基于 Hadoop 实现的 KNN 算法

识别手写数字

1. 背景介绍

识别手写数字是机器学习中一个非常经典的问题，已有的实现算法也非常多，如神经网络、SVM、随机森林等。但是就我所能找到的资料而言，这些算法多是基于本地执行的实现，尚且没有比较完善的基于 Hadoop 的分布式实现。

在众多机器学习算法中，KNN 算法是一种非常适合分布式执行的算法，分布式执行可以有效提高 KNN 程序的执行效率，加快算法反馈速度。因此，在本次实验中，我选择用 Hadoop 实现 KNN 算法，以解决识别手写数字问题。

2. 相关工作

网络上可以找到一些 KNN 算法的 Hadoop 实现，例如[1][2]，但是这些实现都有一些缺陷。例如，网络上已有的实现，其测试文件只能包含一个测试样例，当有多个样例需要进行预测时，只能不断修改测试文件，非常麻烦。而且我在阅读代码的过程中发现，这些实现并不能算严格的“K 近邻”算法，在逻辑上是有问题的：在某些情况下，测试样例可能和带有不同标签的训练样例有相同的距离，但是网上已有的 K 近邻算法实现只能记录其中一种标签。这不符合 K 近邻算法的定义。此外，某些代码实现在处理 reduce 时，针对的是只有一个测试样例的情况，将输入的 key 设置成 NullWritable，导致无法充分利用分布式计算机的资源来并行执行 reduce 操作。

本次实验中，我的实现针对这些问题做了改进。

3. 基本原理

K 近邻算法是一个分类算法。给定一个训练数据集，对于新输入的测试样例，在训练数据集中找到与该样例“距离”最接近的 K 个实例。观察这 K 个实例的类别，将测试样例分到 K 个训练实例中频数最高的类别。

在识别手写数字问题中，每张图片的大小为$28*28$，可以展开为 784 维的向量。我们将两个样例之间的距离定义为两张图片的对应向量之间的欧几里得距离的平方。

4. 程序设计

我们在 4.1 中介绍基本思路，在 4.2-4.4 中进一步讨论具体实现。

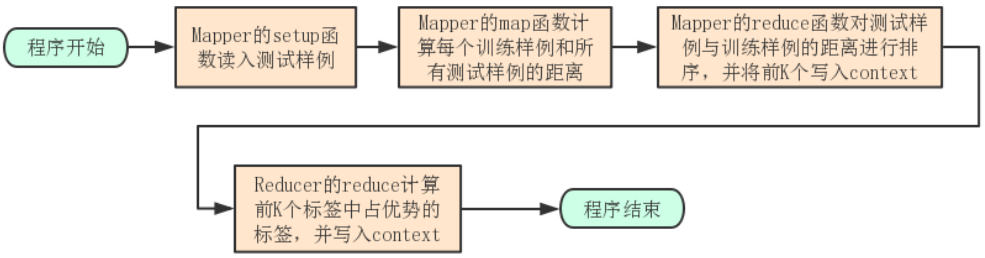
4.1 基本思路

a. 在 Mapper 的 setup 函数中读入测试数据，保存在一个动态二维数组中。
b. Mapper 的 map 函数每次读入一个训练样例，计算该训练样例与各测试样例的距离并保存。
c. 在 Mapper 的 cleanup 函数中，将每个测试样例与所有训练样例的距离进行排序，选出

排在前 K 位的样例。将测试样例的编号和被选出的训练样例的标签写入 context。

d. 在 Reducer 的 reduce 函数每次对一个测试样例进行归约，不同测试样例可以同时进行归约。

流程图如下：



4.2 DistanceLabel 类

这个类主要用来保存训练样例与测试样例之间的距离以及训练样例自身的标签。

```
public static class DistanceLabel implements Comparable<DistanceLabel>{
        public int distance;
        public String label;
        public DistanceLabel(){
            distance = Integer.MAX_VALUE;
            label = null;
        }
        public DistanceLabel(String str, int dis){
            distance = dis;
            label = str;
        }
        public DistanceLabel(int dis, String str){
            distance = dis;
            label = str;
        }
        public int compareTo(DistanceLabel other){
            return this.distance - other.distance;
        }
    }
```

4.3 Mapper 类

在读入训练集之前，我们需要先读入测试样例并保存在 testSamples 中，这部分工作在 Mapper 的 setup 函数中完成。

inputFormat 将 hdfs 上要处理的文件一行一行的读入，形成 Mapper 类的输入<key,value >对。其中，key代表字符偏移量，value代表字符串。在我们的任务中，只需关注value。Mapper 类的输出< key,value >对中，key代表测试样例的编号，value代表与该测试样例距离接近的训练样例的标签。

在 map 函数中，计算不同训练样例与测试样例的距离显然不会相互干扰，因而理论上可以并行计算。代码如下所示。

```
        @Override
        public void map(Object key, Text value, Context context) throws
IOException, InterruptedException{
            String curTrainSample = value.toString();
            StringTokenizer st = new StringTokenizer(curTrainSample,",");
            String curLabel = st.nextToken();
            int[] curTrainArr = new int [featuresNum];
            for (int i = 0; i < featuresNum; ++i){
                curTrainArr[i] = Integer.parseInt(st.nextToken());
            }
            for (int i = 0; i < testId; ++i){
                //计算该训练样例与每个测试样例的距离并保存
                int curDistance = getDistance(testSamples.get(i),
curTrainArr, featuresNum);
                distanceLabels.get(i).add(new DistanceLabel(curDistance,
curLabel));
```

在 Mapper 的 cleanup 函数中，将每个测试样例与所有训练样例的距离从小到大排序，并将前 K 个标签写入 context。

4.4 Reducer 类

Reducer 类输入的< key, value >对中，key代表测试样例的编号，value代表与其距离接近的训练样例的标签。输出的< key, value >对中，key代表测试样例的编号，value代表 KNN 算法预测的标签。具体代码如下。

可以看到，由于每个测试样例有独立的key，预测测试样例标签的操作互不干扰，因而理论上 reduce 函数可以并行预测不同测试样例的标签，充分利用计算机的资源。代码如下。

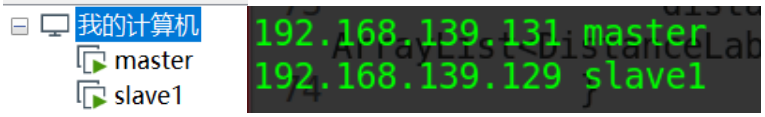
```
    //input key: 测试样例id, input value: 与测试样例距离近样例的标签
    //output key: 测试样例id, output value: 测试样例预测标签
    public static class KnnReducer extends Reducer<IntWritable, Text,
IntWritable, Text>{
        public void reduce(IntWritable key, Iterable<Text> values,
Context context) throws IOException, InterruptedException{
            HashMap<String, Integer> labelCounter = new HashMap<String,
Integer>();
            for (Text val : values){
                if (labelCounter.containsKey(val.toString())){
                    labelCounter.put(val.toString(),
labelCounter.get(val.toString())+1);
                }
                else{
                    labelCounter.put(val.toString(), 1);
                }
            }
```

```
            //选出距离在前K样例的标签中频数最高的标签
            int maxFreq = -1;
            String mostFreqLabel = null;
            for (Map.Entry<String, Integer> entry:
labelCounter.entrySet()){
                if (entry.getValue() > maxFreq){
                    mostFreqLabel = entry.getKey();
                    maxFreq = entry.getValue();
                }
            }
            context.write(key, new Text(mostFreqLabel));
        }
    }
```

5. 编译执行

5.1 配置环境

先配置网络。根据给定的参考资料，设置两台虚拟机，使得其 ip 在同一个子网内，ip 设置如下。



验证两个虚拟机可以相互 ping 通。

```
hadoop@master:/usr/local/hadoop$ ping -c 4 slave1
PING slave1 (192.168.139.129) 56(84) bytes of data.
64 bytes from slave1 (192.168.139.129): icmp_seq=1 ttl=64 time=193 ms
64 bytes from slave1 (192.168.139.129): icmp_seq=2 ttl=64 time=1032 ms
64 bytes from slave1 (192.168.139.129): icmp_seq=3 ttl=64 time=85.7 ms
64 bytes from slave1 (192.168.139.129): icmp_seq=4 ttl=64 time=110 ms

--- slave1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3036ms
rtt min/avg/max/mdev = 85.758/355.685/1032.714/392.928 ms
hadoop@master:/usr/local/hadoop$
```

```
hadoop@slave1:~$ ping -c 4 master
PING master (192.168.139.131) 56(84) bytes of data.
64 bytes from master (192.168.139.131): icmp_seq=1 ttl=64 time=5.36 ms
64 bytes from master (192.168.139.131): icmp_seq=2 ttl=64 time=0.203 ms
64 bytes from master (192.168.139.131): icmp_seq=3 ttl=64 time=0.807 ms
64 bytes from master (192.168.139.131): icmp_seq=4 ttl=64 time=1.00 ms

--- master ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.203/1.844/5.363/2.053 ms
hadoop@slave1:~$
```

然后配置 Hadoop，在 master 运行 bin/hdfs dfsadmin -report，结果如下。可以看到，

配置成功。

```
hadoop@master:/usr/local/hadoop$ bin/hdfs dfsadmin -report
Configured Capacity: 18889830400 (17.59 GB)
Present Capacity: 12627140608 (11.76 GB)
DFS Remaining: 12611956736 (11.75 GB)
DFS Used: 15183872 (14.48 MB)
DFS Used%: 0.12%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
-------------------------------------------------
Live datanodes (1):

Name: 192.168.139.129:50010 (slave1)
Hostname: slave1
Decommission Status : Normal
Configured Capacity: 18889830400 (17.59 GB)
DFS Used: 15183872 (14.48 MB)
Non DFS Used: 6262689792 (5.83 GB)
DFS Remaining: 12611956736 (11.75 GB)
DFS Used%: 0.08%
DFS Remaining%: 66.77%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Wed May 29 20:36:42 PDT 2019
```

5.2 编译程序

由于程序中用到了 commons-io.jar 包，因此在编译命令中加入了这个包。

```
hadoop@master:/usr/local/hadoop/code$ sudo javac -cp ../share/hado
op/common/hadoop-common-2.6.0.jar:../share/hadoop/mapreduce/hadoop
-mapreduce-client-core-2.6.0.jar:../share/hadoop/common/lib/common
s-cli-1.2.jar:/usr/local/Apache/commons-io-2.6/commons-io-2.6.jar
KNN.java -d ./
Note: KNN.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
hadoop@master:/usr/local/hadoop/code$ sudo jar -cvf KNN.jar KNN*.c
lass
added manifest
adding: KNN.class(in = 1642) (out= 907)(deflated 44%)
adding: KNN$DistanceLabel.class(in = 816) (out= 449)(deflated 44%)
adding: KNN$KnnMapper.class(in = 4098) (out= 1890)(deflated 53%)
adding: KNN$KnnReducer.class(in = 2372) (out= 1034)(deflated 56%)
hadoop@master:/usr/local/hadoop/code$
```

5.3 执行程序

训练数据集：train_samples.txt

测试数据集：test_samples.txt

输出预测结果：predictions

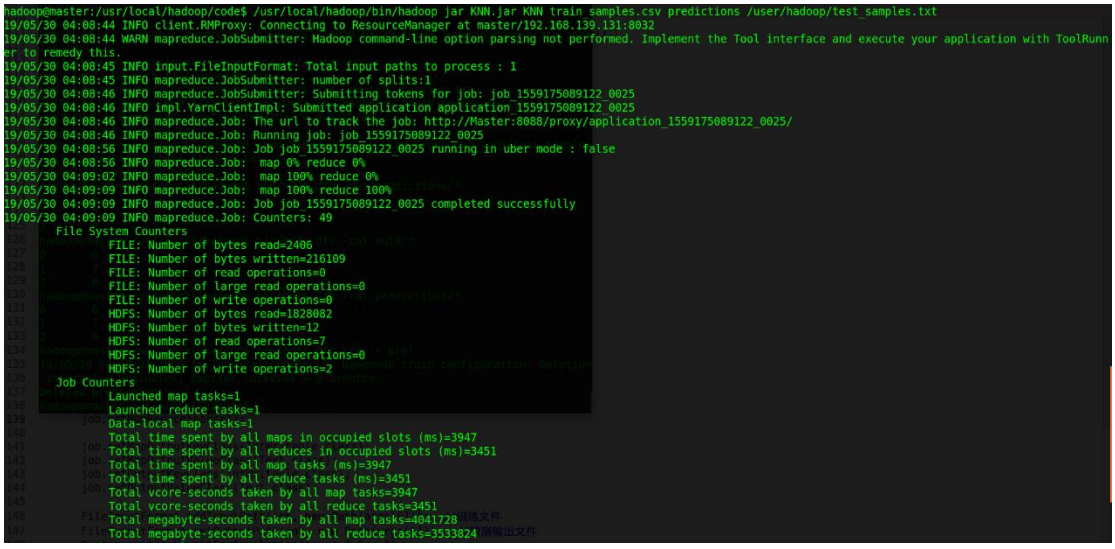
由于本次实验是在两个虚拟机上执行，资源有限，我们的训练数据集中只包含了 1000 个训练样本，预测数据集中只包含了 3 个测试样例。在文件较小的情况下，map 的数量是 1，

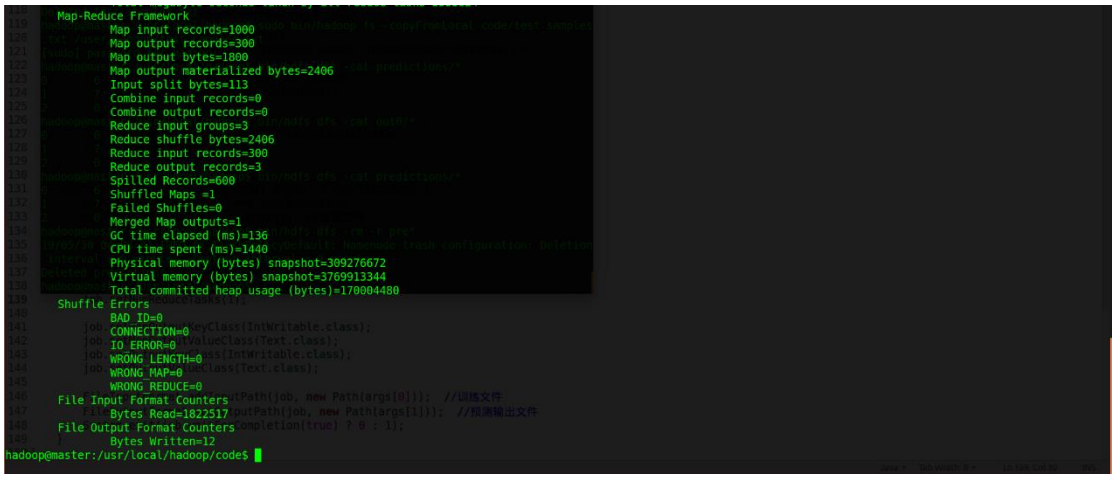
当然也可以通过 setNumMapTasks 函数手动设置，此处我们采用了默认的 map 数量。

一般来说，reduce 的数量是 map 数量的 0.95 倍或 1.75 倍[3]。当 reduce 数量是 map 数量的 0.95 倍时，所有的 reduce 任务能够在 map 任务的输出传输结束后同时开始运行。当 reduce 数量时 map 数量的 1.75 倍时，高速的节点会在完成他们第一批 reduce 任务计算之后开始计算第二批 reduce 任务，这样的情况更有利于负载均衡。同时需要注意增加 reduce 的数量虽然会增加系统的资源开销，但是可以改善负载匀衡，降低任务失败带来的负面影响。

在我们的任务中，我们需要在所有的 map 任务完成后才能执行 reduce 任务，因此采用 0.95 倍的 reduce 数量，即将 reduce 数量设置为 1。

运行结果如下：





6. 结果分析

由于笔记本电脑资源有限，我们只测试了 3 个样例，其真实标签分别为 6，9，0。KNN 算法的预测结果如下：

可以看到第二个样例被错误分类。我们猜测有以下两个原因：

a. 训练数据太少，只有 1000 个数据，无法将测试样例有效分类。
b. 数字“7”和数字“9”在形状上比较接近，因而第二个样例和标签为“7”的样例距离可能比较接近。

7. 总结

本次实验中我们用 Hadoop 实现了 KNN 算法，解决手写数字的分类问题。相较于已有的实现，我们的实现运行一次可以对多个样例进行预测，而不必每预测一个样例就要修改一次测试文件。同时我们的实现理论上可以充分利用节点资源，加快预测速度。

通过这次实验，我对 MapReduce 的特性和运行过程有了进一步的理解，同时对 hadoop 的配置过程和编程实现有了一定的了解，总体来说收获很大。

参考资料：

[1] https://github.com/matt-hicks/MapReduce-KNN
[2] https://github.com/CodeMySky/hadoop-knn
[3] https://my.oschina.net/Chanthon/blog/150500