数据挖掘实验：垃圾邮件检测

查鹏 131220044　Email: zhapenGL@163.com　Tel: 15895987843

(南京大学，计算机科学与技术系，210046)

摘　要: 在学习了很多数据挖掘算法后，我们需要应用到现实生活中。这次实验中，给了我们一个原始邮件数据集，我们需要对这个邮件进行特征提取，然后建立数据挖掘模型，使用这个模型对给的测试样本进行测试，统计出测试样本中哪些是好的邮件，哪些是垃圾邮件。这个实验难就难在特征提取和模型构建，数据挖掘的难点就在于特征提取和数据挖掘算法的设计。在我的设计中，我使用了已有的朴素贝叶斯分类算法，并没有自己设计算法，然后根据不停的调整先验概率使我的模型达到了很高的性能。在这个实验报告中，在实验设计环节我描述了我为什么选择贝叶斯，我怎么设计我的算法和程序结构；在算法比较环节我通过比较朴素贝叶斯和其他分类算法，说明了我为什么会选择朴素贝叶斯，最后说明了我的实验结果和使用结论。

关键词: 数据挖掘；垃圾邮件分类；朴素贝叶斯

1　实验任务介绍

1.1　实验内容

这次实验是一个基于现实世界的数据挖掘实验设计，给定一个初始的实验数据，需要我们对这个数据进行挖掘。实验中给了 8000 多份好邮件和 2000 多份垃圾邮件，需要我们对这 10000 多邮件进行数据挖掘，训练出垃圾邮件检测的模型。然后实验会给 15000 份左右的邮件，我们需要使用我们训练出来的模型对这 15000 份邮件进行判断，判断哪些是好的，哪些是坏的。

1.2　实验重难点

1.2.1　数据是原始的

实验给的数据都是邮件的原始数据，包括收件人，发件人，邮件内容等等。这就意味着需要我们自己对数据进行预处理，提取邮件的特征值，来供我们的数据挖掘算法。

1.2.2　邮件分布不均匀

在现实生活中，我们收到的邮件大多数是正常邮件，垃圾邮件只是少数。垃圾邮件判断的时候可能会因为垃圾邮件少，正常邮件多，然后产生以偏概全的情况。我们使用数据也会有这种不均匀的情况出现，因此这也是难点之一。

综上所述，我们进行这个数据挖掘实验时候，首先要进行邮件特征提取，然后要消除过于“明显”的特征，比如一个特征在垃圾邮件中出现的频率很高，但是正常邮件也会出现这个特征，我们很有可能因此把这个正常邮件判断成垃圾邮件。这样我们需要设置适当的估价函数，把这个特征的估价值调低一点，这样能增加预测的准确率。同样，如果有的特征不足以作为判定正常邮件和垃圾邮件的特征，这样我们就不需要提取这个特征。

举个例子。假设我是用单词出现频率来判断一个邮件是否是垃圾邮件的。对于 a,an,the,and 这样的单词，无论在垃圾邮件还是正常邮件都有很高的出现频率。这种单词是不足以作为特征的。也有其他情况：假设 advertisement 这个词在垃圾邮件中出现频率很高，当一封正常邮件中出现这个词时，我们很可能把这个邮件判断为垃圾邮件屏蔽了，这样明显是对用户不友好的。假设我们给这个词设置不同的系数，在计算这份邮件

是否为 spam 的概率的时候把 adertisement 的频率加个系数 0.9，然后在计算这份邮件是否为 ham 的时候把 advertisement 的频率加个系数 0.1，这样就能减少因为那些太过于“明显”的特征导致的预测失误的几率。

2 实验设计

2.1 特征提取

邮件的特征我使用的方式是使用单词出现频率来计算的。我使用所有的标点符号和空格回车来对每一份邮件内容进行划分，得到邮件的所有单词。这样划分把类似 xxxxx@163.com 这样的发件人手机人也算作一个单词。

```
String line = new String(readBytes);
String[] array = line.split("[\r\n" +
        "|, |@|#|$|%|^{|}|&|\\. |; |: |' |! |? |+ |* |/|\n" +
        "|\r]");
```

然后把类似 a,an 这样的单词去除，因为这种词语不具有判断的说服力。然后使用 Map<words,comunt>存放邮件单词的出现次数，其中用单词作为 key，用单词出现次数作为 value。具体存储方法：如果 Map 中已经有这个词了，则对应的 value 加 1，如果 Map 中没有这个词，判断这个词的长度，如果单词长度大于 2，则加入 MAP 中。

```
for (int i = 0; i < array.length; i++) {
    if (negtive.containsKey(array[i])) {
        int v = negtive.get(array[i]);
        v++;
        negtive.remove(array[i]);
        negtive.put(array[i], v);
    } else {
        if(array[i].length()>2)
            negtive.put(array[i], 1);
    }
```

注意不同邮件特征存放于不同的数据结构中。测试文件存在测试文件 Test 中，ham 的所有单词及其出现次数存放于 postive 中，spam 的所有单词及其出现次数存放于 negtive 中。

```
Map<String,Set<String>>Test=new HashMap<>();
Map<String,Integer> postive=new HashMap<>();
Map<String,Integer>negtive=new HashMap<>();
```

各个 Map 的含义如下：

Test 的 key 为文件名，value 为文件的所有出现的单词；

Postive 的 key 为单词，value 为单词在所有 ham 中出现总次数；

Negtive 的 key 为单词，value 为单词在所有的 spam 中出现的总次数；

2.2 算法选择

2.2.1 算法选择

算法我使用了朴素贝叶斯。

2.2.2 原因

我进行算法选择的时候考虑三个要素：时间空间复杂度，特征提取的难度，算法稳定性。

相比于其他的算法而言，贝叶斯和决策树是比较实用的方法。尤其是特征提取比较简单，完整。其他算法为什么不适用，我说出我的理解：

1) KNN：KNN 是基于距离的分类方法，也就是说我们需要对邮件进行特征提取，然后对特征值进行

离散化，没输入一封待检测邮件，我们都要计算距离最近的 K 的邮件然后进行比较。距离一般使用欧氏距离，我们可以对 K 个距离的贡献进行加权，获得更高的精确率。但是又两个问题：第一就是特征提取和离散化的困难；第二就是，KNN 算法没有训练过程，利用其惰性机制，我们每次判断了一封邮件，便可以把这份邮件加入训练样本中。基于之前的数据挖掘实验，KNN 算法的准确率应该是很高的。但是 KNN 分类算法局限性，还有其巨大的时间复杂度，不适合这个算法。

2) K-Means：K-means 是聚类方法，我们同样需要对特征进行离散化，然后随机获取 K 个簇心，根据每个样本与簇心的距离将样本加入簇心然后更新簇心。算法是时间复杂度还是很可观的，但是有很多问题：第一，K 只能设置为 2，K 太小有时候会造成一些离群点太敏感，导致结果不稳定；第二，K-means 的局部最优不代表全局最优。这个和出现点选取有关，不同初始点会导致不同的结果，所以不稳定。第三，特征选取和离散难。我不选择这个算法。

3) 神经网络：第一，神经网络计算量大，时间复杂度太高；第二，很可能会因为我在特征提取上出现问题，然后就导致 BP 算法的局部搜索陷入极值，然后训练失败的情况。所以不选。

4) 其他类似支持向量机这样的分类算法，由于我对邮件的特征提取不是很擅长，所以就不选择。

综合这些比较，我最后选择了 apriori 算法，决策树算法和朴素贝叶斯算法。因为这些算法特征提取可以使用单词来实现，这样的话就会使特征提取的更加完整，而且算法操作起来也很简单。但是 apriori 和决策树的内存占用率太高空间复杂度不够好，所以我选择了最简单的朴素贝叶斯。
具体的算法比较我写在了后面的“算法比较中”。

2.3 算法介绍

2.3.1 贝叶斯算法

1. 假设 D 是训练元祖和它们相关联的类标号的集合。通常每个元祖用一个 n 唯属性向量 X={x1,x2,x3…xn}表示，描述 n 个属性 A1,A2,A3…An 对元素的 n 个测量。
2. 假设有 m 个类 C1，C2，C3…Cm。给定元祖 X 属于 Ci 的条件是：

对于任意类 $C_j, i < j < m$ 且 $i \neq j$，都满足：

$$P(C_i \mid X) > P(C_j \mid X)。$$

根据贝叶斯定理：

$$P(C_i \mid X) = \frac{P(X \mid C_i)P(C_i)}{P(X)};$$

由于 P（X）是常数，我们只需要保证分子最大就好了。

3. 给定具有许多属性的数据集，计算 P（X|Ci）的代价很大。为了避免过度的开销，我们可以做类条件独立假设。给定元祖的编号，假定属性值又条件的相互独立，也就是属性直接不存在依赖关系。我们可以使用下面的公式：

$$P(X \mid C_i) = \prod_{k=1}^{n} p(x_k \mid C_i);$$

4. 通过上述的步骤，我们就得到了 p(x|c)，再乘以先验概率，得到总概率。我们比较总概率的大小来预测给的 X 属于哪一个类。

2.3.2 我的贝叶斯改进

1. 拉普拉斯修正

在进行单词出现概率计算的时候，如果某个单词的出现概率为 0，那么即使这个邮件的其他单词在其他地方的出现概率再高，最后总的概率还是为 0，这样就没意义了。使用方式是拉普拉斯修正。

假设某个类 ham 包含 2000 个元组，当某个单词出现次数为 0 时，我们把这个单词的概率计算为 1/(2000+2)。这样的话，这种“校准的”概率和与之对应的未校准的概率估计很接近，但是又不是 0。

2. 概率相乘变相加

即使使用了拉普拉斯修正。如果出现修正后概率为 1/100000 这样的大小，程序计算时候还是会当做 0 来进行乘法计算，也就是结果还是 0。所以我使用了另外一种方式：

概率相乘等于对应概率的 log 然后相加

```
double pos = 0.0, neg = 0.0;
for (double d1 : P1)
    pos = pos + Math.log(d1);
```

这样子就不怕出现概率为 0 了。因为是求对数相加，所以结果更加的精确。我在使用概率相乘方法的时候准确率 0.96，改为取对数相加后就变成了 0.99 以上。可以看出这个方法能提高程序的准确率。

2.4 算法应用

2.4.1 数据处理和特征提取

我用下面的格式存储数据：

```
Map<String,Set<String>>Test=new HashMap<>();
Map<String,Integer> postive=new HashMap<>();
Map<String,Integer>negtive=new HashMap<>();
```

其中，Test 存放的是测试样本，KEY 为文件名，value 为样本中出现的单词集合；

Postive 存放的是 ham 中出现的所有词汇，其中 KEY 为单词，value 为出现次数；

Negtive 存放的是 spam 中出现的所有词汇，其中 KEY 为单词，value 为出现次数；

① 将文件夹下的所有 txt 存放到链表中

```
File f1 = new File("D:\\data\\data\\train\\ham");
File f2 = new File("D:\\data\\data\\train\\spam");
File f3 = new File("D:\\data\\data\\test");
List<String> l1 = new ArrayList<>();
List<String> l2 = new ArrayList<>();
List<String> l3 = new ArrayList<>();
l1 = get_file_list(f1);
l2 = get_file_list(f2);
l3 = get_file_list(f3);
```

② 遍历链表中的 txt 文件，对文件内容按照所有标点符号和空格进行划分，得到一个 txt 的所有单词。过滤掉 a,an 这样的单词，然后将单词存放到 map 中。如果当前 txt 是 ham，则单词存入 postive 中，如果 txt 是 spam，则存入 negtive 中。

```
String line = new String(readBytes);
String[] array = line.split("[\r\n|, |@|#|$|{|}|%|^|\\. |; |: |' |! |? |+ |* |/|\r]");//获取txt中的数据

for (int i = 0; i < array.length; i++) {
    if (postive.containsKey(array[i])) {//出现过这个词了，这个词计数+1
        int v = postive.get(array[i]);
        v++;
        postive.remove(array[i]);
        postive.put(array[i], v);
    } else {//没有出现过，词计数减1
        if(array[i].length()>2)
            postive.put(array[i], 1);
    }
```

```
String line = new String(readBytes);
String[] array = line.split("[\r\n" +
        "|, |@|#|$|%|^{|}|&|\\. |; |: |' |! |? |+ |* |/|\n" +
        "|\r]");
for (int i = 0; i < array.length; i++) {
    if (negtive.containsKey(array[i])) {//已经存在这个词，词计数+1
        int v = negtive.get(array[i]);
        v++;
        negtive.remove(array[i]);
        negtive.put(array[i], v);
    } else {//这个词没出现过，加入这个词，计数设为1
        if(array[i].length()>2)
            negtive.put(array[i], 1);
    }
```

上面代码的意思是：统计给的训练样本中 ham 样本里出现的所有词及其出现次数，存入名为 postive 的 MAP 中，再统计给的训练样本中 spam 样本里出现的所有词汇及其出现次数，存入名为 negtive 的 MAP 中。将 ham 和 spam 的出现词汇分开存储，可以为模型训练时候提供方便。

③ 遍历测试文件，将测试文件存入到 Test 中。其中 Test 的 key 是文件名，value 是文件的单词集合（不重复）

```
String line = new String(readBytes);
String[] array = line.split("[\r\n" +
        "|, |\\. |@|#|$|%|^|&|{|}(|)|; |: |' |! |? |+ |* |/|\n" +
        "|\r]");
for(int i=0;i<array.length;i++) {//将出现单词加入set
    if (!set.contains(array[i])&&array[i].length()>2)
        set.add(array[i]);
}
Test.put(l,set);//用<filename,set<word>>的KV存入Test中
```

这里用 set 的好处是自动消除重复出现的单词。

2.4.2 模型训练

模型训练最主要的内容就是进行各个单词出现频率的计算。由于我在 postive 和 negtive 两个 MAP 中分分别存了在 ham 中和在 spam 中出现的所有单词及其出现次数，所以我只需要分别遍历这两个表，然后计算概率存到另外的 MAP 中就好了。我使用如下的格式存放概率：

```
Map<String, Double> posMap = new HashMap<>();//存所有单词在正例中的概率
Map<String, Double> negMap = new HashMap<>();//存所有单词在反例中的概率
```

其中 K 是单词，VALUE 是单词的概率。概率的计算可以用如下方式：

p=a/total。

具体代码如下：

① 计算正例和反例的所有单词的总出现次数：函数 get_pos_num 和 get_neg_num

```
int get_pos_num() {
    int a=0;
    for(String key:postive.keySet())
        a=a+postive.get(key);
      System.out.println(a);
    return a;
}

int get_neg_num() {
    int b=0;
    for(String key:negtive.keySet())
        b=b+negtive.get(key);
      System.out.println(b);
    return b;
}
```

② 计算每个单词在正例中的出现次数和在反例中的出现次数：函数 get_num_of_appear_in_pos(word)和 get_num_of_appear_in_neg(word)：

```
int get_num_of_appear_in_pos(String word) {
    int num = 0;
    if(postive.containsKey(word))
        num=postive.get(word);
    return num;
}
```

③ 训练

训练的过程就是计算各个单词的概率并且存储的过程，其中会涉及到拉普拉斯修正，同时过滤掉出现次数太高或者太低的词汇。由于训练样本中 ham 和 spam 的个数不一样，出现的词汇总数也不一样，所以过滤的词汇要求也不一样。

```
m = get_pos_num();//m为ham中总的单词出现次数
n = get_neg_num();//n为spam中总的单词出现次数
//获取所有单词在正例中的出现次数
Set<String> posSet = postive.keySet();
for (String s : posSet) {
    if (!posMap.containsKey(s)) {//概率表中没存在这个单词，则计算
        double p;
        p = (double) get_num_of_appear_in_pos(s) / (double) m;
        posMap.put(s, p);
    }
}
//获取所有单词在反例中出现的概率
Set<String> negSet = negtive.keySet();
for (String s : negSet) {
    if (!negMap.containsKey(s)) {
        double p;
        p = (double) get_num_of_appear_in_neg(s) / (double) n;
        //              System.out.println(s + " " + p);
        negMap.put(s, p);
    }
}
```

注意这里我先计算了 m 和 n，这样 m 和 n 就只计算了一次。如果 m 是在用到的时候再计算这样就会导致计算了超级大。同样，想判断概率表 posMap 和 negMap 中是否存在了单词，再决定要不要计算概率。我尝试了之后先判断在计算和先计算再判断这两种方式的程序运行时间分别是 8s 和 4mins。在编程中要尤其注意这些情况，尤其是这种大数据计算。因为好的代码流程风格会很好的

减少你的代码时间复杂度和空间复杂度。

2.4.3 数据测试

我在 Test 中存放了<filename,file_content>的键值对，要预测一个文件的时候，我首先需要获取这个文件的所有单词 $w_0, w_1, w_2 .... w_i$，然后计算每一个单词在 ham 中出现的概率 $\mathrm{p}(w_i \mid ham)$，最后求出概率的乘积 pos= $\prod_{i=0}^{n} p(w_i \mid ham)$。q 其中 $p(w_i \mid ham)$ 表示在 ham 中 wi 出现的概率。同样方式计算 $\mathrm{neg} = \prod_{i=0}^{n} p(w_i \mid spam)$，假设 ham 和 spam 的先验概率分布为 a,b 如果 pos*a>pos*b，那么代表是 ham，否则就是 spam。这个先验概率是可以调的，只有满足 a+b=1 就好了。代码如下：

① 首先用 P1 和 P2 存所有的概率，在等会计算时候把链表中的所有概率相乘就好。

```
List<Double> P1 = new ArrayList<>();//所有的概率P(x1|ham),P(x2|ham)......
List<Double> P2 = new ArrayList<>();//所有的概率P(x1|spam),P(x2|spam)......
```

② 然后进行概率计算：

```
for (String s:Test.get(filename)) {//s是单词
    int a = get_num_of_appear_in_pos(s);
    int b = get_num_of_appear_in_neg(s);
    if (a == 0) {//出现次数为0使用拉普拉斯修正
        if(P1.contains())
        double p = (double) 1 / (double) (c + 2);
        P1.add(p);
    }
    if (b == 0) {//同上
        double p = (double) 1 / (double) (d + 2);
        P2.add(p);
    }

    if (a>=1&&a<25000) {//ham中，过滤出现次数大于25000的词语
        P1.add(posMap.get(s));
    }

    if (b>=1&&b<4500) {//spam中过滤掉出现次数大于4500的词语
        P2.add(negMap.get(s));
    }
}//end for 2
```

③ 最后把所有概率相乘，我是使用了取对数相加的方式：

(其中 0.2 和 0.8 是 ham 和 spam 的先验概率)

```
double pos = 0.0, neg = 0.0;
for (double d1 : P1)
    pos = pos + Math.log(d1);
pos=pos+Math.log(0.2);

for (double d1 : P2)
    neg = neg + Math.log(d1);
neg=neg+Math.log(0.8);
```

④ 最后进行判断，如果 pos>neg 代表是 ham，否则就是 spam。

⑤ 对所有的测试文件都执行上述步骤，就可将 14629 个测试文件都遍历结束。

3 算法比较

这里我主要会介绍贝叶斯和其他算法的比较，然后说明我为什么选择这个算法。

3.1 KNN

KNN 算法是基于类比学习，即通过将给定的检验元祖与和它相似的训练元祖进行比较来学习。训练元祖用 n 个属性描述。每个元祖代表 n 唯空间的一个点。这样，所有的训练元祖都被放入了 n 维模式空间中。当给定一个未知元祖时，k 最近邻分类法搜索模式空间，找出了最接近元祖的 k 个训练元祖，这 k 个训练元祖就是未知元祖的 k 个最近邻。

假设两个点 $X_1(\mathrm{x}_{11}, x_{12},..., x_{1n})$，$X_2(\mathrm{x}_{21}, x_{22},..., x_{2n})$，两个点的距离这么计算：

$$d(X_1, X_2) = \sqrt{\sum_{i=1}^{\mathrm{n}}(x_{1i} - x_{2i})\text{^}2}$$ 。

当然，KNN 是可以改进的。

通常在进行计算之前，需要对数据进行离散化，将数据[0,1]区间中。

举个例子：假设给一个数据 D，我们要预测新的点的值，我们计算这个点与数据 D 中每个点之间的距离，然后找出距离最近的 K 个点，在这 K 个点中，如果正例占大多数那么这个新的点就是正例，否则就是反例。

最近邻分析使用基于距离的比较，本质上赋予每个属性相同的权重。因此，当数据中存在噪声或者不相关属性时，它的准确率可能会受到影响。

KNN 的准确不应该是很高的，但是其巨大的时间复杂度不适合垃圾邮件分类。14000 多个测试文件，计算的时间复杂度都太大。而且还要给特征值进行离散化，如果数据处理不好，准确率会受很大影响。

3.2 神经网络

神经网络的设计需要按照下面几个步骤：

3.2.1 初始化权值

网络的权值初始化为小随机数，一般为-1.0 到 1.0，或者-0.5 到 0.5。每个单元都有一个相关联的偏倚，类似的，偏倚也设置为小随机数。

3.2.2 向前传播输入

首先，训练元祖提供给网络的输入层。输入通过输入单元，不发生变化。也就是说，对于输入单元 j，它的输出 O_{j} 等于输入值 I_{j}。然后，计算隐藏层和输出层的每个单元的净输入和输出。隐藏层和输出层单元的净输入用其输入的线性组合计算。在一个隐藏层或输出层单元中，有许多输入，这些输入事实上是连接它的上一层的单元的输出。每个连接都有一个权值。为了计算该单元的净输入，连接该单元的每个输入都乘以其对应的权值，然后求和。给定隐藏层或输出层的单元 j，到单元 j 的净输出 Ij 是：

$$I_{\mathrm{j}} = \sum_i w_{ij} O_i + \theta_j$$

其中 w_{ij} 是由上一次 i 到这一层 j 的权值，O_i 是上一层的输出，而 θ_{j} 是单元 j 的偏倚。

给定单元的净输入 Ij,则单元 j 的输出 Oj 用下面公式计算：

$$O_j = \frac{1}{1+e^{-I_j}}$$

这个函数又称为挤压函数。

3.2.3 计算误差

通过更新权重和反映网络预测误差的偏倚，向后传播误差。对于输出单元 j，误差 Errj 用下面公式计算：

$$O_j = O_j(1-O_j)(T_j - O_j)$$

为计算隐藏层单元 j 的误差，考虑下一层中连接 j 的单元的误差加权和。隐藏层单元 j 的误差是：

$$Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}$$

其中，w_{jk} 是由下一较高层中单元 k 到单元 j 的连接权值，而 Errk 是单元 k 的误差。

3.2.4 更新权值

权值和偏倚的更新，是用来反映误差的传播。权值更新用下面的方式：

$$\Delta w_{ij} = (l) Err_j O_i;$$

$$w_{ij} += \Delta w_{ij}$$

其中 l 是指学习率。然后偏倚的更新用下面方式：

$$\Delta\theta_j = (l) Err_j$$

$$\theta_j += \Delta\theta_j ;$$

3.2.5 代码终止条件

① 前一周期所有的△wij 都太小小于某个指定的阈值，或者

② 前一周期误分类的元祖百分比小于某个阈值，或者

③ 超过预期指定的周期数。

3.3 决策树

J4.8 算法就是决策树分类算法，决策树是一种类似流程图的树结构，其中每个内部节点表示在一个属性上的测试，每个分支代表该测试的一个输出，而每个叶节点存放一个类标号。决策树算法概括如下：

① 用三个参数 D，attribute_list 和 Attribute_selection_method 调用该算法。我们称 D 为数据分组。开始，它是训练元祖和它们相应类标号的完全集。参数 attribute_List 是描述元组属性的列表。Attribute_selection_method 指定属性的启发式过程，用来选择可以按类最好的区分给定元祖的属性。

② 算法伪代码如下：

算法：Generate_decision_tree。由数据分区 D 中的训练元组产生局决策树。

输入：

.数据分区 D：训练元组和他们对应类标号的集合；

Attribute_list：后续属性集合；

Attribute_selection_method：一个确定的“最好的”划分数据元组为个体类的分裂准则的过程。这个准则

由分裂属性 splitting_attribute 和分裂点或划分子集组成。
输出：
一颗决策树
方法：
1) 创建一个节点 N；
2) If D 中的元祖都在同一类 C 中，then
3) 返回 N 作为叶节点，以类 C 标记；
4) If attribute_list 为空，then
5) 返回 N 作为叶节点，标记 D 中的多数类
6) 使用 Attribute_selection_method(D,attribute_list)找出最好的 splitting_criterion；
7) 用 splitting_criterion 标记节点 N；
8) If solitting_criterion 是离散值的，并且允许多路划分//不限于二叉树
9) Attribute_list ←attribute_list-splitting_criterion；//删除分裂属性
10) For splitting_criterion 的每个输出 j：
a) 设 Dj 是 D 中满足输出 j 的数据元组的集合；
b) If Dj 为空，then
c) 加一个树叶到节点 N，标记 D 中的多数类；
d) Else 加一个由 Generate_decision_tree(Dj,attribute,list)返回的节点 N；
11) Endfor
12) 返回 N

通过上述方法就可以实现决策树算法。

3.4 比较

3.4.1 决策树

优点：决策树易于理解和解释；可以在相对短的时间里对大数据做出很好的预测。

缺点：时间复杂度太高；对于训练样本不一致的数据，在决策树中信息增熵会偏向那些具有更多数值的特征。在这次实验中，训练样本中的 ham 就比 spam 多很多。同时还有过度拟合问题。

3.4.2 KNN

优点：简单，有效，不需要训练。可以根据惰性机制，将新预测的样本加入大训练集中，提高预测准确率。

缺点：基于 KNN 的分类局限性，还有时间复杂度的问题，KNN 不适合垃圾邮件分类。

3.4.3 神经网络

优点：分类准确度高，并行分布处理能力强；

缺点：需要提供太多的参数，如网络拓扑结构，权值和阈值的初始值，学习时间又太久甚至打不到学习的目的。

3.4.4 朴素贝叶斯

优点：源于古典数学，有强大的数学基础，稳定的分类效率；朴素贝叶斯不需要决策树那样的内存消耗，因为是用概率计算的，也不会出现预测偏向哪一边的问题；同样也会花费 KNN 那样的时间复杂度；同时又

不用提供项神经网络那样多的不行的参数，只需要提供先验概率就好了。

缺点：需要指定先验概率

综上所述，我选择朴素贝叶斯，朴素贝叶斯也必将适合这个实验。具有可观的时间复杂度和空间复杂度已经准确率。还可以通过调整先验概率来修改模型提高模型的预测正确率。

4 实验结果

4.1 准确率

我的结果准确率是 0.9911。但是这是经过调整的。

最开始我使用统计的是所有的词的概率，准确率是 0.89。后来我修改了代码，将一些出现次数过多或者过少的词过滤掉，又将一些类似 a,an 这样的词过滤掉，准确率立刻上升到 0.95。然后我又修改了先验概率，通过先验概率的修改，可以将准确率吧修改到 0.96。当然，这还不够，我之前用的是概率相乘的方法，后来我修改成取对数相加的方式，准确率立刻上升到了 0.9911。

4.2 程序运行时间

一开始我的程序运行时间是 4 分钟差不多，在我把一些词过滤掉后程序实际到了 1 分钟左右。然后我修改了程序流图，比如先进行判断再计算，不是先计算再判断，通过这样修改，减少了很多计算，将程序运行时间减少到 10s 补不到。

4.3 结论

在垃圾邮件中，确实存在很多的“寡不敌众”现象，举个例子。假设 advertisement 这个词在垃圾邮件中出现频率很高，当一封正常邮件中出现这个词时，我们很可能把这个邮件判断为垃圾邮件屏蔽了，这样明显是对用户不友好的。这种过于“明显”的词对邮件的分类影响很大。对于不同的数据集，我们需要过滤掉不同百分比的单词，这样可以获取更改的准确率。

同样，朴素贝叶斯的先验概率对程序的执行结果也是很重要的，不同的数据集我们需要设定不同的先验概率，这样可以一定程度是模型更完美。

5 结束语

通过这次实验，我第一次完成了一个现实世界的数据挖掘实验，而且准确度很高，感觉还是收获很大。这次实验由于是垃圾邮件分类，我们可以使用简单的单词出现概率来进行朴素贝叶斯分类。在特征提取时候也可以直接用单词作为邮件的特征。但是如果是其他的数据挖掘任务，特征提取就不是很简单，这时候数据挖掘的难点就在于数据特征的提取，这也是我认为数据挖掘最难的地方。

算法本身不是困难，已经有了先人给我们做好了铺垫，数据挖掘重点和难点在于数据分析，即把数据的特征值提取出来，怎么提取数据特征值才是数据挖掘的重点。我们需要在以后慢慢的培养对数据进行抽象和特征提取的能力。

6 参考资料

[1] 《数据挖掘概念与技术》Jiawei Han,Micheline Kamber,Jianpei 编著；范明，孟小峰译；机械工业出版社。

[2] 《机器学习导论》 南京大学周志华编著；机械工业出版社。