27s
6	56.26s	45.09s

Table 2. Comparison between static and dynamic with 36864 points

6. Results

The performance of the sequential algorithm and the different parallel versions is measured using the speedup. The speedup equals to the ratio between the time of execution of the sequential version and the parallel one, using the same input. Moreover, the speedup is computed for different numbers of threads used in the parallel algorithm. The results were produced using a Microsoft Azure Virtual Machine, with Intel Xeon E5-2690 v3 @ 2.60GHz 2.60 GHz, 6 cores.
The first tests are produced comparing the algorithms using the same input size and varying the parameter bandwidth.

Figure 1. Example of Image Segmentation, using MeanShift

BandWidth	1	
Number of points	16384	
Sequential C++	25.05s	
Sequential Java	25.24s	
Threads	OpenMP	Java
2	12.54s (1.99X)	13.07s (1.93X)
3	8.38s (2.98X)	8.62s (2.93X)
4	6.31s (3.96X)	6.52s (3.86X)
5	5.05s (4.95X)	5.25s (4.80X)
6	4.24s (5.90X)	4.47s (5.64X)

Table 3. Comparison between sequential and OpenMP/Java

BandWidth	10	
Number of points	16384	
Sequential C++	31.74s	
Sequential Java	33.34s	
Threads	OpenMP	Java
2	15.07s (2.10X)	17.62s (1.89X)
3	10.05s (3.15X)	11.70s (2.84X)
4	7.53s (4.21X)	8.83s (3.77X)
5	6.04s (5.24X)	7.11s (4.68X)
6	5.04s (6.29X)	5.99s (5.56X)

BandWidth	25	
Number of points	16384	
Sequential C++	47.01s	
Sequential Java	50.30s	
Threads	OpenMP	Java
2	23.18s (2.03X)	25.77s (1.95X)
3	15.52s (3.03X)	17.18s (2.92X)
4	11.59s (4.06X)	12.89s (3.90X)
5	9.26s (5.08X)	10.30s (4.84X)
6	7.76s (6.06X)	8.74s (5.75X)

The speedup grows linearly by increasing the number of threads, and it's almost the same when the bandwidth parameter changes. Other tests are computed comparing the algorithms with different input sizes, with bandwidth = 25.

BandWidth	25	
Number of points	4096	
Sequential C++	2.86s	
Sequential Java	3.05s	
Threads	OpenMP	Java
2	1.43s (2.00X)	1.64s (1.80X)
3	0.98s (2.92X)	1.03s (2.94X)
4	0.72s (3.94X)	0.78s (3.88X)
5	0.57s (4.97X)	0.64s (4.74X)
6	0.47s (6.03X)	0.57s (5.30X)

BandWidth	25	
Number of points	36864	
Sequential C++	228.61s	
Sequential Java	245.05s	
Threads	OpenMP	Java
2	114.37s (1.99X)	131.75s (1.86X)
3	76.30s (2.99X)	87.96s (2.79X)
4	57.05s (4.00X)	66.02s (3.72X)
5	45.77s (4.99X)	53.46s (4.59X)
6	38.20s (5.98X)	44.88s (5.46X)

7. Conclusions

The MeanShift algorithm is a cluster algorithm with good results in terms of quality and with only one parameter. However, its major downside is its high computational cost which is slightly improved using a parallel approach, gaining a speedup that grows almost linearly with the number of threads, even with an increasing input size.

References

[1] K. Fukunaga and L. D. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition, 1975. IEEE Trans. Information Theory, 21:32–40.