5+NaiveBayes

2019 年 3 月 9 日

1 朴素贝叶斯（Naive Bayes）

1.1 一、概念

朴素贝叶斯和高斯判别分析一样，都是生成学习算法，通过学习条件概率来达到分类判别的作用。

先前对高斯判别分析进行前提假定时，假定给定 Y 时 X 的分布是多元正态分布，这也就意味着高斯判别分析适用于连续性数值变量（数据符合正态分布则模型效果更加）。而朴素贝叶斯则是用于离散变量的生成学习算法，朴素贝叶斯曾广泛用在垃圾邮件识别中，通过统计各个词汇是否出现来判断一个邮件是否是垃圾邮件。

在朴素贝叶斯模型中，自变量 X 是一串取值为 0 或 1 的向量，向量长度代表字典中所有词汇的个数，每个位置对应一个单词，该单词出现在邮件中则为 1，不出现则为 0。假设在给定 Y 的条件下各个单词是否出现在邮件中是相互独立的，字典中共有 5000 个词，那么将有如下等式：

$$\begin{aligned} p(x_1,...,x_{5000}|y) &= p(x_1|y)p(x_2|y,x_1)p(x_3|y,x_1,x_2)...p(x_{5000}|y,x_1,x_2,...,x_{4999}) \\ &= p(x_1|y)p(x_2|y)p(x_3|y)...p(x_{5000}|y) \\ &= \prod_{i=1}^{5000} p(x_i|y) \end{aligned}$$

第一行公式是按照条件概率的公式进行展开，第二行公式是运用假定——---各个单词是否出现相互独立，得来的。

假设在给定 Y 的条件下 X 的分布是伯努利分布：

$$\begin{aligned} P(x_i = 1|y = 0) &= \phi_{i|y=0} \\ P(x_i = 0|y = 0) &= 1 - \phi_{i|y=0} \\ P(x_i = 1|y = 1) &= \phi_{i|y=1} \\ P(x_i = 0|y = 0) &= 1 - \phi_{i|y=1} \end{aligned}$$

Y 本身的分布也是伯努利分布：

$$p(y) = \phi^y(1-\phi)^{(1-y)}$$

1.2 二、优化

朴素贝叶斯的优化同样是通过极大化似然函数，根据前述假定，似然函数为：

$$\begin{aligned} L(\phi_y, \phi_{j|y=0}, \phi_{j|y=1}) &= \prod_{i=1}^{n} p(x^{(i)}, y^{(i)}) \\ &= \prod_{i=1}^{n} p(x_1^{(i)}, ..., x_{5000}^{(i)} | y^{(i)}) p(y^{(i)}) \\ &= \prod_{i=1}^{n} \left[\prod_{j=1}^{5000} p(x_j^{(i)} | y^{(i)}) \right] p(y^{(i)}) \end{aligned}$$

之前 x_j 表示字典中的第 j 个词是否出现，为了不造成混淆，此处用 $x^{(i)}$ 表示第 i 个样本。
取对数之后易得使似然函数最大的参数值为：

$$\begin{aligned} \phi_{j|y=1} &= \frac{\sum_{i=1}^{n} I(x_j^{(i)} = 1 \wedge y^{(i)} = 1)}{\sum_{i=1}^{n} I(y^{(i)} = 1)} \\ \phi_{j|y=0} &= \frac{\sum_{i=1}^{n} I(x_j^{(i)} = 1 \wedge y^{(i)} = 0)}{\sum_{i=1}^{n} I(y^{(i)} = 0)} \\ \phi_y &= \frac{\sum_{i=1}^{n} I(y^{(i)} = 1)}{n} \end{aligned}$$

有了上述的参数之后，我们预测一封邮件是否是垃圾邮件时就可以直接用贝叶斯公式：

$$\begin{aligned} P(y = 1|x) &= \frac{P(x|y = 1)P(y = 1)}{P(x)} \\ &= \frac{(\prod_{i=1}^{n} P(x_i|y = 1))P(y = 1)}{(\prod_{i=1}^{n} P(x_i|y = 1))P(y = 1) + (\prod_{i=1}^{n} P(x_i|y = 0))P(y = 0)} \end{aligned}$$

然而实际应用时，经常会有一个单词在垃圾邮件和非垃圾邮件中都没出现，也就是:

$$\begin{aligned} \phi_{j|y=1} &= \frac{\sum_{i=1}^{n} I(x_j^{(i)} = 1 \wedge y^{(i)} = 1)}{\sum_{i=1}^{n} I(y^{(i)} = 1)} = 0 \\ \phi_{j|y=0} &= \frac{\sum_{i=1}^{n} I(x_j^{(i)} = 1 \wedge y^{(i)} = 0)}{\sum_{i=1}^{n} I(y^{(i)} = 0)} = 0 \end{aligned}$$

这也就使得在应用贝叶斯定理时造成：

$$P(y = 1|x) = \frac{(\prod_{i=1}^{n} P(x_i|y = 1))P(y = 1)}{(\prod_{i=1}^{n} P(x_i|y = 1))P(y = 1) + (\prod_{i=1}^{n} P(x_i|y = 0))P(y = 0)} = \frac{0}{0}$$

这显然是我们需要规避的问题，于是便有了**拉普拉斯平滑（Laplace smoothing）**。拉普拉斯平滑的想法就是在所有的参数的分子和分母同时加上一个常数，使得 ϕ 不会为 0。

对于 K 分类问题通常是分母加 K，分子加 1：

$$\phi_m = \frac{\sum_{i=1}^{n} I(y^{(i)} = m) + 1}{n + k}$$
$$\phi_{j|y=m} = \frac{\sum_{i=1}^{n} I(x_j^{(i)} = 1 \wedge y^{(i)} = m) + 1}{\sum_{i=1}^{n} I(y^{(i)} = m) + k}$$

在 2 分类中把 K 变成 2，m 的取值为 0-1 即可。

1.3 三、应用

因为要求变量为离散的，所以直接适用于进行朴素贝叶斯分类的数据并不多，所以这次用生成的数据进行演示。

在实际应用中可以对连续的数据进行分组离散化来应用朴素贝叶斯。

1.3.1 1. 生成数据

首先生成取值在 1-100 的 10000*500 的随机整数矩阵，把 1-1000 的整数视为字典，统计字典中的单词是否出现，整理成 10000*1000 的矩阵。然后按照 7：3 划分训练集和测试集

```
In [25]: import numpy as np

         np.random.seed(2099)
         origin = np.random.randint(1, 1000, (10000, 500))

         data = np.zeros(1000).reshape([1, 1000])
         for i in range(10000):
             tmp = origin[i,:].copy()
             tmp = np.unique(tmp)
             tmp2 = np.zeros(1000).reshape([1, 1000])
             tmp2[:,tmp] = 1
             data = np.vstack([data, tmp2])

         data = np.delete(data, 0, 0)
         data.shape
```

借助回归的形式（其实是感知机，perceptron）将化成取值 0-1 的向量。

```
In [36]: tmp = np.random.randn(1000).reshape([1000,1])
         y = np.dot(data, tmp)
         y = y>y.mean()
         np.unique(y)

Out[36]: array([False,  True])
```

```
In [37]: y.shape
```

```
Out[37]: (10000, 1)
```

```
In [41]: train_x = data[0:7000, :]
         train_y = y[0:7000, :]
         test_x = data[7000:10000, :]
         test_y = y[7000:10000, :]
```

1.3.2 2. 求解

直接应用拉普拉斯平滑计算 $\phi_{j|y=1}$ 和 $\phi_{j|y=0}$ 两个向量。当且仅当单词出现且是垃圾邮件时，代表两个变量的数值的和为 2；当且仅当单词出现且不是垃圾邮件时，前者减去后者的差为 1。以此可以方便计算和统计。

```
In [43]: n_one = np.sum(train_y)
         n_zero = train_y.shape[0] - n_one
         print(n_one)
         print(n_zero)
```

```
3488
3512
```

```
In [56]: phi_1 = (np.sum(((train_x+train_y)==2), axis=0)+1)/(n_one+2)
         phi_0 = (np.sum(((train_x-train_y)==1), axis=0)+1)/(n_zero+2)
         phi_y = np.sum(train_y)/(train_y.shape[0]+2)
```

进行预测较复杂，因为需要元素累乘，这种情况可以将向量处理成对角矩阵然后求行列式。

$X * \phi_1$ 和 $(1-X)*(1-\phi_1)$ 进行元素间的乘法，得到两个矩阵，计算每行的累乘即是 $\prod_{i=1}^{n} P(x_i|y=1)$

```
In [70]: def predict(x, phi_0, phi_1, phi_y):
             n = x.shape[0]
             x_1 = x*phi_1 + (1-x)*(1-phi_1)
             x_0 = x*phi_0 + (1-x)*(1-phi_0)
             tmp_1 = None
             tmp_0 = None
             for i in range(n):
                 tmp = np.linalg.det(np.diag(x_1[i, :]))
                 tmp_1 = np.vstack([tmp_1, tmp])
                 tmp = np.linalg.det(np.diag(x_0[i, :]))
                 tmp_0 = np.vstack([tmp_0, tmp])
```

```
        tmp_1 = np.delete(tmp_1, 0, 0)
        tmp_0 = np.delete(tmp_0, 0, 0)
        return (tmp_1*phi_y)/(tmp_1*phi_y+tmp_0*(1-phi_y))

    train_y_hat = predict(train_x, phi_0, phi_1, phi_y)
    test_y_hat = predict(test_x, phi_0, phi_1, phi_y)
```

根据似然函数的形式，可知似然函数是很多个小于 1 的数相乘的结果，实际似然函数极小，可能接近零，所以这里为了方便评价和比较训练集和测试集，依然选用交叉熵作为损失。

```
In [83]: def loss(y, y_hat):
        return -(np.dot(y.T, np.log(y_hat)) + np.dot(1-y.T, np.log(1-y_hat)))/y.shape[0]

    train_y_hat = train_y_hat.astype('float64')
    test_y_hat = test_y_hat.astype('float64')

    print(loss(train_y, train_y_hat))
    print(loss(test_y, test_y_hat))
```

```
[[0.28723441]]
[[0.37788191]]
```

差距还是挺大的，可能和数据生成有关系。

```
In [ ]:
```