第9讲MapReduce机器学习算法

IDC实验室
华中科技大学计算机学院

Bayes分类器的MapReduce实现

Bayes分类器的MapReduce实现

- 需要分别实现训练和测试
- 训练过程
  - 假设类别集合C={C_1，C_2，…，C_N}
  - 需要计算两种概率：N个先验概率，每个term在每类中出现的条件概率

$$P(C_i)=\frac{N_{C_i}}{N}$$

其中N_{C_i}为训练集中C_i类文档的个数，N为训练集中总文档的个数

$$P(t\mid c)=\frac{T_{ct}+1}{\sum_{t'\in V}T_{ct'}+1}=\frac{T_{ct}+1}{(\sum_{t'\in V}T_{ct'})+B}$$

其中：T_{ct}为单词t在类别c的文档里出现的次数，$\sum_{t'\in V}T_{ct'}$为类别c里单词总数

V为所有单词集合，B为单词集合的大小

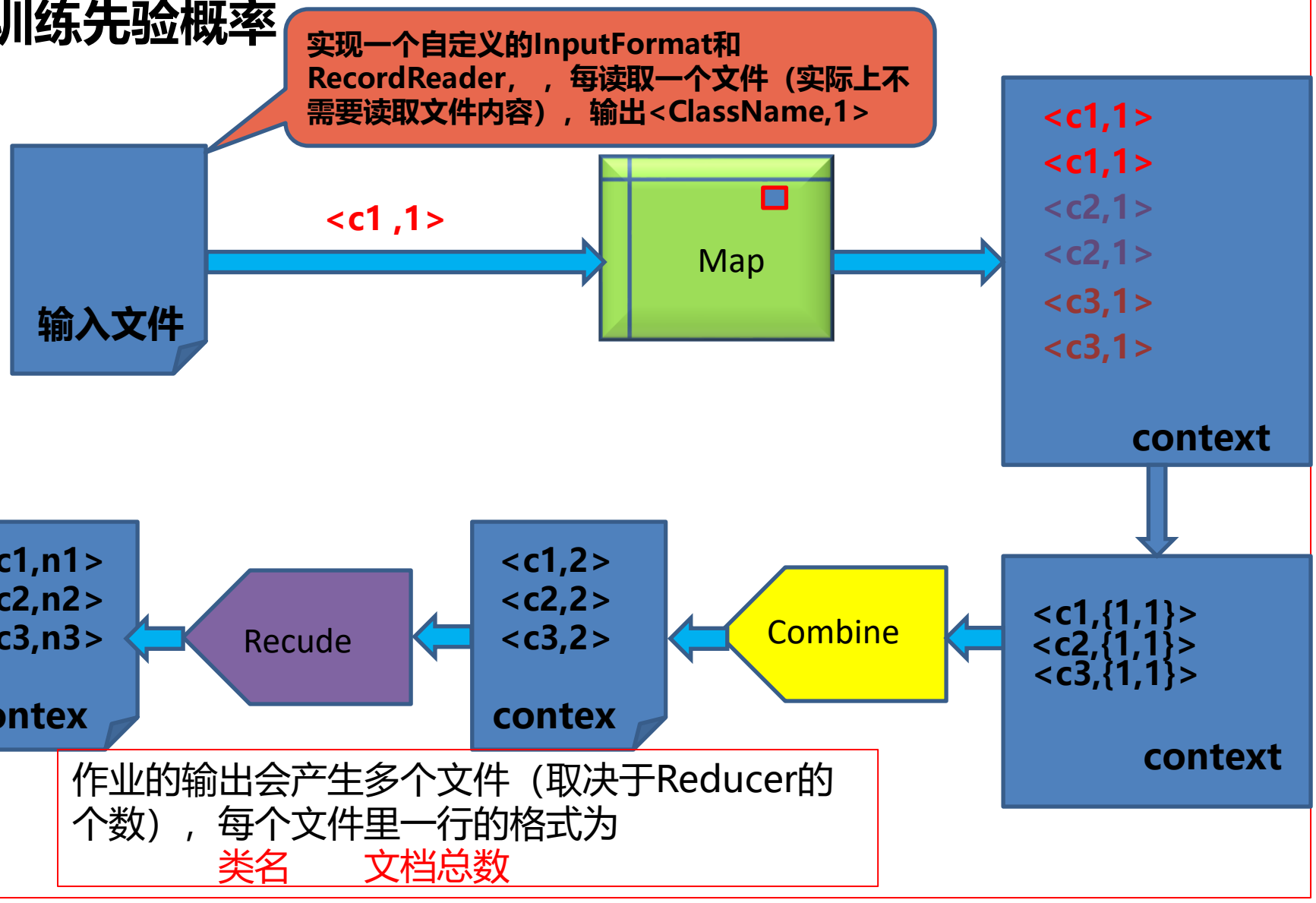
Bayes分类器的MapReduce实现

- **训练先验概率**
  - 需要编写一个单独的MapReduce Job，计算结果写入文件
  - 实现一个自定义的InputFormat和RecordReader，，每读取一个文件（实际上不需要读取文件内容），输出<ClassName,1>,其中ClassName为读取的文件所在的类别目录名，<ClassName,1>为Map的输入，Map不做任何处理，直接输出<ClassName,1>
  - Map的输出交给Combine处理，Combine的输入为<ClassName,{1,1,...,1}>,在Combine中计算1的个数，所以Combine的输出为<ClassName,Count>(Count为属于ClassName类别的文档个数，但是局部的)
  - Combine的输出交给Reducer，Reducer的输入为<ClassName,{count1，count2, ..., countn}>,在Reduce里对count1，count2,...,countn求和，就得到了ClassName的总数TotalCount,Reducer的输出为<ClassName,TotalCount>并写到文件
  - 该作业主要统计了每种类别文档的总数目，具体概率的计算放在了后面。作业的输出会产生多个文件（取决于Reducer的个数），每个文件里一行的格式为

    类名　　文档总数

Bayes分类器的MapReduce实现

- 训练先验概率



作业的输出会产生多个文件（取决于Reducer的个数），每个文件里一行的格式为

类名　　文档总数

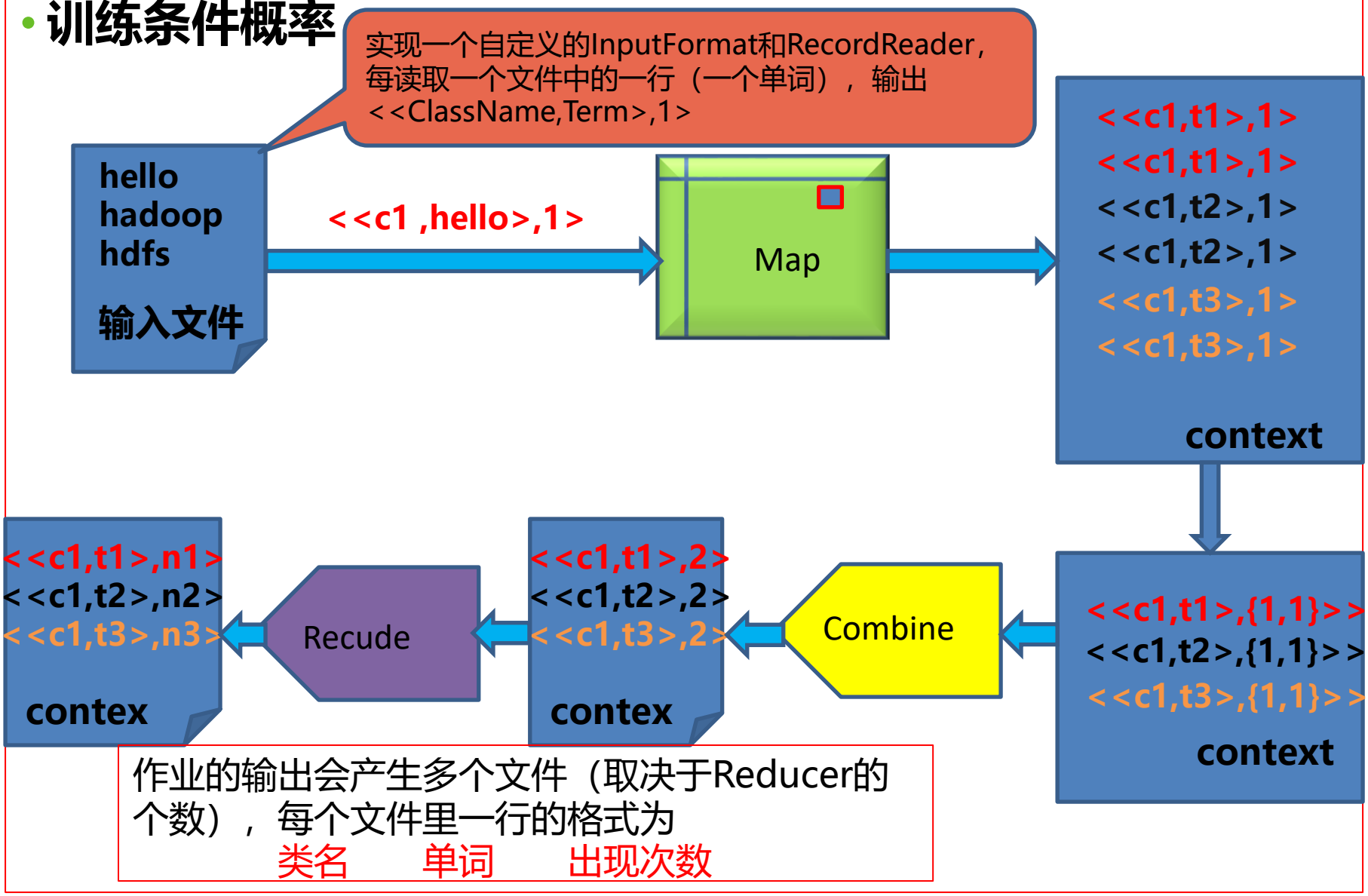
Bayes分类器的MapReduce实现

- **训练条件概率**
  - 需要编写一个单独的MapReduce Job，计算结果写入文件
  - 实现一个自定义的InputFormat和RecordReader，每读取一个文件中的一行（一个单词），输出<<ClassName,Term>,1>,其中key为<ClassName,Term>，ClassName为读取的文件所在的类别目录名，Term为单词，1表示Term在ClassName的类里出现一次
  - Map的输出交给Combine处理，Combine的输入<<ClassName,Term>,{1,1，...,1}>,在Combine中计算1的个数，所以Combine的输出为<<ClassName,Term>,Count>（Count为Term在ClassName的类里出现的次数，但是局部的）
  - Combine的输出交给Reducer，Reducer的输入为<<ClassName,Term>,{count1，count2,...,countn}>,在Reduce里把count1，count2,...,countn求和，就得到了Term在ClassName的类里出现的总次数TotalCount
  - Reduce输出<<ClassName,Term>,TotalCount>
  - 该作业只统计了每个<ClassName,Term>对出现的总次数，具体条件概率计算放在了后面。作业的输出会产生多个文件（取决于Reducer的个数），每个文件里一行的格式为

    类名　　单词　　出现次数

Bayes分类器的MapReduce实现

- 训练条件概率



Bayes分类器的MapReduce实现

- **预测**
  - 预测前将训练得到文件加载到内存里，计算先验概率和每个类别里单词出现的条件概率，可以交给自定义Mapper类和自定义Reducer类的包装类（比如叫Predition类）来处理，在Predition类里定义成类变量来保存这些学习到的概率，这样自定义Mapper类和自定义Reducer类都可以访问到这些概率。保存这些概率的数据结构应该用HashTable，这样可以高效地读取所需的概率值
  - 同时在Predition类实现一个静态方法，计算一个文档属于某类的条件概率P(class|doc)（不需要用MapReduce实现）需要计算其中每个单词出现的频率

    //文件内容作为一个字符串输入

    double Predition.conditionalProbabilityForClass(String content,String className);
  - 做好这些准备工作后，实现MapReduce作业

$$c_{\text{map}} = \arg\max_{c \in \mathbb{C}} \left[\log \hat{P}(c) + \sum_{1 \le k \le n_d} \log \hat{P}(t_k|c) \right]$$

Bayes分类器的MapReduce实现

- **预测**
- **MAP**
  - 每读取一个文件（这里需要把文件内容作为一个整体读取成为一个String,如何做到？），产生<docId,content>作为Map的输入
  - 在Map里做

    for each class in C

    调用 Predition.conditionalProbabilityForClass(String content,String className)

    得到 <docId,<ClassName,Prob>>作为Map的输出

    end for
  - 因此Map的输入为<docId,content>,Map的输出为list<docId,<ClassName,Prob>>
- **REDUCE**
  - 输入为<docId,list<ClassName,Prob>>,找到最大的Prob，输出<docId,最大Prob对应的ClassName>

Bayes分类器的MapReduce实现

- **EVALUATION**
  - **Per class evaluation measures**

	Yes(Ground Truth)	No(Ground Truth)
Yes(Classified)	true positives (tp)	false positives (fp)
No(Classified)	false negatives (fn)	true negatives (tn)

$$P = tp/(tp + fp)$$

$$R = tp/(tp + fn)$$

$$F1 = 2PR/(P + R)$$

P(Precision,精度): 被分类为yes的样本中有多少真实类别是yes

R(Recall,精度): 真实类别是yes的样本中有多少被分为yes

F1: P和R的调和平均

Micro- vs. Macro-Averaging

- If we have more than one class, how do we combine multiple performance measures into one quantity?
- Macroaveraging: Compute performance for each class, then average.
- Microaveraging: Collect decisions for all classes, compute contingency table, evaluate.

Micro- vs. Macro-Averaging: Example

Class 1

	Truth: yes	Truth: no
Classifier: yes	10	10
Classifier: no	10	970

Class 2

	Truth: yes	Truth: no
Classifier: yes	90	10
Classifier: no	10	890

Micro.Av. Table

	Truth: yes	Truth: no
Classifier: yes	100	20
Classifier: no	20	1860

- Macroaveraged precision: (0.5 + 0.9)/2 = 0.7
- Microaveraged precision: 100/120 = .83
- Why this difference?

  Macroaveraging gives equal weight to each class, whereas microaveraging gives equal weight to each per-document classification decision.

Bayes分类器的MapReduce实现

- **EVALUATION**
  - 读取目录，获得每个文档的真实类别
  - 读取预测结果文件，获得每个文档的预测类型
  - 计算

TP=0；TN=0；FP=0；FN=0；
for each c in C
  if(每个文档的真实类别为c and 每个文档的预测类型为c) TP++;
  else if(每个文档的真实类别为c and 每个文档的预测类型不为c) FN++;
  else if(每个文档的真实类别不为c and 每个文档的预测类型为c) FP++;
  else if(每个文档的真实类别不为c and 每个文档的预测类型不为c) TN++;
  class c 的Precison = TP/(TP + FP)； class c 的Recall = TP/(TP + FN)
  class c 的F1 = 2PR/(P+R)
end for
最后分别用Micro-Average和Micro-Average计算总的Precison，Recall，F1

Local Aggregation

MapReduce编程实例和设计模式

- Local Aggregation

  影响MapReduce程序执行效率的一个最重要因素就是中间结果的交换，即mapper的结果传给reducer。因此对mapper产生的中间结果进行局部聚集(Local Aggregation) 非常重要。

- 对中间结果的局部聚集第一个方法就是Combiner。它是一个在MapReduce框架内的通用机制，可以减少mapper产生的中间结果的数量

MapReduce编程实例和设计模式

- WordCound程序的伪代码如下所示（没有Combiner）：程序1

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)

class Reducer
    method Reduce(term t, counts [c1, c2, . . .])
        sum ← 0
        for all count c ∈ counts [c1, c2, . . .] do
            sum ← sum + c
        Emit(term t, count sum)
```

一次读进一个文档，利用WholeFileInputFormat类

MapReduce编程实例和设计模式

- 对WordCount程序加上Combiner：程序2

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)
class Combiner
    method Combine(term t, counts [c1, c2, . . .])
        sum ← 0
        for all count c ∈ counts [c1, c2, . . .] do
            sum ← sum + c
        Emit(term t, count sum)
class Reducer
    method Reduce(term t, counts [c1, c2, . . .])
        sum ← 0
        for all count c ∈ counts [c1, c2, . . .] do
            sum ← sum + c
        Emit(term t, count sum)
```

MapReduce编程实例和设计模式

- Combiner是利用MapReduce API提供的Hook(钩子）。程序员只需要实现Combiner类的Combine方法，MapReduce框架会自动调用自定义的Combiner类的Combine方法。
- 然而，我们无法控制Combiner的执行。例如，Hadoop不保证Combiner被应用了多少次，甚至根本不保证它会被应用。在很多场合，这种不确定性是无法接受的。
- Combiner减少了在网络中shuffled的中间数据量，但实际上并没有减少Mapper首先发出的键值对的数量。
- 我们是否可以自己实现和Combiner类似的功能，但是是自己完全可控的？
- 这就是In-Mapper Combining设计模式

MapReduce编程实例和设计模式

- 对WordCount程序加上In-Mapper Combiner：程序3

```
class Mapper
    method Map(docid a, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(term t, count H{t})
```

AssociativeArray可以是一个Java Map，里面存放是键值对，key是单词（term），value是单词出现次数

统计整个文档d范围内，t出现的次数

Mapper的输出是<t, t在d中出现的次数>,注意：这时map输出的<K,V>对 比程序1的Mapper输出的<K,V>对大大减少了，因为此时的t是不重复的，考虑到一些频繁出现的单词如the，In-Mapper Combiner大大减少了Mapper输出的键值对数量

Reducer的实现和程序1完全一样

MapReduce编程实例和设计模式

- 对In-Mapper Combiner可以进一步改进
- 回忆下在第4章介绍的Mapper类的run方法

```
//Mapper类的run方法
public void run(Context context) throws IOException, InterruptedException {
   setup(context); //在任务开始运行调用一次setup完成任务的设置
   while (context.nextKeyValue()) {
      map(context.getCurrentKey(), context.getCurrentValue(), context);
   }
   cleanup(context); //在任务结束运行调用一次cleanup完成任务的清理
}
```

这里会调用客户定义的map方法（多态）

- 在Mapper对象的map方法处理每个键值对前，setup方法首先被调用。在所有的键值对被map方法处理完后，cleanup方法被调用。
- setup方法就是MapReduce API为我们提供的Hook(钩子),我们可以在我们实现的Mapper子类里覆盖这个方法，去初始化一个AssociativeArray，例如一个Java Map，保存<t, count>(t为单词，count为其出现次数)。这样我们可以跨map方法的多次调用保存状态，如果每次map调用的输入是一个文档，我们就可以在AssociativeArray里跨文档保存<t, count>(t为单词，count为其出现次数)
- 同理，我们可以在Mapper子类里覆盖cleanup方法，在这里输出键值对

MapReduce编程实例和设计模式

- 对WordCount程序实现改进版In-Mapper Combiner：程序4

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

现在是在Initialize方法里初始化AssociativeArray
Initialize方法就是setup方法，后面不再特别说明。
这样可以在Map方法的多次调用间保存状态

由于每次Map调用传进来整个文档，因此现在是跨文档进行单词出现次数统计

现在是在Close方法里输出键值对，Close方法就是cleanup方法，后面不再特别说明。t是跨多个文档的不重复的t，因此In-Mapper Combiner进一步大大减少了Mapper输出的键值对数量

Reducer的实现和程序1完全一样

思考：程序4是在什么范围内跨文档统计词频?

一个Split范围内。一个Split由一个Map Task处理，一个Map Task会实例化一个Mapper对象，运行该对象的run方法处理这个Split。

MapReduce编程实例和设计模式

- 现在可以在Mapper里实现键值对的Combine，我们不再需要使用MapReduce框架为我们提供的Combiner钩子。
- 这是一种MapReduce的设计模式，将其命名为In-Mapper Combiner
- 这个设计模式有二个好处：
  - 完全可控。现在程序员可以完全控制Local Aggregation何时发生，如何实现
  - 比MapReduce框架为我们提供的Combiner钩子效率更高，可以大大减少Mapper输出的键值对数量。而MapReduce框架为我们提供的Combiner是无法减少Mapper输出的键值对数量的
- 这个设计模式的缺点：
  - 依赖MapReduce框架的内部实现
  - 由于要在Split的范围内，去聚合键值对，对内存要求高

Local Aggregation的正确设计

MapReduce编程实例和设计模式

- 考虑这样一个例子：我们有一个海量数据集，其中输入的key为字符串，输入的value为整数，我们希望计算每个key所对应的value的均值，其基本实现为：程序5

```
class Mapper
        method Map(string t, integer r)
                Emit(string t, integer r)
class Reducer
        method Reduce(string t, integers [r1, r2, . . .])
                sum ← 0
                cnt ← 0
        for all integer r ∈ integers [r1, r2, . . .] do
                sum ← sum + r
                cnt ← cnt + 1
        ravg ← sum/cnt
        Emit(string t, integer ravg)
```

Mean(1, 2, 3, 4, 5) ≠ Mean(Mean(1, 2),Mean(3, 4, 5))
因此我们不能直接用Reducer作为Combiner

MapReduce编程实例和设计模式

- 能否这样实现Combiner？程序6

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)
class Combiner
    method Combine(string t, integers [r1, r2, . . .])
        sum ← 0   cnt ← 0
        for all integer r ∈ integers [r1, r2, . . .] do
            sum ← sum + r  cnt ← cnt + 1
        Emit(string t, pair (sum, cnt))
class Reducer
        method Reduce(string t, pairs [(s1, c1), (s2, c2) . . .])
            sum ← 0   cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) . . .] do
            sum ← sum + s   cnt ← cnt + c
        ravg ← sum/cnt
        Emit(string t, integer ravg)
```

Combiner输出的key为string，value为键值对(sum, cnt)

看起来非常完美。不幸是，这个程序是错误的。

MapReduce编程实例和设计模式

- 能否这样实现Combiner？程序6

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)
class Combiner
    method Combine(string t, integers [r1, r2, . . .])
        sum ← 0   cnt ← 0
        for all integer r ∈ integers [r1, r2, . . .] do
            sum ← sum + r  cnt ← cnt + 1
        Emit(string t, pair (sum, cnt))
class Reducer
        method Reduce(string t, pairs [(s1, c1), (s2, c2) . . .])
            sum ← 0   cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) . . .] do
            sum ← sum + s   cnt ← cnt + c
        ravg ← sum/cnt
        Emit(string t, integer ravg)
```

Combiner的输入输出必须是：(K2, list(V2)) → list(K2, V2)
即输入输出的key类型必须一致，输入输出的value类型必须一致。现在其输入的value为Integer，而输出的value为Pair

MapReduce编程实例和设计模式

- Combiner的正确实现：程序7

```
class Mapper
        method Map(string t, integer r)
                Emit(string t, pair(r,1) )
class Combiner
        method Combine(string t, pairs [(s1, c1), (s2, c2) . . .])
                sum ← 0  cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) . . .] do
                sum ← sum + r   cnt ← cnt + c
        Emit(string t, pair (sum, cnt))
class Reducer
        method Reduce(string t, pairs [(s1, c1), (s2, c2) . . .])
                sum ← 0   cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) . . .] do
                sum ← sum + s  cnt ← cnt + c
        ravg ← sum/cnt
        Emit(string t, integer ravg)
```

现在Map输出value是pair(r,1)
Combine输入value是pairs [(s1, c1), (s2, c2) . . .])

MapReduce编程实例和设计模式

- 利用In Mapper Combiner：程序8

```
class Mapper
        method Initialize
                S ← new AssociativeArray
                C ← new AssociativeArray
        method Map(string t, integer r)
                S{t} ← S{t} + r
                C{t} ← C{t} + 1
        method Close
                for all term t ∈ S do
                        Emit(term t, pair (S{t},C{t}))
```

在Mapper对象初始化时，创建二个AssociativeArray S保存string对应值的sum，C保存string对应值的count

计算string对应值的sum，计算string对应值的count

Map函数执行完，在关闭Mapper对象时输出键值对 Key是string t，value是键值对（sum，count）。 这时整个InputSplit处理完

Reducer的实现和程序7完全一样

统计单词的共现性

MapReduce编程实例和设计模式

- 在基于文本的机器学习算法中，除了需要统计单词词频（Word Count）外，统计单词之间的共现性也是经常需要的
- 即需要从海量文档里，计算单词共现矩阵M（Co-occurrence Matrices）
- 单词共现矩阵M为n×n的矩阵，n为文档集合里所有单词的并集的单词个数，m_{ij}为单词w_i和单词w_j在某个上下文共同出现的次数，这里的上下文可以是相邻、相隔为k个单词(k>1)，在同一个句子，同一个段落、在同一篇文档等等

MapReduce编程实例和设计模式

- MapReduce实现共现矩阵算法1：程序9

```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            for all term u ∈ Neighbors(w) do
                Emit(pair (w, u), count 1)


class Reducer
    method Reduce(pair p, counts [c1, c2, . . .])
        s ← 0
        for all count c ∈ counts [c1, c2, . . .] do
            s ← s + c
        Emit(pair p, count s)
```

在嵌套的for循环里，只要是二个单词满足Neighbors关系，就输出(pair (w, u), count 1)
Neighbors(w) 取决于具体上下文（二个单词是邻居的具体定义忽略）

这种方法称为“Pairs”方法，因为将(w,u)对作为Mapper的输出Key

MapReduce编程实例和设计模式

- MapReduce实现共现矩阵算法2：程序10

```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            H ← new AssociativeArray
            for all term u ∈ Neighbors(w) do
                H{u} ← H{u} + 1
            Emit(Term w, Stripe H)
class Reducer
    method Reduce(term w, stripes [H1,H2,H3, . . .])
        H_f ← new AssociativeArray
        for all stripe H ∈ stripes [H1,H2,H3, . . .] do
            Sum(H_f,H)
        Emit(term w, stripe H_f )
```

对每个单词w，创建一个AssociativeArray H，这个矩阵为行矩阵(stripe)，这一行的每一列为w的邻居单词出现的次数

对单词w的每个邻居u，更新H中u列的内容

输出key为单词w，value为w对应的stripe H

对每个单词w，创建为行矩阵(stripe)H_f

将H合并到H_f

将最后输出(w, H_f) ， H_f为和单词w共现的完整统计

这种方法称为“Stripes”方法，因为将Stripe H作为Mapper的输出Key

统计单词的共现相对频率

MapReduce编程实例和设计模式

- 单词共现矩阵M记录的是单词之间的共现次数，这个共现次数是绝对次数
- 统计单词的绝对共现次数的缺点是：它没有考虑到有些词比其他词出现得更频繁这一因素。
  - 单词w_i可能与单词w_j共现次数更多，仅仅是因为w_i是一个非常常见的单词（如the）
- 更好的度量是将绝对共现次数转换成相对频率，f(wj|wi)

$$f(w_j|w_i)=\frac{N(w_i,w_j)}{\sum_{w'}N(w_i,w')}$$

在上式中，分母为与单词w_i共现的其他单词的共现次数之和，称为单词w的marginal

- 如何利用MapReduce计算海量文本文档的单词共现相对频率矩阵R，其中r_{ij}为单词w_i和单词w_j在某个上下文共同出现的相对频率

MapReduce编程实例和设计模式

- MapReduce实现共现相对频率矩阵算法：程序11

```
class Reducer
    method Reduce(term w, stripes [H1,H2,H3, . . .])
        Hf ← new AssociativeArray
        Hr ← new AssociativeArray
        for all stripe H ∈ stripes [H1,H2,H3, . . .] do
            Sum(Hf,H)
        //计算相对频率
        marginal ← 0
        for all term u in Vocabulary
            if u 不等于w
                marginal ← marginal + Hf{u}
            Hr ← Hf / marginal
        Emit(term w, stripe Hr )
```

创建另一个Stripe（行向量）H_r，含义与H_f类似，不同的是每个分量保存的是单词w与另外单词u的共现相对频率

Vocabulary是所有文档的单词的并集，其大小为一个Stripe（行向量）的维度

计算单词w的marginal

计算H_r，其每个分量= H_f对应分量/marginal

基于“Stripes”方法计算相对频率很简单，只需要修改程序10的Reducer部分，如上所示

MapReduce编程实例和设计模式

- 如何基于“Pairs”方法（程序9）计算单词相对频率？
- “Pairs”方法里，Reducer接受(w_i,w_j)作为key，然后计算(w_i,w_j)的出现次数，因此没法计算单词w的marginal
- 幸运的是，和In Mapper Combiner的设计模式思想类似，我们可以在Reducer的Initialize方法里创建一个数据结构buffer，这个数据结构可以跨reduce方法（也就是跨不同的输入key）将所有的属于该Reducer对象的Partion里的(w_i,w_j)缓存到该buffer里，然后在该buffer里计算每个单词的marginal

Method Summary

protected void	**cleanup**(org.apache.hadoop.mapreduce.Reducer.Context context) Called once at the end of the task.
protected void	**reduce**(KEYIN key, Iterable<VALUEIN> values, org.apache.hadoop.mapreduce.Reducer.Context context) This method is called once for each key.
void	**run**(org.apache.hadoop.mapreduce.Reducer.Context context) Advanced application writers can use the run(org.apache.hadoop.mapreduce.Reducer.Context)
protected void	**setup**(org.apache.hadoop.mapreduce.Reducer.Context context) Called once at the start of the task.

MapReduce编程实例和设计模式

- 要基于“Pairs”方法计算共现相对频率，必须做到以下二点：
  - 必须定义(w_i, w_j)对的排序：首先按左字排序，然后按右字排序。基于这样的排序，Reducer很容易check是否和单词w_i相邻的所有(w_i, w_j)对都已经遇到，从而很容易计算单词w_i的marginal。还记得TextPair吗？
  - 我们必须确保所有左字相同的(w_i, w_j)对都被发送到同一个Reducer。即我们必须定义一个只关注左边单词的自定义Partitioner。也就是说，分区器应该只基于左单词w_i的哈希进行分区。
- 这个算法确实可以工作，但是它也有缺点：随着语料库的大小增加，词汇量也会增加，可能没有足够的内存来存储为了计算单词w_i的marginal而必须cache的所有(w_i, w_j)对及其计数

MapReduce编程实例和设计模式

- 采用一种设计模式，称为“order inversion”，可以极大地节省内存空间。该模式需要使用MapReduce的几种内在机制
- 该设计模式的关键就是就是将呈现给Reducer的数据进行排序：如果可以让Reducer在看到具体的((w_i, w_j),[cnt1,cnt2])对之前，能先计算出单词w_i的marginal，那么将很快算出(w_i, w_j)的相对频率：

  (cnt1+cnt2)/ w_i的marginal

MapReduce编程实例和设计模式

- 如何让Reducer能先算出单词w_i的marginal？
  - 修改Mapper，让map除了输出正常的$((w_i, w_j), 1)$对之外，还额外输出$((w_i, *), 1)$对，其目的是为了让Reducer能先算出单词w_i的marginal（因此，和w_i共同出现的另外一个单词我们不关心，用*表示）
  - 利用MapReduce的Combiner或者In Mapper Combiner设计模式，将这些特殊的$((w_i, *), 1)$对进行局部聚合，得到单词w_i的局部marginal，同时也对正常的$((w_i, w_j), 1)$对进行聚合，得到$((w_i, w_j), cnt)$
  - 在将Map输出的键值对送到Reducer前，会根据键对键值对进行排序。要保证特殊的键值对$((w_i, *), cnt)$对一定排在正常的$((w_i, w_j), cnt)$对之前。如何做到？定义特殊的TextPair类型
  - 最后别忘了，必须确保所有左字相同的(w_i, w_j)和$(w_i, *)$对都被发送到同一个Reducer

MapReduce编程实例和设计模式

- “order inversion” 的具体例子

Key	Values	Computation
(dog *)	[6327, 8514, …]	计算marginal：$\sum_{w'} N(\text{dog}, w') = 42908$
(dog, cat)	[2,1]	f(cat|dog) = 3/42908
(dog, tiger)	[1]	f(tiger|dog) = 1/42908
…		
(dog, zebra)	[2,1,1,1]	f(zebra|dog)=5/42908
(doge, *)	[682, …]	计算marginal：$\sum_{w'} N(\text{doge}, w') = 1267$

第一个键值对（第一行）首先进入Reducer的reduce方法（为什么）

后面以dog为左字的键值对依次进入Reducer的reduce方法

以dog为左字的键值对按排序次序处理完了，再开始处理以doge为左字的键值对