//王泽宇_181250148_金融大数据处理技术作业 7
//Git 仓库链接：https://github.com/ZeyuuOne/MapReduceKMeans

一、文件组织结构说明

Git 仓库的文件组织结构如下：

目录名或文件名	说明
build/libs	jar 文件所在目录
gradle/wrapper	gradle-wrapper 所在目录
src/main/java	源代码所在目录
.gitignore	.gitignore 文件
build.gradle	gradle 编译配置文件
gradlew	gradlew Linux Shell 脚本
gradlew.bat	gradlew Windows 脚本
README.pdf	作业报告
settings.gradle	gradle 配置文件

二、MapReduce 程序设计

程序目的为使用 K-Means 算法实现对给定点集的划分，其中划分的数量由用户输入。设划分数量为 k，则程序设计如下：

1. 随机选取点集中的 k 个点作为初始划分集合的中心；
2. 对点集中的所有点，计算其与各划分中心之间的距离，取距离最小的划分中心所对应的划分集合作为该点所在的划分集合；
3. 对所有划分集合，重新计算其中心；
4. 重复步骤 2 和 3，直到计算得到的划分集合不再变化。

其中距离指欧氏距离，集合中心即各点坐标的算数平均值对应的点。

观察发现，只需划分中心的坐标，即可确定一个点集上的划分。同时，对每个点与各划分中心之间的距离的计算只需用到该点坐标和各划分中心的坐标。因此，该程序可以并行运行，各分布式节点只需存储点集中部分点的坐标和所有划分中心的坐标。

上述程序设计只需 1 个 MapReduce Job 即可实现，其中第 2 步在 Map 阶段完成，第 3 步在 Reduce 阶段完成，最终输出新的划分中心坐标。但为了输出点集的划分结果，即输出点集中各个点对应的类别，需要额外定义 1 个 MapReduce Job，在 Map 阶段执行与上述第 2 步相同的计算，输出各个点对应的类别。

三、MapReduce 程序实现

观察参考代码，发现其 Cluster 类定义了划分集合的数据结构，Instance 类定义了点的数据结构，KMeans 类定义了上述程序设计中的第 1 个 Job 所需的 Mapper、Combiner 和 Reducer，KMeansCluster 类定义了上述程序设计中的第 2 个 Job 所需的 Mapper，

KMeansDriver 类定义了上述两个 Job 和主程序，RandomClusterGenerator 类用于随机生成初始的划分中心。各个模块的代码清晰明了，且注释充分，此处不再一一分析。

观察参考代码 KMeansDriver 类发现，参考程序未考虑划分的收敛情况，即使划分集合已不再变化，程序仍会运行用户指定的迭代次数，非常浪费算力和时间。

因此在 KMeansDriver 类中定义 readPoints()函数，用于在 Job 之外读取文件中的点集坐标。由于 MapReduce Job 在默认情况下输出的文件会被分为输出路径下的多个文件块，因此读取文件时需要读取目录下的所有文件。

```
1.  public static ArrayList<ArrayList<Double>> readPoints(Configuration conf, St
    ring inputPath) throws IOException {
2.      FileSystem fs = FileSystem.get(conf);
3.      FileStatus[] fileList = fs.listStatus(new Path(inputPath));
4.      ArrayList<ArrayList<Double>> points = new ArrayList<ArrayList<Double>>()
    ;
5.      for (FileStatus fileStatus : fileList) {
6.          BufferedReader fis = new BufferedReader(new InputStreamReader(fs.ope
    n(fileStatus.getPath())));
7.          String line = null;
8.          while ((line = fis.readLine()) != null) {
9.              ArrayList<Double> point = new ArrayList<Double>();
10.             StringTokenizer itr = new StringTokenizer(line, ",");
11.             if (!itr.hasMoreTokens()) break;
12.             int i = 0;
13.             while (itr.hasMoreTokens()) {
14.                 if (i != 0 && i != 1) point.add(Double.parseDouble(itr.nextT
    oken()));
15.                 i++;
16.             }
17.             points.add(point);
18.         }
19.         fis.close();
20.     }
21.     return points;
22. }
```

而后在 KMeansDriver 类中定义 run()函数，在每一轮迭代后读取当前的划分中心，与上一轮迭代得到的划分中心做比较，若相同则跳出循环。同时，为了展现迭代过程中划分情况的变化，在每一轮迭代后都使用 KMeansClusterJob()函数输出划分结果。

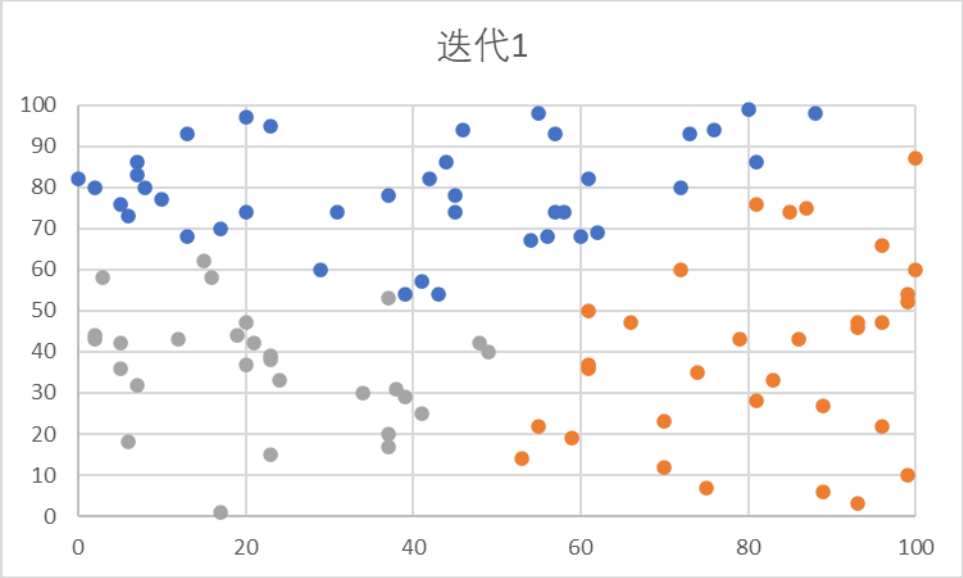
```
1.  public void run() throws IOException, ClassNotFoundException, InterruptedExc
    eption {
2.      generateInitialCluster();
3.      ArrayList<ArrayList<Double>> clusters = readPoints(conf, outputPath + "/
    cluster-0/");
```

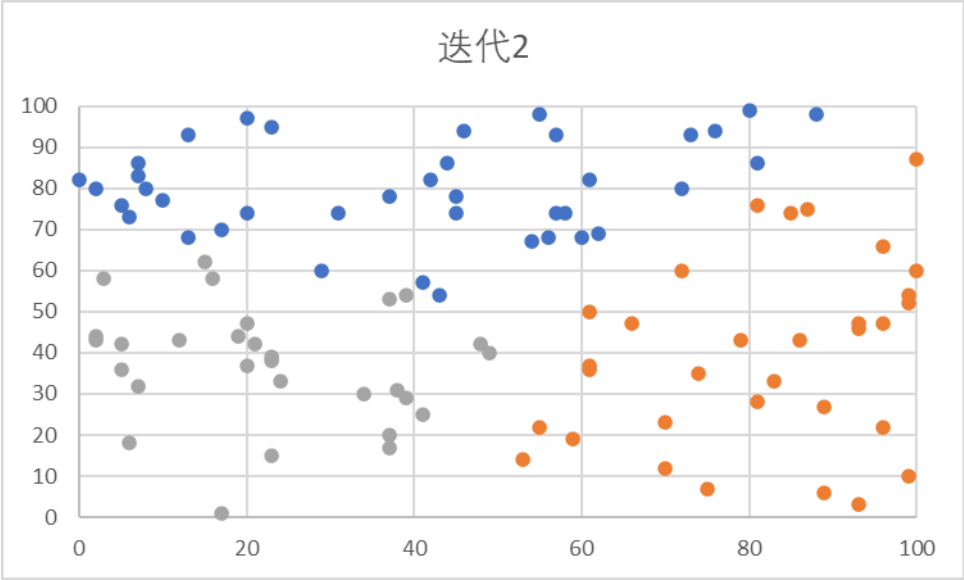
```
4.      int i = 0;
5.      for (; i < iterationNum; i++) {
6.          clusterCenterJob(i);
7.          KMeansClusterJob(i);
8.          ArrayList<ArrayList<Double>> newClusters = readPoints(conf,outputPat
    h + "/cluster-" + (i + 1) + "/");
9.          if (newClusters.equals(clusters)) break;
10.         clusters = newClusters;
11.     }
12. }
```

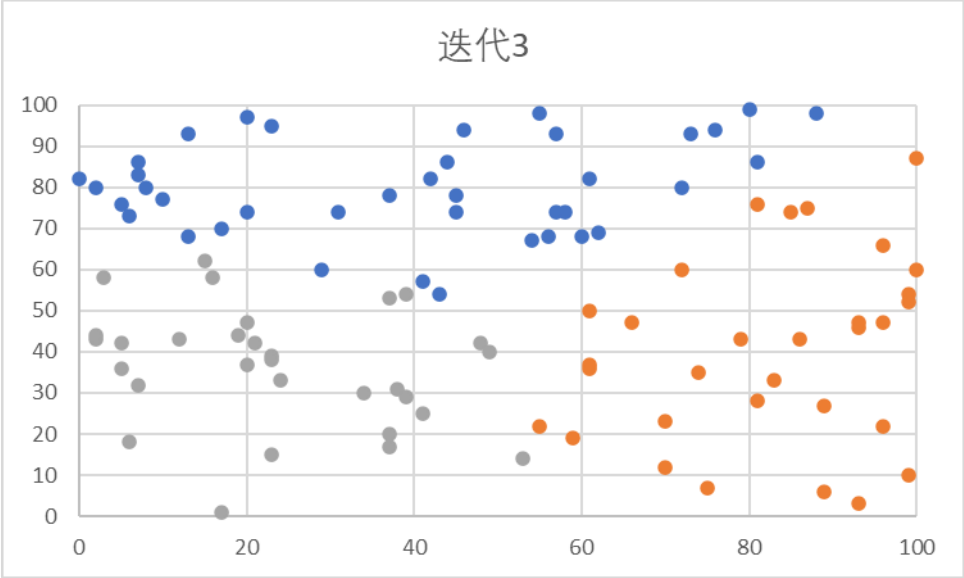
最后修改 main()函数，直接调用 driver.run()，MapReduce 程序即编写完成。

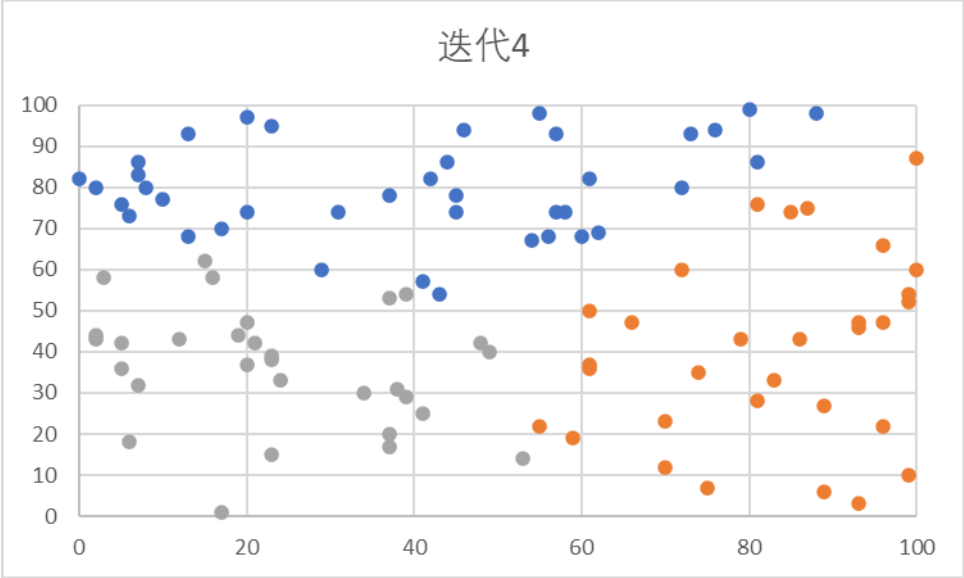
四、集群运行测试

在 Hadoop 集群运行程序，首先设置 k 为 3，发现 4 次迭代后结果收敛。



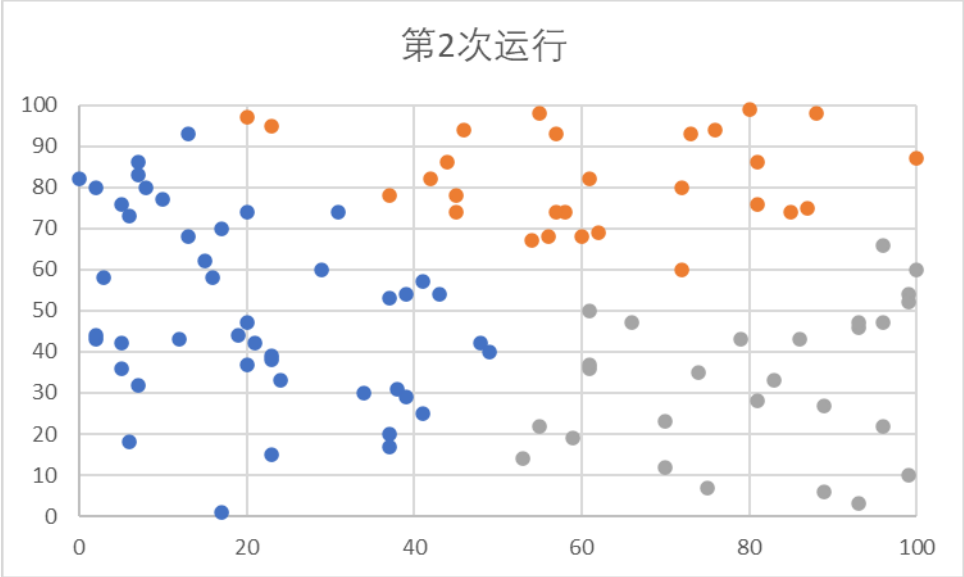


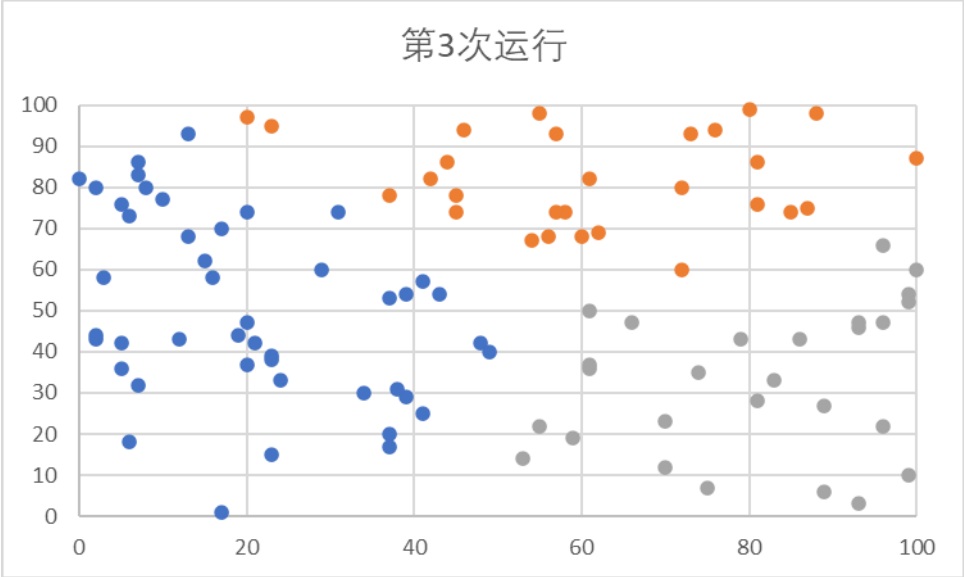




观察发现，每次迭代时仅有一个点的分类所有变化，初始划分和最终结果很接近。

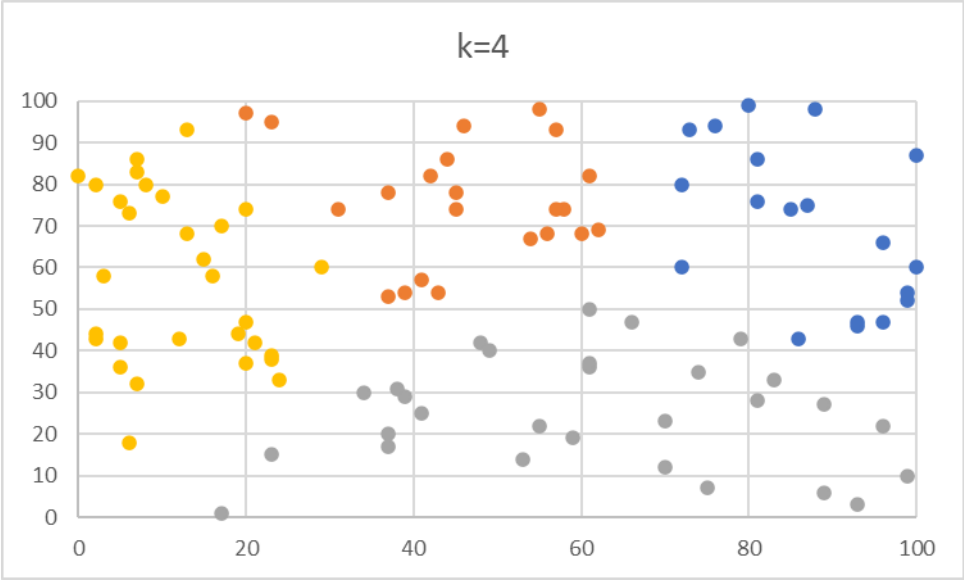
仍然将 k 设置为 3，再运行 2 次程序，观察最终划分结果。

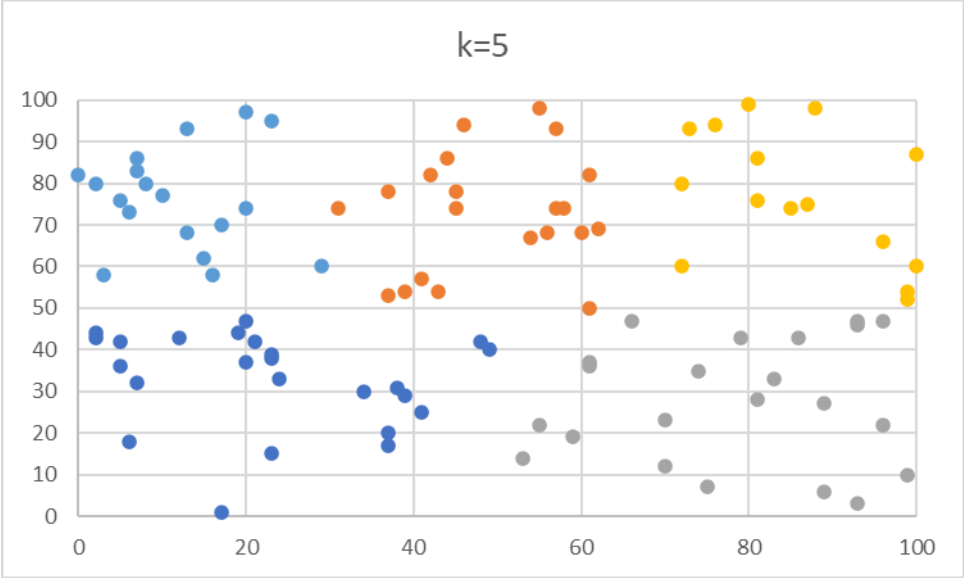


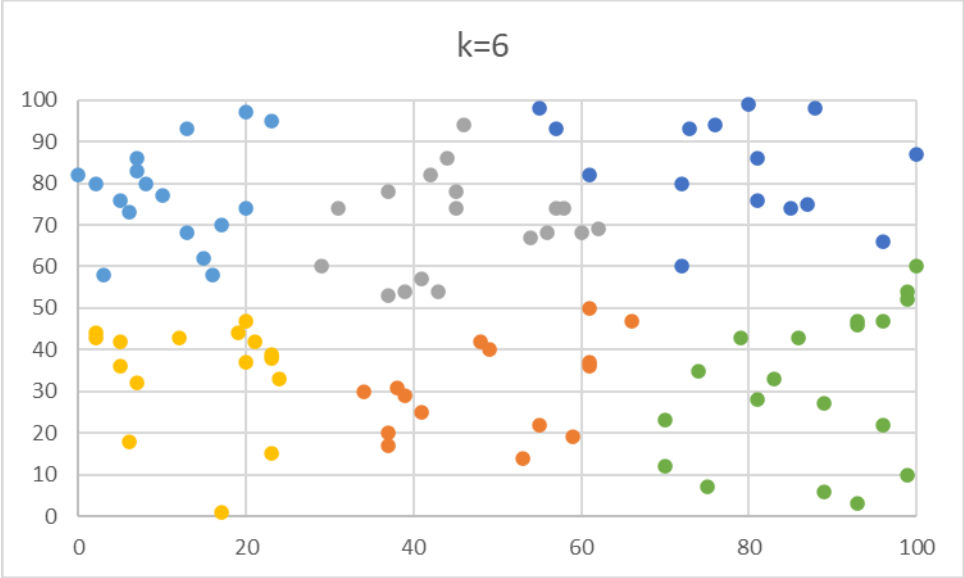


发现第 2 次和第 3 次运行得到的结果完全一致，与第 1 次运行得到的结果有所不同。同时也发现，第 2 次和第 3 次运行时初始划分中心并不相似，其迭代轮数分别为 7 和 4。可见即使是随机数据集，K-Means 算法仍具备一定稳定性。

将 k 设置为 4 至 6，分别运行 1 次程序，观察最终划分结果。


k=4
100
90
80
70
60
50
40
30
20
10
0
0
20
40
60
80
100


k=5
100
90
80
70
60
50
40
30
20
10
0
0
20
40
60
80
100


k=6
100
90
80
70
60
50
40
30
20
10
0
0
20
40
60
80
100

可见划分效果符合预期。

五、思考与讨论

实际上，第 2 个 Job 输出划分结果所需要的每个点的划分信息在第 1 个 Job 的计算过程中就已得到，但由于 Hadoop MapReduce 框架中同一个 Job 仅支持输出一种类型的 Key-Value 对，因此无法将 Job 内部的中间结果输出，只能再定义一个 Job 专门负责输出划分结果，这样浪费了算力和时间。若能够通过一些方式保留 Job 的中间结果，应当能够大幅提升程序运行效率。