MSBD5003 Project Final Report
Implementation of OPTICS on Spark

Group 18 LAU Ka Ho , SUN Qian , XUAN Yueming

Abstract

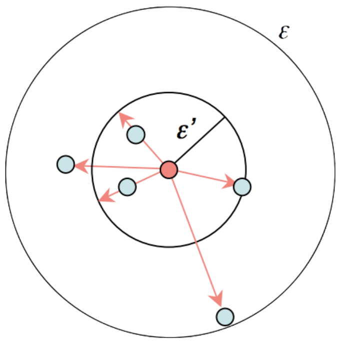
Implementing parallel mechanism for OPTICS using PySpark in order to overcome the shortage of high complexity and reduce running time of sequential OPTICS. The dataset is splitted using KD-Tree and then combine after the parallel process to collect a ordered list of dataset for tuning the clustering result without re-running the OPTICS.

1. Introduction

1.1 Project Background

Clustering algorithms are the most common type of machine learning algorithms, such as K-Means which we are all familiar with, its principle or the realization is very simple. However, we expect the clustering algorithm to satisfy some additional features:
1.Support for custom distance functions: Many clustering algorithms do not support this feature. Even a small system, the daily input data is near the TB level. It is a basic requirement for algorithms can execute concurrently.
2.No parameters, or parameter insensitive. Because most of the algorithms we used to analysis our data belong to unsupervised learning algorithms, users do not need to master the knowledge of methods to adjust the parameters. OPTICS is a parameter insensitive algorithm.

Clustering algorithms can be broadly divided as Partitioning Methods, Hierarchical Methods, Density-based Methods, Grid-based Methods, Model-based Methods. OPTICS is a kind of density-based clustering algorithm. OPTICS(Ordering point to identify the clustering structure) is an algorithm for finding density-based[1] clusters in spatial data. The main idea of this algorithm is that when the density of points in an area is greater than a certain threshold, those points are grouped together. Therefore, this density-based clustering algorithm has strong ability to find outliers. In general, the clustering algorithm output the specific classification results under fixed parameters. However, OPTICS is not that case. OPTICS can get all the classification eventually under a certain parameter interval (ε-neighborhood). Each of the possible points in a sequence records its two properties: the core distance and the reachability distance. For any given MinPts and ε, the core-distance of a point o is a minimal distance εi such that point o is a core point. The reachability-distance of a point p with respect to a point o is the smallest distance such that o is a core point and p belongs to its ε-neighbourhood.Through this sequence, we can easily get the classification result of the data points. In summary, OPTICS has two very important features:
1. Ability to resist out-of-range noise (look for outliers)
2. Insensitive to the parameters



OPTICS is improved from DBSCAN to overcome the disadvantage caused by using global parameter values which do not reflect the structure of clusters in-depth. OPTICS addresses that issue by ordering the objects in a database which represents its density-based clustering structure. OPTICS still use MinPts value and the similar principle of DBSCAN to processed the ε-neighborhood. The final output as ordered list of objects with two variables stored for each one of them: core-distance and reachability-distance provided sufficient information in extracting clusters. Due to high temporal and spatial complexity, OPTICS can not adapt well to the large data sets in modern society. With the development of cloud computing and parallel computing, a method to solve the complexity of OPTICS algorithm is provided. In this paper, we propose a parallel OPTICS algorithm based on the

Spark memory computing platform. The experimental results show that it can greatly reduce the time and space needs of the OPTICS algorithm.

The serial OPTICS algorithm has four input parameters: points(input points to cluster), minpts(the minimum points required to form a cluster), epsilon(a threshold to make a cluster) and order file(used to store the points' order). It can also output the follow results: rd(each point's reachability distance), cd(each point's core distance), clusters(a struct containing each cluster) and order(the order of points in the reachability graph). The pseudocode for OPTICS and its update function:

```
OPTICS(objects, e, minpts, order_file):
    for each unprocessed object in objects:
        neighbors = objects.get_neighbors(e, object)
        object.set_core_distance(neighbors, e, minpts)
        order_file.write(object)
        if object.core_distance != None:
            order_seeds.update(object, neighbors)
            for seed in order_seeds:
                neighbors = objects.get_neighbors(e, seed) seed.set_core_distance(neighbors, e, minpts)
                order_file.write(seed)
                if seed.core_distance != None:
                    order_seeds.update(seed, neighbors)
```

```
UPDATE(center, neighbors):
    d = center.core_distance
    for each unprocessed object in neighbors:
        new_reach_dist = max(d, dist(center, object))
        if object.reachability == None:
            object.reachability = new_reach_dist
            insert(object, new_reach_dist)
        elif new_reach_dist < object.reachability:
            object.reachability = new_reach_dist
            decrease(object, new_reach_dist)
```

1.2 Description of Parallel OPTICS

In OPTICS, there are two parts that can be taken as parallel. The first part is the distance function which calculates the distance between one point and the rest of the data points. We know that the time complexity to calculate the distance between two elements of a dataset is O(n^2). After parallelized, the time complexity compressed to O(n). The second part which can be parallelized is the update function. Every points need to be updated are independently for each other. Therefore, Parallel OPTICS can be obtained based on serial OPTICS by modifying its distance and update functions.

2. Design

We have tried to implement the OPTICS algorithm using Spark RDD operation internally in the iterations. However, the result is very bad due to the lazy evaluation of RDD, the Directed Acyclic Graph grows huge when the iteration continue and finally end with out of memory. Therefore, we plan to run the OPTICS sequentially in the workers with preprocessed splitted data.

After detail investigation, KD Tree is popular to partition the data into equal amount set. KdTree is used for partitioning by separating the data into partitions using maximum variance. Firstly, map all points with indexes(position in the dataset list) and partition label 0. Then partition the points by, first, finding the axis with the greatest variance, then split the points into two RDD as partition according to the mean point along that axis. The bounding of the partitions are recorded correspondingly. Then, those new partitions will be put back to the queue for partitioning again. The partitioning will be stop when number of partitions equal to the defined number of Spark partitions. Finally, a RDD union all partition with element in format ((point_index, partition_label), vector)). Each partition will has an associated bounding record which contains the lower and upper of boundary along the each axis. The bounding is stored in a Python dictionary.

The clustering result from each partition is **local** result of that partition. In order to combine the result of each partition to form **global** result, we need to share the points information of each partition. It is shared by using the expanded bounding boxes to create certain intersection between partitions' dataset. Each bounding box is expanded by adding epsilon. Points within the expanded bounding box will have addition partition label appeared in multiple partitions.

After running the sequential OPTICS in each worker, each point will get the local cluster label, order index and determination of core point, neighbour or noise. The local clusters will be merged to form a global cluster if there is core point existed in their overlapping area. The duplicated points will be filtered if it has the or minimum core distance for core or minimum reachability distance for neighbour only. The final collected list of point will be sorted by cluster label firstly, partition label secondly and order index finally .

3. Implementation

We use pypardis[2] as a blueprint in partitioning the data into equal size partition. Our work is modifying pypardis to suit for OPTICS, program a sequential OPTICS algorithm and design a approach to merge the parallel OPTICS result. The max number of partition is defined as the number of partition when parallelize data into a RDD in the very beginning. The implementation will be explained with capture of coding. PointObject that contains global index which is the index in the original dataset, local index which is the index in the partition dataset, partition label, reachability distance, core distance, processed boolean, order index which is index in the ordered list after the local OPTICS process and core point boolean.

3.1 Partitioning

Each data is converted into BoundingBox object and then merge to form a bigger box under the KD-Tree partition algorithm. BoundingBox is just recording the upper and lower limit along the axises in the partition.

```
box = data.aggregate(BoundingBox(k=self.k), lambda total, (_, v): total.union(BoundingBox(v)),
lambda total, v: total.union(v))
```

The means and variances along each axis are calculated using moments in each partition.

```
moments = partition.aggregate(np.zeros((3, k)), lambda x, (keys, vector): x + np.array( [np.ones(k), vector,
vector ** 2]), add)
means = moments[1] / moments[0]
variances = moments[2] / moments[0] - means ** 2
axis = np.argmax(variances)
```

Each time there will be two partitions generated since KD-Tree is doing a binary split and stop until max partitions are generated.

```
counts = np.abs(partition.aggregate(np.zeros(7), lambda x, (_, v): x + 2 * (v[axis] < bounds) - 1, add))
boundary = bounds[np.argmin(counts)]
part1 = partition.filter(lambda (_, v): v[axis] < boundary)
part2 = partition.filter(lambda (_, v): v[axis] >= boundary).map(lambda ((key, _), v): ((key, next_label), v))
```

BoundingBox will also be splitted according to the same mean value used in the same round for partition.

Each BoundingBox will be expanded and then a RDD will content all points with assigned partition label. The number of iteration is equal to number of patition. Therefore, the speed should not be slow.

```
for label, box in bounding_boxes.iteritems():
    expanded_box = box.expand( eps)
    expanded_boxes[label] = expanded_box
```

```
        neighbors[label] = data.filter(lambda (k, v): expanded_box.contains(v)).map(lambda (k, v):
((k, label), v))
        new_data = new_data.union(neighbors[label])
```

The points with partition label is ready. Then, we use mapPartitions to run the OPTICS algorithm. The points will obtain label of combination of partition and local cluster as format “partition:local_cluster” as key.

```
data = data.mapPartitions(lambda iterable: optics_do(iterable,params)).cache()
```

The result of data is separated to two part, one is global index with the combine label and other one is the global index with point objects. The RDD of combined label will be groupByKey so as to aggregate to a list of dictionary in format as {global_cluster : “partition:local_cluster”} will be the global cluster label mapper. The ClusterAggregator is provided by library pypardis and finds the global cluster using nested iteration to search whether the multiple local cluster labels of a point exist in the current global cluster map.

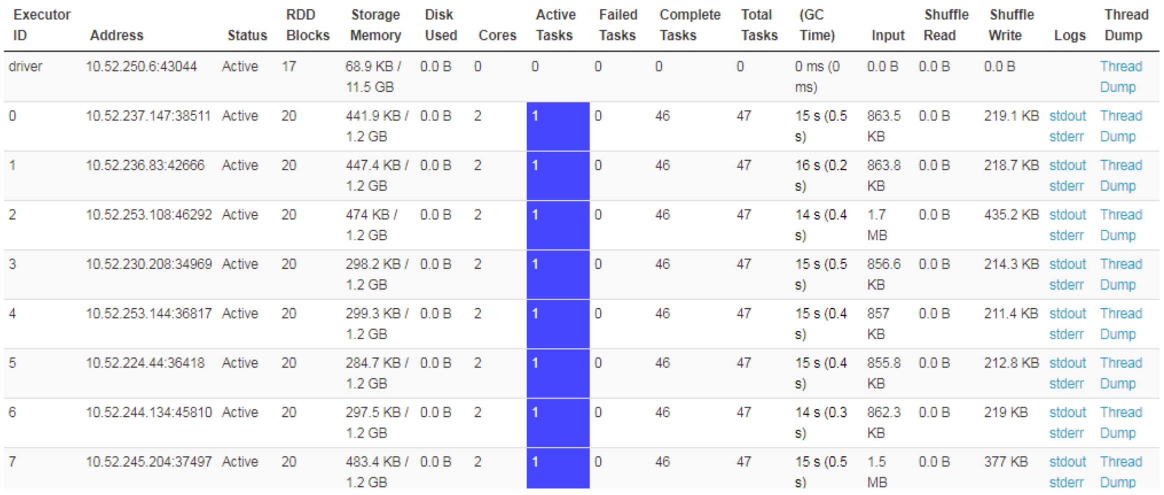
```
labeled_points = data.map(lambda x:x[0]) #combine label
object_points = data.map(lambda x:x[1]).groupByKey().map(lambda x:(x[0],min(list(x[1]),key = (lambda
x:x.reachDis)))).cache() #point object
mapper = labeled_points.aggregate(ClusterAggregator(), add, add)
b_mapper = data.context.broadcast(mapper.fwd)
object_result = object_points.map(lambda x:map_object_cluster_id(x[1],b_mapper)).sortBy(lambda x:x[0])
```

Only the point object with minimum reachability distance will be kept which will result in a more dense cluster. The cluster label in point object will be updated using the global cluster map. The, the result is sorted by the key that contains (global_cluster_label,partition,order_index). Finally, the result list of reachability distance is exported into .csv for tuning the result of clustering using different epsilon.

4. Evaluation

4.1 Environment Setup

The evaluation is built on Databricks with the cluster created using AWS EC2 instances. The initial setting is 1 driver node and 2 worker node with the same configuration. Each node has 8.0GB memory and 2 cores. In the testing, we also add worker nodes to 4 and 8 to enable more executors.



Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	(GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	10.52.250.6:43044	Active	17	68.9 KB / 11.5 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
0	10.52.237.147:38511	Active	20	441.9 KB / 1.2 GB	0.0 B	2	1	0	46	47	15 s (0.5 s)	863.5 KB	0.0 B	219.1 KB	stdout stderr	Thread Dump
1	10.52.236.83:42666	Active	20	447.4 KB / 1.2 GB	0.0 B	2	1	0	46	47	16 s (0.2 s)	863.8 KB	0.0 B	218.7 KB	stdout stderr	Thread Dump
2	10.52.253.108:46292	Active	20	474 KB / 1.2 GB	0.0 B	2	1	0	46	47	14 s (0.4 s)	1.7 MB	0.0 B	435.2 KB	stdout stderr	Thread Dump
3	10.52.230.208:34969	Active	20	298.2 KB / 1.2 GB	0.0 B	2	1	0	46	47	15 s (0.5 s)	856.6 KB	0.0 B	214.3 KB	stdout stderr	Thread Dump
4	10.52.253.144:36817	Active	20	299.3 KB / 1.2 GB	0.0 B	2	1	0	46	47	15 s (0.4 s)	857 KB	0.0 B	211.4 KB	stdout stderr	Thread Dump
5	10.52.224.44:36418	Active	20	284.7 KB / 1.2 GB	0.0 B	2	1	0	46	47	15 s (0.4 s)	855.8 KB	0.0 B	212.8 KB	stdout stderr	Thread Dump
6	10.52.244.134:45810	Active	20	297.5 KB / 1.2 GB	0.0 B	2	1	0	46	47	14 s (0.3 s)	862.3 KB	0.0 B	219 KB	stdout stderr	Thread Dump
7	10.52.245.204:37497	Active	20	483.4 KB / 1.2 GB	0.0 B	2	1	0	46	47	15 s (0.5 s)	1.5 MB	0.0 B	377 KB	stdout stderr	Thread Dump

4.2 Dataset

We randomly pick up some datasets from [3] Clustering benchmark datasets. The table below shows dataset details:

Dataset	No. of points
Toy set	150
A-sets A1	3,000

A-sets A3	7,500
Birch 2 subset 53	5,3000

4.3 Testing Plan

Efficiency is the most critical factor to evaluate our parallel OPTICS. Original OPTICS is efficient and effective in clustering given ordered sequence and it is easy to adjust radius. However, the order and distance calculation is time consuming when dataset is large, where we aim to improve using parallel OPTICS.
We will apply parallel OPTICS to 4 datasets mentioned above and capture the total running time. Different number of worker nodes and number of partitions will be tested.

No. of worker nodes	2		4		8	
No. of partitions	2	4	4	8	8	16

4.4 Testing Result

4.4.1 Toy set

In all cases, OPTICS can finish within 5s. Running time and improvement is negligible using parallel algorithm.

4.4.2 A-sets A1

No. of worker nodes	2		4		8	
No. of partitions	2	4	4	8	8	16
Partition time	3s	2s	3s	3s	3s	5s
OPTICS and merge time	68s	37s	47s	34s	42s	32s
Total time	71s	39s	50s	37s	45s	37s

4.4.3 A-sets A3

No. of worker nodes	2		4		8	
No. of partitions	2	4	4	8	8	16
Partition time	3s	4s	6s	5s	5s	6s
OPTICS and merge time	323s	417s	281s	213s	226s	189s
Total time	326s	421s	287s	218s	231s	195s

As the dataset size becoming large, the running time has increased significantly. Partition is a very quick process and running OPTICS and aggregation takes most of time. Adding worker node can improve the efficiency but not linearly. Our guess is although the parallel running of OPTICS can improve the computation efficiency linearly, there is loop and sorting when aggregating the result from each partition.

4.4.4 Birch 2 subset 53

No. of worker nodes	2		4		8	
No. of partitions	2	4	4	8	8	16
Partition time					10s	24s
OPTICS and merge time					1h18min	56min
Total time	>1h	>1h	>1h	>1h	1h18min	57min

Using our largest data set of 53,000 points, even parallel OPTICS with low number of worker nodes cannot finish within 1 hour so we stop the program after 1.5h. Similar to previous case, running OPTICS and aggregation takes huge amount of time. Monitoring the SparkUI for jobs, tasks on 8 nodes are not finished at similar time. For the aggregation step, 1/16 task finish is monitored at 23min however 16/16 task finish is monitored at 52min. We think there are still some iteration steps remaining where tasks need to queue.

5. Future Development

ClusterAggregator has complexity O(n^2). The global cluster mapper can be generated using graph method which can be more clean code and potentially fast. Actually, we tried implemented BFS using GraphX to find the connected partition. However, the result is very slow due to the union in while loop and count each iteration.

```
while layers.count() > 0:
        d1 = layers.join(g.edges, layers['id'] == g.edges['src'])
        layers = d1.select(d1['dst'].alias('id')) .subtract(visited).distinct()
        visited = visited.union(layers)
```

Spark provides BFS can be tried in the future development but required additional design to find the connected nodes since that BFS just provide the result of shortest path.

Our testing is performed on 2 dimensional data only. The performance and accuracy is not tested for high dimensional data.

Reference:

[1] Hans-Peter; Kröger,Peer; Sander,Jörg; Zimek,Arthur(May2011). "Density-basedclustering". Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery.1(3):231–240. doi:10.1002/widm.30
[2] https://github.com/bwoneill/pypardis
[3] http://cs.joensuu.fi/sipu/datasets/