朴素贝叶斯分类器实现垃圾邮件分类

1.实验环境

- OS: Windows 10
- CPU: intel CORE i5
- Python 3.6
- Sckikit_learn 0.18.1
- 其他 Python 科学计算模块
- 编译器：PythonCharm 2017.3.3

2.主要参考文档：

Sklearn 0.18 官方文档

3.实验过程：

3.1 实验问题描述

朴素贝叶斯分类器是一个较为简单的监督学习方法，在文本分类方面应用较为广泛。文本分类往往是一种标称型的二分类问题，基于朴素贝叶斯分类器的原理，虽然其方法简单，但是对这种二分类问题往往可以取得非常不错的效果。本实验主要解决对于垃圾邮件的分类，考虑停用词汇，生成统计词典，对已经做了标记的邮件进行预测分类，并且得到分类的正确率。

3.2 实验资源库

通过查阅资料，本实验的语料库使用了滑铁卢大学的“2006 年 TREC 中国公共语料库”中文邮件数据集，共有 60000 封已经做好标注的中文邮件，选择了其中的 30000 封，其中，29400 封用作训练集，600 封作为测试集。停用词汇列表附带在文件中。

语料库网址：https://plg.uwaterloo.ca/cgi-bin/cgiwrap/gvcormac/foo06

3.3.实验原理与过程：

这个实验参考了一些互联网上的博客，github 等，并在充分理解的基础上做

出了自己的修改和补充。

首先是朴素贝叶斯原理在本实验中的运用：

对于邮件文本$d=\{w1,w2,\cdots,wn\}$的邮件的属性属于$C=\{C_p \ ,C_n\}$，在特征相互独立的情况下，考虑特征词的权值，其分类算法如下：

$$C_{NB}=\underset{C_j\epsilon C}{argmax}\ \{P(C_j)\prod_{i=1}^{n}P(w_i,c_j)^{wt(wi)}\}$$

其中： P(Cj)是类别 cj 的先验概率，P（wi， cj）是特征词 wi 在类别中的后验概率；wt(wi)是测试语料库中词 wi 的权值，对于先验概率 P(cj)，可以视每类是相同的，也可以是预先估计，本实验根据已经正确标注的训练样本预先估计，估计算法如下：

$$P(C_j)=\frac{doc(C_j)}{\sum_{c_j\epsilon}c\ doc(C_j)}$$

其中：doc(C_j)是属于类别 Cj 的文本数。

对于后验概率 P(wi， cj)，即特征词 wi 出现在类别 cj 中的概率，一般是从训练语料中通过计算进行估计。本实验采用 wi 在属于类别 cj 的文本中的权值之和除以类别 cj 的文本中所有词的权值之和。为了避免 P(wi, cj)等于 0，可以采用 Laplace z 转换，由此得出的概率计算方法如下：

$$P(w_{i,}c_j)=\frac{weight(w_{i,}c_j)+\delta}{\sum_{i=1}^{n}weight(w_{i,}c_j)+\delta|V|}$$

$$(V=\sum_{C_j\in C}\sum_{i=1}^{n}weight(w_{i,}c_j),\delta=\left|\frac{1}{V}\right| \ )$$

由上述公式，只要统计 doc(cj)和 weight(wi, cj)就可以计算出先验和后验概率， 因此对于训练模型的存储问题也就简化为了保存这两部分；而其中的 doc(cj)， 对于训练样本而言有所要求——训练样本已经是分类好的，实验采用的语料库对于这些都是满足的。

下面结合具体的代码说明，实验的过程大致如下：

首先需要从邮件路径中得到文本内容，由于从语料库获取的邮件是 gbk 编码，有一些是无法解码的，所以，应该忽略这一部分内容。

```
1.  def get_content(path):
2.
3.
4.      with open(path, 'r', encoding='gbk', errors='ignore') as f:
5.          lines = f.readlines()
6.      for i in range(len(lines)):
7.          if lines[i] == '\n':
8.              # 去除空行
9.              lines = lines[i:]
10.             break
```

```
    content = ''.join(''.join(lines).strip().split())
    return content
```

接下来是生成邮件的字典。我们使用 jieba 库来分词，把邮件内容分成一个一个离散的词汇，并且要除去其中的停用词汇，生成一个词典记录所有的词汇。然后统计所有的词汇在正常邮件与垃圾邮件中出现的次数。
生成邮件词典的代码如下：

```
def create_string_word_dict(string, stop_list):

    string = re.findall(u'[\u4E00-\u9FD5]', string)
    string = ''.join(string)

    word_list = []

    # 结巴分词
    seg_list = jieba.cut(string)
    for word in seg_list:
        if word != '' and word not in stop_list:
            word_list.append(word)
    word_dict = dict([(word, 1) for word in word_list])
    return word_dict
```

为了得到分类器，需要统计不同的词汇在垃圾邮件与正常邮件中出现的次数，以此与不同类邮件的总数相除，得到词汇出现的概率。由训练样本得到词典，得到的字典中记录了以上数据，并且最后需要返回垃圾邮件总数，正常邮件总数以及垃圾邮件总数。
代码如下：

```
def train(df):

    train_word_dict = {}
    for word_dict, spam in zip(df.word_dict, df.spam):
        # 统计测试邮件中词语在垃圾邮件和正常邮件中出现的总数
        for w in word_dict:
            train_word_dict.setdefault(w, {0: 0, 1: 0})
            train_word_dict[w][spam] += 1
    ham_count = df.spam.value_counts()[0]
    spam_count = df.spam.value_counts()[1]
    return train_word_dict, spam_count, ham_count
```

最后是在输入测试集的邮件之后，通过与训练得到的字典进行比较，得到结果。在具体测试的过程中，只输入待测试邮件的一行数据测试即可，根据的式子如下：

$$C_{NB} = \underset{C_j \epsilon C}{argmax} \{\mathrm{P}(C_j) \prod_{i=1}^{n} P(w_i, c_j)^{wt(wi)}\}$$

为了避免在等式的右边出现直接相乘，将所有的概率都改为对数的形式，之所以可以改为对数，是因为最后只要统计正常邮件和垃圾邮件两种情况下的大小比较，并不需要直接得到大小。使用对数表示，则每次只需要相加就可以得到最后相乘的结果，即所谓的对数似然函数

值得注意的是，这里为了与实际结果更为接近，减小误差，防止某次计算的时候出现概率值为 0 的情况，还需要在每次相加的时候做拉普拉斯平滑处理：平滑处理的方法是在分子的词汇出现次数加 1，而在分母的分类的邮件总数加 2。具体的这部分代码如下：

```
def predict(train_word_dict, spam_count, ham_count, row):
    """
    预测函数
    :param train_word_dict: 训练函数产生的统计词典
    :param spam_count: 垃圾邮件数
    :param ham_count: 正常邮件数
    :param row: 一行测试邮件数据
    """
    total_count = ham_count + spam_count
    word_dict = row['word_dict']

    # 正常邮件概率
    hp = math.log(float(ham_count) / total_count)
    # 垃圾邮件概率
    sp = math.log(float(spam_count) / total_count)

    for w in word_dict:
        w = w.strip()
        # 给该词在词典中设定一个默认数 0
        train_word_dict.setdefault(w, {0: 0, 1: 0})

        # 该词在词典中正常邮件出现的次数
        pih = train_word_dict[w][0]
        # 平滑处理，每个词汇基数+1，正常邮件数+2
        hp += math.log((float(pih) + 1) / (ham_count + 2))

        pis = train_word_dict[w][1]
        sp += math.log((float(pis) + 1) / (spam_count + 2))
    # 预测结果
    predict_spam = 1 if sp > hp else 0
    # 返回预测是否准确，用于统计
    return 1 if predict_spam == row['spam'] else 0
```

与邮件的标签进行比较，如果预测正确，则返回 1，否则，就返回 0。
整个步骤可以总结如下：

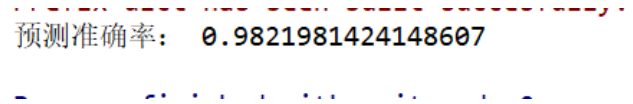
初始化。构造可以表征文本的特征向量（字典），根据特征向量把训练集表征出来。从训练集中分离出部分数据作为测试集，本实验采用了 98%样本为训练集，2%样本为测试集。

```
1.  # 产生训练集与测试集
2.  train_mails = df.loc[:len(df) * 0.98]
3.  test_mails = df.loc[len(df) * 0.98:]
```

学习。计算类的先验概率和特征向量（字典中的词汇）对应每一类的条件概率向量。

分类（参数估计）。计算测试集中待分类邮件在每一类的分类概率，取最大值作为其分类，并与给定标签比较，得到分类的正确率。

3.4.实验结果与分析：



最后，得到的结果是，预测的准确率为 98.23%，效果较为理想，这样的结果是让我非常惊讶的，出乎意料。

总结来说，朴素贝叶斯分类器在实际运行存在一类问题，特征的个数远大于训练集的个数或者与训练集个数相当，则容易出现过拟合现象。我们的目标是对邮件进行分类，将邮件中的除了停用词汇以外的词汇建立了字典，这样，每封邮件都会有一个字典，有大量的特征出现，如果训练样本不够大，就会出现过拟合。但是朴素贝叶斯提出了简单的处理方法，认为给定的文档分类标号的情况下，词的出现是相互独立的，假设一个文档被标记为正常邮件，那么分类与回归被认为是相互独立的。虽然这样的假设看上去并不合理，但是在显示中的效果很好，因为它不是完全假设任意两个词的出现都是独立的，独立的前提条件是文档的类别已知，实际上可以证明，条件独立的两个变量在条件未的情况下并不一定独立，因此朴素贝叶斯算法仍然可以通过类别的先验概率体现两个变量之间的关系。

朴素贝叶斯的成功之处在于通过简单的条件“类别“，使得原本不独立的变量近似的认为是独立的，大大地减小了模型的参数。