垃圾邮件甄别项目报告

一、基础原理

朴素贝叶斯介绍

这次项目我们主要是基于`naive bayes`方法进行分类实践。故先介绍朴素贝叶斯方法的原理，其主要基于贝叶斯公式。

$$P(c|d) = \frac{P(d|c)P(c)}{P(d)}$$

我们想要判断一个文档属于某个类，一个朴素的想法是计算这个文档属于每个类的概率，然后将概率最大的类判定为这个文档所属的类别。而计算一个文档`d`中出现某个`class`的概率可通过贝叶斯公式及先验概率：通过给定的样本，我们可以计算出$P(C)$， 即出现某个class的概率和$P(d|c)$, 即已知该文档属于某个class，它属于某个文档d的概率。

而在对文档d进行判别分类的时候，我们可以舍弃掉对P(d)的求解，因为不管该文档属于垃圾邮件还是正常邮件，P(d)并不改变。

$$C = argmax\{\frac{P(d|c)P(c)}{P(d)}\}$$

$$C = argmax\{P(d|c)P(c)\}$$

同时我们可以把文档d视为特征集合，在该情况下即把一篇邮件看作是许多个单词的集合。

$$C = argmax\{P(x_1, x_2 \dots x_n|c)P(c)\}$$

为简化该模型，我们可以借助独立性假设，即在已知该邮件所属类别之后，出现某个单词x_j的概率和出现x_i的概率相互独立。

$$C = argmax\{P(x_1|c)P(x_2|c) \dots P(x_n|c)P(c)\}$$

为了避免浮点数的溢出，我们可以通过取log运算得到以上式子。

$$C = argmax\{logP(c) + \sum_{i=1}^{n} logP(x_i|c)\}$$

而通过这个式子，我们可以把朴素贝叶斯算法拆分成三个部分：

1. 计算Spam邮件数目`NumberOfSpam`，Ham邮件数目`NumberOfHam`
2. 计算Spam邮件中出现的单词ω_i，以及每个单词ω_i出现次数。
3. 计算$P(c)$和$logP(x_i|c)$, 将结果取log相加，求出概率最大的类别。

项目大致思路

- 本地对 `train` 文件夹进行数据清洗，传入云端，map reduce 进行模型训练
- 本地加载训练后的模型，读取测试集，数据清洗，打印预测结果

二、数据清洗

- 基本思路：本次项目数据集质量良好，清洗步骤较为简单
  - 去除每篇邮件头的 `Subject:`
  - 利用正则去除文件中所有符号
  - 利用正则进行分词
  - 去除长度小于3的单词（视为噪声）
  - 利用 python nltk 包获取停用词列表，去除邮件中的停用词
  - 利用 python nltk, 进行词性还原（lemmatize）或词根还原（stem），这两种不同模型有不同表现，我们将在结果分析中阐述

```
from nltk.stem import WordNetLemmatizer
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
## 停用词列表
en_stopwords = stopwords.words("english")
## 词性还原
lemmatizer = WordNetLemmatizer()
## 词根还原
stemmer = PorterStemmer()
## 用于去除符号的正则表达式
pattern = re.compile(
    r"[\s+\.\!\/_,$%^*(+\"\')]+|[+——()?【】“”！，。？、~@#￥%……&*（）]+")

## 去除subject同时去除所有标点符号
file = re.sub(pattern, " ", file[8:].lower())
words_list = re.split(r"\s", file)
clean_list = [lemmatizer.lemmatize(word) for word in words_list if not
is_noise(word)]
clean_file = " ".join(clean_list)
## 写回文件，格式为 map reduce 接受的格式
with open(out_path, "a") as outfile:
    outfile.write(tag + " " + clean_file + "\n")
```

三、model training

正如原理介绍所言，模型训练算法主要由3个 job 实现。

先来看一下main函数。我们利用 `GenericOptionsParser` 将读取命令行参数，设置每一个Job的输入输出路径。

```
String[] otherArgs = (new GenericOptionsParser(conf,
args)).getRemainingArgs();
String inPath = otherArgs[0];
String job1OutPath = otherArgs[1];
String job2OutPath = otherArgs[2];
String countjobOutPath = otherArgs[3];
String job3OutPath = otherArgs[4];
```

依次运行 job 的函数，检测是否运行成功，并向用户报告。

```
if (Job1(inPath, job1OutPath) && Job2(inPath, job2OutPath) &&
CountJob(job2OutPath, countjobOutPath)
    && Job3(job1OutPath, job2OutPath, countjobOutPath, job3OutPath)) {
    System.out.println("## All jobs has finished!");
    System.exit(0);
} else {
    System.out.println("## Failed!");
    System.exit(1);
}
```

Job1计算邮件集的单词数

job1的 map 阶段：

```
输入value：类别 + 以空格分开的单词
示例： h good great ...
目的：通过空格将文档中的单词分开，计数整个文档的单词数量
输出key：单词的类别 h or s
输出value：单词数目
```

主要利用 `value.toString().split(" ")` 这个函数将传入的文本解析成单个的单词，输出该文本所有的单词数目。

核心代码：

```
public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {

    String[] fields = value.toString().split(" ");
    String Class = fields[0];
    int num_words = 0;
    num_words = fields.length - 1;

    // example:
    // key: h
    // value: 127
    context.write(new Text(Class), new IntWritable(num_words));

}
```

job1 的 reduce 阶段：

```
计数同一个类别的文档数目和单词总数
输出key：类别h or s
value：该类别的邮件数目 + 单词数 如：20 2000
```

主要利用 `for (IntWritable num : values)` for 循环来计数同一个 `key` 的每个单词数和邮件数目。

核心代码：

```
public void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {

    int class_num_words = 0;
    int class_num_emails = 0;

    for (IntWritable num : values) {
        class_num_words += num.get();
        class_num_emails++;
    }

    String result = Integer.toString(class_num_words) + " " +
Integer.toString(class_num_emails);
    // example:
    // key: h
    // value: 1324 34505
    context.write(key, new Text(result));
}
```

job2: 计算每个单词出现的邮件数目

job2 的 map 阶段：

```
输入value：文档（类别 + 以空格分开的单词）
判断输入邮件的类型，将spam和ham文件区分开
输出key：每一个单词
输出value：所属文件类型，第一个数字为1表明属于正常邮件，第二个数字为1表明属于垃圾邮件
示例：0 1 表明该邮件为垃圾邮件
```

主要利用`value.toString().split(" ")`函数将输出的文本解析成单词数组，计数每一个文本的`h_count s_count`，输出每个单词及对应的`h_count`和`s_count`.

核心代码：

```
public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
    String[] fields = value.toString().split(" ");
    int h_count = 0;
    int s_count = 0;
    if (fields[0].equals("h"))
        h_count++;
    else
        s_count++;

    // 从第一项开始
    for (int i = 1; i < fields.length; i++) {
        String output_value = Integer.toString(h_count) + " " +
Integer.toString(s_count);

        context.write(new Text(fields[i]), new Text(output_value));
    }
}
```

job2的reduce阶段：

```
将map输出的字符串“0 1”解析，对应数字相加即可得到该单词对应出现在ham和spam两个类中的数目
输出key：单词
输出value：对应出现的ham邮件数目 + spam邮件数目
```

核心代码：

```
public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
    int ham_count = 0;
    int spam_count = 0;

    for (Text value : values) {
```

```
            String line = value.toString();
            String[] count_in_class = line.split(" ");
            ham_count += Integer.parseInt(count_in_class[0]);
            spam_count += Integer.parseInt(count_in_class[1]);
        }
        String output_value = Integer.toString(ham_count) + " " +
Integer.toString(spam_count);

        context.write(key, new Text(output_value));

    }
```

CountJob

countjob实质是Job2的附属，接收Job2的输出，主要用来记录整个文件夹中不一样的单词总数。

```
int numberOfUniqueWords = 0;
        for (IntWritable value : values) {
            numberOfUniqueWords++;
    }
```

job3计算概率

Job3 的mapper有一个 `setup` 函数，主要用来计算邮件的先验概率 $logP(Class = Ham)$ 与 $logP(Class = Spam)$

```
Integer totalMails = numberOfHams + numberOfSpams;
Double logHamPrior = Math.log((double) (numberOfHams) / (double)
(totalMails));
Double logSpamPrior = Math.log((double) (numberOfSpams) / (double)
(totalMails));
```

job3的 map 阶段：

主要通过前面统计的结果计算对每个单词w_i来说的 $log(P(w_i|C = Ham))$ 和 $log(P(w_i|C = Spam))$

```
public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
    StringTokenizer valueTokenizer = new StringTokenizer(value.toString());
    // The while loop is supposed to loop exactly 3 times
    // The first time to parse the count in ham, the second to parse the
count in
    int countInHam = 0, countInSpam = 0;
    int iterationCounter = 0;
    String[] splittedStrings = value.toString().split("\\s+");
    if (splittedStrings.length != 3) {
```

```
            throw new IOException("There should be two numbers in value field,
but more are provided.");
        }

        // parse some parameters
        String word = splittedStrings[0];
        countInHam = Integer.valueOf(splittedStrings[1]);
        countInSpam = Integer.valueOf(splittedStrings[2]);
        Integer totalItems = numberOfUniqueWords;
        Double logHamProbability = Math.log((double) (countInHam + 1) /
(double) (numberOfWordsInHam + totalItems));
        Double logSpamProbability = Math
            .log((double) (countInSpam + 1) / (double) (numberOfWordsInSpam +
totalItems));
        Text text = new Text(String.valueOf(logHamProbability) + " " +
String.valueOf(logSpamProbability));
        context.write(new Text(word), text);
    }
```

job3 的 reduce 阶段直接接受 map 的 key, value 并输出，最终输出文件即为训练好的 model。

模型格式

通过之前的训练，我们得到了在此邮件数据集上的 Naive Bayes 模型，我们使用如下的格式进行输出：

```
    -0.707216549518245 -0.6792730155238255 131567 2229642 1797187
aa  -13.576072045294508 -13.373772467871296
aaa -11.902095611722835 -11.10508892655293
aaaa    -14.674684333962617 -13.373772467871296
aaaacy  -14.674684333962617 -13.77923757597946
```

第一行有五个数字，分别是 $logP(C = Ham)$ 和 $logP(C = Spam)$， UniqueCount (出现的互异的单词总数)、h_count（出现在Ham邮件里的所有单词总数） 、s_count（出现在Spam邮件里所有单词的总数）。

从第二行开始分别是单词，对应的 $P(w_i|C = Ham)$ 和 $P(w_i|C = Spam)$

四、模型优化

增大subject权重

- 基础思路：在给定数据集中，每一封邮件都有自己的主题，主题词也许更能体现一篇邮件的特性，于是我们将主题词作为权重更大的 feature, 给予这些词汇更大的权重，观察能否在 test 集上获得更好的效果
- 实现思路：在清洗数据集过程中，让 subject 中的单词重复出现 k 次，观察不同k对结果的影响
- 结果：非常遗憾，在 k = [2, 20] 的区间内，没有一个k值可以让结果变好。

stemming / lemmatizing

在进行数据清洗的过程中，我们尝试了两种去词根的方法，一是 `lemmatize` (词性还原)，`stem` (词根还原)

- stem

  stem 比较简单粗暴，在我们用 nltk, 进行 stemming 之后，发现很多词被简单粗暴地还原为一个词根，例子如下

  ```
  produces produc
  produced produc
  productive product
  producing produc
  productivity product
  ```

- lemmatize

  lemmatize 的结果更加合理，只进行基本的词性还原

  ```
  produces produce
  produced produced
  productive productive
  producing producing
  productivity productivity
  ```

  通过 `nltk.pos_tag()` 可以达到更加准确的结果

  ```
  produces produce
  produced produce
  productive productive
  producing produce
  productivity productivity
  ```

- 从结果来看，lemmatize 方法要比 stemming 高一点点，最终我们还是选取了进行 lemmatize 的模型

AdaBoost

Adaptive Boost 是一种较为常用的机器学习方法，它能够对训练数据集进行自适应，具体来说，Adaptive Boost 能够将之前的分类器错误分类的样本用来训练下一个分类器，经验表明，Adaptive Boost 方法不会太容易出现过拟合的情况，并且其附加的分类器即便效果并不显著，也能够起到提高模型表现的效果。在关于 Adaptive Boost 的尝试中，我们参考了 Qing Liu 的解决方案，并针对此项目的细节进行了一些优化。虽然我们在 Adaptive Boost 上的尝试并没有能为我们带来更高的识别准确率，但我们依然认为此次尝试是值得记录的。

从具体问题来说，Adaptive Boost 通过如下实际意义下的方法来解决垃圾邮件分类的问题：通过 Naive Bayes 方法获得的相关后验概率仅仅形容了相应的词汇在其所属类别中出现的数量，但在预测过程中每个单词都拥有相同的权重，仅仅通过其出现次数进行相应的预测；而 Adaptive Boost 通过学习原始模型在训练数据集上预测错误的邮件，赋予部分单词不同的权重，以此对邮件进行更深一步的区分。为了实现 Adaptive Boost，我们需要实现以下几个步骤：

- 建立一个长度等于 Naive Bayes 模型行数的列表并初始化为全 1 作为权重向量 w
- 在训练数据集上运行模型，获取预测错误的文件序号以及错误时样本在两个类别中的后验概率之差 α
- 对 w 进行调整：对于预测错误的文件中的每个单词，依据错误的种类 b （spam 误当成 ham / ham 误当成 spam）调整 w 中的相应项 $w_i = w_i + f(\alpha, b)$，f 作为用于逐步调整权重的函数有待更进一步讨论
- 在预测过程中，对于 spam 的判断需要乘以相应单词的权重
- 重复之前步骤给定次数，或者直到错误率满足要求

为了将错误种类 b 进行数值表示，我们引入示性函数 δ_b ：

$$\delta_b = \begin{cases} 1 & mispredict\ ham\ as\ spam \\ -1 & mispredict\ spam\ as\ ham \end{cases}$$

在此模型上，我们的调整主要在对于 f 的选取以及迭代次数的选取上。我们尝试了一些做法，但都没能取得比纯粹的 Naive Bayes 方法更高的准确率，以下我们所进行的一些尝试：

指数权重调整

我们的首次尝试遵循了我们的参考内容，使用如下的方式来对权重进行调整：

$$f(\alpha, b) = \frac{\delta_b \cdot e^{\alpha}}{w_i}$$

然而，通过观察错误样本在两个类别中的后验概率之差 α ，我们发现 α 的取值的绝对值可以高达三个数量级，这个数量级的数字出现在指数位上通常会造成灾难，而实验结果也验证了这一点：在计算 e^{α} 时出现了 overflow，相关的准确率跌到了 70% 以下，因此我们没有在此方法上进行更多的测试。

常量权重调整

在此部分中，我们尝试一种极为简单，忽略掉 α 的做法：选定迭代次数 `iteration` 并给定步长 `step` ，将 f 设置为如下形式：

$$f(\alpha, b) = \delta_b \cdot step$$

在整个迭代过程中，我们使用 Python 的 `min` 和 `max` 方法来在每轮迭代的末尾显示迭代中权重的最大值和最小值。我们测试了 `iteration` 为 3、5、10 的情况，各种情况下的 F 值均在 85% 左右，且更多的迭代次数并不会带来显著的改变，只是略微的提高相应的准确度。

Sigmoid 权重调整

由于在之前的尝试中都发现了最终的权重向量中出现极端值（可能会高达五个数量级）的情况，我们希望使用 sigmoid 函数对权重进行合理的分配，让其落在一个合理的区间上，同时，我们也希望引入 α 作为一个参照，因此，我们使用如下的方式进行权重更新：

$$f(\alpha, b) = \delta_b \cdot \alpha$$

同时，在生成一封邮件在垃圾邮件类别上的后验概率过程中，我们将计算出的原始概率乘以 $f(\alpha, b)$ 的 sigmoid 函数值，而不是乘以 $f(\alpha, b)$ 本身，即：

$$E\ mail = p \cdot (\frac{2c}{1 + e^{-f(\alpha, b)}} + c)$$

其中的 c 用于表示所映射的 $f(\alpha, b)$ 的区间范围，在实验中我们根据结果对 c 进行适应性的调整，选取的取值包括 10、40、50、80。使用此方法获得的结果明显好于之前的两种方法，准确率已经接近使用单纯的 Naive Bayes 方法获得的结果，甚至在 Spam Precision 上有所超越，达到了 99.25%，但是，此方法使 Ham Precision 较低 （大约 96%），并且随着迭代次数的增加，两个值有接近的趋势，总体来说，此方法已然不如 Naive Bayes 方法。

总的来说，我们在 Adaptive Boost 上的尝试在本次试验中并没有令模型取得决定性的提升，但由于我们尝试方法的有限，我们依然不排除 Adaptive Boost 能够作为 Naive Bayes 模型的一个重要优化方向的可能。

Feature Selection

在本部分中，我们尝试了一种根据 Adaptive Boost 的思想衍生出的 Feature Selection 方法，经过一定的调整，取得了在测试数据集上 20% 的错误率下降。我们的 Feature Selection 同样是通过回顾模型在训练数据集上发生的错误而调整参数，并建立在以下两个假设上：

- 整个数据集清洗过后的单词包含三类：典型的非垃圾邮件词汇（例如 seminar）、典型的垃圾邮件词汇（例如 credit）以及中性词汇（例如 christmas）
- 对于某些中性词汇，可能会产生其单词数量在两个类别上的偏斜，例如 christmas 在垃圾邮件中出现较多，但这种偏斜并不能说明相应邮件所属的类别

在确定了以上的两个假设之后，我们给出了降低预测错误率的思路：对于之前预测错误的邮件中大量出现的中性词汇予以去除，即在预测中忽略这些词汇的存在，转而使用 Laplace Smooth 替代。

依据我们在训练集上运行测试的结果，并对结果进行词汇量统计，我们给出了在预测错误的邮件中出现的单词，并按照其出现次数进行降序排名，以下是部分结果：

```
com 346
please 171
http 130
service 114
www 112
free 109
information 109
get 109
time 101
register 99
```

在首次测试中，我们选取了此列表的前 100 词，并对之后的模型进行修改：对于每封测试邮件中的单词，如果其属于我们给出的词汇列表，则将其剔除而使用 Laplace Smooth 进行替代。这种方法使模型的准确率有了可见的提升。之后的实验中，我们主要针对选取的剔除词的数量进行调整，以及尝试从上方列出的列表中去除拥有明显的非中性特征的词汇（例如 seminar 或 credit），最终，我们在选取 75 词，并进行非中性词汇去除的条件下获得了目前在测试数据集上最高的准确率：99 %。

加大数据量

模型再好耐不住数据多，在这个模型中我们加入了前三次作业利用的邮件数据集，简单粗暴地加大了数据量，这个模型虽然在 `test` 集上表现不好，但是在最终测试集上有碾压级的表现。

五、成果展示

在对垃圾邮件进行分类的时候，人们往往无法容忍正常邮件被认定为垃圾邮件，但对少量垃圾邮件成为漏网之鱼却可以忍受。所以往往人们更看重这几个指标，一个是准确率（precision），一个是召回率（recall），但我们可以看到这两个指标并不能同时达到最优，所以还有一个f1指标，综合了准确率和召回率，是二者的调和平均。

准确率

准确率在分类问题中指的是在被预测为正的样本中，实际为正样本所占比例。在垃圾邮件分类问题中，可以认为是预测为垃圾邮件的样本中，实际为垃圾邮件的比例。较高的准确率意味着对垃圾邮件的误报较少。

$$Percision = \frac{\text{垃圾邮件数目}}{\text{被预测为垃圾邮件的数目}}$$

召回率

recall在分类问题中指的是在所有的正样本里，预测为正的样本数所占比例。在垃圾邮件分类问题中，我们可以认为是垃圾邮件中预测正确的比例。较高的召回率说明垃圾邮件出现在正常邮件中的情况较少。

$$recall = \frac{\text{预测为垃圾邮件的数目}}{\text{垃圾邮件的总数目}}$$

F1指标

$$F = \frac{(\beta^2 + 1) * precision * recall}{\beta^2 * precision + recall}$$

不妨取$\beta = 1$ 则 $F = \frac{2 * precision * recall}{precision + recall}$

- Model 1
  - train 上 lemmatize 之后训练的模型

```
Summary:
ham mispredicted as spam: 29
spam mispredicted as ham: 23
ham precision:  0.9885          ham recall:     0.9855
spam precision: 0.9855          spam recall:    0.9885
ham_f:  0.987
spam_f: 0.987
Running time: 0.5226 seconds.
```

- Model 2
  - train 上 lemmatize 并进行 feature selection

```
Summary:
ham mispredicted as spam: 24
spam mispredicted as ham: 21
ham precision:  0.9895          ham recall:     0.988
spam precision: 0.988           spam recall:    0.9895
ham_f:  0.9887
spam_f: 0.9888
Running time: 0.4902 seconds.
```

- Model 3
  - 加大数据量的模型

```
Summary:
ham mispredicted as spam: 20
spam mispredicted as ham: 136
ham precision:  0.9357          ham recall:     0.99
spam precision: 0.9894          spam recall:    0.932
ham_f:  0.9621
spam_f: 0.9598
Running time: 0.5034 seconds.
```

- Model 4
  - train 上进行 stemming

```
Summary:
ham mispredicted as spam: 30
spam mispredicted as ham: 25
ham precision:  0.9875          ham recall:     0.985
spam precision: 0.985           spam recall:    0.9875
ham_f:  0.9862
spam_f: 0.9863
Running time: 0.4673 seconds.
 Yee@livifeideMacBook-Pro  ~/Downloads/Bayes_Predictor  master ●
```