链滴

Flink 流数据 api 实战之实现机器学习密度峰值聚类算法

作者：Sakura6868

原文链接：https://ld246.com/article/1628939359434

来源网站：链滴

案例背景

● 此案例的数据源为通过GPS定位产生的经纬度信息返回到服务器，然后通过调用特定的定位接口来成一片特定区域的平面图的x和y坐标。同一个人收集到的坐标集加上特定的id作为标签。

● 此案例的数据源的类型为实时流式数据，其中最大的特点就是有头无尾，只要开启收集程序，就会集到源源不断的流式数据

● 此案例的算法思想为：每隔极短的时间收集一次每个人的位置信息数据传入到flink,在flink中按照id行分类区分每个人，对每个人的数据在固定时间类（1天or1周）进行一次密度峰值聚类算法，得到这人这段时间类活动范围的多个聚类中心（涉及到自适应的问题，下文只返回最大的那一个）。从而可析这个人的行动轨迹习惯。

具体实现

定义一个简单POJP类作为数据源的数据类型

通过案例背景可知道

数据源包含三个参数

- x: 位置x坐标的值
- y: 位置y坐标的值
- id:每个人的编号

具体代码如下

```
public class ClusterReading {
    //位置信息 x 和 y
    public double x,y;
    // 编号 id
    public int id;
    // 默认无参构造函数   (必须有这个，不然进行keyby分组的时候会报错)
    public ClusterReading()  {    }
    // 带参构造函数
    public ClusterReading(double x, double y,int id)
    {
        this.id = id;
        this.x = x;
        this.y = y;
    }

    // 定义所有获得方法
    public int getId()
    {
        return id;
    }

    public double getX() {
        return x;
    }
    public double getY(){
        return y;
    }

    // 定义所有设置方法
    public void setId(int id) {
        this.id = id;
    }

    public void setX(double x) {
        this.x = x;
    }

    public void setY(double y) {
        this.y = y;
    }
```

```
    // 重写 tostring()
    @Override
    public String toString()
    {
        return "ClusterReading{" + "id='" + id + '\'' + ", x=" + x + ", y=" + y + '}';
    }
}
```

模拟数据源

为什么要弄这个呢，因为只是实验，不可能和真是环境一样，只能通过自定义数据源来模拟整个算法程

为了方便：数据源只有三个id标签

具体代码如下

```
import java.util.Random;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.SourceFunction;

public class CustomSource {
    public static void main(String [] args) throws Exception {
        // 获取执行环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnviron
ent();
        // 获取Source数据源
        DataStreamSource<ClusterReading> inputDataStream = env.addSource(new DPCCusto
Source());
        // 打印输出
        inputDataStream.print();
        //执行
        env.execute();


    }

    public static class DPCCustomSource implements SourceFunction<ClusterReading> {
        // 定义无线循环，不断产生数据，除非被cancel
        boolean running = true;
        @Override
        public void run(SourceContext<ClusterReading> sourceContext) throws Exception {
            Random random = new Random();
            while (running)
            {
                int a = random.nextInt(3)+1;   //
                for(int i = 0;i<5;i++){
                    if(a == 1)
                    {
                        double x = Math.round((10.00 + random.nextDouble() * 20.00)*100)/100.0; //
围[10,30]
                        double y = Math.round((10.00 + random.nextDouble() * 20.00)*100)/100.0;//
围[10,30]
```

```
                    sourceContext.collect(new ClusterReading(x,y,a));
                    Thread.sleep(100L);
                }
                if(a == 2)
                {
                    double x = Math.round((3.00 + random.nextDouble() * 25.00)*100)/100.0;  //
围[3,28]
                    double y = Math.round((3.00 + random.nextDouble() * 25.00)*100)/100.0;  //
围[3,28]
                    sourceContext.collect(new ClusterReading(x,y,a));
                    Thread.sleep(100L);
                }
                if(a == 3)
                {
                    double x = Math.round((10.00 + random.nextDouble() * 20.00)*100)/100.0;  //
范围[10,30]
                    double y = Math.round((3.00 + random.nextDouble() * 25.00)*100)/100.0;  //
围[3,28]
                    sourceContext.collect(new ClusterReading(x,y,a));
                    Thread.sleep(100L);
                }
            }
                Thread.sleep(1000L);


            }

        }

        @Override
        public void cancel() {
            running = false;
        }
    }
}
```

案例具体实现

整体框架（main里面）

通过前面的学习，我们知道flink流处理流程包括 source->transform->sink

我们就通过这个流程来分析

- source：数据源为自己模拟的数据源
- transform：主要包括以下操作
  - keyby来通过id分组
  - 开窗通过时间窗口将数据收集
  - 通过全窗口函数来对数据一起处理
- sink:直接print()打印出最大的聚类中心点坐标

具体代码如下

```
public class DPC_Realize {
    public static void main(String[] args) throws Exception {
        // 获取执行环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnviron
ent();
        // 获取Source数据源
        DataStreamSource<ClusterReading> inputDataStream = env.addSource(new CustomSou
ce.DPCCustomSource());
        DataStreamSink<String> resultDataStream = inputDataStream.keyBy("id")
                .timeWindow(Time.seconds(10L))
                .process(new CustomProcessWindowFunction())
                .print();
        env.execute();
    }
```

密度峰值聚类算法的实现

由于Java中没有python中的numpy库对大量数据的批分析很困难

这里我们运用了GitHub上的开源项目Deep Java Library中的NDArray这个api，其中有很多类似numy的操作，但是相对于numpy还是比较僵硬难以上手。

从maven引入依赖

```
<dependency>
            <groupId>ai.djl</groupId>
            <artifactId>api</artifactId>
            <version>0.12.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/ai.djl.pytorch/pytorch-engine -->
        <dependency>
            <groupId>ai.djl.pytorch</groupId>
            <artifactId>pytorch-engine</artifactId>
            <version>0.12.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/ai.djl.pytorch/pytorch-native-auto -->
        <dependency>
            <groupId>ai.djl.pytorch</groupId>
            <artifactId>pytorch-native-auto</artifactId>
            <version>1.8.1</version>
            <scope>runtime</scope>
        </dependency>
```

算法整体框架

```
private long DPC(NDArray nd) {
        NDManager manager = NDManager.newBaseManager();
        // 计算距离矩阵
        NDArray dists = getDistanceMatrix(nd);
        // 计算dc
        NDArray dc = select_dc(dists);
```

```
        //用高斯方程计算局部密度
        String method = "Gaussion";
        // 计算局部密度
        NDArray rho = get_density(dists, dc, method);
        //计算密度距离
        NDArray deltas = get_deltas(dists, rho);
        //获取聚类中心点
        NDArray centers =find_centers_K(rho, deltas);
        //返回最大一个 （目前这样）
        long centersmax = centers.get(new NDIndex("0")).toLongArray()[0];
//          double deltasmax =deltas.get(new NDIndex("{}",centersmax)).toDoubleArray()[0];
//          double rhomax =rho.get(new NDIndex("{}",centersmax)).toDoubleArray()[0];
        return centersmax;
    }
```

计算数据点两两之间的距离

```
private NDArray getDistanceMatrix(NDArray nd) {
        //计算数据点两两之间的距离
        NDManager manager = NDManager.newBaseManager();
        // 获取 nd的维度数（n,d）
        Shape e = nd.getShape();
        //dists初始化为维度为（n，n）
        NDArray dists = manager.zeros(new Shape(e.size()/e.dimension(),e.size()/e.dimension()
);
        //求出每个点到其它点的距离
        for(int i = 0;i < dists.getShape().dimension();i++)
        {
            for(int j = 0;j < dists.getShape().dimension();j++)
            {
                NDArray vi = nd.get(new NDIndex("{}",i));
                NDArray vj = nd.get(new NDIndex("{}",j));
                dists.set(new NDIndex("{},{}",i,j), array -> {
                    array = ((vi.sub(vj)).dot(vi.sub(vj))).sqrt();
                    return array;
                });
            }
        }
        return dists;
    }
```

寻找密度计算的阈值

```
// 找到密度计算的阈值dc
    // 要求平均每个点周围距离小于dc的点的数目占总点数的1%-2%
    private NDArray select_dc(NDArray dists) {
        //获取 dists的维度数（n,d）
        Shape e = dists.getShape();
        //求出 n 值
        long N = e.get(0);
        //把 dists的形状改为一维 列数N * N
        NDArray tt = dists.reshape(N*N);
        //定义筛选百分比
```

```
        double percent = 2.0;
        // 位置
        int position = (int)(N * (N - 1) * percent / 100);
        //返回dc值
        return (tt.sort()).get(new NDIndex ("{}",position + N));
    }
```

计算局部密度

```
private NDArray get_density(NDArray dists,NDArray dc,String method  )
    {
        //获取 dists的维度数（n,d）
        Shape e = dists.getShape();
        //求出 n 值
        long N = e.get(0);
        //初始化rho数组
        NDManager manager = NDManager.newBaseManager();
        NDArray rho = manager.zeros(new Shape(N));
        for (int i = 0;i<N;i++)
        {
            //如果没有指定用什么方法，默认方法
            if (method == null)
            {

                //筛选出 （dists[i, :] < dc）条件下的行
                NDArray g = dists.get(new NDIndex("{}", i));
                NDArray s = g.get(g.lte(dc)).get(new NDIndex("0"));
                // rho[i]为s的维度-1
                Shape c = s.getShape();
                long a = c.get(0)-1;
                NDArray r =manager.create(a);
                rho.set(new NDIndex("{}", i),aa->r);
            }
            else   //使用高斯方程计算
            {
                // 没想让你们看懂
                NDArray t = ((dists.get(new NDIndex("{}", i)).div(dc)).pow(2).neg().exp()).sum().sub
1);
                rho.set(new NDIndex("{}", i),aa->t);
            }

        }
        return rho;

    }
```

计算密度距离

```
private NDArray get_deltas(NDArray dists, NDArray rho) {
        //获取 dists的维度数（n,d）
        Shape e = dists.getShape();
        //求出 n 值
        long N = e.get(0);
```

```
        //初始化deltas数组
        NDManager manager = NDManager.newBaseManager();
        NDArray deltas = manager.zeros(new Shape(N));
        NDArray index_rho = rho.argSort().flip(0);
        for (int i = 0;i<N;i++)
        {
            //写出值
            long index =index_rho.get(new NDIndex("{}",i)).toLongArray()[0];
            // 对于密度最大的点
            if(i == 0)
                continue;
            //对于其它的点
            // 找到密度比其它点大的序号
            NDArray index_higher_rho = index_rho.get(new NDIndex(":{}", i));
            //获取这些点距离当前点的距离,并找最小值
            //下面这段对应python语句为：deltas[index] = np.min(dists[index, index_higher_rho])
            //看了懂算我输
            NDArray z = dists.get(new NDIndex("{}",index));
            long Z = index_higher_rho.size();
            for (int c = 0;c<Z;c++)
            {
                NDArray C = manager.zeros(new Shape(Z));
                int finalC = c;
                C.set(new NDIndex("{}",c), aa->z.get(new NDIndex("{}",index_higher_rho.toLongAr
ay()[finalC] )));
                deltas.set(new NDIndex("{}", index),aa->C.min());
            }
        }
        //导入最大值
        long max = index_rho.get(new NDIndex("{}",0)).toLongArray()[0];
        deltas.set(new NDIndex("{}", max),aa->deltas.max());
        return deltas;
    }
```

获取聚类中心点

```
private NDArray find_centers_K(NDArray rho, NDArray deltas) {
        //每个点都相乘
        NDArray rho_delta = rho.mul(deltas);
        //从大到小排序返回下标NDArray数组
        NDArray centers = rho_delta.argSort().flip(0);
        // 返回最大的三个下标
        return centers.get(new NDIndex(":3"));

    }
```

全窗口函数的具体实现

具体思想：

先遍历全部数据获取数据数量，创建二维数组

把数据导入到二维数组

通过二维数组创建NDArray数组

执行DPC算法返回聚类中心点的下标

通过下标返回结果

具体代码如下

```
public static  class CustomProcessWindowFunction extends ProcessWindowFunction <Cluste
Reading,String,Tuple, TimeWindow>
    {


        @Override
        public void process(Tuple tuple, Context context, Iterable<ClusterReading> iterable, Coll
ctor<String> collector) throws Exception {
            // 统计数据的数量
            int count = 0;
            //增强for循环遍历全窗口所有数据
            for (ClusterReading cluster : iterable) {
                count++;
            }
            //初始化一个二维数组来记录这个窗口里面的所有数据
            double data [][] = new double[count][];
            //增强for循环遍历全窗口所有数据输入到二维数组
            int i =0; // 计数
            for (ClusterReading cluster : iterable){
                data[i] = new double[] {cluster.getX(), cluster.getY()};
                i++;
            }
            //调用 djl中的 NDArray数组 将data导入进行算法分析
            NDManager manager = NDManager.newBaseManager() ;
            NDArray nd = manager.create(data); // 创建NDArray数组

            long centersmax = DPC(nd);     //进行dp算法找出聚类中心点的下标
            String resultStr = "id:"+tuple.getField(0) +"聚类中心为：  "+"x="+data[Math.toIntExact(
entersmax)][0]+"y="+data[Math.toIntExact(centersmax)][1];
            collector.collect(resultStr);
        }
```