Lab 2: K-means in Spark

Youfei Zhang

Part 1: NBA Log File

Inplement the K-Means algorithm without using SparkML or SparkMLlib package.

For each player, we define the comfortable zone of shooting is as matrix of,

```
{SHOT DIST, CLOSE DEF DIST, SHOT CLOCK}
```

Develop an K-Means algorithm to classify each player's records into 4 comfortable zones. Considering the hit rate, which zone is the best for James Harden, Chris Paul, Stephen Curry and Lebron James.

Design

In this lab, I tried to implement kmeans with PySpark in two ways. One is an RDD based iteration, the other is based on Spark Dataframe. By comparision, the RDD based iteration is more efficient than the Spark Dataframe one.

1. RDD based Kmeans

1. Intialize spark session

```
spark = SparkSession.builder.appName("NBA-kmeans").getOrCreate()
```

2. Preprocessing: clean and filter into RDD Load the csv into a spark context as a Spark DataFrame, and filter based on player name and the matrix column names. Convert the DataFrame into RDD through map to prepare for the iteration.

```
df = spark.read.format("csv").load(sys.argv[1], header = "true", inferSchema = "true")
dataPts = df.filter(df.player_name == 'james harden').select('SHOT_DIST','CLOSE_DEF_DIST',
'SHOT_CLOCK').na.drop()
dataRDD = dataPts.rdd.map(lambda r: (r[0], r[1], r[2]))
```

3. K-Means Iteration

After preprocessing, I start to implement the Kmeans algorithm. First, randomly select 4 rows as intial centroid:

```
k = 4
int_centroid = dataRDD.takeSample(False, k)
```

Then, I run the K-Means algorithm iteratively. For each data point, we calculate their distances to the 4 initial centroids, and assign them to the cluster of their closest centroid. Next, for each cluster, we recalculate the new centroid by getting the mean of each column. By doing so, we have 4 new centriods, and we recalculate the distance between each data points to the new centroids. We iterate this process until the new centroids equal to the old centroids, or until the maximum times of iteration is reached.

- calculate each data's distance to the centroid
- asign each data to the closet cdntroid
- calculate the new centroid per cluster by finding their mean
- iteration: calculate each data's distance to the NEW centroid
- stop iteration when old_centroids = new_centroids or when the maximum number of iteration is reached

```
iters = 0
old_centroid = int_centroid

# set the maximum iteration as 40
for m in range(40):
    map1 = dataRDD.map(lambda x: closestCenter(x, old_centroid))
    reduce1 = map1.groupByKey()
    map2 = reduce1.map(lambda x: cal_centroid(x)).collect() # collect a list
    new_centroid = map2
    converge = 0
    for i in range(k):
        if new_centroid[i] == old_centroid[i]:
            converge += 1
        # check if the different is smaller than 0.03
        else:
            diff = 0.0009
            closeDiff = [round((a - b)**2, 6) for a, b in zip(new_centroid[i], old_centroid[i])]
            if all(v <= diff for v in closeDiff):
                converge += 1
    if converge >= 4:
        print("Converge at the %s iteration\n" %(iters))
        print("\nFinal Centroids: %s" %(new_centroid))
        break
    else:
        iters += 1
        print("Iteration - %s round" %(iters))
        old_centroid = new_centroid
        print('Update:',old_centroid,'\n')
```

The Result

Final cluster:

```
2019-04-23 15:27:59 INFO  DAGScheduler:54 - Job 11 finished: collect at /spark-examples/labs/
lab2/part1/./nba-kmeans-1.py:87, took 0.679742 s
Converge at the 7 iteration


Final Centroids: [[4.1054, 2.8148, 18.3343], [6.3922, 2.5587, 9.1581], [23.7822, 5.0147, 16.2
881], [21.3225, 4.2283, 5.7786]]
```

The job process:

```
2019-04-23 15:27:58 INFO  BlockManagerInfo:54 - Added broadcast_20_piece0 in memory on 10.150
.0.7:43801 (size: 5.4 KB, free: 413.8 MB)
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 393
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 372
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 396
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 376
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 407
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 406
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 412
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 417
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 403
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 384
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 410
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 380
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 413
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 401
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 370
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 389
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 386
2019-04-23 15:27:58 INFO  ContextCleaner:54 - Cleaned accumulator 383
2019-04-23 15:27:58 INFO  MapOutputTrackerMasterEndpoint:54 - Asked to send map output locati
ons for shuffle 6 to 10.150.0.7:32950
```

2. Dataframe based Kmeans

1. Intialize spark session
2. Preprocessing: clean and filter

Load the csv into a spark context as a Spark DataFrame, and filter based on player name and the matrix column names.

```
df = spark.read.format("csv").load(sys.argv[1], header = "true", inferSchema = "true")
dataPts = df.filter(df.player_name == 'james harden').select('SHOT_DIST','CLOSE_DEF_DIST',
'SHOT_CLOCK').na.drop()
```

3. K-Means Iteration

First, randomly select 4 rows as intial centroid:

```
k = 4
int_centroid = dataPts.takeSample(False, k)
```

Then, run the iteration on Dataframe by adding a column 'Center' that idicate the datapoint's closet cluster from last round.

The function **closestCentroid** is wrapped into a udf to perform operations on the dataframe columns with:

```
def closestCentroid(col1, col2, col3):
    '''
    input: float value
    output: integer
    '''
    points = [col1, col2, col3]
    dist_list = []
    for c in newCenter:
        dist_list.append(euclDist(points, c))
    closest = float('inf') # an unbounded upper value for comparison
```

```
    index = -1
    for i, v in enumerate(dist_list):
        if v < closest:
            closest = v
            index = i
    return int(index)

minCenter = udf(closestCentroid, IntegerType())
```

The dataframe is first new column 'Center' that idicate the datapoint's closet cluster from the randomly chosen initial centroids. Then, for each cluster, new centers are computed based on Euclidean distance. Each centroid is compared to the prior centroids. If they are the same, or the absolute difference is within 0.03, then a convergence is achieved.

```
converge = 0
# calculate the new centroids for each cluster
for i in range(k):
    kCluster = rddCluster.filter(rddCluster.Center == i)
    n = kCluster.count()
    sumCol = [0] * 3
    sumCol[0] = calNewCentroid(kCluster, 'SHOT_DIST')
    sumCol[1] = calNewCentroid(kCluster, 'CLOSE_DEF_DIST')
    sumCol[2] = calNewCentroid(kCluster, 'SHOT_CLOCK')
    sumOfCols = [round(x / n, 4) for x in sumCol]
    newCenter[i] = sumOfCols
    print(newCenter)
    if newCenter[i] == oldCenter[i]:
        converge += 1
    elif:
        diff = 0.0009
        closeDiff = [round((a - b)**2, 6) for a, b in zip(newCenter[i], oldCenter[i])]
        if all(v <= diff for v in closeDiff):
            converge += 1
```

When convergence are larger than 4 (all the four new centroids are similar to the prior centroids), we break out of the loop. Otherwise, we keep within the iteration. At the end of every round of iteration, the old centroid is updated to be the new centroid.

```
    iters += 1
    print("Iteration - %s round" %(iters))
    if converge >= 4:
        print("Converge at the %s iteration\n" %(iters))
        break
    else:
        iters += 1
        print("Iteration - %s round" %(iters))
        old_centroid = new_centroid
        print('Update:',old_centroid,'\n')
```