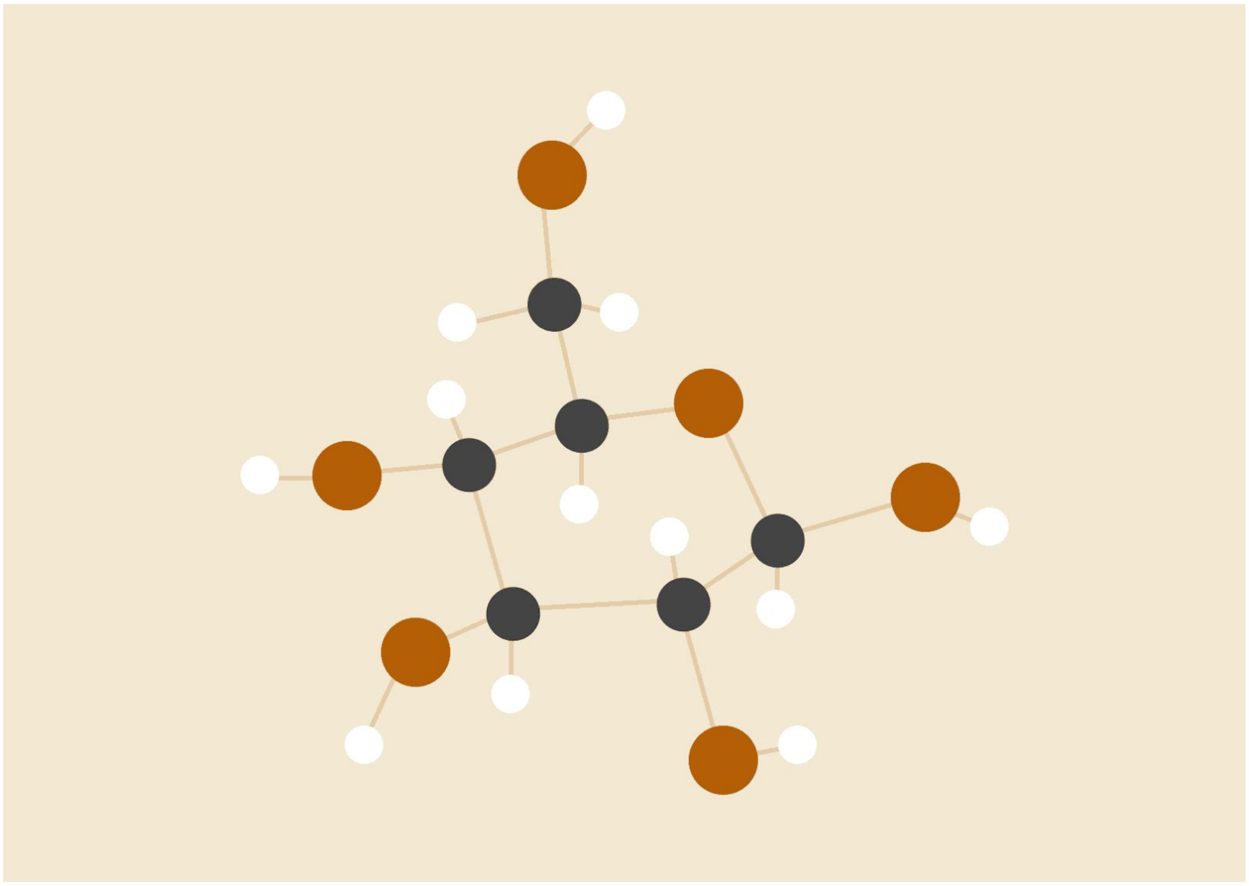
Distributed KMeans - Spark REPORT



Omar Ahmed Fouad - 40

Mostafa Fathy Youssef - 67

Youmna Moataz Dwidar - 73

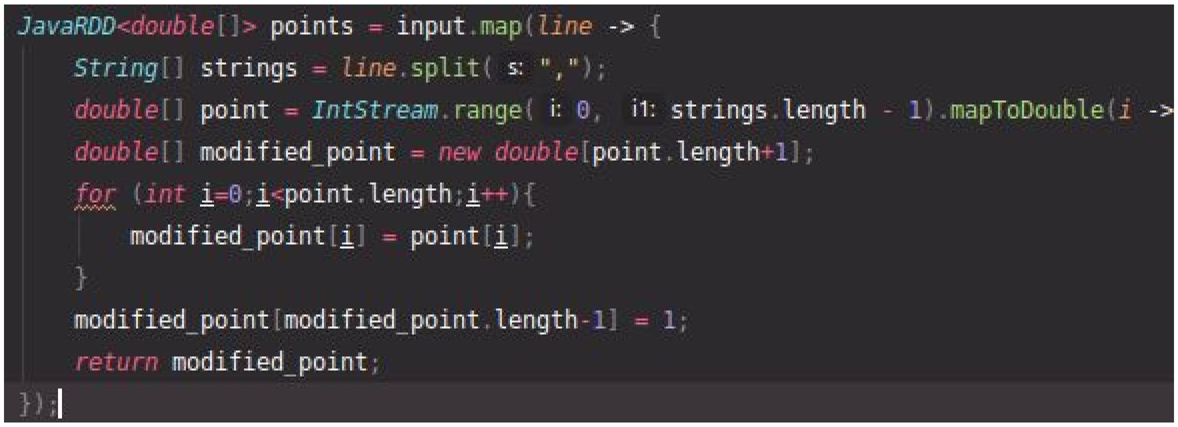
15.04.2020

Problem Statement

To implement a new parallel version of K-means clustering algorithm using the Spark map-reduce framework.

Then, evaluate your clustering algorithm using the IRIS dataset as compared to the original one. In terms of run time and clustering accuracy.

Spark - Implementation

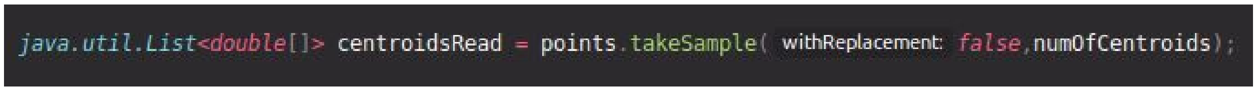


```
JavaRDD<double[]> points = input.map(line -> {
    String[] strings = line.split( s: ",");
    double[] point = IntStream.range( i: 0,  i1: strings.length - 1).mapToDouble(i ->
    double[] modified_point = new double[point.length+1];
    for (int i=0;i<point.length;i++){
        modified_point[i] = point[i];
    }
    modified_point[modified_point.length-1] = 1;
    return modified_point;
});
```

We mapped each line in the dataset to a point leaving last string which represents the class.

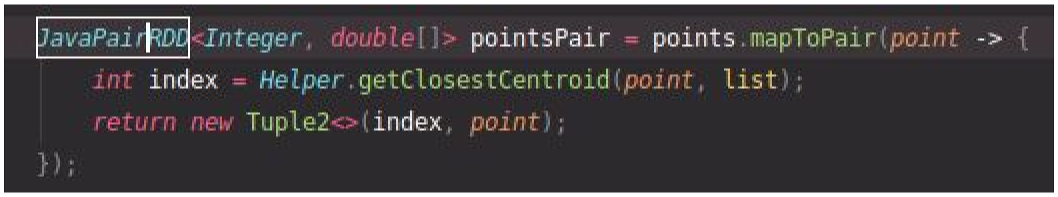
We've also added a new slot in each point which represent the number of points we averaged, It's 1 initially

After that we took number of random points to act as centroids



```
java.util.List<double[]> centroidsRead = points.takeSample( withReplacement: false,numOfCentroids);
```

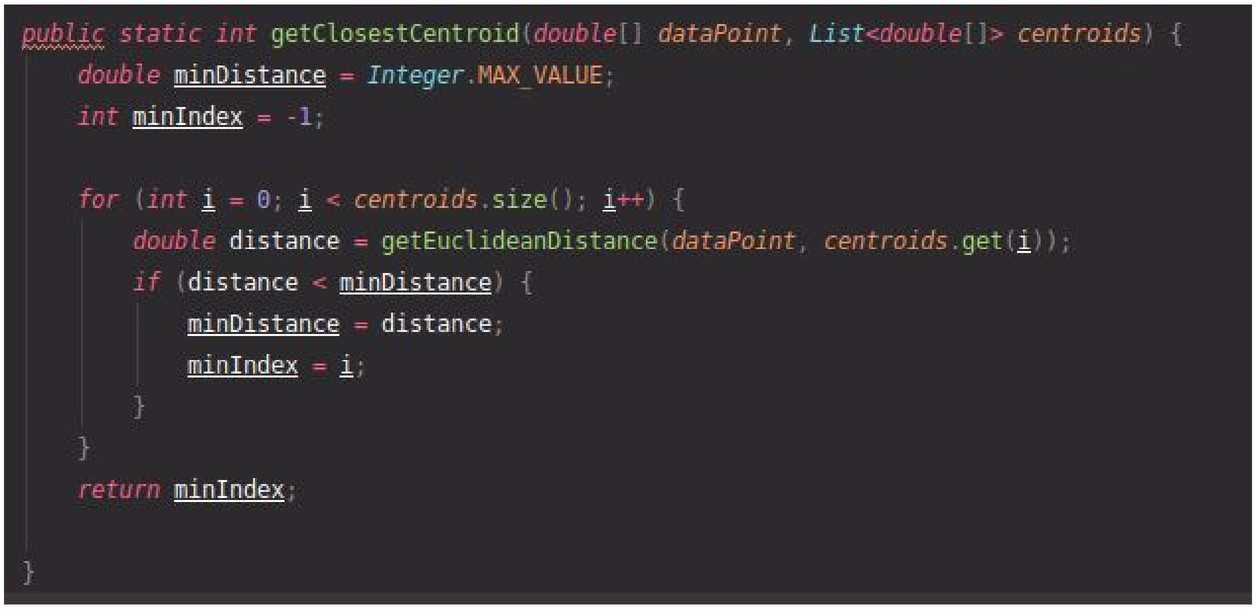
MAP



```java
JavaPairRDD<Integer, double[]> pointsPair = points.mapToPair(point -> {
    int index = Helper.getClosestCentroid(point, list);
    return new Tuple2<>(index, point);
});
```

Here we depend on a Helper function to identify the nearest centroid.

Every point is mapped to one of the centroids

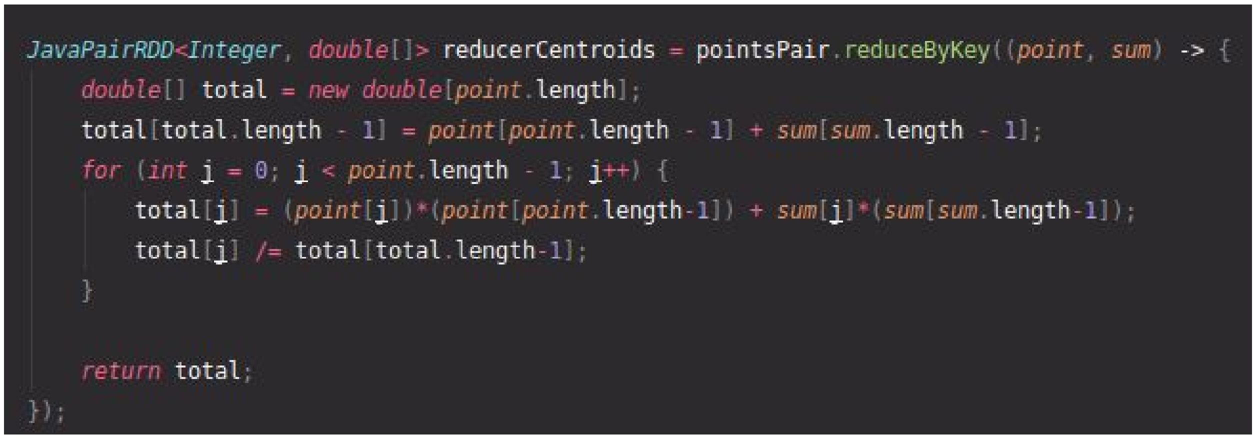


```java
public static int getClosestCentroid(double[] dataPoint, List<double[]> centroids) {
    double minDistance = Integer.MAX_VALUE;
    int minIndex = -1;

    for (int i = 0; i < centroids.size(); i++) {
        double distance = getEuclideanDistance(dataPoint, centroids.get(i));
        if (distance < minDistance) {
            minDistance = distance;
            minIndex = i;
        }
    }
    return minIndex;

}
```

Reduce



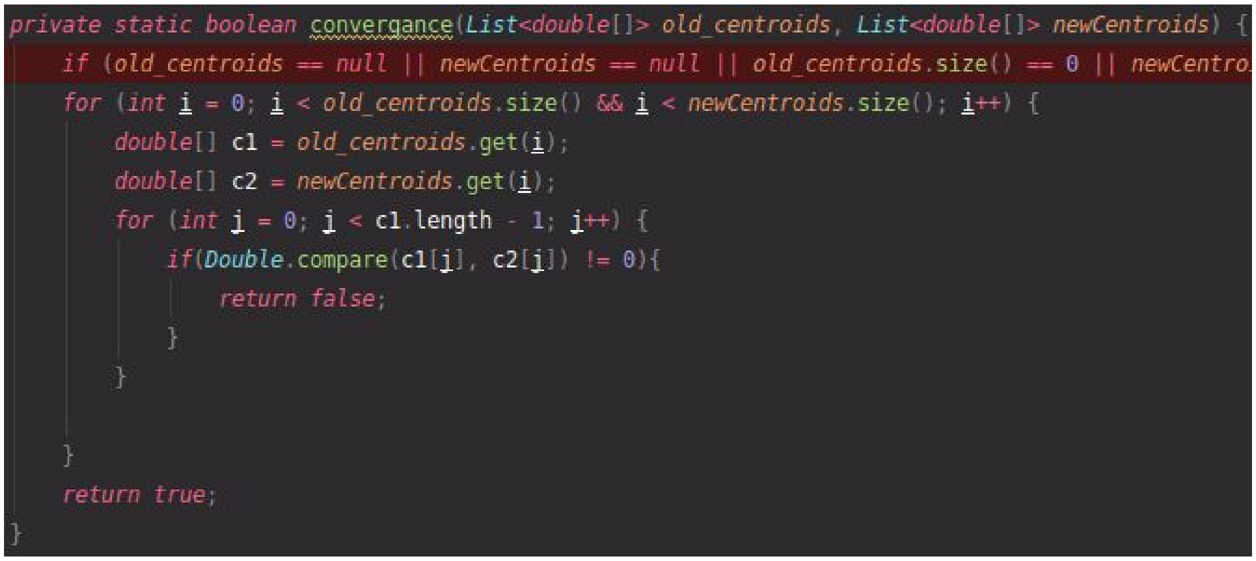
```
JavaPairRDD<Integer, double[]> reducerCentroids = pointsPair.reduceByKey((point, sum) -> {
    double[] total = new double[point.length];
    total[total.length - 1] = point[point.length - 1] + sum[sum.length - 1];
    for (int j = 0; j < point.length - 1; j++) {
        total[j] = (point[j])*(point[point.length-1]) + sum[j]*(sum[sum.length-1]);
        total[j] /= total[total.length-1];
    }

    return total;
});
```

In the Reducers we add the number of points in each mean to get the number of points merged.

At any mean point, we can get the count of points used and the final average.

Convergence



```
private static boolean convergance(List<double[]> old_centroids, List<double[]> newCentroids) {
    if (old_centroids == null || newCentroids == null || old_centroids.size() == 0 || newCentro
    for (int i = 0; i < old_centroids.size() && i < newCentroids.size(); i++) {
        double[] c1 = old_centroids.get(i);
        double[] c2 = newCentroids.get(i);
        for (int j = 0; j < c1.length - 1; j++) {
            if(Double.compare(c1[j], c2[j]) != 0){
                return false;
            }
        }

    }
    return true;
}
```

Every iteration we try to compare the previous centroids to the new centroids, if any change was observed we return false meaning that it didn't converge yet.

Accuracy

We decided to measure the model performance by calculating the sum of the euclidean distance between the centroid and the points that belong to its cluster.

The smaller the resulted sum, the better the model.

This table presentes this sum for the 3 clusters for multiple runs.

Runs / cluster	1	2	3
Run 1	0.7198385488470926	0.738152369268767	0.4841322496689401
Run 2	0.73184587835359	0.4841322496689401	0.7311084910642589
Run 3	0.7311084910642589	0.4841322496689401	0.73184587835359
Run 4	0.4841322496689401	0.7311084910642589	0.73184587835359
Run 5	0.7311084910642589	0.4841322496689401	0.73184587835359
Run 6	0.4841322496689401	0.7311084910642589	0.73184587835359