

金融大数据-作业6

施宇 191250119

题目要求

Iris数据集是常用的分类实验数据集，由Fisher, 1936收集整理。Iris也称鸢尾花卉数据集，是一类多重变量分析的数据集。数据集包含150个数据，分为3类，每类50个数据，每个数据包含4个属性。可通过花萼长度，花萼宽度，花瓣长度，花瓣宽度4个属性预测鸢尾花卉属于（Setosa, Versicolour, Virginica）三个种类中的哪一类。在MapReduce上任选一种分类算法（KNN，朴素贝叶斯或决策树）对该数据集进行分类预测，采用留出法对建模结果评估，70%数据作为训练集，30%数据作为测试集，评估标准采用精度accuracy。可以尝试对结果进行可视化的展示（可选）。

基本思路

拆分数据集得到训练集和测试集

数据集中包含150个数据，每个类别包含50个数据，使用70%数据作为训练集，30%数据作为测试集。这里选择每个类别的数据中的前35个数据作为训练集，后15个数据作为测试集。将训练数据及标签放入train.csv文件中，测试集数据放入test_data.csv文件，测试集标签放入test_label.csv文件，去掉开头的特征名称一行，每个文件的数据示意如下：

train.csv:

```
5.1,3.5,1.4,0.2,setosa
4.9,3,1.4,0.2,setosa
4.7,3.2,1.3,0.2,setosa
4.6,3.1,1.5,0.2,setosa
5,3.6,1.4,0.2,setosa
.....
```

test_data.csv:

```
5,3.2,1.2,0.2
5.5,3.5,1.3,0.2
4.9,3.6,1.4,0.1
4.4,3,1.3,0.2
5.1,3.4,1.5,0.2
.....
```

test_label.csv:

```
setosa
setosa
setosa
setosa
setosa
.....
```

输入数据的文件目录组织如下：

```
input:
|- test
|   |- test_data.csv
|   |- test_label.csv
|- train.csv
```

特征向量间距离的计算方式

为了能够量化不同数据间的差异，他们都包含了N个维度的特征，即 $X = (x_1, x_2, \dots, x_n)$, $Y = (y_1, y_2, \dots, y_n)$ ，考虑到本题中的特征值都为数值，且不含布尔值，所以这里只考虑距离度量。为了检测不同距离度量的表现，选用三种距离度量方式测试KNN的表现。

欧几里得距离(Euclidean)

欧氏距离是最常见的距离度量，衡量的是多维空间中各个点之间的绝对距离。公式如下：

$$dist(X, Y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

因为计算是基于各维度特征的绝对数值，所以欧氏度量需要保证各维度指标在相同的刻度级别，比如对身高（cm）和体重（kg）两个单位不同的指标使用欧式距离可能使结果失效。

切比雪夫距离(Chebyshev)

切比雪夫距离起源于国际象棋中国王的走法，只关注距离最大的那个维度的距离，计算公式如下：

$$dist(X, Y) = \lim_{p \rightarrow \infty} \left(\sum_{i=1}^n |x_i - y_i|^p \right)^{1/p} = \max |x_i - y_i|$$

曼哈顿距离(Manhattan)

曼哈顿距离来源于城市区块距离，是将多个维度上的距离进行求和后的结果，计算公式如下：

$$dist(X, Y) = \sum_{i=1}^n |x_i - y_i|$$

KNN算法实现

由于数据集分为训练集和测试集，所以首先需要考虑哪一部分作为mapreduce的对象。通常情况下，训练集的数据是巨量的，测试集的数据量则较少，本题同样满足这个特点。因此选择在Mapper类的setup阶段就直接读取测试集，将训练集作为mapreduce的对象。

当Mapper类中存储了测试集数据后，可以在map阶段完成测试集和训练集的距离计算，在reduce阶段完成训练数据距每个测试数据的距离的排序和筛选。但是由于训练数据可能是非常大量的，在reduce中没法直接完成排序，因此可以在shuffle的过程中自动的完成排序。另外，考虑到KNN算法只关注距离最近的K个数据点，因此可以通过Combiner来充分减少map输向reduce的网络通信量，在Combiner中只发送每个测试数据对应的训练数据点中的前K个即可，这里需要满足这些训练数据是按照距测试数据的距离从大到小排序过的。

为了满足排序和将同一个测试数据的数据集中在一起的目标，这里创建一个新的数据类型KNNVec。KNNVec存储了一个训练数据对应的一个测试数据及两者间的距离：

```
Integer id; //测试数据id
Double dis; //测试数据与训练数据的距离
```

KNNVec之间进行大小比较时，先按照测试数据id进行比较，如果id相同，再按照距离进行比较：

```
public int compareTo(Paris o) {
    if(o.id.equals(id))
        return dis.compareTo(o.dis);
    return id.compareTo(o.id);
}
```

为了将同一个测试数据id的KNNVec发送到同一个reduce上，需要让KNNVec的hashCode与其存储的测试数据id的hashCode相同：

```
public int hashCode(){
    return id.hashCode();
}
```

Map阶段

在setup函数中，读取测试数据的特征值并存储。在map阶段中输入为训练数据的特征值和标签，通过一个循环输出每个训练数据与所有测试数据的距离等信息：

```
for(String i: testData){
    String[] f = i.split(",");
    Double dis = getDistance(features, f, disFun);
    try{
        // map输出的Key为KNN(id, dis), value为训练数据的标签
        context.write(new KNNVec(id, dis), new Text(label));
    }catch (Exception e){
        e.printStackTrace();
    }
    id++;
}
```

其中getDistance函数通过不同的方式计算两个数据间的距离：

```
private Double getDistance(String[] f1, String[] f2, String disFun){
    Double dis = 0.0;
    switch (disFun){
        case "Euclidean": //欧几里得距离
            for(int i=0;i<f2.length;i++){
                dis += Math.pow(Double.parseDouble(f1[i]) -
                    Double.parseDouble(f2[i]), 2);
            }
            dis = Math.sqrt(dis);
            break;
        case "Chebyshev": //切比雪夫距离
            for(int i=0;i<f2.length;i++){
                dis = Math.max(Double.parseDouble(f1[i]) -
                    Double.parseDouble(f2[i]), dis);
            }
            break;
        case "Manhattan": //曼哈顿距离
            for(int i=0;i<f2.length;i++){
                dis = Math.abs(Double.parseDouble(f1[i]) -
                    Double.parseDouble(f2[i]));
            }
    }
}
```

```

        break;
    }
    return dis;
}

```

Combine阶段

在Combiner中，输入的Key为KNNVec，Value为训练数据对应的标签。因为这里KNNVec已经按照之前的规则排好序，所以直接输出每个测试数据的前K个训练数据即可：

```

protected void reduce(KNNVec key, Iterable<Text> value, Reducer<KNNVec, Text,
KNNVec, Text>.Context context)
    throws IOException, InterruptedException{
    if(current_id == null){
        current_id = key.id;
    }
    if(current_id != key.id){
        j = k;
        current_id = key.id;
    }
    if(j != 0){
        for(Text i: value){
            j--;
            context.write(key, i);
            if(j == 0)
                break;
        }
    }
}

```

Reduce阶段

在reduce阶段中，由于输入的KNNVec也是排好序的，所以与Combiner中的思路类似，记录每个测试数据对应的K个训练数据中的类别，当该测试数据对应的K个训练数据都记录后，将具有最多训练数据的类别作为该测试数据的类别。

在cleanup函数中，输出距离计算方式、类别数和准确率：

```

protected void cleanup(Reducer<KNNVec, Text, Text, NullWritable>.Context
context)
    throws IOException, InterruptedException{
    context.write(new Text("距离计算: "+disFun), NullWritable.get());
    context.write(new Text("类别数: "+k), NullWritable.get());
    context.write(new Text("准确率accuracy:
"+right_count/(right_count+false_count)), NullWritable.get());
}

```

模块封装

为了提高程序的可拓展性，对KNN算法的相关模块进行封装。将Mapper、Combiner和Reducer类封装为KNN类的子类，在KNN类中添加run函数作为KNN算法的程序入口：

```

public static void run(String input, String output, String test_data_path,
String disFun){

```

```

try{
    Configuration conf = new Configuration();
    conf.set("disFun", disFun);
    Job job = Job.getInstance(conf, "Iris Classification - KNN");
    job.setJarByClass(KNN.class);
    job.setMapperClass(KNNMap.class);
    job.setCombinerClass(KNNCombiner.class);
    job.setReducerClass(KNNReduce.class);
    job.setMapOutputKeyClass(KNNVec.class);

    test_data = test_data_path + "test/test_data.csv";
    test_label = test_data_path + "test/test_label.csv";

    //      job.addCacheFile(new Path(test_data_path +
"test/test_data.csv").toUri());
    //      job.addCacheFile(new Path(test_data_path +
"test/test_label.csv").toUri());

    FileInputFormat.addInputPath(job, new Path(input));
    FileOutputFormat.setOutputPath(job, new Path(output));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}catch (Exception e){
    e.printStackTrace();
}

}

```

为了减少硬编码，将训练数据路径、输出数据路径，测试数据路径，计算距离的方式作为run函数的参数，分别为input, output, test_data_path, disFun。

最后，对于该任务，新建一个类IrisClassification，该类为完成鸢尾花分类任务的启动类，在该类中，设定好输入、输出、计算距离的方式等参数设定后，调用不同方法的模块来完成任务，因为这里使用的是KNN算法，于是调用KNN类的run函数启动任务：

```

public class IrisClassification {
    public static void main(String[] args){
        String input = args[0];
        String output = args[1];
        String test_data_path = args[2];
        String disFun = "Euclidean";
        //      Euclidean, Chebyshev, Manhattan
        try{
            KNN.run(input, output, test_data_path, disFun);
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}

```

算法伪代码

Mapper

```
class KNNMapper{
  setup(...){
    //读取全局测试样本数据文件，转入本地内容的数据表TR中
  }
  map(key, value){
    f1 = value.split(","); // 训练数据特征
    label = f1[f1.length-1]; //训练数据标签
    for i=0 to TR.length{
      f2 = TR[i].split(","); //测试数据特征
      dis = getDistance(f1, f2, disFun); // 根据disFun指定的距离计算方式计算f1
和f2之间的距离
      id, dis -> v // 根据id和dis得到KNNVec对象v
      emit(v, label);
      id++;
    }
    id=1;
  }
}
```

Combiner

```
class KNNCombiner{
  0 -> current_id;
  reduce(key, value){
    如果key的id发生变化，令current_id=key.id, j=k
    if j != 0:
      for i=0 to value.length{
        j--;
        emit(key, value[i]);
        if j == 0:
          break;
      }
  }
}
```

Reducer

```
class KNNReducer{
  HashMap -> results;
  0 -> right_count;
  0 -> false_count;
  setup(...){
    //读取全局测试样本标签文件，转入本地内容的数据表TR中
  }
  reduce(key, value){
    如果key的id发生变化，令current_id=key.id, j=k
    if j != 0:
      for i=0 to value.length{
        1 -> count;
        if results contain value[i] as Key:
          count += results[value[i]];
        put (i, count) in results;
        j--;
      }
  }
}
```


```

        if j == 0:
            get max key from results as newKey;
            emit(newKey, null);
            clear results;
            if labels[false_count + right_count] == newKey:
                right_count += 1;
            else false_count += 1;
        }
    }
    cleanup(..){
        emit(距离计算方式, null);
        emit(类别数, null);
        emit(准确率accuracy, null);
    }
}

```

运行结果

集群运行截图：



All Applications

Cluster

- About
- Nodes
- Node Labels
- Applications
- NEW
- NEW_SAVING
- SUBMITTED
- ACCEPTED
- RUNNING
- FINISHED
- FAILED
- KILLED
- Scheduler

Tools

Cluster Metrics

| Apps Submitted | Apps Pending | Apps Running | Apps Completed | Containers Running | Memory Used | Memory Total | Memory Reserved | VCores Used | VCores Total | VCores Reserved | Ac |
|----------------|--------------|--------------|----------------|--------------------|-------------|--------------|-----------------|-------------|--------------|-----------------|----|
| 1 | 0 | 0 | 1 | 0 | 0 B | 16 GB | 0 B | 0 | 16 | 0 | 2 |

Scheduler Metrics

| Scheduler Type | | Scheduling Resource Type | | Minimum Allocation | |
|--------------------|--|--------------------------|--|-------------------------|--|
| Capacity Scheduler | | [MEMORY] | | <memory:1024, vCores:1> | |

Show 20 entries

| ID | User | Name | Application Type | Queue | StartTime | FinishTime | State |
|--------------------------------|------|---------------------------|------------------|---------|--------------------------------|--------------------------------|----------|
| application_1636477025899_0001 | root | Iris Classification - KNN | MAPREDUCE | default | Wed Nov 10 00:57:50 +0800 2021 | Wed Nov 10 00:58:21 +0800 2021 | FINISHED |

Showing 1 to 1 of 1 entries

输出结果在output文件夹下的part-r-000000文件中，在末尾处给出了测试的结果信息：

欧几里得距离：

```

42      virginica
43      virginica
44      virginica
45      virginica
46      距离计算: Euclidean
47      类别数: 3
48      准确率accuracy: 1.0
49

```

可以看到使用欧几里得距离可以达到100%的准确率。另外两种距离计算方式的运行结果如下：

切比雪夫距离：

```

40     virginica
41     versicolor
42     versicolor
43     setosa
44     versicolor
45     versicolor
46     距离计算: Chebyshev
47     类别数: 3
48     准确率accuracy: 0.6888888888888889
49

```

曼哈顿距离:

```

40     virginica
41     virginica
42     virginica
43     virginica
44     virginica
45     virginica
46     距离计算: Manhattan
47     类别数: 3
48     准确率accuracy: 1.0
49

```

可以看到，欧几里得距离和曼哈顿距离都取得了较好的结果，而切比雪夫距离的表现不佳。

考虑到本题中数据具有4个特征，不能直接进行可视化，于是这里使用一组图片绘制两两特征作为x, y坐标的情况下，每个数据点的位置。使用Python的Matplotlib库完成这一功能，代码如下：

```

import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import numpy as np
import pandas as pd

feature_names = ['Sepal.Length', 'Sepal.Width', 'Petal.Length', 'Petal.Width']

feature_datas = pd.read_csv('../code/test_hadoop/input/test/test_data.csv')
labels = pd.read_csv('../code/test_hadoop/input/test/test_label.csv')
#
data = []
label = []

for i, item in feature_datas.iterrows():
    data.append([item[0], item[1], item[2], item[3]])

for i, item in labels.iterrows():
    if item[0] == 'setosa':
        label.append(0)
    elif item[0] == 'versicolor':
        label.append(1)
    else:
        label.append(2)

```



```

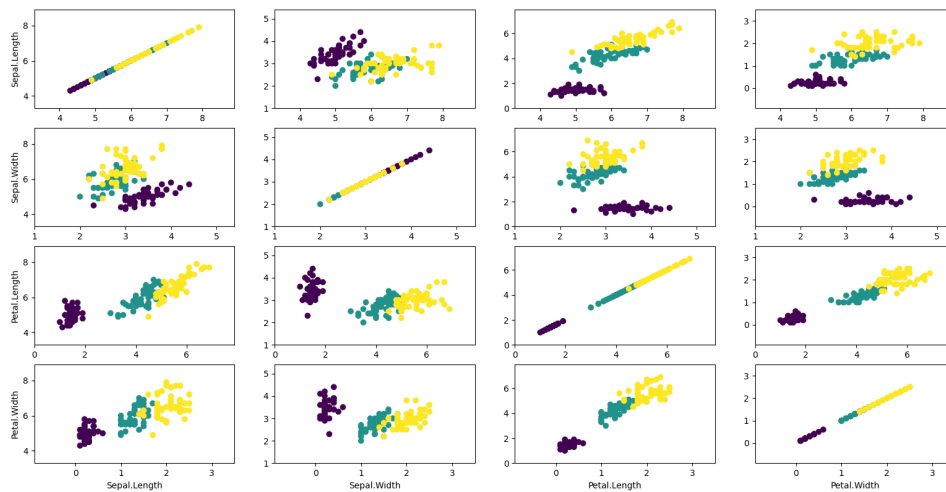
# df = pd.read_csv('../code/test_hadoop/iris.csv')
#
# for i, item in df.iterrows():
#     data.append([item[1], item[2], item[3], item[4]])
#     if item[5] == 'setosa':
#         label.append(0)
#     elif item[5] == 'versicolor':
#         label.append(1)
#     else:
#         label.append(2)

h = 0.1
for i in range(4):
    for j in range(4):
        plt.subplot(4, 4, 1 + 4*i + j)
        if j == 0:
            plt.ylabel(feature_names[i])
        if i == 3:
            plt.xlabel(feature_names[j])
        x = [data[k][i] for k in range(len(data))]
        y = [data[k][j] for k in range(len(data))]
        plt.scatter(x, y, c=label)
        plt.xlim(min(x)-1, max(x)+1)
        plt.ylim(min(y)-1, max(y)+1)

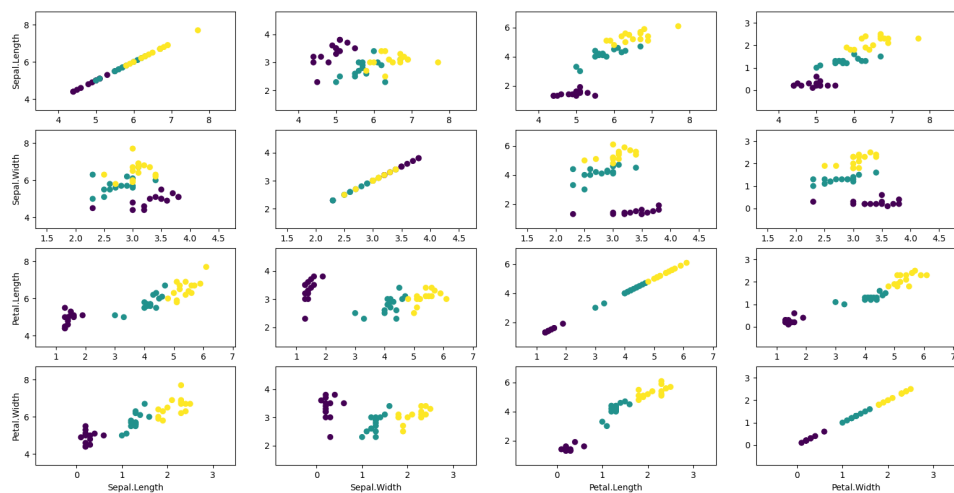
plt.show()

```

对于训练数据的可视化如图：



测试数据的可视化如图：



可以看到只通过Petal.Width这一个特征就能将测试数据中的三个类别区分开来。

性能与可拓展性分析

通过将KNN算法放入KNN类作为一个模块被入口程序调用，能够提高程序的可扩展性。另外，本程序是将训练数据作为mapreduce的对象，而测试数据则是在map和reduce阶段通过setup函数直接读取，因此不适用于存在大量测试数据的情况。考虑到KNN算法设计过程中的对称性，可以对称的设计出针对大量测试数据情况下的程序，这样能覆盖更多的算法适用情景。

在性能方面，可以通过事先对已知样本点进行剪辑，去除对分类作用不大的样本，具体方法有KD树等。