题目：使用 MapReduce 实现 KNN 算法　　2018 年 1 月 19 日

一、　内容：

KNN 算法（K-NearestNeighbor，也称 K 近邻算法）指每个样本都可以用与它最接近的 K 个邻居来代表。其核心思想是如果一个样本在特征空间中的前 k 个最相邻的样本中的大多数都属于某一类别，那么这个样本就判定为该类别。

Hadoop 是一种分布式算法基础架构，用户在不了解分布是底层细节的情况下，开发分布式程序，利用集群来进行高速运算和存储。Hadoop 的架构的核心设计是 HDFS（分布式文件系统）和 MapReduce，其中 HDFS 为海量数据提供存储，MapReduce 提供了高速运算。MapReduce 是一种并行编程模型，它将大数据集按块进行分割，然后将数据块分到各个节点上，根据用户自定义的 Map 和 Reduce 函数执行相应的操作，它的优点是多个节点并行工作，大大提高了运算速度。

综合 KNN 算法自身的特点和 Hadoop 并行计算的优点，我们此次实验的算法是基于 Hadoop 的架构来实现 KNN 算法，以提高 KNN 算法处理大数据的能力。我们通过设计 KnnInstance 类、Map 函数和 Reduce 函数来实现 KNN 算法的并行化。

二、　实现过程：包括对框架结构（什么环境搭建、每一部分实现的内容）、代码的解释、实现难点、关键技术点等；

1、 框架架构

环境搭建：

- 启动 hdfs 和 yarn，如下图，各个结点的对应的服务开启。

```
hadoop@master:~$ jps
1392 NameNode
1890 ResourceManager
1747 SecondaryNameNode
1559 DataNode
2235 Jps
2028 NodeManager
hadoop@master:~$
```

```
hadoop@slaver1:~$ jps
1155 DataNode
1300 NodeManager
1433 Jps
hadoop@slaver1:~$
```

```
hadoop@slaver2:~$ jps
1156 DataNode
1301 NodeManager
1420 Jps
hadoop@slaver2:~$
```

- 编译 KnnInstance.java 和 Knn.java 成 class 文件

```
hadoop@master:~$ javac KnnInstance.java
hadoop@master:~$ javac Knn.java
```

- 将编译的 KnnInstance.class 和 Knn.class 文件打包成 Knn jar 包。

```
hadoop@master:~$ jar -cvf Knn.jar Knn*.class
added manifest
adding: Knn.class(in = 1609) (out= 809)(deflated 49%)
adding: KnnInstance.class(in = 1015) (out= 573)(deflated 43%)
adding: Knn$KnnMap.class(in = 4194) (out= 1984)(deflated 52%)
adding: Knn$KnnReducer.class(in = 2902) (out= 1335)(deflated 53%)
hadoop@master:~$
```

- 可以看到在 home 目录下有关于 Knn 的代码文件和生成的 Knn.jar

题目：使用 MapReduce 实现 KNN 算法　　2018 年 1 月 19 日

```
hadoop@master:~$ ls
apache-hive-1.2.0-bin.tar.gz  KnnInstance.java                    setenv.sh
authorized_keys               Knn.jar                             spark-1.6.3-bin-hadoop2.6.tgz
hadoop-2.6.0.tar.gz           Knn.java                            SparkPiRes.txt
id_rsa.pub                    Knn$KnnMap.class                    test.txt
jdk-8u60-linux-x64.tar.gz     Knn$KnnReducer.class                traindata.txt
Knn.class                     META-INF
KnnInstance.class             mysql-connector-java-5.1.44-bin.jar
hadoop@master:~$
```

- 将训练文本和测试文本放至 hdfs 上

```
hadoop@master:~$ /usr/local/hadoop/bin/hdfs dfs -put traindata.txt /input
hadoop@master:~$ /usr/local/hadoop/bin/hdfs dfs -put test.txt /input
hadoop@master:~$
```

- 通过 Knn.jar 包，Knn 作为主函数，以 traindata.txt 为输入参数，执行 Knn.jar 中的代码，并将运行结果防止 hdfs 上 output/RESULT 文件夹中。

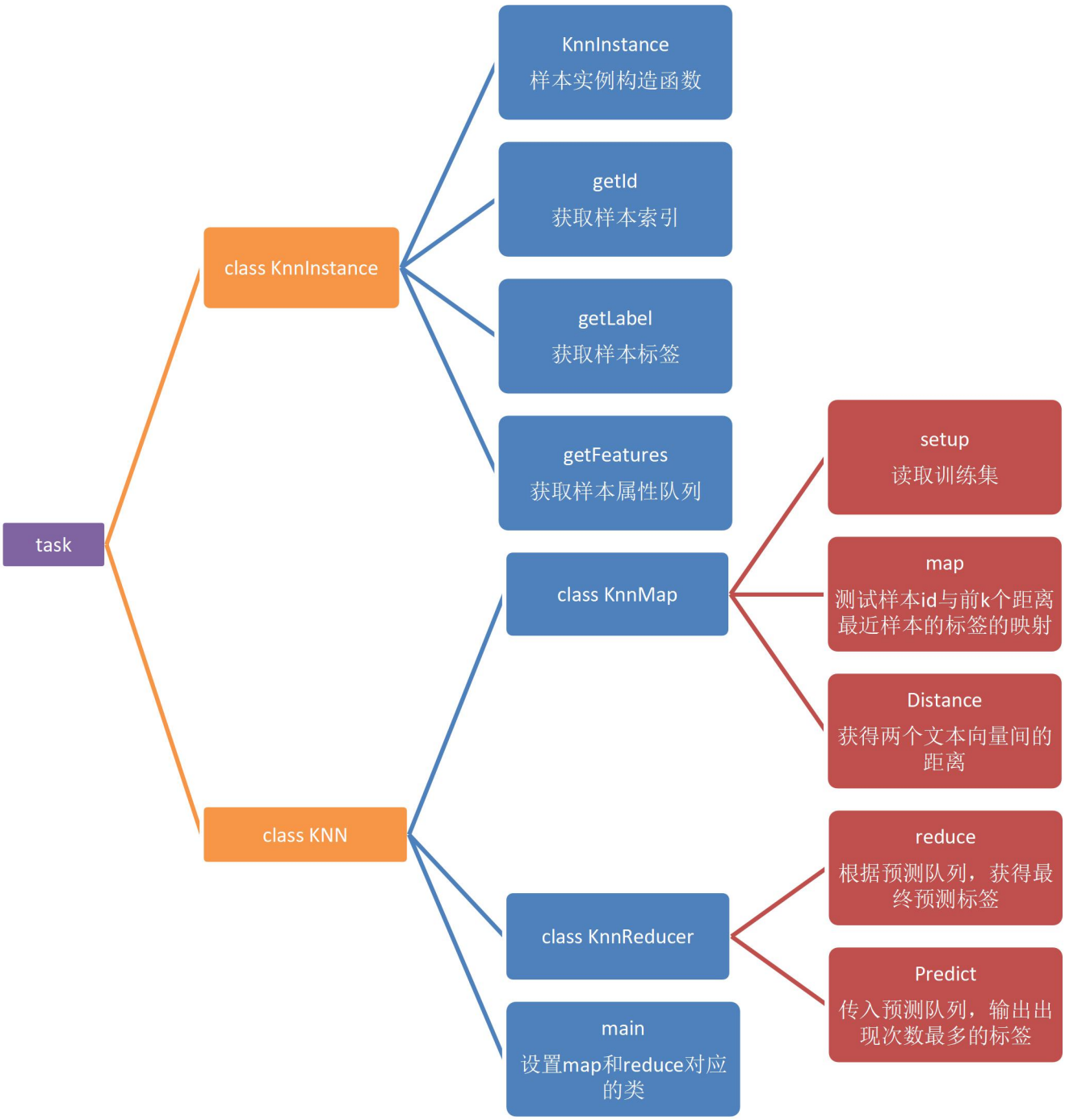
```
hadoop@master:~$ /usr/local/hadoop/bin/hadoop jar Knn.jar Knn /input/traindata.txt /output/RESULT
```

- 程序运行时截图

```
hadoop@master:~$ /usr/local/hadoop/bin/hadoop jar Knn.jar Knn /input/traindata.txt /output/RESULT
18/01/20 22:06:49 INFO client.RMProxy: Connecting to ResourceManager at master/192.168.142.130:8032
18/01/20 22:06:51 WARN mapreduce.JobSubmitter: Hadoop command-line option parsing not performed. Implem
ent the Tool interface and execute your application with ToolRunner to remedy this.
18/01/20 22:06:53 INFO input.FileInputFormat: Total input paths to process : 1
18/01/20 22:06:53 INFO mapreduce.JobSubmitter: number of splits:1
18/01/20 22:06:53 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1516456301671_0001
18/01/20 22:06:55 INFO impl.YarnClientImpl: Submitted application application_1516456301671_0001
18/01/20 22:06:55 INFO mapreduce.Job: The url to track the job: http://master:8088/proxy/application_15
16456301671_0001/
18/01/20 22:06:55 INFO mapreduce.Job: Running job: job_1516456301671_0001
18/01/20 22:07:13 INFO mapreduce.Job: Job job_1516456301671_0001 running in uber mode : false
18/01/20 22:07:13 INFO mapreduce.Job:  map 0% reduce 0%
18/01/20 22:07:32 INFO mapreduce.Job:  map 58% reduce 0%
18/01/20 22:07:34 INFO mapreduce.Job:  map 100% reduce 0%
18/01/20 22:07:44 INFO mapreduce.Job:  map 100% reduce 100%
18/01/20 22:07:45 INFO mapreduce.Job: Job job_1516456301671_0001 completed successfully
18/01/20 22:07:45 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=33135
                FILE: Number of bytes written=277013
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=13615283
                HDFS: Number of bytes written=5458
                HDFS: Number of read operations=7
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=1
                Launched reduce tasks=1
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=17136
```

```
                Total time spent by all reduce tasks (ms)=7448
                Total vcore-seconds taken by all map tasks=17136
                Total vcore-seconds taken by all reduce tasks=7448
                Total megabyte-seconds taken by all map tasks=17547264
                Total megabyte-seconds taken by all reduce tasks=7626752
        Map-Reduce Framework
                Map input records=623
                Map output records=3115
                Map output bytes=26899
                Map output materialized bytes=33135
                Input split bytes=103
                Combine input records=0
                Combine output records=0
                Reduce input groups=623
                Reduce shuffle bytes=33135
                Reduce input records=3115
                Reduce output records=623
                Spilled Records=6230
                Shuffled Maps =1
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=653
                CPU time spent (ms)=9910
                Physical memory (bytes) snapshot=323612672
                Virtual memory (bytes) snapshot=3752804352
                Total committed heap usage (bytes)=170004480
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=6807590
        File Output Format Counters
                Bytes Written=5458
hadoop@master:~$
```

函数各部分实现内容：


task
class KnnInstance
KnnInstance
样本实例构造函数
getId
获取样本索引
getLabel
获取样本标签
getFeatures
获取样本属性队列
class KNN
class KnnMap
setup
读取训练集
map
测试样本id与前k个距离最近样本的标签的映射
Distance
获得两个文本向量间的距离
class KnnReducer
reduce
根据预测队列，获得最终预测标签
Predict
传入预测队列，输出出现次数最多的标签
main
设置map和reduce对应的类

2、 代码解释

构造一个 KnnInstance 类，每个类代表一个样本，类中有样本的索引 id、样本的各个属性值、样本的标签这三种数据成员。

```
class KnnInstance
{
        public double[] attributeset;//存放样例属性
        public String lable;//存放样例标签
        public String id; //索引
```

KnnInstance 构造函数：传入 String 类型的一个样本，样本的格式为“索引/t 属性值_1/t 属性值_2/t…属性值_n/t 标签”，因此需要用 split 函数进行分割，所得字符串数组的第一个为样本索引，最后一个是样本标签，中间的是样本属性值。

```
public  KnnInstance(String line)
{
    String[] splited = line.split("\t"); //用tab键分割
    attributeset = new double[splited.length-2]; //各属性值
    id = splited[0] + ""; //设置样本索引
    //设置样本各属性
    for(int i=1;i<attributeset.length;i++)
    {
        attributeset[i-1] = Double.parseDouble(splited[i]);
    }
    //设置样本标签
    lable = splited[splited.length-1] + "";
}
```

定义获取样本各值的函数：

```
//获取样本属性
public double[] getFeatures()
{
    return attributeset;
}
//获取样本标签
public String getLabel()
{
    return lable;
}
//获取样本索引
public String getId(){
    return id;
}
```

自定义 KnnMap 类，继承了 Mapper 类。

```
public static class KnnMap extends Mapper<LongWritable, Text, Text, Text>
```

首先定义一个队列，存放训练集文本；定义 knn 的 k 值：

```
public ArrayList<KnnInstance> train = new ArrayList<KnnInstance>();   //存储训练集
public int k = 5; //k值
```

定义一个 setup 函数，用来从文件中读取训练集，并将其存放于上面定义的训练集队列中：

```
protected void setup(Mapper<LongWritable, Text, Text, Text>.Context context)
    throws IOException, InterruptedException {
        FileSystem fs = null;
        try {
            fs = FileSystem.get(new URI("hdfs://192.168.142.130:9000"), new Configuration());
        }
        catch (Exception e) {
        }
        FSDataInputStream fi = fs.open(new Path(                    //获取训练文本
                "hdfs://192.168.142.130:9000/input/traindata.txt")); //文件路径
        BufferedReader bf = new BufferedReader(new InputStreamReader(fi));
        String line = bf.readLine(); //读取每行文本
        while (line != null) {
            KnnInstance sample = new KnnInstance(line);//传入字符串line构建KnnInstance实例
            train.add(sample); //将创建的实例放入训练集中
            line = bf.readLine();
        }
    }
```

定义欧式距离计算公式：

```
//欧氏距离计算公式
private double Distance(double[] a, double[] b) {
    double sum = 0.0;
    for (int i = 0; i < a.length; i++) {
        sum += Math.pow(a[i] - b[i], 2);
    }
    return Math.sqrt(sum);
}
```

定义 map 函数：

定义两个队列，一个用来存储前 k 个距离，一个用来存储对应的标签：

```
protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
    ArrayList<Double> distance = new ArrayList<Double>(k); //存k个距离
    ArrayList<String> trainlabel = new ArrayList<String>(k); //存k个距离对应的标签
```

对两个队列进行初始化：

```
for (int i = 0; i < k; i++) {
    distance.add(Double.MAX_VALUE); //初始化k个距离为Double的最大值
    trainlabel.add(String.valueOf("-1.0")); //初始化k个标签为-1.0
}
```

将测试样本转化为 KnnInstance 实例：

```
KnnInstance test = new KnnInstance(value.toString()); //创建测试集样本实例
```

获取测试样本索引：

```
//计算测试集样本和每个训练集样本的距离
String id = test.getId(); //测试集索引
```

定义两个队列，distancelist 队列和 labellist 队列，distancelist[i]存放当前测试样本与训练集第 i 个训练样本的距离，labellist[i]存放第 i 个训练样本对应的标签：

```
Double[] distancelist = new Double[train.size()]; //距离队列，存放测试样本与验证集对应的距离
String[] labellist = new String[train.size()]; //标签队列，存放验证集对应的标签
//计算测试样本与训练集的各个距离，并存到list和label队列中
for (int i = 0; i < train.size(); i++) {

    double dis = Distance(train.get(i).getFeatures(), test.getFeatures());
    distancelist[i] = dis; //存放距离
    labellist[i] = train.get(i).getLabel() + ""; //存放标签

}
```

使用冒泡排序，对 distancelist 队列进行从小到大冒泡排序，同样需要更新 labellist：

```
//对距离进行排序
for(int i=0; i<train.size(); i++) {
    for(int j=i; j<train.size(); j++) {
        //大于则交换
        if(distancelist[i]>distancelist[j]) {
            Double temp = distancelist[i];
            String temps = labellist[i];
            distancelist[i] = distancelist[j];
            labellist[i] = labellist[j];
            distancelist[j] = temp;
            labellist[j] = temps;
        }
    }
}
```

将 distancelist 和 labellist 的前 k 个距离和 label 按顺序存放进最终的 distance 和 trainlabel 中：

```
//存前k个距离及对应的label
for(int i=0; i<k; i++) {
    distance.set(i, distancelist[i]);
    trainlabel.set(i, labellist[i]);
}
```

测试样本的索引作为 key，前 k 个标签作为 value 写入

```
for (int i = 0; i < k; i++) {
    context.write(new Text(id),new Text(trainlabel.get(i) + ""));
}
```

Reduce 函数：

编写 Predict 函数，通过传入的预测标签队列，来选出其中出现次数最多的标签。建立哈希映射，将标签与其出现的次数映射起来：

```
private String Predict(ArrayList<String> arr) {

    HashMap<String, Double> tmp = new HashMap<String, Double>(); //map映射，预测值->出现次数
```

对于预测队列中的每个标签，如果之前已经记录了，就更新其出现的次数；而如果该标签是第一次出现，就加入到哈希映射中：

```
for (int i = 0; i < arr.size(); i++) {
    //原先存在该预测值
    if (tmp.containsKey(arr.get(i))) {
        double frequence = tmp.get(arr.get(i)) + 1; //更新次数
        tmp.remove(arr.get(i)); //移除原先的映射
        tmp.put((String) arr.get(i), frequence); //更新预测+次数
    }
    //原先没有该预测值
    else
        tmp.put((String) arr.get(i), new Double(1)); //加入map中
}
```

取出哈希映射中的所有键值集合，查找每一个标签对应的次数，记录出现次数最多的标签就是最终预测的标签：

```
Set<String> s = tmp.keySet(); //键值的集合
Iterator it = s.iterator();
double lablemax = Double.MIN_VALUE; //初始化最大出现次数lablemax为最小值
String predictlable = null; //预测值
while (it.hasNext()) {
    String key = (String) it.next();
    Double lablenum = tmp.get(key);
    //找次数最大的预测值
    if (lablenum > lablemax) {
        lablemax = lablenum;
        predictlable = key;
    }
}
return predictlable;
```

reduce 函数：

对于每个测试样本，根据 map 得到的 k 个键值对，选出出现最多的标签作为测试样本最终的预测标签：

```
protected void reduce(Text k, Iterable<Text> values, Context context) //k为测试集id，values为预测队列
        throws IOException, InterruptedException {

    ArrayList<String> labellist = new ArrayList<String>();

    for (Text t : values) {
        labellist.add(t.toString());
    }
    //l为预测label队列
    String predict = Predict(labellist); //根据预测值队列，选出出现次数最多的，作为预测值
    context.write(new Text(k.toString() + "\t" + predict), NullWritable.get());
}
```

main 函数中设置 map 和 reduce 对应的类以及定义输入、输出类型：

```
public static void main(String[] args)
throws IOException,ClassNotFoundException, InterruptedException {
    FileSystem fs = FileSystem.get(new Configuration());

    Job job = new Job(new Configuration());
    job.setJarByClass(Knn.class); //通过传入的class 找到job的jar包

    FileInputFormat.setInputPaths(job, new Path(args[0]));
    job.setMapperClass(KnnMap.class); //设置map class
    job.setMapOutputKeyClass(Text.class); //设置map输出key的类型为text
    job.setMapOutputValueClass(Text.class); //设置map输出value的类型为text

    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setReducerClass(KnnReducer.class); //设置reduce class
    job.setOutputKeyClass(Text.class); //设置reduce输出key的类型为text
    job.setOutputValueClass(NullWritable.class);

    job.waitForCompletion(true);

}
```

3、 实现难点

[1]. 如何将 KNN 算法的各个阶段的工作分配到 Map 函数和 Reduce 函数中。

我们很清楚 KNN 算法的基本流程：算出测试样本与所有训练样本的距离 -> 得到前 k 个距离最近的样本对应的标签 -> 选取出现次数最多的标签作为预测标签，查阅了 MapReduce 的相关资料之后也知道 Map 函数做的主要是将文件解析成<key,value>键值对，然后将相同 key 值的记录送到 Reduce 结点，Reduce 结点再根据所定义的函数对键值对中的 value 执行计算操作，最后得出计算结果。但是要怎么把两者结合起来就是我们面临的第一个问题。

参考网上基于 MapReduce 的 KNN 多种实现，首先可以确定 key 值应该是唯一确定一个测试样本的索引，而最终的结果是要输出预测标签。对于 map 函数，输入应该是一个测试集样本和所有训练集，输出应该是测试集样本索引以及前 k 个距离所对应的标签；在 reduce 函数中，输入应该是 key 值相同的所有 map 函数的键值对输出，输出则是测试样本索引和最终预测结果。

确定了每个函数的输入输出之后，分别要执行什么任务也就清晰了。在 map 函数中，我们可以先计算测试集样本到所有训练集样本的距离，得到前 k 个距离最近的样本的标签，作为 value 输出；在 reduce 函数中，对得到的所有标签作一个统计，选出出现次数最多的作为最后的结果。

[2]. 训练集、测试集文本的读取。

参考网上样例，先将文本上传到 hdfs，然后设置路径读取文件，使用 BufferReader 读取每行文本，并存放在 ArrayList 中。

4、 关键技术点

[1]. 将 Knn 算法的各个阶段的工作分配到 Map 函数和 Reduce 函数中。

[2]. spilt 分割文本的索引、属性、标签。
[3]. 距离公式的选取：欧氏距离。
[4]. 冒泡排序得到前 k 个距离对应的标签。
[5]. 使用 Hashmap 将标签映射到出现的次数。

三、 结果

训练集样本共 623 行，测试集样本共 311 行，测试输出结果为“索引+标签”，预测均正确：

1 -1.0	11 -1.0	120 -1.0	131 -1.0	142 -1.0	153 -1.0	164 -1.0
10 -1.0	110 -1.0	121 -1.0	132 -1.0	143 -1.0	154 -1.0	165 -1.0
100 -1.0	111 -1.0	122 -1.0	133 -1.0	144 -1.0	155 -1.0	166 -1.0
101 -1.0	112 -1.0	123 -1.0	134 1.0	145 -1.0	156 -1.0	167 -1.0
102 -1.0	113 -1.0	124 -1.0	135 -1.0	146 -1.0	157 -1.0	168 -1.0
103 -1.0	114 -1.0	125 -1.0	136 -1.0	147 -1.0	158 -1.0	169 -1.0
104 -1.0	115 -1.0	126 1.0	137 -1.0	148 -1.0	159 -1.0	17 -1.0
105 -1.0	116 -1.0	127 -1.0	138 -1.0	149 -1.0	16 -1.0	170 -1.0
106 -1.0	117 -1.0	128 -1.0	139 -1.0	15 -1.0	160 -1.0	171 -1.0
107 -1.0	118 -1.0	129 -1.0	14 -1.0	150 -1.0	161 -1.0	172 -1.0
108 -1.0	119 -1.0	13 -1.0	140 -1.0	151 -1.0	162 -1.0	173 -1.0
109 -1.0	12 -1.0	130 -1.0	141 -1.0	152 -1.0	163 -1.0	174 -1.0

四、 总结

KNN 算法的思路比较简单，即给定一个待分类的样本，首先要计算该样本与所有训练集样本的距离，然后选取与该样本距离最接近的 k 个训练样本，根据这 k 个训练样本中出现次数最多的标签来确定待分类样本的标签。KNN 算法简单易理解，编程实现也不难，但是它的缺点就是时间复杂度高，对于有 n 个训练样本、m 个测试样本的数据集来说，每个测试样本都需要计算 n 次才能完成一次预测，时间复杂度是 O（mn），如果数据集很大，串行执行就会很耗时。而 Hadoop 多个节点并行执行的方法在大数据集上可以一定程度地提高运算速度，节省运算时间。