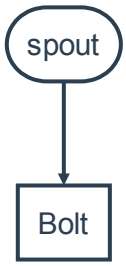
Open-Streamer 开发文档

项目介绍

Open-Stream 是一个基于Storm的在线学习算法库，包含几大算法类型，如分类算法，聚类算法，回归算法，基数计数算法等，本算法库中的所有算法均不是我的原创算法，这些算法已经有多个可用的实现版本，但是，所有的实现都是基于单机的，在大数据时代，将算法仅仅部署在单机已经无法满足实际需求了，本算法尝试将已经被广泛使用的算法应用在大数据平台上，如今，已经有多个分布式的大数据处理平台，如**Hadoop**，**Storm**，**SummingBird**等，hadoop致力于处理批量数据，依靠强大的分布式文件系统**HDFS**来处理大批量数据，Storm则用来处理流式数据，与hadoop不同，Storm运行的任务称为Topology，而不是Job，一个Topology就是一个Storm任务，Topology由一些组件合成，最后的Topology需要是一个DAG（有向无环图），有向指的是数据的流向，无环是说一个数据实例不要再次回到已经处理过它的组件里面，这是没有意义的。组件分为两类，**Spout**负责生成数据，这些数据可以随机生成，也可以从**Kafka**订阅，只要开发出来的Java类继承或者实现了Spout的基类，就可以做为Spout，**Bolt**组件则用来处理数据，Bolt处理完从上游（Spout或者Bolt）来的数据之后，可以选择将自己作为一个新的Spout，向下游Bolt发射数据，下面是一个最简单的Topology：



可以使用Spout和Bolt组成任意复杂的Topology，Storm平台将会将各个组件部署在合适的CPU上运行，本算法库基于Storm平台，该算法库将可以快速的将经典的算法运行在Storm平台上。核心代码与平台无关，组装时建议使用**Trident** 来描述，Trident是Storm的一部分，是一种可以更加方便的去操作数据流的语言，本项目中有丰富的样例代码供您参考，几乎每一个算法都有与之对应的例子，您可以在项目中发现它们。
接下来的部分，我将对该项目的开发过程中的思考与参考的资料，总结成一篇开发文档，可能不能将所有的算法都顾及到，但是我所实现的算法，在**github**或者其他地方可以实现更加优美的实现。

算法概览

- **分类算法**
  - **Committee Classifier**
  - **Passive Aggressive Classifier**
  - **Perceptron Classifier**
  - **Winnow Classifier**
  - **Balanced Winnow Classifier**
  - **Modify Balanced Winnow Classifier**

- **聚类算法**
  - **Birch**
  - **Canopy**
  - **K-means**

- **求平均算法**
  - **Moving Average**
  - **EWMA average**

- **基数计数算法**
  - **LogLog Cardinality**
  - **HyperLogLog cardinality**
  - **Adaptive Counting Cardinality**
  - **Linear Counting**

- **估计算法**
  - **Count Sketch**
  - **Lossy Counting**
  - **Stick Sampling Counting**
  - **Space Saving**
  - **Top-k**

- **回归算法**

- **Ftrl regression**
- **Perceptron Regression**
- **Passive Aggression Regression**

分类算法（Classification Algorithms）

package: com.hujian.trident.ml.classifier

- **Committee Classifier**
  该算法在[论文][1]中被提出，改算法具有多类分类的能力，而且适合部署在流处理平台之上，在论文中，作者提出了该算法，并且证明了该算法在多分类问题上的优良表现特性，并给出了适合工业实现的算法伪代码，你可以在论文的**3.1**部分找到如何分类（Predict）的算法，在**3.3**部分可以找到如何更新分类器模型的算法，作者描述比较清晰，本算法实现基于作者在论文**3.4**节给出的伪代码，根据**3.1-3.4**节的描述，我们就可以实现Committee分类算法，接下来的部分是对该算法的数学描述以及一些证明。
  需要注意的是，该算法只有在分类错误的时候才会更新分类器模型，在分类正确的时候，分类器模型将会保持下去，这可以看成是一种贪心算法，既然对当前特征实例可以得到正确的分类结果，为什么还要更新模型呢？如果更新了模型之后无法正确分类该实例，那么算法的表现不就受影响了？当然有些算法不仅在算法分类错误的时候更新模型，在分类正确的时候也会去更新模型，这些算法的特点就是维护多维模型向量，一些向量是正向量，会对分类策略长生正影响，有些向量是负向量，会对预测长生负影响，这样当算法表现正确的时候，给予一个正反馈，当算法表现错误的时候，给予一个负反馈，这样算法就具有更高的合理性，分类结果也将更可信。
  该算法没有采用多维模型向量，而是采用一个单一权重向量来做出分类决策，你可以在论文中看到算法的具体做法，再次不再赘述。
  下面是该算法用于Storm平台的样例代码，因为这是首次向您展示如何将普通的算法部署在大数据平台之上，所以，我将会为您介绍整个流程与注意事项，接下来的算法将会略过这些描述。
  （1）、本项目的所有特征向都将被包装成一个个的实例（Instance），然后再讲该实例发射到下游去，接收处理，你可以在【com.hujian.trident.ml.core】发现所有用于将特征向量转化为Instance的对象，其实只是对从源Spout流下来的数据做了一层包装，将其包装成一个全局通用的对象，然后我们就可以在下游使用类型转化来得到一个可用的特征实例，在最简单的情况下，一个实例将包含特征向量本身，需要注意的是，该项目的所有算法均不接受字符类型的特诊向量，对于分类算法，只接受浮点型特征向量，所以，在您使用该项目中的算法之前，您需要将您的特征数据转换为一个double类型的特征向量。在某些时候，一个特征实例将会被给予一个特征向量的数据类型，不同的数据类型到达下游Bolt时会有不同的处理方式，对于分类算法来说，可以设置一个特征实例为Train或者Test，分别代表这个向量是用来训练模型的或者是来查询分类的，参考源代码将对您理解整个实例的包装过程更加了解。
  （2）、在包装完一个特征向量之后，我们需要Trident的APIS来执行我们的算法，您应该先去学习一下Trident的基本内容，然后再实现您的Topology，基本的Trident内容包括如何产生一个数据流，如何过滤数据，如何对数据执行各种运算操作，如何更新运行时状态等等，当然，您也应该了解一些非常典型的数据流操作原语，比如如何对每个数据都执行某个操作，如何将两个数据流汇聚在一起，如何将结果存储起来等等，这些内容都可以在Storm的Trident子项目下发现。
  （3）、一个特别简单的分类Topology在第三步就可以执行分类或者训练操作，而执行分类或者执行训练模型操作是根据从上游流下来的特征实例上的数据类型标签来决定的，当然，您也可以在这一步介入，改变特征实例的数据类型，做您想做的操作。我们需要一个Trident的状态更新函数来训练模型，或者做分类，然后将新的模型（状态）保存起来，在下一次特征实例到达的时候再把模型取出来，做相应的操作。

  接下来为您展示如何用本项目的算法组装出一个Topology，您会发现原来将普通的算法应用于大数据平台是如此的简单，当然，在此应该感谢Trident Apis，否则，您还需要新建一个类来实现一个Spout，然后新建若干类来实现您的Bolt们，然后还要考虑如何组装才能最大化并行度，使用Trident Api，这些问题我们都不需要考虑太多，Trident会用最优的方案将Stream转化为Topology。

```
TridentTopology tridentTopology = new TridentTopology();
        TridentState tridentState = tridentTopology
                .newStream("committee",new
RandomFeaturesForClassifySpout(10,4,3,true))
                .each(new Fields("class","c0","c1","c2"),
                new InstanceCreator<Integer>(true),
                        new Fields("instance"))
                .partitionPersist(new MemoryMapState.Factory(),new Fields("instance"),
                        new ClassifierModelUpdate("committee-classifier",
                                new CommitteeClassifier(2.0,3)));
```

- **Passive Aggressive Classifier**
  该算法在[论文][2] 中被提出，这个算法不仅可以用来解决二分类问题，还可以改造成可以解决多分类问题的算法，算法还可以被设计用来解决回归问题。

（1）、二分器

在解决二分类问题上，PA算法使用一个权重数组来决策分类，位特征实例的每一纬提供支持，支持度越高，表示该纬向量对分类的结果贡献越大，也就说明，整个维度可以将特征区分开来的能力越强。下面是该算法解决二分类问题的伪代码，您也可以直接参考论文，您将可以收获更多。

```
 *           Algorithm describe
*   input: aggressiveness parameter C > 0
*   initialize: w1 = (0,0,...,0)
*   For t = 1,2....
*    * receive instance: Xt of (R^n)
*    * predict: Yt = sign( Wt.Xt)
*    * receive correct label: Yt of {1,-1}
*    * suffer loss l = max{0,1- Yt(Wt.Xt)}
*    * update:
*          1. set :
*                pi = l / |Xt|^2              ......PA
*                pi = min{C,l/|Xt|^2}         ......PA-I
*                pi = l / (l/|Xt|^2 + 1/2C)   ......PA-II
*
*         2. update: Wt+1 = Wt + pi*Yt*Xt
```

需要注意的是，该分类算法不仅在分类错误的时候调整模型，在分类正确的时候也会调整模型，这也是用的贪心算法，如果分类错误，说明当前对[可区分特征维度][3]的选取存在问题，需要修改，当分类正确的时候，说明目前的调整方向没有问题，可以往该方向上再次调整模型，让可区分维度更加明显，所以每次实例面对的都是新的模型，这样的分类算法更具有动态性，可以很好的适应不同的特征向量。

还需要注意一点，在更新权重向量的时候，可以选择PA，PA-I或者PA-II来更新，不同的更新策略适应不同分布模型的特征向量。或者，您可以使用不同的更新策略来形成三个分类器，在末尾的demo中，我将使用这三个分类器组装一个Hybrid模式的分类器，您可以参考那个分类器的实现来实现自己的分类器。

下面是引导您使用该算法来建立您的项目：

```
TridentTopology tridentTopology = new TridentTopology();
TridentState tridentState = tridentTopology
        .newStream("PA",new RandomBooleanDataForClassifySpout(10,3,true))
        .each(new Fields("label","x0","x1","x2"),
             new InstanceCreator<Boolean>(true),new Fields("instance"))
        .partitionPersist(new MemoryMapState.Factory(),new Fields("instance"),
                new ClassifierModelUpdate("pa",
                        new PassiveAggressiveClassifier()));
```

（2）、多分类器

该算法的多分类器基于二分类器，算法会为没个类别维护一个权重数组，其实就是将二分类器升级到多分类器。
一个有趣的问题是，多分类算法里面，依然是无论分类正确与否都会更新模型，但是只会更新两个类别的权重向量，一个是算法分类出来的类别的权重向量，一个是该特征实例本来属于的向量：

```
*          {  W(i) + adjust_vector     i is the true label.
*  W(i) =  {
*          {  W(i) - adjust_vector     i is the prediction label
```

当算法分类正确的时候，权重数组实际上是不更新的，因为加了然后减掉了啊！
但是当算法分类错误的时候，对于本来属于的那个类，需要刺激一下，对于分类出来的那个类，需要抵制一下，这个时候模型会更新。

多分类器的demo和二分类的demo区别不大，您可以参考项目examples包下的相关算法的demo来了解详情，在此不再赘述。

- **Perceptron Classifier**
  该算法是最简单也是经典的在线分类算法，您可以在很多地方找到相关资料，[在这里][4]您可以了解关于这个算法的大致情况。
  该算法翻译过来叫感知器分类算法，和人的感觉一样，觉得对就是对，觉得错就是错，但是该算法使用了更加科学的判断标准来决策分类。该算法维护一个权重数组，对特征向量的每一维度都保持一个vote值，这个值越大说明这个维度对区分特征向量的能力越强，在分类的时候，您需要提供一个阈值，算法将会计算出到来的特征实例与权重数组的乘积，然后与您提供的阈值比较，然后得出分类结果，这是感知器算法的二分类算法的分类算法，在训练模型时，算法只有在predict出一个错误的类别的时候才会更新模型，在更新的时候会考虑分类的结果，您还可以提供一个学习比例，感知器算法将会根据这个比例来更新分类模型，对于这个算法，您需要提供众多参数，但是如果您提供的参数足够合理（对于您的输入数据模型），那么将会得到很好的分类结果。

下面将向您展示如何使用感知器算法来分类：

```
TridentState tridentState = tridentTopology
        .newStream("perceptron",
                new RandomBooleanDataForClassifySpout(100,4,true))
        .each(new Fields("label","x0","x1","x2","x3"),
                new InstanceCreator<Boolean>(true),new Fields("instance"))
        .partitionPersist(new MemoryMapState.Factory(),new Fields("instance"),
                new ClassifierModelUpdate("perceptron",
                        new PerceptronClassifier()));
```

- **Winnow Classifier**
  这个算法有很多版本，可以在[论文][5]里面找到各种实现版本的伪代码，本算法库实现了三个版本的Winnow 分类算法，分别是基础的Winnow算法，平衡的Winnow算法，和改进的平衡Winnow算法。
  对于Winnow算法，需要注意的是，算法只在分类错误的时候会更新模型，分类和更新的策略非常简单，可以阅读论文来了解为什么这样做是有效的。
  对于平衡Winnow分类算法，算法会维护两个向量，一个是正反馈向量，一个是负反馈向量，算法对模型的更新代码如下：

```
for( int i = 0 ; i < features.length ; i ++ ){
            if( features[i] > 0 ){
                //demotion update
                this.u[ i ] *= this.alpha;
                this.v[ i ] *= this.beta;
            }else{
                //promotion update
                this.u[ i ] *= this.beta;
                this.v[ i ] *= this.alpha;
            }
        }
```

在分类决策的时候，算法会综合考虑正反馈向量u和负反馈向量v。

对于改正后的平衡Winnow算法，算法直接继承了平衡Winnow算法，然后再训练模型的时候，更新u/v向量的代码如下：

```
for( int i = 0 ; i < features.length ; i ++ ){
            if( features[i] > 0 ){
                //demotion update
                this.u[ i ] *= (this.alpha * ( 1 + features[i]));
                this.v[ i ] *= (this.beta * (1 - features[i]));
            }else{
                //promotion update
                this.u[ i ] *= (this.beta * ( 1 - features[i] ));
                this.v[ i ] *= (this.alpha * ( 1 + features [i]));
            }
        }
```

这样，Winnow算法的分类能力就更强大了。具体的证明以及介绍可以参考相关[论文][5]。

聚类算法（Clustering Algorithm）

package: com.hujian.trident.ml.clustering

- **K-means**
  K-means算法是最著名的聚类算法之一，具有简单有效的有点，本算法库也实现了基于Storm的K-means算法，需要注意的一点是，对于Storm平台上的算法来说，是不能假设数据的完整性的，数据是以流式来的，而且源源不断，这也是和以日志形式组织数据的大数据处理平台Hadoop的主要区别。
  聚类的目的就是将相似的数据聚合在一起，形成一个类，k-means算法使用距离计算公式来计算新到来的特征实例与各个聚类中心的聚类，然后找到一个聚类最近的类，将该特征归到这个聚类里面，然后用这个特征向量来更新聚类中心，一直这样下去，我们就可以大概知道我们的数据的分布。
  在此项目中，k-means算法的实现依然尊重上面的算法描述，需要注意的是，该算法有一个启动时间，在算法最初，聚类算法将不会运行，为了消除随机性，算法将等待第**K*10**个实例到来之后才运行聚类算法，k是聚类的个数，我们可以调整这个大小，越大数据的随机性就越小，但是为了实时性考虑，我们的初始时间不要太长。在算法初始化成功之后，对新到来的实例，运行算法将其归到聚类中的某一个类，然后使用算法来更新聚类中心，我们需要更新的仅仅是刚刚吸收了实例的那个聚类就可以了，而且这个速度是非常快的，整个过程十分简单，下面将介绍如何将其部署到Storm平台上，您可以使用下面的代码运行一个符合Storm规范的Topology。

```
TridentTopology tridentTopology = new TridentTopology();
TridentState kmeansState = tridentTopology
        .newStream("samples",new RandomFeaturesForClusteringSpout(3,4))
        .parallelismHint(1)
        .each(new Fields("x0","x1","x2"),
                new InstanceCreator<Integer>(),new Fields("instance"))
        .parallelismHint(1)
        .partitionPersist(new MemoryMapState.Factory(), new Fields("instance"),
                new ClusterModelUpdater("kmeans",new Kmeans(4)))
        .parallelismHint(1);
```

再说一句，对于如何确认一个实例是用来训练还是用来聚类，您应该在上游就确定，仅仅需要给数据打一个类型标签就可以搞定了。当然如果您熟悉RPC的话，可以使用Storm自带的RPC系统来实现分布式查询，而Topology则专心训练模型。

求平均算法

package:com.hujian.trident.ml.average

- **Moving Average**

对数据流进行求平均值是一个非常常见的需求，也是一个很简单的算法，我们只需要知道数据的大小和其和，就可以很快的计算出平均值。Moving Average算法是一种改进的求平均值的算法，该算法应用在一些特殊的场景下，比如我们需要知道某段固定长度的时间内的数据的平均值，我们就可以运行该算法来得到我们想要的结果。我们需要做的就是输入一个固定的长度，然后算法就会给我们结果，因为该算法过于简单，所以不再赘述，下面附上使用方法：

```
IAverage average = new MovingAverage(10);

        for( int i = 0 ; i < 100; i ++ ){
            average.update( Math.random() * 100 );
        }

        System.out.println("average:"+average.average());
```

基数计数算法

package:com.hujian.trident.ml.cardinality

所谓基数计数，就是找出数据集中存在多少个不一样的数据，返回这些不一样的数据的大小，在一般的算法中，我们很自然的想到用hash来做，只需要一个数组就可以了，但是这样的代价就是过多的内存使用，这样势必无法适应大数据环境下的基数计数任务，本项目实现的都是基数估计算法，但是估计的误差可以控制，所以都是概率算法，对那些要求精确的场景并不适用。

- **LogLog Cardinality**
  这是基数估计经典算法，源于[Linear Counting算法][6]，待会会介绍LinearCounting算法，对于LogLog算法，需要hash函数的支持，本项目中包含众多可用的hash算法。
  下面介绍该算法，假设有一个hash函数，可以将所有的输入数据都hash到一个固定长度的数值上面，我们找到第一个为1的位置，记做N，那么算法将估计出现的基数为(2^N)，为了消除不确定性，使得结果更加准确，算法引进**分桶平均算法**，使用多个桶，然后将数据hash到不同的同里面，用相同的算法算出N，与原先的桶合并，然后最后估计的时候求每个桶的平均值M，然后最后的结果就是（2^M）。
  算法的实现细节可以参考项目源代码，或者在各大搜索引擎搜索LogLog关键词，就可以获取更多关于该算法的内容。
- **[HyperLogLog cardinality][7]**
  该算法基于LogLog算法，可以在[论文][7]中找到这个算法，论文里面详细介绍了该算法，并且证明了为什么会选择那样的参数，相比于LogLog算法，该算法的改进点在于，LogLog的关键点在**分桶平均**，求平均值之后得到预测结果，使用的是几何平均，而该算法在此使用的是调和平均，您可以在该[链接][7]上找到一段精彩关于调和平均的描述。
- **Adaptive Counting Cardinality**
  该算法在[论文][8]里面被提出来，他的思想非常朴素，根据预测误差来选择使用LogLog还是使用Linear Counting，论文中给出了分界点，并且给出了详细的数据证明。
- **Linear Counting**

这是一个非常典型的基数估计算法，可以使用非常有限的内存来做相对精确到基数统计，您可以在[论文][9]中看到关于这个算法的一切。
该算法一样很朴素，假设有一个hash函数可以将任意的输入数据hash到一个固定长度的数值上（m），使用一个长度为m的bitmap，每个bit为一个桶，均初始化为0，设一个集合的基数为n，此集合所有元素通过H哈希到bitmap中，如果某一个元素被哈希到第k个比特并且第k个比特为0，则将其置为1。当集合所有元素哈希完成后，设bitmap中还有u个bit为0：

```
cardinality = ( -1 ) * m * ln u / m
```

您可以在[链接][10]找到一段关于该算法的敏锐的解释。

估计算法

package:com.hujian.trident.ml.frequency

- **Count Sketch**

您可以在[论文][11]中找到关于这个算法的信息，这个算法基于Basic Sketch算法，Basic Sketch算法的基本思想上，使用两个哈希函数h和g，对于输入的一个数据，将会得到两个hash值，即h(x) and g(x)，h(x)的值在1到k之间，k是计数数组的大小，而g(x)将会长生一个无偏估计，用于调整计数，计数数组记做C，那么一次计数模型的更新操作如下：

```
C[h(x)] = C[h(x)] + g(x)
```

Count Sketch算法相当于运行t次上面的算法，这就是基本的Count Sketch算法。

- **Lossy Counting**

该算法可以在[论文][12]里面找到，该算法的基本描述为：按照窗口来处理数据，窗口可以理解为一段时间之内的数据的统计情况，对于流处理，该窗口随着时间推移被陆续到来的数据填满，当一个窗口溢出时，对该窗口里面的统计情况做处理，基本的做法是对所有的计数都减1，然后继续等待后面的数据，然后做类似的处理，在本项目的Lossy counting算法实现时，采用了更加合理的算法，当窗口溢出时，对存在的数据做一次判断，如果出现的次数没有达到阈值，那么就直接将其删除，因为假设它没有达到我们的阈值频率，那就认为该数据是干扰的噪声，可以去掉。

- **Stick Sampling Counting**

该[算法][12]称为采样估计法，字面意思就是采样来估计，在本项目中对该算法的实现上，采用了随机函数来产生一个随机数，如果产生的随机数达到了我们设定的阈值，那么该实例就会被采样，然后对该实例采用Lossy Counting算法就可以了，在数据流量非常大的时候，可以使用此算法来代替Lossy counting算法，我们可以设定一些参数以控制误差。您可以在源代码中找到那些参数的意义，在您的项目中，您可以设定为符合您预期的参数来测试您的数据。

- **Space Saving**

该算法大概是计数算法中比较著名的了，根据算法名字就可以看出该算法致力于使用最少的内存来达到最高的准确率，该算法可以在[论文][13]里面找到，该算法提出一个数据结构，这个数据结构由很多桶组成，一个桶里面保存了那些出现次数一致的数据，相同的桶里面的数据的次数一样，它们之间用双向链表来维护。该算法使用该数据结构来维护模型，算法运行时，对于一个新到来的数据项，如果这个数据项在桶里面，那么将该数据项从桶里面拿出来，然后将其计数加1，然后查看该数据项所在桶的后一个桶的计数是不是等于该数据项的计数值，如果相等，那么将该数据项增加到该桶中，如果不等，那么就要新建一个桶，然后将该数据项填充到该新桶里面，设置新桶的计数值。如果原来的桶中没有数据项了，那么就要删除那个老的数据桶。如果新到的数据项不在当前维护着的桶里面，那么就要删除最小的那个数据项，然后将该数据项填充到那个桶里面。

综合实例

link：https://github.com/pandening/open-streamer/tree/master/src/main/java/com/hujian/trident/hybrid

该实例致力于提高分类算法的分辨率，以往单一分类算法的表现太过单一，在该实例中，我将使用4个多分类器，来实现一个多个分类器协调合作的分类器，当然，您还可以增加或者减少分类器的数量，所以该实例具有一定的通用性，您可以按照您的项目需求来组装或者卸载分类器。
该算法的基本思想是，找到大多数，比如我们需要将我们的数据分为三类，那么该实例的运行过程如下：

```
classification_1 = Committee.classify(instance)
classification_2 = PA.classify(instance)
if( classification_1 == classification_2 )
       then final_classification = classification_1
else
   classification_3 = PA_I.classify(instance)
   if( classification_1 == classification_3 )
       then final_classification = classification_3
   if( classification_3 == classification_2 )
       then final_classification = classification_2
   else
      classification_4 = PA_II.classify(instance)
      if( get the final classification )
          then end.
      else
          get the weight vector of classifiers
          get the max value index of weight vector
          then final_classification = classificationResult[index]

in the end, you should update the weight vector, the rules of updating the vector:
for each the classifier's classification list
    if( result of this classifier == final_classification )
       then weight[ index of this classifier ] ++
```

上面大概描述了整个算法运行的流程，一个特征实例的生命周期从Spout开始，经过每一个classifier都有可能终结，只要判断出了整个实例的最终类别，整个实例在接下来的Classifier里面将被抛弃，我使用了一个单例类来保存这些信息，除了保存一个实例是否被完全处理之外，还需要保存每个Classifier给该实例的分类建议，我将其保存在一个Map里面，key为实例的id，在此说明一下，为了辨别一个特征，我为每个特征增加了id属性，这个id属性是每个特征向量的唯一标识。value为一个数组，保存过程中Classifier给他的分类建议，这个数组的长度最大和分类器的数量一样，那是最坏的情况，当数组的长度达到了分类器的数量的时候，说明所有分类器都已经给过这个特征实例分类建议了，而且根据这些分类器的建议该实例没有办法决定自己属于哪个类别，这个时候就需要一个权重数组来帮助决策了，这个权重数组维护一个向量，向量的没一个维度代表一个分类器的权重，权重越大说明对应的分类器识别率越好。在给出了最终的分类结果之后，无论经过了几个分类器，都应该更新这个权重向量，更新策略非常简单，扫描整个特征实例的分类建议数组（由各个分类器给出），只要某个分类器的分类结果等于最终的分类结果，那么权重数组就加1，为了防止在大数据项目中数组值溢出，每次更新完成之后需要将数组归一化处理，将数组的和归一化为100，这个值可以自己决定，只要不溢出就可以。

该实例只是在给出一种在大数据平台之上的分类算法部署建议，可能在某些场景下确实有效，但是在某些情况下可能会适得其反，需要根据项目的属性和需求来使用。

总结

在开发项目的过程中，感谢github上的大量开源项目，以及google以及百度。
整个项目贯穿着几个关键词，**流处理**，**流数组**，**Storm**,**Trident**,**大数据**，**机器学习**，**在线学习**，**增量学习**，**等量学习**，这些关键字是我搜索相关资料和思考问题的指引，您也可以用这些关键字去搜索论文，可以搜索到很多相关的学术或者过程论文。
虽然很多算法都可以部署到大数据平台Storm之上，但是某些算法并不能那么简单的直接移植过来，需要做特别多的努力，而且效果不佳，所以在此，根据整个开发过程中的思考，总结出适合在Storm平台上实现的算法所需要具备的特征：

- （1）、对数据不要求完整，具备流式数据处理的特点，返利如排序算法，普通的排序算法需要完整的数据参与计算，这样的算法不具备移植到Storm平台上的特性
- （2）、对内存或者CPU的占用比例不会随着时间而增高，或者一定会在某个可以控制的节点上停止增长，因为在Storm平台上部署的算法，是假设可以永久运行下去的，如果算法的内存占用量随着时间不断增长，那么总有一个时间点，Storm集群的内存将会被盛满，整个集群将会瘫痪。

至少具备上面两点特性的算法才能考虑部署到Storm之上，当然，一些复杂度特别高的算法，可能也不适合部署到Storm之上，因为一个数据实例处理的时间过长会阻塞整个Topology。

如果您不幸只看到该文档而没看到源码，那么下面的地址可以直接跳转到源码：
https://github.com/pandening/open-streamer

作者简介

胡健，南开大学（2013.9 - 2017.6），计算机与控制工程学院，计算机科学与技术专业。
联系邮箱：<1425124481@qq.com>

参考文献

[1]: A Multi-class Linear Learning Algorithm Related to Winnow
[2]:Online Passive-Aggressive Algorithms
[3]:可区分特征维度：可以用该特征维度来区分不同的特征向量
[4]:http://www.cnblogs.com/jerrylead/archive/2011/04/18/2020173.html
[5]:Single-Pass Online Learning: Performance, Voting Schemes and Online Feature Selection
[6]:http://blog.csdn.net/keshixi/article/details/46730231
[7]:hyper LogLog algorithm,LogLog algorithm
[8]:Fast and Accurate Traffic Matrix Measurement Using Adaptive Cardinality Counting
[9]:A Linear-Time Probabilistic Counting Algorithm for Database Applications
[10]:http://blog.codinglabs.org/articles/algorithms-for-cardinality-estimation-part-ii.html
[11]:http://dimacs.rutgers.edu/~graham/pubs/papers/freqvldbj.pdf
[12]:Approximate Frequency Counts over Data Streams
[13]:Efficient Computation of Frequent and Top-k Elements in Data Streams