基于 Hadoop MapReduce 并行计算框架的邮件自动分类

曹佳涵　白家杨　刘笑今
2019st04 小组

摘　要

本实验的目标是通过 MapReduce 编程来实现邮件的自动分类，通过本课程设计的学习，可以体会如何使用 MapReduce 完成一个综合性的数据挖掘任务，包括全流程的数据预处理、样本分类、样本预测等。我们使用 Hadoop MapReduce 并行计算框架对原始邮件文本进行特征选择、特征向量权重计算、文本分类和样本预测任务，从而搭建起一个完整的文本分类处理流程。我们采用了不同的特征提取方法和分类算法并进行比较和分析。此外，我们使用 Spark 同样完成了实验。

关键词： 邮件分类，并行计算，MapReduce，Spark

1　引言

本篇论文主要按预处理-算法原理-算法实现-结果分析的顺序进行。第二部分简要分析了训练数据集的特征，以对后续处理的方法起到指导性作用，便于接下来的处理。第三部分介绍了整个完整的运行流程，包括文本特征选择、特征向量权重计算、文本分类和样本预测，其中文本分类任务使用了不同的分类方法进行实现，包括 Naive Bayes，KNN，SVM。第四部分为使用 Spark 实现邮件自动分类的完整流程，同样分为在 Spark 中进行数据特征的选择和提取，分别使用 TF-IDF 特征和 Word2Vec 特征，使用 Spark 的 ml 库中的 SVM，Naive Bayes，Random Forest 进行分类预测。最后第五部分汇总不同分类器下的为分类预测结果，并进行分析和总结，第六部分列出了一些实验过程中对我们起到很大帮助作用的文献和网页。

实验分工见表一：小组分工。

2　数据分析

这是一个有监督的多分类问题。对于训练数据全集，经过分析，共有 20 个类别，它们编号，名称，以及包含的邮件文本个数见表 2：训练文本数据分析。

可以看到，训练样本全集共有 20 个类别，训练样本总数为 19997 个，每个类别下训练样本的数量是平衡的，几乎都是 1000 个，因此在进行训练时，不需要考虑类别不平衡问题。

表 1: 小组分工

小组成员	负责任务
曹佳涵	文本预处理，文本特征选择算法实现和改进，文本特征向量权重计算 TF-IDF 算法的实现，Spark Random Forest 模型以及实验结果分析，实验报告框架搭建和内容撰写及排版修改。
白家杨	中间文件格式处理，朴素贝叶斯的 MapReduce 和 Spark 实现，对比结果分析，Spark 特征提取中的 TF-IDF 和 Word2Vec 实现，对应部分的实验报告。
刘笑今	xxx

表 2: 训练文本数据分析

类别编号	类别名称	包含邮件文本个数	类别编号	类别名称	包含邮件文本个数
1	sci.electronics	1000	11	rec.sport.baseball	1000
2	comp.windows.x	1000	12	rec.autos	1000
3	talk.politics.mideast	1000	13	talk.politics.misc	1000
4	comp.os.ms-windows.misc	1000	14	talk.politics.guns	1000
5	rec.sport.hockey	1000	15	comp.graphics	1000
6	sci.crypt	1000	16	soc.religion.christian	997
7	comp.sys.mac.hardware	1000	17	rec.motorcycles	1000
8	sci.space	1000	18	misc.forsale	1000
9	talk.religion.misc	1000	19	sci.med	1000
10	comp.sys.ibm.pc.hardware	1000	20	alt.atheism	1000

3 MapReduce 处理流程

3.1 特征选择

本任务的主要工作是对原始的邮件文本中进行特征选择，选择出能够表征邮件主题的特征词，为后续的文本分类做准备。

对于输入的未分词的邮件训练样本全集和停词表，我们需要输出全局邮件文本特征，并对它们进行相应的编号，此外，对于训练数据集的目录，将目录名（即文本类别）转换为相应的类别序号。

对于该任务，我们采用了两种计算方法，分别为半并行化和并行化计算方法。

3.1.1 半并行化计算方法

半并行化计算方法是指将该任务划分为两个步骤，第一步读取训练样本全集和停词表，生成干净文本；第二步读取干净文本，生成全局邮件特征集。

第一步的主要思路是顺序执行，首先读取 20_newsgroup 文件夹下的子文件夹，将子文件夹名（即类别名）转化为类编号并进行存储。将 Lucene 的 Standard Analyzer 分词器作为基类，输入停词

表，自定义自己的带有停词表功能的停词器，然后顺序依次读取每个子文件夹下的所有文件，使用自定义的停词器对文本进行分词，并将分词结果储存在目标文件中，我们将分词后的文本称为干净文本，生成干净文本的过程是顺序执行，非并行化的。为了后续处理的方便性，我们修改输出文件的格式为：filename-classNum，filename 为原文件的文件名，classNum 为它所属的类别的编号，两者用分隔符'-' 隔开。

Algorithm 1 特征选择半并行化算法：第一步

Input: 邮件训练样本全集 U，停词表 S

Output: 用停词表分割后的干净文本

初始化类别名-类别编号对应表 classMap
初始化停词器 stopAnalyzer=StopAnalyzer(S)
for each 文本 u in U **do**
    新文本 u' = stopAnalyzer(u)
    通过类别名-编号对应表查找文本 u 的文件名 filename 对应的类别编号 num=classMap(u)
    在目标目录下创建 filename-num 文件，并将新文本 u' 的内容写入该文件。
end for

第二步从干净文本中提取全局邮件文本特征集的过程是并行化的，使用 MapReduce 并行计算框架，在 Map 阶段读取干净文本，对于每一个单词 word，发射 (word,1) 键值对。在 Reduce 阶段，设置一个全局缓存变量 n，为每个单词维护一个编号，并将每个单词和相应的编号输出到全局邮件特征集中。

Algorithm 2 特征选择半并行化算法：第二步

Input: 干净文本 U'

Output:

Map 阶段：
function MAP($filename$, $text$)
    for each word w in $text$ **do**
        Emit(w,1)
    end for
end function
Reduce 阶段：
function REDUCE($word$, $value$)
    全局缓存变量 $num = num + 1$
    将 $word$ 和 num 写入到全局邮件文本特征集中
    Emit($word$,$value$)
end function

3.1.2 并行化计算方法

非并行化顺序执行，缺点也显而易见：计算速度慢，效率低，因此我们设计了并行化的计算方法，利用 MapReduce 计算框架，很好地并行处理大量的训练文本，极大地加快了处理速度。

在 Map 阶段读取原文本，利用停词器将其分词后输出到目标目录下，同时对于每一个单词，发射 (word,filename-classNum) 键值对。值得注意的是，因为停词表是所有 Mapper 都需要各自初始化的，因此将其初始化放在 Map 阶段的 setup 过程中，首先将停词表文件路径存在 DistributedCache 中，然后在 setup 过程中读取该路径并利用其初始化停词器，在 map 过程中使用该停词器。

在 Reduce 阶段，设置一个全局缓存变量 n，用来表示每一个单词的唯一标号。输入为 (word, value)。由于在 Reduce 阶段，已经将相同 key 的键值对都整合到了一起，因此读取的 word 值是唯一不重复的，只需要利用全局变量 n 为每一个单词分配相应的标号，并将单词和编号输出到目标文件中即可。

Algorithm 3 特征选择并行化算法

Input: 邮件训练样本全集 U，停词表 S

Output: 用停词表分割后的干净文本，全局邮件文本特征集

初始化类别名-类别编号对应表 classMap
将停词表 S 存入 DistributedCache 中
Map 阶段：
function SETUP
    从 DistributedCache 中读取停词表 S
    初始化停词器 stopAnalyzer=StopAnalyzer(S)
end function
function MAP($filename$, $text$)
    $text'$ = stopAnalyzer($text$)
    将 $text'$ 写入到目标干净文本中
    for each word w in $text'$ **do**
        Emit(w,$filename$)
    end for
end function
Reduce 阶段：
function REDUCE($word$, $value$)
    全局缓存变量 $num = num + 1$
    将 $word$ 和 num 写入到全局邮件文本特征集中
    Emit($word$,$filename$)
end function

3.1.3 运行结果

该任务的输出结果见图 1：全局文本特征。

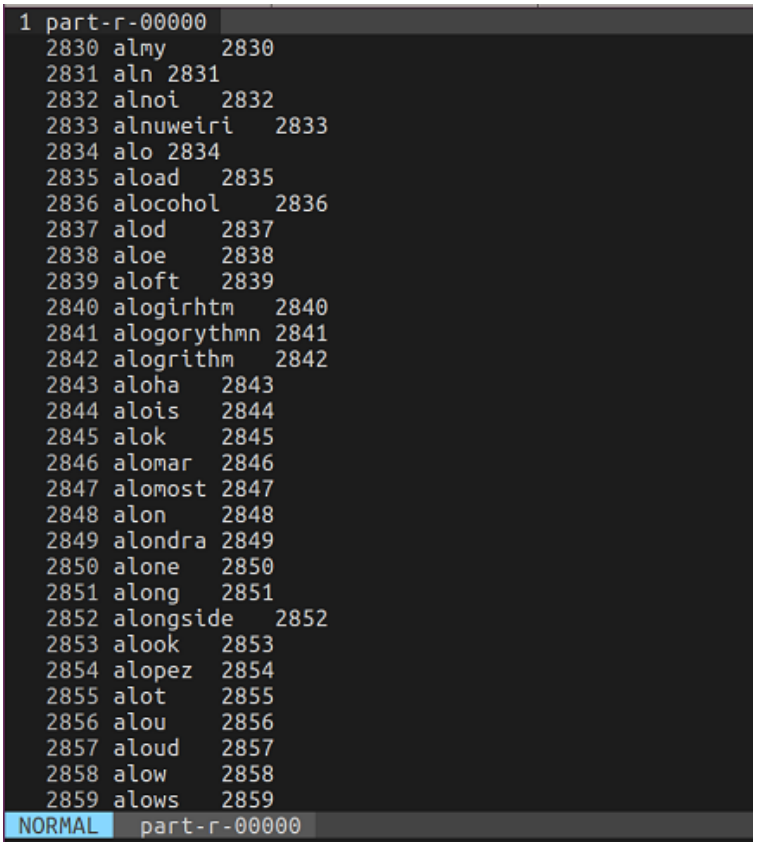


图 1: 全局邮件文本特征

3.2 特征向量权重计算

本任务主要是基于任务一输出的特征词向量，计算出每个邮件样本的特征词权重，特征词权重用来刻画特征词在描述此文本内容时所起的重要程度。

目前流行的特征向量权重算法大多是通过构造评估函数, 对特征集合中的每个特征进行评估, 并对每个特征打分，这样每个词语都获得一个评估值, 又称为权值。然后将所有特征按权值大小排序, 提取预定数目的最优特征作为提取结果的特征子集。显然, 对于这类型算法, 决定文本特征提取效果的主要因素是评估函数的质量。

目前常见的特征向量权重算法有 TF-IDF，词频方法 (Word Frequency)，文档频次方法 (Document Frequency)，互信息 (Mutual Information)，期望交叉熵 (Expected Cross Entropy)，x2 统计量方法，文本证据权 (The Weight of Evidence forText) 等。在本任务中，我们采用 TF-IDF 算法。

3.2.1 原理解释

TF-IDF（term frequency–inverse document frequency）是一种用于资讯检索与资讯探勘的常用加权技术。TF-IDF 是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降。TF-IDF 加权的各种形式常被搜寻引擎应用，作为文件与用户查询之间相关程度的度量或评级。

在一份给定的文件里，词频（term frequency，TF）指的是某一个给定的词语在该文件中出现的频率。这个数字是对词数 (term count) 的归一化，以防止它偏向长的文件。（同一个词语在长文件里可能会比短文件有更高的词数，而不管该词语重要与否。）对于在某一特定文件里的词语 t_i 来说，它的重要性可表示为：

$$\mathrm{tf}_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

以上式子中 $n_{i,j}$ 是该词 t_i 在文件 d_j 中的出现次数，而分母则是在文件 d_j 中所有字词的出现次数之和。

逆向文件频率（inverse document frequency，IDF）是一个词语普遍重要性的度量。某一特定词语的 IDF，可以由总文件数目除以包含该词语之文件的数目，再将得到的商取对数得到：

$$\mathrm{idf_i} = \log \frac{|D|}{|\{j : t_i \in d_j\}|}$$

其中 $|D|$ 是语料库中的文件总数，$|\{j : t_i \in d_j\}|$ 是包含词语 t_i 的文件数目（即 $n_{i,j} \neq 0$ 的文件数目）如果该词语不在语料库中，就会导致被除数为零，因此一般情况下使用 $1 + |\{j : t_i \in d_j\}|$

将 TF 值和 IDF 值相乘，即获得最终的 TF-IDF 值。

$$\mathrm{tfidf_{i,j}} = \mathrm{tf_{i,j}} \times \mathrm{idf_i}$$

某一特定文件内的高词语频率，以及该词语在整个文件集合中的低文件频率，可以产生出高权重的 TF-IDF。因此，TF-IDF 倾向于过滤掉常见的词语，保留重要的词语。

3.3 Hadoop MapReduce 实现

由于 TF-IDF 设计到 TF 和 IDF 两种不同指标的计算，所以所有计算难以直接在一个 job 中完成。我们通过分解，将其划分为三个 Job，分别对应于计算的三个阶段：TF 计算（TF-Job），IDF 计算（IDF-Job）和 TF、IDF 相乘的整合阶段（Integrate-Job）。

3.3.1 TF-Job

在 TF Job 中，主要任务是统计每个文件所有单词个数以及每个单词的出现次数，并计算每个单词在每个文件中的出现比例。

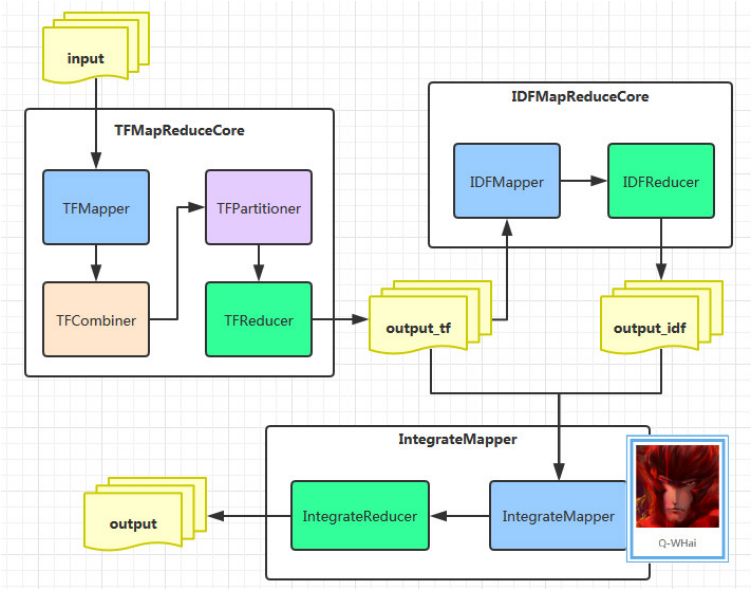


图 2: TF-IDF 算法框架

在 Map 阶段，读取在上个任务中生成的干净文本，输入为 (文件名 filenanme-文本内容 text) 键值对，对于每一个单词 word，发射 (word:filename, 1) 键值对，同时文本单词数量计数器 count 加一，当整个文本遍历完成后，发射 (!:count) 键值对。值得注意的是，我们还需要自定义 Partitioner，因为我们希望传入 Reducer 的数据按文件名排序（即 key 中':' 后面的字符串），所以在自定义 Partitioner 中需要分割 key 并取':' 后面的 filename 作为划分和排序标准。

在 Reduce 阶段，输入为 (xxx:filename, list[v1, v2, ...]) 键值对，此处 xxx 可能是单词 word，也可能是标记符'!'，当它是'!' 时，value 中存储的是该 file 中所有单词的数量，由于'!' 是键盘上可输出的字符的 ASCII 码最小的字符，并且我们的自定义 Partitioner 使得来自相同文本（即 filename 相同）的键值对排列在一起并且传入同一个 Reudcer，因此对于任意一个文本，所有来自它的键值对中 (!:filename,value) 是第一个被 Reducer 接收到的键值对，因此我们将其存储到全局缓存变量 allWordCnt 中；当它是 word 时，list[v1,v2,...] 中存储的是该 word 在该 file 中的出现次数列表，将其相加即可获得该单词在该文本中的频数，此时全局缓存变量 allWordCnt 存储的是该文本中所有单词的数量，将二者相除即可得到该单词在该文本中的 TF 值，最后发射键值对 (word:filename, tf)。

3.3.2 IDF-Job

在 IDF-Job 中，需要计算两个值，一是文档数目，这个可以在 Reducer 外读取上下文信息来获得，另一个是包含某个单词的文件数目，由于 TF-Job 中已经输出 (word:filename, tf) 键值对，因此只需要读取 TF-Job 的输出，解析 word 和 filename 的对应关系，统计同一个包含同一个 word 的文件数目即可，这是一个典型的 WordCount 任务。

在 Map 阶段，从输入的 (key, text) 键值对的 text 字符串按':' 分割，':' 前面的子字符串即为单词 word，发射键值对 (word,1)。

Algorithm 4 TF-IDF 算法: TF-Job

Input: 干净训练文本全集 D

Output: 所有文本下所有单词的 TF 值

Map 阶段：
function SETUP
    初始化全局缓存变量 $n = 0$
end function
function MAP($filename, text$)
    for each word w in $text$ **do**
        Emit(w : $filename$,1)
    end for
end function
Reduce 阶段：
function REDUCE(key, $value = list[v_1, v_2, ...]$)
    if key.beginWith('!') **then**
        全局缓存变量 $n = value$
    else
        $sum = 0$
        for each v in $list[v_1, v_2, ...]$ **do**
            $sum = sum + v$
        end for
        $tf = \frac{sum}{n}$
        Emit(key, tf)
    end if
end function

在 Reduce 阶段，输入为 (word, list[v1,v2,...])，只需要将 list[v1,v2,...] 累加起来，即为包含单词 word 的文档个数，利用公式 $\text{idf}_\text{i} = \log \frac{|D|}{|\{j:t_i \in d_j\}|}$ 可以求得 IDF 值，最后发射键值对 (word,idf)。

3.3.3 Integrate-Job

在 Integrate-Job 中，由于 TF-Job 和 IDF-Job 的输出文件格式不统一，因此主要任务是读取两个不同格式的文件，并将 TF 值和 IDF 值解析出来，最后相乘得到 TF-IDF 值。

TF-Job 的输出文件格式为：

```
word1:filename1    tf11
word2:filename1    tf21
```

Algorithm 5 TF-IDF 算法: IDF-Job

Input: TF-Job 的输出文本 T

Output: 所有单词的 IDF 值

Map 阶段：
function MAP($filename$, $text$)
    $word$ = split($text$, '!')[0]
end function
Reduce 阶段：
function REDUCE(key, $value = list[v_1, v_2, ...]$)
    $sum = 0$
    for each v in $list[v_1, v_2, ...]$ **do**
        $sum = sum + v$
    end for
    n = getProfileNum()
    $idf = \log \frac{n}{sum}$
    Emit(key, idf)
end function

```
    word1:filename2    tf12
    ...
```

IDF-Job 的输出文件格式为：

```
word1   idf1
word2   idf2
...
```

因此在处理时，在 Map 阶段，将读入的文本分割回原本的格式，发射 (word:filename, tf) 键值对或 (word,idf) 键值对。在 Reduce 阶段，对两种不同类型的 key 分开处理。此处利用了按 key 排序的特性，对于包含同一个 word 的 key，一定是 (word,idf) 键值对先被读入，然后再读入 (word,filename,idf) 键值对，因此，先将 idf 值存入全局缓存变量中，然后将该 idf 和后续的每个属于该单词的不同文本的 TF 值相乘，即可获得最终每个单词每个文本的 TF-IDF 值。

3.3.4 运行结果

通过将 TF-IDF 任务的输出格式整理后，最终的输出结果如下图所示：

Algorithm 6 TF-IDF 算法: Integrate-Job

Input: TF-Job 的输出文本 T，IDF-Job 的输出文本 I

Output: 所有单词所有文本的 TF-IDF 值

Map 阶段：
function MAP($filename$, $text$)
    key' = split($text$, '\t')[0]
    $value'$ = split($text$, '\t')[1]
    Emit(key',$value'$)
end function
Reduce 阶段：
function REDUCE(key, $value$)
    if key.contains(':') **then**
        $tfidf = value \times idf$
        Emit(key, tfidf)
    else
        全局缓存变量 $idf = value$
    end if
end function

图 3: 特征向量权重计算（TF-IDF）

3.4 分类预测：朴素贝叶斯

3.4.1 原理推导

朴素贝叶斯算法是生成模型中分类算法之一。所有朴素贝叶斯分类器都假设特定特征的值独立于给定类变量的任何其他特征的值。例如，如果水果是红色的，圆形的，直径约 10 厘米，则可以认为它是苹果。朴素贝叶斯分类器认为这些特征中的每一个都独立地贡献于该果实是苹果的概率，而不管任何可能的颜色，圆度和直径特征之间的相关性。抽象来看，朴素贝叶斯是一个条件概率模型：

给定一个要分类的问题实例，用向量表示 $\mathbf{x} = (x_1, \ldots, x_n)$ 代表一些 n 个特征（自变量），它为这个实例分配概率 $p(C_k \mid x_1, \ldots, x_n)$ 其中 C_k 表示 K 个可能的类别中的一个。使用贝叶斯定理，条件概率可以分解为

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k)p(\mathbf{x} \mid C_k)}{p(\mathbf{x})}$$

“朴素”的条件独立假设发挥作用：假设所有特征都在 $\mathbf{x}$ 是相互独立的，对类的条件 C_k。在这个假设下，

$$p(x_i \mid x_{i+1}, \ldots, x_n, C_k) = p(x_i \mid C_k)$$

因此，可以表示为

$$p(C_k \mid x_1, \ldots, x_n) = \frac{p(C_k)}{p(\mathbf{x})} \prod_{i=1}^{n} p(x_i \mid C_k)$$

因此只要比较 $p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k)$ 的值就可以判断类别。即为求下式

$$\hat{y} = \underset{k \in 1,\ldots,K}{\operatorname{argmax}}\ p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k)$$

常用的有高斯朴素贝叶斯，伯努利朴素贝叶斯 Bernoulli 和多项式朴素贝叶斯 Multinomial 三种。高斯朴素贝叶斯当处理连续数据时，典型的假设是与每个类相关联的连续值根据正态（或高斯）分布分布。多项式朴素贝叶斯使用多项事件模型，样本（特征向量）表示多项式生成某些事件的频率 $(p_1, \ldots, p_n)$ 其中 p_i 是事件 i 发生的概率（或多类情况下的 K 个多项式）。在词袋模型中事件表示单个文档中单词的出现。伯努利朴素贝叶斯：在多变量伯努利事件模型中，特征是描述输入的独立布尔值（二元变量）。尽管朴素贝叶斯分类器具有朴素设计和明显过于简单的假设，但它在许多复杂的现实世界中仍然运行良好。综合比较表明，贝叶斯分类的表现优于其他方法，特别是在文本处理方面。

3.4.2 文本分类中公式推导

词袋模型就是表示文本特征的一种方式。给定一篇文档，它会有很多特征，比如文档中每个单词出现的次数、某些单词出现的位置、单词的长度、单词出现的频率……而词袋模型只考虑一篇文档中单词出现的频率 (次数)，用每个单词出现的频率作为文档的特征（或者说用单词出现的频率来代表该文档）。利用词袋模型表示文本特征来进行朴素贝叶斯分类器的实现。实际上就是求

$$\hat{y} = \underset{k \in 1,\ldots,K}{\operatorname{argmax}}\ p(C_k) \prod_{i=1}^{n} p(w_i \mid C_k)$$

其中 w_i 表示文档中的第 i 个单词。由于每个概率值很小（比如 0.0001）若干个很小的概率值直接相乘，得到的结果会越来越小。为了避免计算过程出现下溢，最终无法比较，引入对数函数 Log，在 Log 空间中进行计算。然后得到如下公式

$$\hat{y} = \underset{k \in 1,\ldots,K}{\operatorname{argmax}}\ log\ p(C_k) + \sum_{i=1}^{n} log\ p(w_i \mid C_k)$$

训练朴素贝叶斯的过程其实就是计算先验概率 $p(C_k)$ 和后半部分似然函数的过程。

3.4.3 先验概率 $p(c)$ 的计算

$P(C_k)$ 的意思是：在所有的文档中，类别为 C_k 的文档出现的概率有多大。假设训练数据中一共有 N_{doc} 篇文档，只要数一下类别 C_k 的文档有多少个就能计算 $P(C_k)$ 了，类别 C_k 的文档共有 N_k 篇，先验概率的计算公式如下：

$$p(C_k) = \frac{N_K}{N_{doc}}$$

3.4.4 似然函数 $p(w_i|C_k)$ 的计算

由于是用词袋模型表示一篇文档 d，对于文档 d 中的每个单词 w_i，找到训练数据集中所有类别为 C_k 的文档，数一数单词 w_i 在这些文档（类别为 C_k）中出现的次数：$count(w_i, C_k)$。然后，再数一数训练数据集中类别为 C_k 的文档一共有多少个单词

$$\sum_{w \in v} count(w, C_k)$$

其中 V 是词库，即所有在文本中出现过的单词。（有些单词在词库中，但是不属于类别 C_k，那么 $count(w, C_k) = 0$ 计算二者之间的比值，就是似然函数的值。似然函数计算公式如下：

$$p(w_i \mid C_k) = \frac{count(w_i, C_k)}{\sum_{w \in v} count(w, C_k)}$$

需要考虑特殊情况某一类的文档不包含某个词时，考虑 add-one-smoothing，类似于拉普拉斯修正，其实就是将“出现次数加一”。此时似然函数变为：

$$p(w_i \mid C_k) = \frac{count(w_i, C_k) + 1}{\sum_{w \in v}(count(w, C_k) + 1)} = \frac{count(w_i, C_k) + 1}{\sum_{w \in v} count(w, C_k) + |V|}$$

其中 $|V|$ 表示词库的单词数。

3.5 Hadoop MapReduce 的实现

首先需要数据预处理，计算每个类对应的文件数便于接下来计算先验概率 $p(c)$。用一个 Job 来实现这个功能。

在这个 Job 的 Map 阶段，输入的是前面经过处理的文件格式：每个样本是一个文件，文件名为“文件名-类编号”的格式。在 Map 阶段中利用 split 得到相应的文件名和类编号，输出键值对 < 类编号，文件名 >

在 Reduce 阶段，用一个全局缓存变量 Set 来保存一个类编号的文件名，合并计算得到 Set 的元素个数就是类编号对应的文件数，最终输出键值对 < 类编号，文件数 > 到输出文件中。

算法伪代码如下。

最终得到的输出结果截图见图 4：训练 Job 输出结果。

在 Hadoop MapReduce 的实现中分成了两个 Job 来完成，分别是训练阶段的 Job 和预测阶段的 Job。

Algorithm 7 预处理得到类与对应的文件数关系

Input: 邮件训练样本全集 U

Output: 输出键值对 < 类编号，文件名 >

```
Map 阶段：得到相应的文件名和类编号，输出键值对 < 类编号，文件名 >
function MAP(key, text)
    for each 单词 w in text do
        类名编号 classnum = getFileName(w)).split("-")[1]
        文件名 filename = getFileName(w)).split("-")[0]
        Emit(classnum , filename)
    end for
end function
Reduce 阶段：合并并输出 < 类编号，文件数量 > 到输出文件中
定义全局缓存变量 Map<String,int> list 记录类编号对应文件数量
function REDUCE(key, values)
    定义全局缓存变量 Set<String> files
    for value in values do
        files.add(value)
    end for
    list.put((key, files.size())
end function
function CLEANUP
    for (key, value) in list do
        Emit(key, value)
    end for
end function
```

在训练阶段的 Job 中实现的功能是对每个样本中的每个单词，输出键值对 < 类编号 # 单词，单词出现的次数 >。使用这种结构的好处是：可以通过 Map 映射方便找到每个单词在不同类中的出现次数。

在 Map 阶段中，利用 FileSplit 结构获得文件名，由于文件名由“文件名-类编号”格式构成，从文件名中就可以的获得单词所属的类别。用 StringTokenizer 划分出单词，对每个单词输出键值对 < 类编号 # 单词, 1>

在 Combine 和 Reduce 阶段中，对读入的键值对 < 类编号 # 单词，单词出现的次数 > 进行合并，最后输出即可。

第一个 Job 算法伪代码如下算法。

Algorithm 8 朴素贝叶斯训练阶段第一个 Job

Input: 邮件训练样本全集 U

Output: 输出训练结果文件，记录键值对 < 类编号 # 单词，数量 >

```
Map 阶段：分割每个单词并根据文件名获得类名编号，输出 < 类编号 # 单词，数量 >
function MAP(key, text)
    for each 单词 w in text do
        类名编号 classnum = getClassNum(getFileName(w))
        Emit(classnum + "#" + w, 1)
    end for
end function
Combine 阶段：合并 < 类名 # 单词，num1>、< 类名 # 单词，num2> 为 < 类名 # 单词，num1+num2>
Reduce 阶段：合并并输出 < 类名 # 单词，数量 > 到输出文件中
function REDUCE(key, values)
    sum = 0
    for value in values do
        sum+ = value
    end for
    Emit(key, sum)
end function
```

最终得到的输出结果截图见图 5：类编号与对应文件数

预测阶段 Job 实现根据样本数据，计算朴素贝叶斯方法下每个样本对应各个类的概率，比较得到最大概率对应的类别就是预测类别。

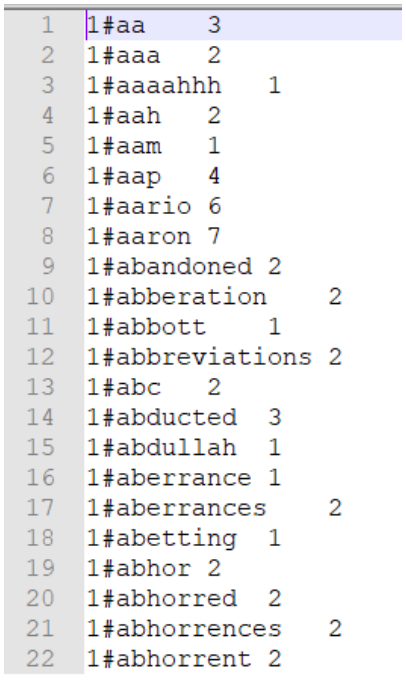
在 Map 的 setup 阶段中，先将训练阶段的键值对输出结果 < 类名 # 单词，出现次数 > 保存到全局缓存变量 Map 结构体中，并计算得到每个类对应的单词总数，作为 < 类名，单词总数量 > 也加入到 Map 中。同时构建全局缓存变量 Set 结构体作为词库，将出现过的单词加入到 Set 中，Set 中元素个数就是词库的大小。

在 map 阶段，对每个样本的每个单词 w_i，计算对每个类 c 的似然函数的值 $v = p(w|C_k) = \frac{count(w_i,c)+1}{\sum_{w\in v}(count(w,C_k)+1)} = \frac{count(w_i,c)+1}{\sum_{w\in v} count(w,C_k)+|Set|}$ 并输出键值对 < 文件名，c#v>。其中 $count(w_i, c)$ 可以通过 Map 映射找到。

在 Reduce 阶段，输入是 map 阶段输出的键值对 < 文件名，c#v>，对于每个相同的类别 c，计算 v 的乘积，并乘上先验概率 $p(c)$ 得到结果就是该样本对应类别 c 的概率，取其中概率最大的类别作为预测结果，输出 < 文件名，预测类编号 >

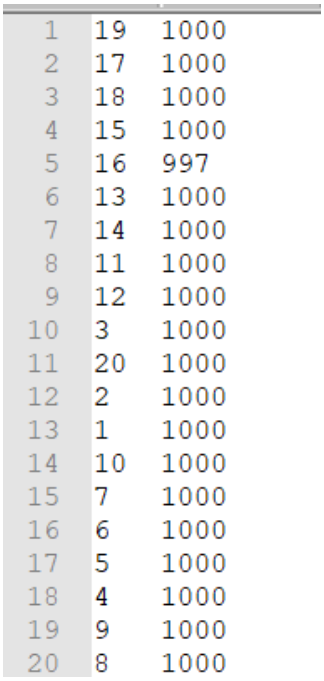
第二个 Job 算法如下：

得到的结果截图见图 6：预测结果。



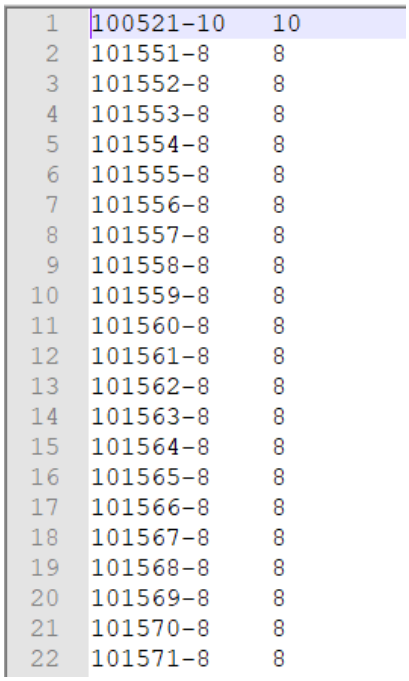
```
1#aa    3
1#aaa   2
1#aaaahhh   1
1#aah   2
1#aam   1
1#aap   4
1#aario 6
1#aaron 7
1#abandoned 2
1#abberation    2
1#abbott    1
1#abbreviations 2
1#abc   2
1#abducted  3
1#abdullah  1
1#aberrance 1
1#aberrances    2
1#abetting  1
1#abhor 2
1#abhorred  2
1#abhorrences   2
1#abhorrent 2
```

图 4: 训练 Job 输出结果



```
19  1000
17  1000
18  1000
15  1000
16  997
13  1000
14  1000
11  1000
12  1000
3   1000
20  1000
2   1000
1   1000
10  1000
7   1000
6   1000
5   1000
4   1000
9   1000
8   1000
```

图 5: 类编号与对应文件数



```
100521-10   10
101551-8    8
101552-8    8
101553-8    8
101554-8    8
101555-8    8
101556-8    8
101557-8    8
101558-8    8
101559-8    8
101560-8    8
101561-8    8
101562-8    8
101563-8    8
101564-8    8
101565-8    8
101566-8    8
101567-8    8
101568-8    8
101569-8    8
101570-8    8
101571-8    8
```

图 6: 预测结果

4 Spark 实现

4.1 简介

ML 和 MLlib 都是 Spark 中的机器学习库，目前常用的机器学习功能 2 个库都能满足需求。spark 官方推荐使用 ML, 因为 ML 功能更全面更灵活，未来会主要支持 ML，MLlib 很有可能会被废弃 (据说可能是在 Spark3.0 中 deprecated）。

两者的不同在于 ML 主要操作的是 DataFrame, 而 MLlib 操作的是 RDD，也就是说二者面向的数据集不一样。相比于 MLlib 在 RDD 提供的基础操作，ML 在 DataFrame 上的抽象级别更高，数据和操作耦合度更低。

DataFrame 是 Dataset 的子集，也就是 Dataset[Row], 而 DataSet 是对 RDD 的封装，对 SQL 之类的操作做了很多优化。相比于 MLlib 在 RDD 提供的基础操作，ML 在 DataFrame 上的抽象级别更高，数据和操作耦合度更低。ML 中的操作可以使用 pipeline, 跟 sklearn 一样，可以把很多操作 (算法/特征提取/特征转换) 以管道的形式串起来，然后让数据在这个管道中流动。ML 中无论是什么模型，都提供了统一的算法操作接口，比如模型训练都是 fit；不像 mllib 中不同模型会有各种各样的 trainXXX。MLlib 在 Spark2.0 之后进入维护状态, 这个状态通常只修复 BUG 不增加新功能。

所以综上我们在这里使用 ML 库来实现操作。

Algorithm 9 朴素贝叶斯预测阶段第二个 Job 的 Map 阶段

Input: 第一个 Job 训练结果键值对集合 A，训练邮件预测样本全集 U，类编号和类文件数对应表 B

Output: 输出键值对 < 文件名，类别 # 似然函数数值 >

```
定义全局缓存变量 Map<String, int> trainSet, classSet
定义全局缓存变量 Set<String> wordSum    % 记录词库
Map 阶段：分割每个单词并根据文件名获得类名编号，输出 < 类名 # 单词，数量 >
function SETUP(A, B)
    for each (key, value) in A do
        trainSet.put(key, value)
        wordSum.add(key.GetWord())
    end for
    for each (key, value) in B do
        classSet.put(key, value)
    end for
end function
function MAP(key, text)
    filename = key.getFileName()
    for each 单词 w in text do
        类名编号 classnum = GetClassNum(GetFileName(w))
        for each 类别 c in ClassSet.keys do
            key = c + "#" + word;
            wordcount = 1 + trainSet.get(key);//类中对应的词数
            sumClass = trainSet.get(c);   % 类对应的单词总数
            sumWord = wordSum.size();   % 词库中词数量
            result = log(wordcount / (sumClass + sumWord));
            Emit(filename, c + "#" + result);
        end for
    end for
end function
```

4.2 数据预处理：特征提取

4.2.1 计算 TF-IDF

第一步：读取原文本转换成 DataFrame

注意：由于 Spark2.0 起，SQLContext、HiveContext 已经不再推荐使用，改以 SparkSession 代之，故本文中不再使用 SQLContext 来进行相关的操作，关于 SparkSession 的可以参看 Spark2.0 的官方

Algorithm 10 朴素贝叶斯预测阶段第二个 Job 的 Reduce 阶段

Input: Map 阶段的输出，类编号和类文件数对应表 B

Output: 输出预测结果文件，记录键值对 < 文件名，预测类编号 >

```
1: 定义全局缓存变量 Map<String, int> classSet
2: 定义全局缓存变量 sumFile 表示文件总数
3: function SETUP(B)
4:     for (key, value) in B do
5:         ClassSet.put(key, value)
6:         SumFile += value
7:     end for
8: end function
9: function REDUCE(key, values)
10:    Map<String, double> mapResult
11:    for value in values do
12:        类名编号 classno = value.GetClassNum()
13:        number = value.GetNum()
14:        if mapResult.contains(classno) then
15:            number += mapResult(classno)
16:        else
17:            number += log(classSet.get(classname) / SumFile)
18:        end if
19:        MapResult.put(classno, number)
20:    end for
21:    maxClass = Max(mapResult) // 找到数值最大对应的类编号
22:    Emit(key, maxClass) // 输出结果
23: end function
```

文档。Spark2.0 以上版本的 spark-shell 在启动时会自动创建一个名为 spark 的 SparkSession 对象，当需要手工创建时，SparkSession 可以由其伴生对象的 builder() 方法创建出来。RDD 转 DataFrame 的实现通过 toDF 接口来实现。使用举例如下：

我们的文档是已经经过前面预处理过的文档，每个文件表示一个样本，每个样本名字为“文件名-类名编号”，文件内容由邮件单词构成，每一行为一个单词。

第二步: 分词处理

分词使用的是 RegexTokenizer 库。根据官方文档，RegexTokenizer 通过正则化式子来划分句子。在前面的处理中我们已经得到了样本中单词的划分使用的是换行符，所以正则表达式使用的是“$\backslash s$”

表示切分符为包括空格、制表符、换页符等空白字符的其中任意一个。使用示例如下：

```
val regexTokenizer = new RegexTokenizer()
  .setInputCol("sentence")
  .setOutputCol("words")
  .setPattern("\\s")
```

第三步：将单词转换成对应的矩阵, 计算频率

根据官方文档 API，FeatureHasher 的作用是将一组分类或数字特征投影到指定尺寸的特征向量中（通常远小于原始特征空间的特征向量）。这是使用散列技巧将要素映射到特征向量中的索引来完成的。

其中特征列每列可能包含数字或字符串列。列数据类型的行为和处理如下：

数字列：对于数字要素，列名称的哈希值用于将要素值映射到要素向量中的索引。默认情况下，数字要素不被视为字符串列（即使它们是整数）。要将它们视为字符串列，请使用 categoricalCols 参数指定相关列。

字符串列：对于字符串，字符串"column_name = value" 的哈希值用于映射到矢量索引，指示符值为 1.0。因此，类别特征是“一热”编码（类似于使用 OneHotEncoder）。

布尔列：布尔值的处理方式与字符串列相同。也就是说，布尔特征表示为“column_name = true”或“column_name = false”，指示符值为 1.0。忽略空（缺失）值（在结果特征向量中隐式为零）。

使用举例如下：

```
val hasher = new FeatureHasher()
  .setInputCols("real", "bool", "stringNum", "string")
  .setOutputCol("features")
```

第四步：计算 TF-IDF

ml 库中的 IDF 类根据给出的文档计算相应的 IDF，IDFModel 采用特征向量（通常从 HashingTF 或 CountVectorizer 创建）并缩放每个特征。直观地，它降低了在语料库中频繁出现的特征的权重。使用示例如下：

```
val idf = new IDF().setInputCol("rawFeatures").setOutputCol("
   features")
```

4.2.2 计算 Word2Vec

Word2Vec 是一种著名的词嵌入（Word Embedding）方法，它可以计算每个单词在其给定语料库环境下的分布式词向量（Distributed Representation，亦直接被称为词向量）。词向量表示可以在一定程度上刻画每个单词的语义。

如果词的语义相近，它们的词向量在向量空间中也相互接近，这使得词语的向量化建模更加精确，可以改善现有方法并提高鲁棒性。词向量已被证明在许多自然语言处理问题，如：机器翻译，标注问题，实体识别等问题中具有非常重要的作用。

Word2vec 是一个 Estimator，它采用一系列代表文档的词语来训练 Word2VecModel。该模型将每个词语映射到一个固定大小的向量。Word2VecModel 使用文档中每个词语的平均数来将文档转换为向量，然后这个向量可以作为预测的特征，来计算文档相似度计算等等。

Word2Vec 具有两种模型，其一是 CBOW ，其思想是通过每个词的上下文窗口词词向量来预测中心词的词向量。其二是 Skip-gram，其思想是通过每个中心词来预测其上下文窗口词，并根据预测结果来修正中心词的词向量。两种方法示意图如下图所示：总结简单来说，Word2Vector 实际上

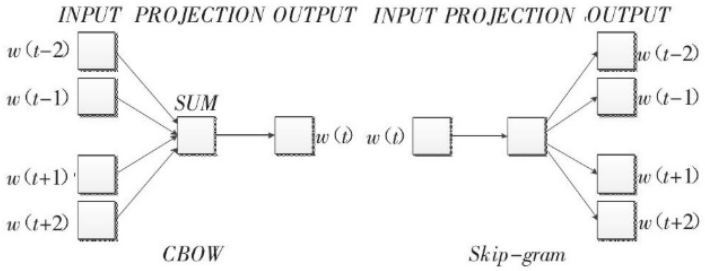


图 7: CBOW 和 Skip-gram 模型

就是用神经网络来实现词向量的生成。使用举例如下：

```
val word2Vec = new Word2Vec()
  .setInputCol("text")
  .setOutputCol("features")
  .setVectorSize(100).setMinCount(0)
```

4.3 朴素贝叶斯 Naive Bayes

ML 库中的 NaiveBayes 类实现了朴素贝叶斯模型。支持多项式朴素贝叶斯，可以直接用 TF-IDF 的结果进行训练，模型可调参数有：特征值列名 (featuresCol)、标签列名 (labelCol)、模型类型 (modelType，默认为多项式朴素贝叶斯)、平滑系数 (smoothing) 等。使用示例如下：

```
val model = new NaiveBayes()
  .setFeaturesCol("features")
  .setModelType("multinomial")
```

4.4 支持向量机 SVM

4.5 随机森林 Random Forest

随机森林是一个包含多个决策树的分类器，并且其输出的类别是由个别树输出的类别的众数而定。这个方法则是结合 Breimans 的"Bootstrap aggregating" 想法和 Ho 的"random subspace method" 以建造决策树的集合。随机森林的构造过程是：

1. 假如有 N 个样本，则有放回的随机选择 N 个样本 (每次随机选择一个样本，然后返回继续选择)。这选择好了的 N 个样本用来训练一个决策树，作为决策树根节点处的样本。
2. 当每个样本有 M 个属性时，在决策树的每个节点需要分裂时，随机从这 M 个属性中选取出 m 个属性，满足条件 $m << M$。然后从这 m 个属性中采用某种策略（比如说信息增益）来选择 1 个属性作为该节点的分裂属性。
3. 决策树形成过程中每个节点都要按照步骤 2 来分裂（很容易理解，如果下一次该节点选出来的那一个属性是刚刚其父节点分裂时用过的属性，则该节点已经达到了叶子节点，无须继续分裂了）。一直到不能够再分裂为止。注意整个决策树形成过程中没有进行剪枝。
4. 按照步骤 1~3 建立大量的决策树，这样就构成了随机森林了。

我们使用 Spark MLlib 中的 RandomForestClassifier 模型，调用方法和参数设置如下：

```
1  val model = new RandomForestClassifier()
2  .setFeaturesCol("features")
3  .setMaxDepth(30)
4  .setNumTrees(num)
5  .fit(trainDataRdd)
```

我们设置每一棵决策树的最大深度为 30（RandomForestClassifier 模型限制 maxDepth 参数最大为 30），num 为决策树的数量，我们通过改变 num 的值来观察随机森林预测效果随着决策树数量变化而发生的变化。

对于样本特征提取，我们使用 TF-IDF 和 Word2Vec 两种方式，并进行了对比，实验结果如下图所示：

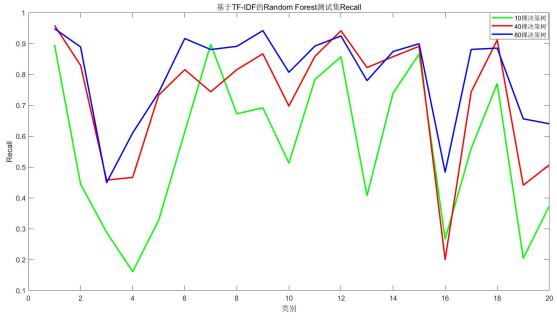


图 8: 基于 TF-IDF 的 RF 测试集 Recall

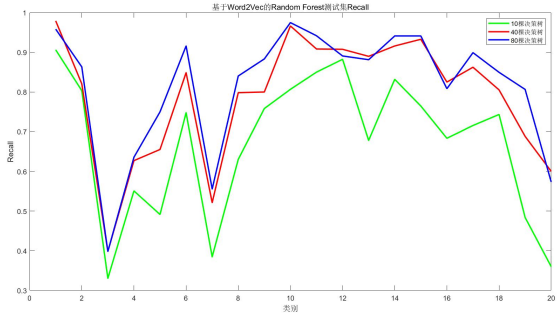


图 9: 基于 Word2Vec 的 RF 测试集 Recall

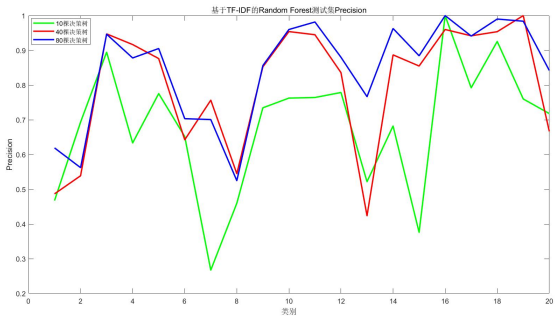


图 10: 基于 TF-IDF 的 RF 测试集 Precision

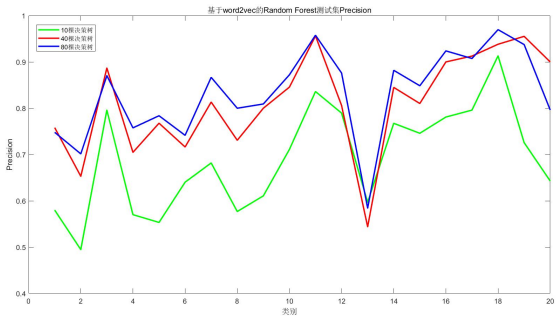


图 11: 基于 Word2Vec 的 RF 测试集 Precision

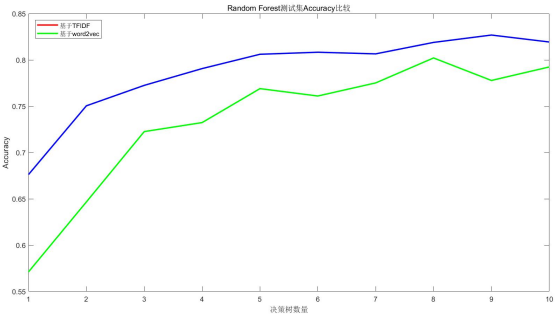


图 12: RF 测试集 Accuracy 比较

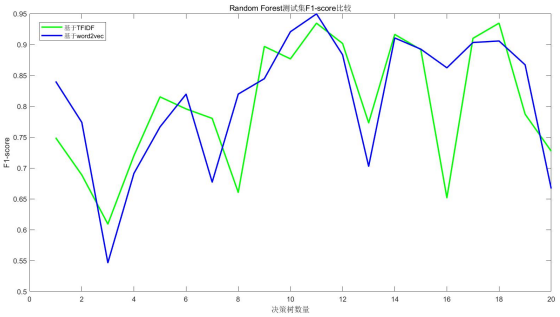


图 13: RF 测试集 F1-score 比较

分析实验结果，我们可以看到，总体上来说 Random Forest 在这个任务下的表现较为平庸，accuracy 只能达到 80% 多，但我们认为这和 Spark ML 库提供的 RandomForestClassifier 模型对决策树 maxDepth 参数的限制有关，由于 maxDepth 最大只能为 30，而本任务是一个 20 个类别的多分类问题，显然深度 30 的决策树是远远不够的，因此达到 80% 左右的 accuracy 可以说是在意料之中。

此外，无论是通过 TF-IDF 还是 Word2Vec 来提取特征，随着决策树数量的上升，accuracy 总体上来说是呈上升趋势，并且最终都逐渐趋向于 80% 多附近，增速趋向平稳，并且在同一种特征提取方法下，预测结果是同分布的，它们的 recall 和 precision 值都服从同一种分布，通过横向对比，这两种特征提取方式的预测结果的分布也大致类似，在大多数类别上，它们的预测表现都十分相似，总体上来说，TF-IDF 不同类别的预测表现较为均衡，而 Word2Vec 的预测表现则波动很大，在某些类别上表现很差。总体来说，如果关注 accuracy，那么 Word2Vec 显然更好，它在少量决策树的情况下就已经比 TF-IDF 高出很多，并一直表现得更优。但如果关注 precison，recall 或者 F1-score 的话，两者区别不是很大，在每个类别上各有优劣，总体来说表现相近。

5 结果分析

使用 hadoop 的 MapReduce 多种方法实现出来的准确率如下表所示：

5.1 朴素贝叶斯结果分析

使用了 Hadoop 中 MapReduce 自己实现了朴素贝叶斯和在 Spark 中调库实现的朴素贝叶斯进行比较，通过对比 20 个类各自的召回率（recall）、精确率（precision）和 F1 score，画出对应对比表格。

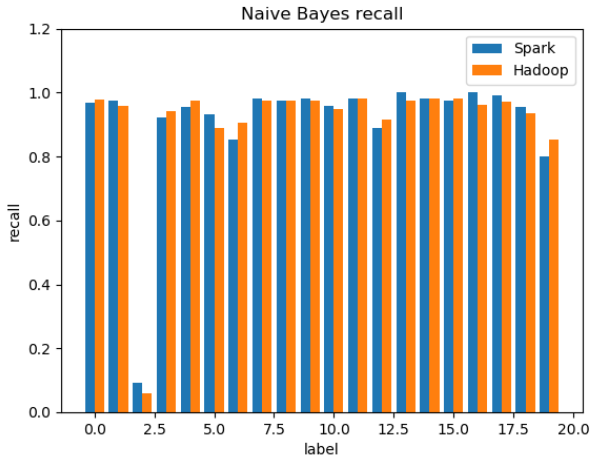


图 14: 两种方式下 20 个类的 Recall

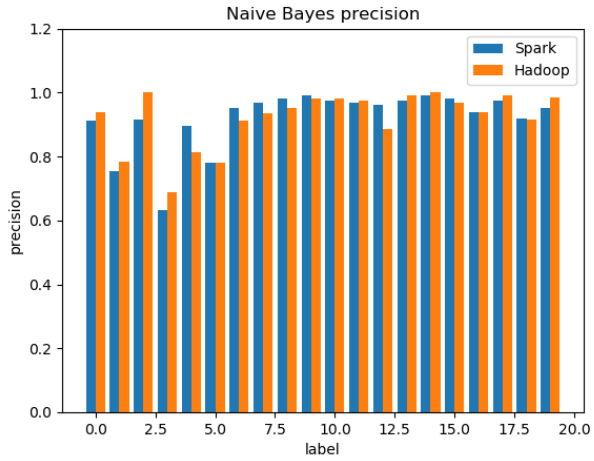


图 15: 两种方式下 20 个类的 Precision

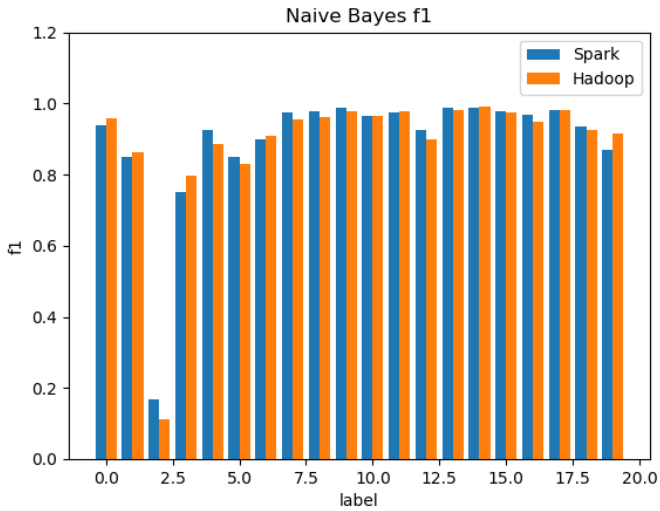


图 16: 两种方式下 20 个类的 F1 Score

分析实验结果，可以发现，总体上来说 Hadoop 下自己实现的朴素贝叶斯预测分类效果和 Spark 实现效果相近，在处理分类第二类问题上都做的不是很好，这与文本特性有关，但是在其他类中均能达到相似的较好的效果，这也侧面说明了自己实现方法基本没有问题。在准确率上，Hadoop MapReduce 的准确率在测试集上为 90.9894%，而 Spark 分类的准确率为 90.7244%，造成的这个差距的原因是在 FeatureHasher 时设置的参数为 10000，而实际单词数在 10000 以上，损失了部分词语。总体上看，横向对比其他方法，准确率有了极大的提升，说明朴素贝叶斯在这个任务上分类效果最

好。

6 参考文献

【1】Hadoop 分布式缓存
【2】TF-IDF: Wikipedia
【3】IDEA 将依赖的第三方 jar 包打入 jar 包
【4】Spark MLlib Document
【5】Spark 教程
【6】NaiveBayes 介绍