

# Homework 1: Clustering, Spark MLlib, and Hadoop

---

Yi Yang (yy3089)

October 7, 2021

## 1 ITERATIVE K-MEANS CLUSTERING ON SPARK

### 1.1 L1-distance clustering

```
1 import operator
2 import sys
3 from pyspark import SparkConf, SparkContext
4 import numpy as np
5 import matplotlib.pyplot as plt
6 from scipy import linalg
7
8 conf = SparkConf()
9 sc = SparkContext(conf=conf)
10
11 # Macros.
12 MAX_ITER = 20
13 # gs://yy3089_data_storage/data/HW1_data/
14 DATA_PATH = "gs://yy3089_data_storage/data/HW1_data/data.txt"
15 C1_PATH = "gs://yy3089_data_storage/data/HW1_data/c1.txt"
16 C2_PATH = "gs://yy3089_data_storage/data/HW1_data/c2.txt"
17 NORM = 1
```

In order to conduct kmeans clustering, we need to define some helper functions:

```
1 # Helper functions.
2 def closest(p, centroids, norm):
3     """
4     Compute closest centroid for a given point.
5     Args:
6         p (numpy.ndarray): input point
7         centroids (list): A list of centroids points
8         norm (int): 1 or 2
```

```

9     Returns:
10         int: The index of closest centroid.
11     """
12     closest_c = min([(i, linalg.norm(p - c, norm))
13                     for i, c in enumerate(centroids)],
14                     key=operator.itemgetter(1))[0]
15     return closest_c
16
17 def toArray(l):
18     return np.array([float(x) for x in l])
19
20 def mergeSet(x, y):
21     l = min(len(x), len(y))
22     return tuple(x[_] + y[_] for _ in range(l))

```

This is my kmeans function:

```

1  # K-means clustering
2  def kmeans(data, centroids, norm):
3      """
4      Conduct k-means clustering given data and centroid.
5      This is the basic version of k-means, you might need more
6      code to record cluster assignment to plot TSNE, and more
7      data structure to record cost.
8      Args:
9          data (RDD): RDD of points
10         centroids (list): A list of centroids points
11         norm (int): 1 or 2
12     Returns:
13         RDD: assignment information of points,
14         a RDD of (centroid, (point, 1))
15         list: a list of centroids
16         and define yourself...
17     """
18     # data: list of np.array
19     data_np = data.map(lambda l: l.split(' ')).map(lambda x: toArray(x))\
20     .collect()
21
22     points = []
23     costs = []
24
25     # iterative k-means
26     for _ in range(MAX_ITER):
27         # Transform each point to a combo of
28         point, closest centroid, count=1
29         # point -> (closest_centroid, (point, 1))
30         # print('Iter %d' % _)
31
32         points = []
33         distance = 0
34
35         for p in data_np:

```

```

36         index = closest(p, centroids, norm)
37         centroid = centroids[index]
38
39         distance += linalg.norm(centroid-p, norm)**norm
40         points.append((index, (p, 1)))
41
42     # record total cost
43     costs.append((_, distance))
44     # generate points RDD
45     points = sc.parallelize(points)
46
47     # Re-compute cluster center
48     # For each cluster center (key), aggregate its values
49     # by summing up points and count
50     # Average the points for each centroid:
51     # divide sum of points by count
52     # points_sum = (index, (sum_points, sum_count))
53     points_sum = points.reduceByKey(lambda x, y: mergeSet(x, y))\
54         .collect()
55
56     for key, val in points_sum:
57         # print('For centroid %d, there are %d points' % (key, val[1]))
58         centroids[key] = val[0]/val[1]
59
60     costs = sc.parallelize(costs)
61     return points, centroids, costs
62
63     # Use collect() to turn RDD into list

```

Then conduct the clustering and plot the cost curve:

```

1 data = sc.textFile(DATA_PATH)
2 c1 = sc.textFile(C1_PATH)
3 c2 = sc.textFile(C2_PATH)
4
5 # cx_np: list of float
6 c1_np = c1.map(lambda l: l.split(' ')).map(lambda x: toArray(x)).collect()
7 c2_np = c2.map(lambda l: l.split(' ')).map(lambda x: toArray(x)).collect()
8
9 points1, centroids1, costs1 = kmeans(data, c1_np, NORM)
10 iter_list = costs1.keys().collect()
11 cost_list = costs1.values().collect()
12 plt.plot(iter_list, cost_list, 'b-', label = 'Cost_for_c1.txt')
13
14 points2, centroids2, costs2 = kmeans(data, c2_np, NORM)
15 iter_list = costs2.keys().collect()
16 cost_list = costs2.values().collect()
17 plt.plot(iter_list, cost_list, 'r-', label = 'Cost_for_c2.txt')
18
19 plt.xticks(iter_list, [_ for _ in range(1, 21)])
20
21 plt.ylabel("Cost", fontdict={'weight': 'normal', 'size': 12})

```

```

22 plt.xlabel("Iterations", fontdict={'weight': 'normal', 'size': 12})
23 plt.legend(loc = 'upper_right', prop={'size' : 11})
24 plt.title('Cost-Iteration Curve for L%d' % NORM)
25
26 plt.show()

```

I put the two cost-iterations curve together in one picture to show the difference:

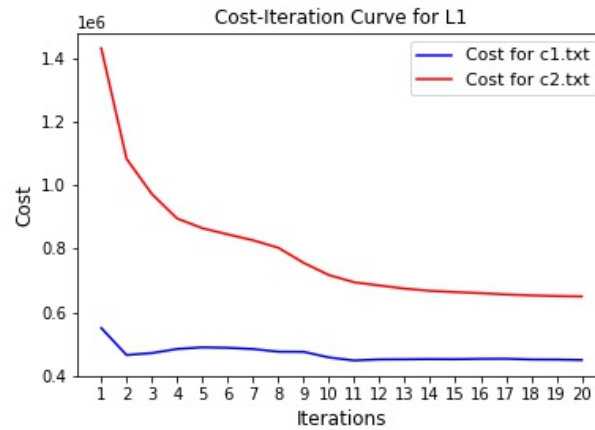


Figure 1.1: Screenshot for '*L1 distance clustering*'

## 1.2 L2-distance clustering

In order to perform L2-distance clustering, we only need to set  $NORM=2$ . I put the two cost-iterations curve together in one picture to show the difference:

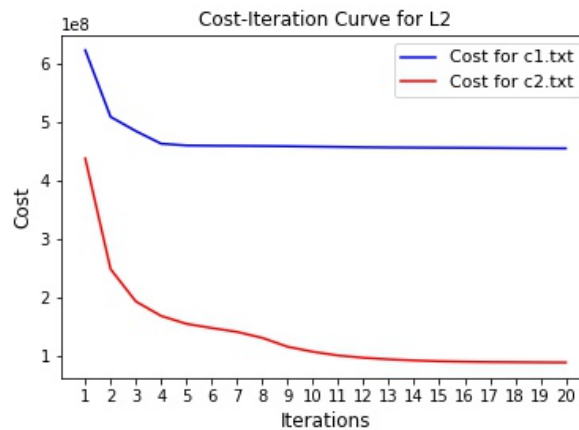


Figure 1.2: Screenshot for '*L2 distance clustering*'

## 1.3 t-SNE plot

```

1 from sklearn.manifold import TSNE
2
3 indexes = points1.map(lambda x: x[0]).collect()
4 coordinators = points1.map(lambda x: x[1][0]).collect()

```

```

5 coordinators_embedded = TSNE(n_components=2, perplexity=100,
6                               random_state=100).fit_transform(coordinators)
7
8 vis_x = coordinators_embedded[:, 0]
9 vis_y = coordinators_embedded[:, 1]
10 plt.scatter(vis_x, vis_y, cmap=plt.cm.get_cmap('jet', 10))
11 plt.title('Original Data')
12 # plt.show()
13 plt.savefig('origin.jpg')
14
15 scatter = plt.scatter(vis_x, vis_y,
16                       marker='o', c=indexes, cmap='jet')
17
18 plt.title('Clustering Data for L%d on c1.txt' % NORM)
19 plt.savefig('c1_clustering.jpg')

```

Original data:

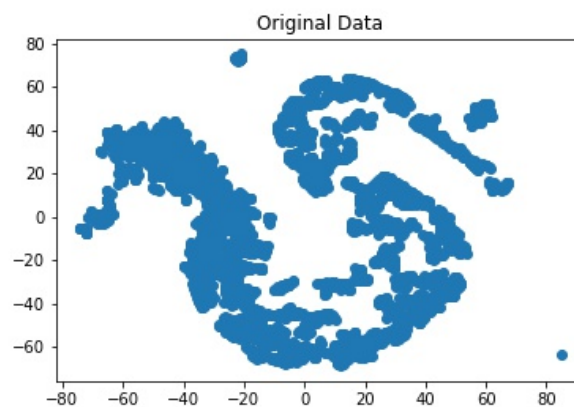


Figure 1.3: Screenshot for 'Original data'

Using c1.txt as initial centroids:

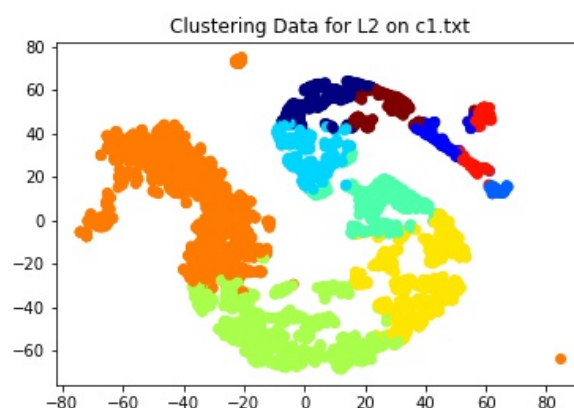


Figure 1.4: Screenshot for 'c1.txt as initial centroids'

Using c2.txt as initial centroids:

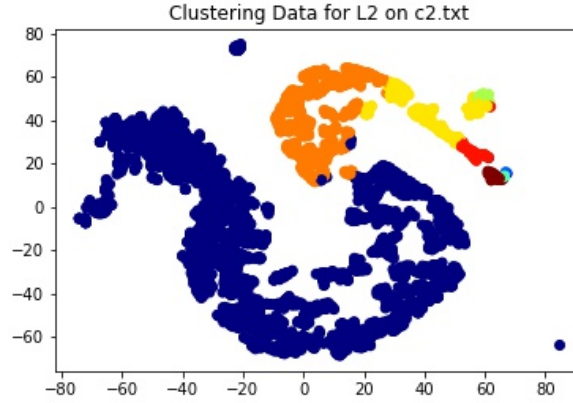


Figure 1.5: Screenshot for '*c2.txt as initial centroids*'

### 1.4 Performance Analysis

When we use L1 distance, initialization with c1.txt is better than that with c2.txt. When we use L2 distance, initialization with c2.txt is better than that with c1.txt.

For L2 distance, initial points which are as far away from each other as possible make clustering converge better during the optimization process. The computed centroids are more accurate to the true centroids of each cluster.

However, when L1 distance is employed for clustering, points from c2.txt are not very suitable for clustering. I guess there are two reasons. The first reason is points in c2.txt are computed using L2 distance. Maybe they are not the farthest points from each other for L1 distance. The second reason is that 20 iterations are not enough for the clustering model to converge.

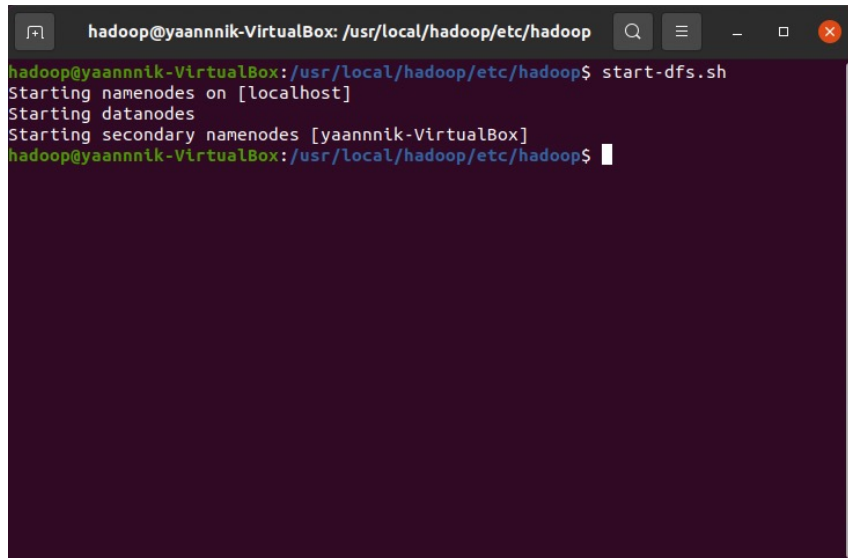
### 1.5 Time Complexity

The time complexity of kmeans is  $O(iter \times K \times n \times d)$ , where  $iter$  is the iteration number,  $K$  is total clusters,  $n$  is the number of data samples and  $d$  is the dimension of each sample.

## 2 MONITORING HADOOP METRICS

### 2.1 Download and setup Hadoop

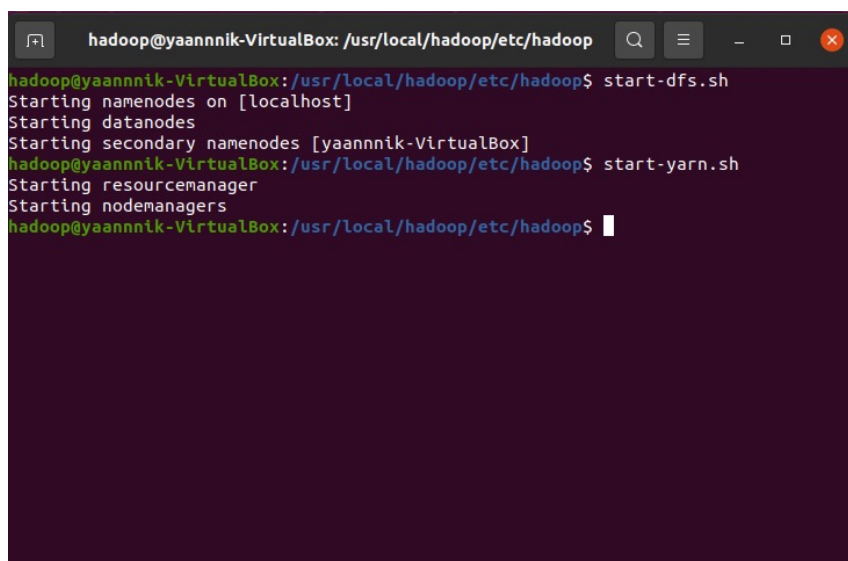
start-dfs.sh:

A terminal window titled 'hadoop@yaannik-VirtualBox: /usr/local/hadoop/etc/hadoop' with search, menu, and window control icons. The terminal shows the command 'start-dfs.sh' being executed. The output is: 'Starting namenodes on [localhost]', 'Starting datanodes', and 'Starting secondary namenodes [yaannik-VirtualBox]'. The prompt returns to 'hadoop@yaannik-VirtualBox: /usr/local/hadoop/etc/hadoop\$' with a cursor.

```
hadoop@yaannik-VirtualBox: /usr/local/hadoop/etc/hadoop$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [yaannik-VirtualBox]
hadoop@yaannik-VirtualBox: /usr/local/hadoop/etc/hadoop$
```

Figure 2.1: Screenshot for 'start-dfs.sh'

start-yarn.sh:

A terminal window titled 'hadoop@yaannik-VirtualBox: /usr/local/hadoop/etc/hadoop' with search, menu, and window control icons. The terminal shows the command 'start-yarn.sh' being executed. The output is: 'Starting resource manager' and 'Starting nodemanagers'. The prompt returns to 'hadoop@yaannik-VirtualBox: /usr/local/hadoop/etc/hadoop\$' with a cursor.

```
hadoop@yaannik-VirtualBox: /usr/local/hadoop/etc/hadoop$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [yaannik-VirtualBox]
hadoop@yaannik-VirtualBox: /usr/local/hadoop/etc/hadoop$ start-yarn.sh
Starting resource manager
Starting nodemanagers
hadoop@yaannik-VirtualBox: /usr/local/hadoop/etc/hadoop$
```

Figure 2.2: Screenshot for 'start-yarn.sh'

### 2.2 HDFS metrics monitoring

The five most important metrics to me are:

- **CapacityRemaining:** This metric monitors the total available capacity remaining across the entire HDFS cluster. DataNodes that are out of space are likely to fail on boot and running DataNodes will not be able to be written to.

- **CorruptBlocks/MissingBlocks:** This metric monitors corrupt or missing blocks which can point to an unhealthy cluster.
- **NumDeadDataNodes:** This metric monitors the number of DataNodes in the cluster, by state. If this number unexpectedly drops, it may warrant investigation.
- **FilesTotal:** This metric monitors the the number of files being tracked by the NameNode. The number of files (or blocks, or directories) is positively related to the more memory required by NameNode.
- **TotalLoad:** This metric monitors the current number of concurrent file accesses (read-/write) across all DataNodes. It can illuminate issues related to job execution.

Figure 2.3: Screenshot for 'localhost:9870'

Figure 2.4: Screenshot for *'localhost:9870'*

The MapReduce framework exposes a number of counters to track statistics on MapReduce job execution. MapReduce counters can be broken down into four categories: job counters, task counters, custom counters and file system counters. We can monitor the execution of MapReduce tasks through metrics:



- **MILLIS\_MAPS/ MILLIS\_REDUCE**S
- **NUM\_FAILED\_MAPS/ NUM\_FAILED\_REDUCE**S
- **DATA\_LOCAL\_MAPS/ RACK\_LOCAL\_MAPS/ OTHER\_LOCAL\_MAPS**
- **REDUCE\_INPUT\_RECORDS**
- **SPILED\_RECORDS**
- **GC\_TIME\_MILLIS**

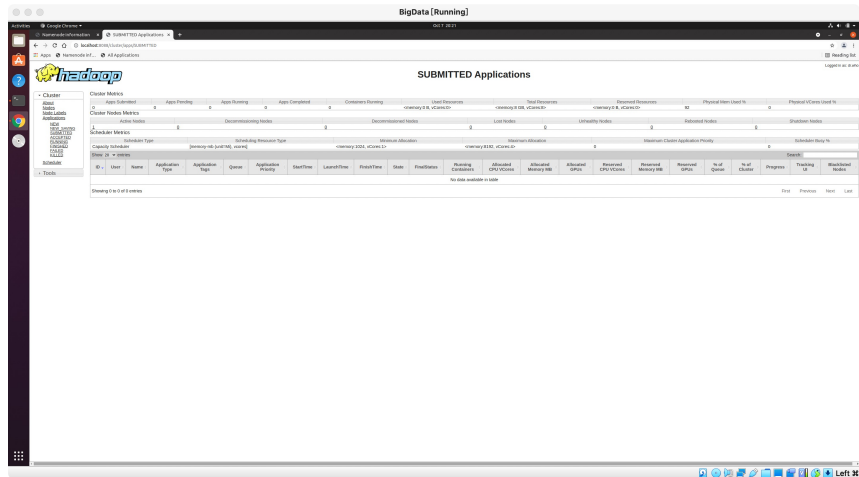


Figure 2.5: Screenshot for 'mapreduce metric'

## 2.4 YARN metrics monitoring

The five most important metrics to me are:

- **activeNodes/lostNodes:** The metric monitors the current count of active, or normally operating, nodes. If a NodeManager fails to maintain contact with the ResourceManager, it will eventually be marked as “lost” and its resources will become unavailable for allocation.
- **appsFailed:** This metric monitors The percentage of failed tasks. If the percentage of failed map or reduce tasks exceeds a specific threshold, the application as a whole will fail.
- **totalMB/allocatedMB:** This metric monitors the cluster’s memory usage.
- **progress:** This metric monitors the real-time window into the execution of a YARN application.
- **containersFailed:** This metric monitors the number of containers that failed to launch on that particular NodeManager.

