COMP5347 Assignment 2 Document

Design

Task 1

There are three main jobs in Task 1:

- extract (sample, researcher) pairs from `/share/cytometry/experiments.csv` . Because there might be many corresponding researchers for a certain sample, we firstly firstly get a list of (sample, researcher) pairs, and flat this map later.
  The functions applies to map is like below:

```
# for each record: sample, ..., researchers, ...
# we get [(sample, reseacher1), (sample, researcher2),...]
def extract_reserchers_function (record):
     sample, date, experiment, day, subject, kind, instrument, researchers
     = record.strip().split(",")
     result = []
     if sample != 'sample':
         researchers_list = researchers.strip().split('; ')
         for name in researchers_list:
             # store the researchers in result
             result.append((sample,name))
         return result
     else:
         return (sample,researchers) # return the title of sample and researchers
```

  We firstly apply this to map `/share/cytometry/experiments.csv` and the flat it.

```
extract_resercher = experiments.map(extract_reserchers_function ).flatMap(lambda xs:
[x for x in xs])
```

- extract measurements for each sample from `/share/cytometry/large` and reduce its number of measurements. During the map process, we mark every valid measured sample as 1, and 0 vice versa. And then reduce it to get the total number of measurements.
  The function applied is as follows:

```
def extract_measurement_function (record): # return the valid measurement and 1 or 0
      data = record.strip().split(',')
      sample,FSC,SSC = data[0],data[1],data[2]
      if FSC != 'FSC-A': # Check if this is the title
          if int(FSC) >=0 and int(FSC)<=150000 and int(SSC)>=0 and int(SSC)<=150000:
              return (sample,str(1))
          else:
              return (sample,str(0))
      else:
          return (sample,str(0))
```

And the job:

```
extract_measure =
measurements.map(extract_measurement_function ).repartition(1).reduceByKey(lambda
before,after: int(before)+int(after))
```

Here, since the valid measured sample is marked as 1, simply sum up the value will get the total measure ments for each sample

- join the above two results by sample, and then reduce it by researchers to get the total number of measurements for each researcher.

```
join_resercher_measurement =
extract_resercher.join(extract_measure).values().reduceByKey(lambda before,after:
int(before)+int(after)).collect()
```

The `collect()` function is to convert the result into array.
After this, we sort the result array by the number of measurements. If there is a tie in measurements, sort in alphabet order by researcher's name and save the result in required format.

```
join_resercher_measurement =
sc.parallelize(sorted(join_resercher_measurement ,key=lambda tup:(-tup[1],
tup[0]))).map(lambda record: record[0]+'\t'+str(record[1]))

join_resercher_measurement .repartition(1).saveAsTextFile("pyspark/q1")
```

Task 2

There are four major steps involved in this task.

- Read, filter and extract data.
  The filter function and job are as follows:

```
def measure_filter(record): # Selected the valid measurement
      data = record.strip().split(',')
      sample,FSC,SSC = data[0],data[1],data[2]
      if FSC != 'FSC-A': # If this is a tilte
          if int(FSC) >=0 and int(FSC)<=150000 and int(SSC)>=0 and int(SSC)<=150000:
              return True
          else:
              return False
      else:
          return False
```

```
 def extract_measurement_function (record): # data ---> (sample,(Ly6C,CD11b,SCA1))
      data = record.strip().split(',')
      sample,Ly6C,CD11b,SCA1 = data[0],data[11],data[7],data[6]
      return (sample,(Ly6C,CD11b,SCA1))
```

```
measurements = sc.textFile("/share/cytometry/large" )
  after_filter_measurement =
measurements.filter(measure_filter ).map(extract_measurement_function )
```

- Initiate cluster centers (5 clusters by default)

  Here we use the **Forgy** method for our initialization. We randomly generate five centers to be the initial centers of 5 clusters respectively.

```
number_of_cluster = 5
initial_cluster = np.random.rand(number_of_cluster,3) # generate 5 center randomly
```

- Learning process

  In this k-means learning process, we set iteration times to 10 by default. Within each iteration, we calculate the Euclidean distance for each observation (each record here) and find the closest center to map this observation to the corresponding cluster. And we use this clustered data set to get the new centers, which are the centers of current clusters. The whole process is presented in the following code:

```
# learning_time is set to 10 by default
for num in range(learning_time ):
    cluster_ini = after_filter_measurement .map(cluster_function )

    # group by cluster number and re-calculate the new center (which is the center
of the cluster)
    new_cluster_center = cluster_ini.groupByKey().map(lambda x : (x[0],
np.sum((np.asarray(list(x[1]))),axis=0)/len(np.asarray(list(x[1])))))
    data = new_cluster_center .collect()
    data_list_tp = []
    for i in range(number_of_cluster ):
        for j in data:
            if j[0] == i:
                data_list_tp.append(j[1])
    broad_cluster_center = sc.broadcast(data_list_tp )
```

The function `cluster_function` which calculates the distance is like below:

```
# get the closest center, and set the data into its cluster
def cluster_function(record):
    center = np.asarray(broad_cluster_center.value)
    data = np.asarray([record[1][0],record[1][1],record[1][2]]).astype('float64')
    cluster_number = np.sum((center-data)**2,axis=1).argmin()
    return (cluster_number,(float(record[1][0]),float(record[1][1]),float(record[1]
[2])))
```

- After the above learning process, we get a **better** center (this is also the centroids of output) for each cluster. Now we can derive the final clusters for the data set, and reduce the output by cluster ID. After sorting the above result by number of measurements, we can get the desired result.
  The codes are as follows:

```
new_cluster_center_result =
after_filter_measurement.map(map_result).repartition(1).reduceByKey(lambda
before,after: int(before)+int(after)) # Give the data a cluster number
number_of_cluster = new_cluster_center_result.map(lambda x :
(x[0],x[1],np.asarray(broad_cluster_center.value)[x[0]-1])).sortBy(lambda record:
int(record[0]))
result = number_of_cluster.map(lambda record:
str(record[0])+'\t'+str(record[1])+'\t'+str(record[2][0])+'\t'+str(record[2]
[1])+'\t'+str(record[2][2]))
result.repartition(1).saveAsTextFile("pyspark/q2")
```

  There is a slightly different in the function applied to map. Now we only care about the measurements time:

```
def map_result(record):
    center = np.asarray(broad_cluster_center.value)
    data = np.asarray([record[1][0],record[1][1],record[1][2]]).astype('float64')
    cluster_number = np.sum((center-data)**2,axis=1).argmin()
    return (cluster_number+1,1)
```

Task 3

Task 3 is quite similar with task 2, except a outlier removal process before clustering.

The whole process is as follows:

1. we use the result of task 2, more precisely, the centroid of each cluster.

```
# read the result of task2
q2_center = np.asarray(sc.textFile("pyspark/q2").map(lambda x:
x.strip().split('\t')).collect()).astype('float')
initial_center = q2_center[:,-3:]
number_of_data_each_cluster = q2_center[:,1]
broad_cluster_center = sc.broadcast(initial_center)
broad_number_of_data_each_cluster = sc.broadcast(number_of_data_each_cluster)
```

2. we apply the same calculation method used in task 2. But this time we keep distance in the value for removing 10% data with largest residual error

```
new_data_9_percent =
measurements_.map(cluster_function_1).groupByKey().flatMap(group_map_function)
```

The function calculating distance and the function to remove the 10% least correct data are as follows respectively:

```
# calculate the distance and cluster
def cluster_function_1(record):
    record = record[1]
    center = np.asarray(broad_cluster_center.value)
    data = np.asarray([record[0],record[1],record[2]]).astype('float')
    dis_number = (np.sum((center-data)**2,axis=1)**0.5)
    cluster_number = dis_number.argmin()
    dis = dis_number.min()
    # keep distance, not the same as previous cluster function
    return (cluster_number,(float(record[0]),float(record[1]),float(record[2]),dis))
```

```
# keep 90% most correct data
def group_map_function(record):
    number_ = broad_number_of_data_each_cluster.value
    total = []
    cluster_number = record[0]
    data = sorted(list(record[1]), key=lambda tup: tup[-1])
    number_of_cluster = int(number_[cluster_number]*0.9)
    for i in data[:number_of_cluster]:
        total.append(i[:-1])
    return total
```

3. The rest is just what we did in task 2, use the new 90% data to perform learning and get the final result.

Performance

Large Data Set

Task	Time
Task 1	Approximate 17s
Task 2	Approximate 6min 45s
Task 3	Approximate 6min

Small Data Set

Task	Time

Task 1	Approximate 8s
Task 2	Approximate 1min 40s
Task 3	Approximate 1min 20s

Appendix

To run the assignment, there is a `total.sh` to run in a whole. And for task 2 and task 3, an argument can given to define the learning iterations.

For *large data set*:

The output is in /home/lhan9852/pyspark/, with q1, q2, q3 folder corresponding to certain task.

For *small data set*:

The output is in /home/lhan9852/pyspark-small/, with q1, q2, q3 folder corresponding to certain task.