Group6 Project Report

Team Members

JL, LQ, ZW, JW
Github repo link: https://github.ccs.neu.edu/cs6240-f19/group6-Project

Project Overview

Have you ever experienced a flight delay? Have you ever thought about why your flight got delayed? Driven by this curiosity and backed up by the solid flight delay data from the Department of Transportation, our team, group 6, decided to use data mining techniques to help understand the flight delay data. K-means clustering algorithm was our choice of tool, as it could efficiently gather similar flight delay situations together, and could easily be applied to even large datasets.

We first implemented k-means clustering algorithm to run parallely on a fixed k program. Then as we want to find the best possible k for the given data, we also implemented the algorithm to run a range of k and starting configurations in parallel. The optimization, speed up, scalability of our implementation are also addressed. With some optimizations, our program can cluster 2 million data points with 20 different K and starting configuration in parallel on AWS. Once the clustering process finished, whenever a new data point comes in, we can quickly assign the new data point to one of the clusters and potentially infer the delay reason.

Input Data

We collected our dataset from a website data.world. Our data is about airline on-time statistics and delay causes. The dataset contains on-time performance of domestic flights operated by large air carriers tracked by the U.S. Department of Transportation's (DOT) Bureau of Transportation Statistics (BTS). BTS began collecting details on the causes of flight delays in June 2003 and renewed the report every month. It is stored in CSV format. There are about 2 million records, each with 24 attributes describing properties such as year, month, day, day of week, departure time, arrival time, CRS departure time, departure delay, airtime and distance. Our dataset only has data in year 2008.

For the data preprocessing part, we first filter the dataset and only keep the numeric data for our K-Means model, then transfer the ‘date’ into ‘Day of Year’ for more meaningful use and ‘time clock’ into ‘minute of day’, and we scale all columns to a range of 0 to 1 using MinMaxScaler method.

Here is an example for our input data: after data preprocessing, we have about 2 million records, each with 13 attributes, and every attribute is of type Double.

DayO	DayO	CRSDe	CRSAr	DepTir	ArrTim	CRSEla	ArrDe	DepD	AirTim	Distanc	TaxiIn	TaxiOut
0.01	0.5	0.83	0.93	0.835	0.92	0.25	0.02	0	0.175	0.161	0.019	0.0209

[K-means clustering, Spark, (1)]

Overview

This program is focusing on implementing the distributed K-means clustering algorithm with a predefined k and number of iterations. Using multiple tasks for each iteration, we have every task computing a subset of data concurrently, and the task will assign each data point to a new center and then renew the centers. The iterations are stopped until convergence or when it reaches 200 iterations.

After that, we calculate the sum of squared error, and applied it to different ks, trying to find the best k. We also analyze the speedup and scaleup of our program by increasing the number of machines and change the input size respectively.

Pseudo-code

Link to our code:
https://github.ccs.neu.edu/cs6240-f19/group6-Project/blob/master/task1/src/main/scala/kmeans/DistributedKmeans.scala

```
//--------------------main program----------------------
//define parameters
//threshold is used to check whether the result is converge
val threshold = 0.001
//k is the number of clusters
val k = 25

//read from csv
var df = spark.read.option("header", true).csv(args(0))

//change all columns to double
val colNames = Seq("DayofYear","DayOfWeek","CRSDepTime","CRSArrTime","DepTime","ArrTime","CRSElapsedTime","ArrDelay","DepDelay",
  "TaxiIn","TaxiOut")
val myschema = StructType(colNames.map(field => StructField(field, DoubleType,true)))
val toDouble = udf[Double,String](_.toDouble)
for (i <- 0 to colNames.size - 1){
  df = df.withColumn(colNames(i), toDouble(df(colNames(i))))
}

//initialize k centroidsd by randomly select k points from dataset
var centroid = df.sample(false, 0.5).limit(k)

//main part of k means clustering
//iteration begins, set a break condition if the result is converge
breakable {
  for (i <- 1 to 200) {

    //broadcast centroids to all machines
    val centerlist = broadcast(centroid).collect()

    //assgin data points based on current k centers
    val new_assignment = df.rdd.map {
          //store as an rdd in which key is the cluster id, value is the coordinates
      row => {
        val rowSeq = row.toSeq.asInstanceOf[Seq[Double]]
        val cluster_idx = nearestCenter(rowSeq, centerlist)
        (cluster_idx, rowSeq)
      }
    }
```

```
    //get new center by taking the average of those points in the cluster
    //change value to (value, 1) in order to count number of points inside each cluster
    val new_center_coord = new_assignment.mapValues(value => (value, 1)).reduceByKey {
      case ((p1, c1), (p2, c2)) =>
        val res = Seq[Double](p1(0) + p2(0), p1(1) + p2(1), p1(2) + p2(2), p1(3) + p2(3), p1(4) + p2(4), p1(5) + p2(5), p1(6) +
        (res, c1 + c2)
    //sort by key to guarantee that further update step works good
    }.sortBy(_._1).map {
      case (cluster, (p1, c1)) =>
        val res = Seq[Double](p1(0) / c1, p1(1) / c1, p1(2) / c1, p1(3) / c1, p1(4) / c1, p1(5) / c1, p1(6) / c1, p1(7) / c1, p1
        (cluster, res)
    }

    val new_center = new_center_coord.map(x => x._2)
    //check convergence
    val total_sum_distance = calculateTotalDistance(new_center.collect(), centerlist)
    if (total_sum_distance < threshold || i == 200) {

      //calculate SSE(sum of squared error)
      val sse = new_assignment.join(new_center_coord).mapValues{
        case (center, data) => calculateDist(center, data)
      }.map(_._2).reduce(_ + _)

      println("Loop stop here, when i = " + i.toString)
      println("Sum of squared when k = " + k.toString + " , is " + sse.toString)
        break
    }

    //convert rdd back to dataframe and update centroid
    centroid = spark.createDataFrame(new_center.map(Row.fromSeq(_)), myschema)
  }
}
println("Debug String is here")
logger.info(centroid.rdd.toDebugString)
centroid.rdd.saveAsTextFile(args(1))
```

Algorithm and Program Analysis

In this part, we try to distribute our task in parallel in each iteration. To achieve this, we need to broadcast k centroids into different machines and parallel our dataset, then re-assign part of points in different machines and update our centroids.

To be more precise, first we randomly select k sample points from our dataset as initial state of k centroids, and broadcast it. Then we write a function to calculate which cluster should this point be assigned to given that point and k coordinates of cluster centers. In our codes, this function is called *nearstCenter*. Secondly, convert dataset to rdd and use map to find its corresponding cluster number that it has been assigned to, store the cluster index and point coordinate as a key-value pair, so that we have a new pair RDD, named *new_assignment*.

Our next step is how to update our center based on the new assignment result. Here we need two values, the sum of all the points in each cluster and how many points in each cluster. To calculate this, we first add 1 to each record, in other words, for each key in our pair RDD *new_assignment,* use mapValues to change the value from point coordinate to a tuple (points coordinate, 1). Then customize the reduceByKey function to sum up the value, so that we get a new tuple (sum of point coordinate, total count) as value for each key (cluster index), which is exactly the updated centroid, named *new_center*. Finally, convert *new_center* into the DataFrame that has the same schema with previous centroids, and update it. And this is one iteration in this task. Repeat this process until converge, we get our final result. Here we sum up the distance between the new centers and the old one, check whether this number is less than a predefined threshold as a standard to figure out whether the result is converge or not.

Final step is to calculate the sum of squared error in order to find the optimal value of k.

Experiments

To test our code, we use the data mentioned in previous section. Set the number of iteration = 200 and in each iteration, check whether our center coordinate is converged or not by summing up all the distance between old and new centers. If this sum value less than a threshold, here we take threshold = 0.001, then we think the result is converged and break the loop, save our result.

After getting the result, we then need to evaluate our choice of k by calculating the sum of squared error(SSE), here we defined SSE as below:

$$SSE = \sum_{K} \sum_{i \in cluster\ k} distance(point_i, center_k)$$

Here K is the total number of clusters. We sum up the distance between all the points in each cluster and the center of that cluster as an evaluation score for comparing among different options of k value. The result will be shown in Result Sample part.

Task 1 Output file on aws:
https://github.ccs.neu.edu/cs6240-f19/group6-Project/tree/master/task1/output_aws

Task 1 Log file on aws:
https://github.ccs.neu.edu/cs6240-f19/group6-Project/tree/master/task1/log_aws

Speedup:

Number of workers(1 million)	Running time	Number of workers(2 million)	Running time
5	303s	5	960s
10	194s	10	384s

We try 5 workers (m4.large, following the same) and 10 workers on 1 million data and 2 million data, the result is in above table. We can see that when we add 5 more workers, the running time becomes around 40-60% of previous time. This result might be influenced by initialization of k centers, but the overall speedup is good.

Scaleup:

Input size	Running time change
100,000	+ 37s (97s)
1,000,000	+ 206s (303s)
2,000,000	+ 657s (960s)

Here we try three different input size, 0.1 million, 1 million and 2 million as a simplification of calculating scaleup. From the above table, we can indicate that when the input increase to 10 times, which is, input size increase from 0.1 million to 1 million, the running time only increases to about 3 times, which is from 97s to 329s. But if we double the input size, in other words, increase input size from 1 million to 2 million, the running time increase 3 times.

Result Sample

Our output are the final values of centers, and figure 1 shows one sample center.

```
[0.2566655509256553,0.8210113982897153,0.7278727359243095,0.8161752836181235,0.7567256953038172,0.84420762691199,0.2248497869035598,0.04339814617906072,0.01489540653528401,0.15984822069289356,0.14717237532514468,0.033578332973277114,0.044836777455525646]
```

Figure 1: center sample

For the later part, we are going to choose the best k and print out the k and its corresponding SSE. Figure 2 shows one sample of k and its SSE.

```
Loop stop here, when i = 43
Sum of squared when k = 15 , is 328191.35317630426
```

Figure 2: printed k and SSE

Gathering all the results for k and corresponding SSEs, we plotted a graph to better known how to choose the best k. In figure 3, we have k of value 5 to 20 and their SSE. The SSE decreases steeply from k = 5 to k = 10. After 10, there is no obvious better performance and the SSE decreases much more slower than previous by increasing k. Also in reality, there are not so many reasons or clusters for delay, so we choose k = 10 as our best k.

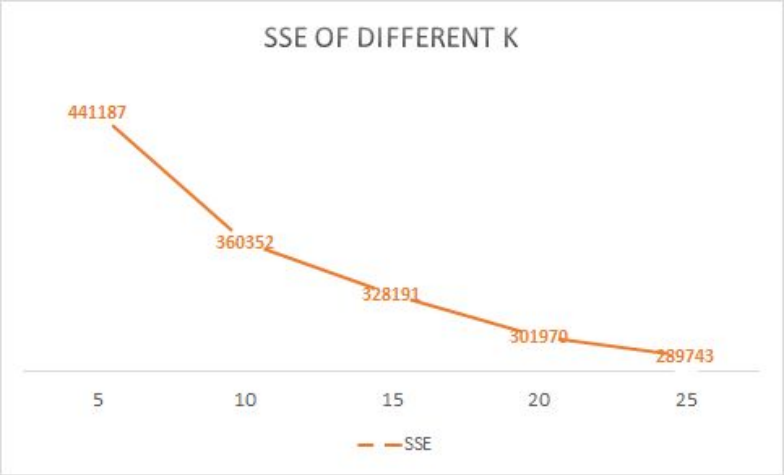

Figure 3: SSE of different k for distributed k means program

[K-means clustering, Spark, (2)]

Overview

This program is focusing on implementing a local k-means algorithm that computes an entire clustering for a single K in a single task. The parallelism is achieved by running with different starting centers and different values of K concurrently. Running with various starting configurations and Ks can potentially show what's the best cluster number and starting configuration for the given input.

Pseudo-code

Link to our code:
https://github.ccs.neu.edu/cs6240-f19/group6-Project/blob/master/task2/src/main/scala/localkmeans/LocalKMeans.scala

```
val getCenter // All the initial center which we got through preprocessing
val resultCenter: RDD[(String, List[List[Double]])] = getCenter.map{
```

```
        case(k,v)=> (k,kmeans(v,tmpData, NumberOfIteration)) // call local kmeans algorithm
}
resultCenter.saveAsTextFile(args(1))

// calculate the distance between point and one center
def calculateDist(data:List[Double], centerPoint:List[Double]): Double = {
        val n = centerPoint.length
        var distance: Double = 0.0
        for (i <- 0 until n - 1){
                //for each dimension, calculate distance
                val tmp_dist = math.pow(data(i) - centerPoint(i), 2)
                distance += tmp_dist
        }
        math.sqrt(distance)
}

/**
 *  local kmeans algorithm.
 * @param klist: a list of all the candidates of k and it's initial centers
 * @param data: all the data in an array of row
 * @return: a list of final centers after k-means for all the candiddates k
 */
def kmeans(klist:List[List[Double]], data: Array[Row], iter:Int) : List[List[Double]] ={
        var kOriginal = klist
        // multiple iteration of algorithm before converge
        for(iteration<- 1 to iter){
                // record number of data in the cluster
                var countForEachK = Array.fill(kOriginal.length)(0)
                 // record the sum of the distance of all point to cluster
                var dataForEachK= new ListBuffer[List[Double]]()
                for (i<-kOriginal.indices){
                        var oneCenter = List.fill(kOriginal(i).length)(0.0)
                        dataForEachK += oneCenter
                }
                for (i<-data.indices){
                        val one_data = dataStringToDouble(data(i))
                        var min_distance = Double.MaxValue
                        var index = -1
                        val one_data_asList = one_data.toSeq.asInstanceOf[List[Double]]
                        for(i<-kOriginal.indices){
                                val new_distance = calculateDist(kOriginal(i), one_data_asList)
                                if(min_distance>new_distance){
                                        min_distance = new_distance
```

```
                                        index = i
                                    }
                                }
                    countForEachK(index) = 1+countForEachK(index)
                    for (k <- dataForEachK(index).indices) {
                                dataForEachK(index)          =          dataForEachK(index).updated(k,
dataForEachK(index)(k) + one_data_asList(k))
                    }
          }
          for (i<-dataForEachK.indices){
                    var tmp = dataForEachK(i)
                    for (k<-dataForEachK(i).indices){
                                tmp = tmp.updated(k,dataForEachK(i)(k)/countForEachK(i))
                    }
                    kOriginal=kOriginal.updated(i,tmp)
           }
          }
kOriginal
}
```

Algorithm and Program Analysis

To parallelize the calculation of different K-means models, we run each K-means model independently on one working machine and different Ks on different machines concurrently. Therefore, the different initial centers (K) should be distributed on all machines while data points should be broadcast to each working machine. We achieve this by doing the following things:

First, we pre-generated several different settings (with a different number of clusters and different initializations for each cluster center) for K as the input. The program will read the input and partition it on different machines by making it as a paired RDD. The key of the paired-RDD is the number of clusters and the value is a list of all centers, in which each center is a list of Doubles (represent the dimension of input data). Then, for each pair-RDD, we run a map function, within which, the program locally runs the K-means model until convergence. The convergence condition is the same as the one mentioned in task 1. Here, we set the number of iterations to 100 since Kmeans model usually converges quickly within 30-50 iterations.

Second, the program will read in the data file, which is stored as a DataFrame and is collected, then broadcast to all machines. The local K-means is achieved by the K-means function. K-means function takes one set of centers and all data points as input, runs several iterations, and outputs new centers of the model.

We also use SSE which is the distance between all the points in each cluster and the center of that cluster to evaluate our model result. The total SSE of all clusters will decrease as long as K is increasing. But the model could be overfitting if k is too large. This is the classic problem of balancing variance and bias.

We use elbow method (https://uc-r.github.io/kmeans_clustering#elbow) to find the best K for our data. The best K would be the number that adding another cluster does not give much better modeling our data. We embed the evaluation in our program. After getting all the converged centers, we calculate the sum of SSE of all centers within in one model and output the corresponding result. Figure 4 below is the SSE graph for k ranges from 2 to 27. And we can easily see the elbow is around k = 12 - 15.

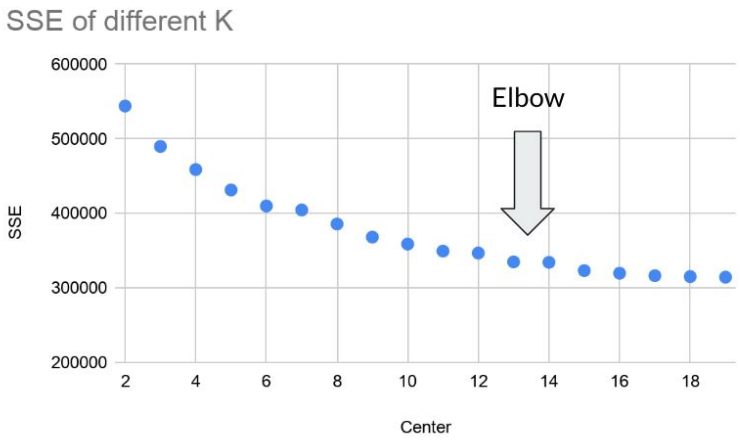

Figure 4: SSE for different K for local k means program

Experiments

Task 2 Output file on aws:
https://github.ccs.neu.edu/cs6240-f19/group6-Project/tree/master/task2/aws-log-output
Task 2 Log file on aws:
https://github.ccs.neu.edu/cs6240-f19/group6-Project/tree/master/task2/aws-log-output

Speedup:

Input size	Partition size	Running time change
1,000,000, k = 2-27	10 (5 machines)	1 hour 21 mins
1,000,000, k = 2-27	20 (5 machines)	57 mins (24 mins less)
1,000,000, k = 2-27	20 (10 machines)	57 mins (same as 5 machines)
1,000,000, k = 2-27	26 (7 machines)	45mins (36mins less)

We try different machines and different partitions as shown above. The program showed good speedup when having more partitions. If we use the same partition, running time does not change. That is because the input K is too small and using more machines is not necessary.

Scaleup:

Input size	Running time change
100,000	5min
500,000	26min
1,000,000	57min
2,000,000	Out of memory error

As the result shown above, the program exhibited good scaleup - running time is linearly increasing with the input dataset. However, we may encounter memory problem, which limits the input size our program could run.

Result Sample

Our output is a paired RDD. For each row in the output, key is the number of clusters (K). The value consists of two elements, one is the sum of SSE of all clusters, one is the final values of all centers, which is stored as list of list. Here is one example:

```
(2,(5336.994031657168,List(List(0.4652901128335397, 0.4774201297141483, 0.461375215210305,
0.5092979525531528, 0.4700617989278799, 0.5209826064587991, 0.1733444770240796,
0.06754852482745191, 0.02688980528806771, 0.17212728281969214, 0.14948897753409926,
0.0783003328643463, 0.046375080340970265), List(0.46893035606062433, 0.5143168646353344,
0.7288692364779006, 0.8181770590185143, 0.7607595556055877, 0.790917402975492,
0.1714879589807219, 0.07383345682985644, 0.033290716828813426, 0.16771796327227567,
0.1436776814489142, 0.0816057185483884, 0.05006676499188139))))
```

Figure 5: center sample

Conclusion

We successfully implemented the distributed k-means algorithm and the local k-means algorithm with different configurations of k computed distributely.

Task 1 mainly uses DataFrame to implement k-means, the algorithms is that we broadcast center points and parallel all the data points into different machines, then in different machines, re-allocate the data points. During implementation, we face a problem that when using foreach to loop through DataFrame, change of variable will not be applied once the loop is ended. Therefore, we try another thoughts by using map to get each row of DataFrame and call a user-defined function and get a pairRDD. Moreover, we also set a break condition once the

result is converged by checking the sum of distance between new and old cluster centers. And then, an evaluation method, Sum of Squared Error(SSE) is applied for k value choosing.

In task 2, we do local kmeans and compute different kmeans models parallely. By doing so, we could get several model results for a dataset within one run. However, the bottleneck of task 2 is memory, since the whole dataset is broadcast to all executors. If we would like to work with big data, we have to use large machines. Meanwhile, it is pretty hard to configure Spark. We modify the java heap size of our program in the makefile to make the most of the memory of our machines, still, it seems that we do not have 100% control about how Spark allocate resources when running the program. The better part is that we could leverage the advantages of these 2 tasks to achieve better results. We could first use task 2 with different configurations of k to find the best possible cluster initialization with a randomly selected sample data. Then we could use task 1 program with the best k initialization to get the result of the whole dataset.