拉普拉斯修正的朴素贝叶斯分类器

杨启航

2022 年 4 月 13 日

目录

1 实验目的

了解朴素贝叶斯分类器和拉普拉斯修正的理论，使用 Python 实现，并在西瓜数据集上进行测试。

2 贝叶斯分类器

2.1 最小化错误率贝叶斯最优分类器

设假设 h 错将分类为 y_i 的样本 x 分类为任意 $y_j \neq y_i$ 的风险 λ_{ji} 均相等，则可以在后验概率上运用贪心思想，得出最小化分类错误率的贝叶斯最优分类器为：

$$h^*(\mathbf{x}) = \arg\max_{c \in \mathcal{Y}} P(c|\mathbf{x}) \tag{2.1}$$

2.2 朴素贝叶斯分类器

根据贝叶斯定理，可以由数据集估计先验概率 $P(c)$ 和条件概率 $P(\mathbf{x}|c)$ 以计算后验概率，则可以将式 (2.2) 改写为：

$$h^*(\mathbf{x}) = \arg\max_{c \in \mathcal{Y}} \frac{P(c)P(\mathbf{x}|c)}{P(\mathbf{x})} \tag{2.2}$$

上式中先验概率 $P(\mathbf{x}|c)$ 往往因为组合爆炸问题，在有限的数据集上难以进行有效统计。

朴素贝叶斯分类器 (naive Bayes classifier) 采用了“属性条件独立性假设”，有 $P(\mathbf{x}|c) = \prod_{i=1}^{n} P(x_i|c)$。单个属性的条件概率 $P(x_i|c)$ 在估计时不容易失真，而式 (2.2) 可以写为：

$$h^*(\mathbf{x}) = \arg\max_{c \in \mathcal{Y}} \frac{P(c)}{P(\mathbf{x})} \prod_{i=1}^{n} P(x_i|c) \tag{2.3}$$

2.3 先验概率和条件概率的估计

主要利用大数定理和极大似然估计，并假设连续属性服从高斯分布。

令 D_c 表示训练集 D 中第 c 类样本组成的结合，则有先验概率估计：

$$P(c) = \frac{|D_c|}{|D|} \tag{2.4}$$

对于离散属性而言，令 D_{i,x_i} 表示 D_c 在第 i 个属性上取值为 x_i 的样本集合，则有条件概率估计：

$$P(x_i|c) = \frac{|D_{i,x_i}|}{|D_c|} \tag{2.5}$$

对于连续属性而言，假定 $P(x_i|c) \sim \mathcal{N}(\mu_{c,i}, \sigma_{c,i}^2)$，其中 $\mu_{c,i}$ 和 $\sigma_{c,i}^2$ 分别是第 c 类样本在第 i 个属性上取值的均值和方差，则有条件概率估计：

$$P(x_i|c) = \frac{1}{\sqrt{2\pi}\sigma_{c,i}} e^{-\frac{(x_i \mu_{c,i})^2}{2\sigma_{c,i}^2}} \tag{2.6}$$

2.4 拉普拉斯修正

在有限的数据集上，即使使用朴素贝叶斯方法，对于条件概率较小的属性，在估计的时候也很有可能出现 $P(x_i|c) = 0$ 的情况，此时式 (2.3) 中的连乘为 0，信息失真，分类趋于极端化。若出现这种情况，则在估计时引入“拉普拉斯修正”(Laplacian correction)：

$$\hat{P}(c) = \frac{|D_c| + 1}{|D| + N} \tag{2.7}$$

$$\hat{P}(x_i|c) = \frac{|D_{i,x_i}| + 1}{|D_c| + N_i} \tag{2.8}$$

其中 N 表示训练集 D 出现的类别种数，N_i 表示第 i 个属性可能的取值数。

3 Python 实现

3.1 NaiveBayes 类

```
1  from numpy import float64
2  import pandas as pd
3  from scipy import stats as st
4
5
6  class NaiveBayes:
7      class U:
8          def __init__(self) -> None:
9              self.is_discrate = True
10             self.cnt = {}
11
12         is_discrate, cnt, = None, None
```

```
13
14      data = None
15      Y = None
16      tot = None
17      ycnt = None
18      Ni = None
19
20      def malloc( self ):
21          self .data = []
22          self .Y = []
23          self .tot = 0
24          self .ycnt = {}
25          self .Ni = []
26
27      def __init__(self, D) -> None:
28          self .malloc()
29          self .tot = len(D)
30          self .Y = set(D.iloc [:, -1])
31          Dy = {}
32          for y in self .Y:
33              Dy[y] = D.loc[D[D.keys()[-1]] == y]
34              self .ycnt[y] = len(Dy[y])
35          for attr in D.keys ()[:-1]:
36              self .Ni.append(len(set(D[attr ])))
37              self .data.append(self .U())
38              if D[attr ].dtype == float64:
39                  self .data[-1]. is_discrate = False
40                  for y in self .Y:
41                      = Dy[y][attr ].mean()
42                      s = Dy[y][attr ].std()
43                      self .data[-1].cnt[y] = [ , s]
44              else :
45                  self .data[-1]. is_discrate = True
```

```
46                  for y in self.Y:
47                      self.data[-1].cnt[y] = {}
48                      for xi in Dy[y][attr]:
49                          self.data[-1].cnt[y][xi] = \
50                                  self.data[-1].cnt[y].get(xi, 0) + 1
51
52      def predict(self, x):
53          laplace = False
54          res, maxp = None, 0.0
55          # is need laplacian correction
56          for y in self.Y:
57              for xi, i in zip(x, range(len(x))):
58                  if self.data[i].is_discrate:
59                      if self.data[i].cnt[y].get(xi, -1) == -1:
60                          laplace = True
61
62          for y in self.Y:
63              p = 1.0
64              for xi, i in zip(x, range(len(x))):
65                  if self.data[i].is_discrate:
66                      if laplace:
67                          p *= (self.data[i].cnt[y].get(xi, 0) + 1) / (
68                              self.ycnt[y] + self.Ni[i]
69                          )
70                      else:
71                          p *= self.data[i].cnt[y][xi] / self.ycnt[y]
72                  else:
73                      p *= st.norm.pdf(
74                          x=xi,
75                          loc=self.data[i].cnt[y][0],
76                          scale=self.data[i].cnt[y][1]
77                      )
78
```

```
79              if laplace:
80                  p *= (self.ycnt[y] + 1) / (self.tot + len(self.Y))
81              else:
82                  p *= self.ycnt[y] / self.tot
83
84              if p > maxp:
85                  maxp, res = p, y
86          return res
```

封装 NaiveBayes 类，在对象化的时候需要传入数据集，此时会立刻完成统计并保存统计结果，之后只需要调用该对象的 Predict(x) 函数，传入待预测样本，返回预测结果。

NaiveBayes 对象以属性为单元储存统计信息，并封装在类内类 U 中，U 对象内记录了对应属性是否为离散属性，并对于离散属性记录 $|D_{c,x_i}|$，对于连续属性记录 $\mu_{c,i}$ 和 $\sigma_{c,i}$。

预测时，先判断是否需要拉普拉斯变换，即是否有 $\exists xi \exists c |D_{c,xi}| = 0$，后选用合适的估计方式以计算各个分类 c_i 的后验概率，取后验概率最大者返回。

3.2 测试代码

```
1  from NaiveBayes import NaiveBayes
2  import numpy as np
3  from sklearn import datasets
4  from time import time
5  import pandas as pd
6
7
8  df = pd.read_csv("watermelon3_0_Ch.csv")
9  df.drop("编号", axis=1, inplace=True)
10 t0 = time()
11 nb = NaiveBayes(df)
12 print("统计消耗时间: {:.3f}S".format(time() - t0))
13 precision = 0.0
```

```
t0 = time()
for i in range(len(df)):
    x = df.iloc[i, :].tolist()
    y = nb.predict(x[:-1])
    print("{}: 真实分类: {}, 预测结果: {}".format(i + 1, x[-1], y))
    if x[-1] == y:
        precision += 1 / (len(df) + 1)
print("准确率: {:.3f}".format(precision))
print("预测消耗时间: {:.3f}S".format(time() - t0))

iris = datasets.load_iris()
df = pd.DataFrame(
    data=np.c_[iris["data"], iris["target"]],
    columns=iris["feature_names"] + ["target"]
)
print(df)
t0 = time()
nb = NaiveBayes(df)
print("统计消耗时间: {:.3f}S".format(time() - t0))
precision = 0.0
t0 = time()
for i in range(len(df)):
    x = df.iloc[i, :].tolist()
    y = nb.predict(x[:-1])
    print("{}: 真实分类: {}, 预测结果: {}".format(i + 1, x[-1], y))
    if x[-1] == y:
        precision += 1 / (len(df) + 1)
print("准确率: {:.3f}".format(precision))
print("预测消耗时间: {:.3f}S".format(time() - t0))
```

测试中除了西瓜数据集，还测试了 Sk-learn 库中自带的鸢尾花数据集，以检验 NaiveBayes 类处理其它任务的能力，且鸢尾花数据集更加大，相比西瓜数据集更适合探究分类器性能。

3.3 测试结果

```
统计消耗时间: 0.002S
1: 真实分类: 是, 预测结果: 是
2: 真实分类: 是, 预测结果: 是
3: 真实分类: 是, 预测结果: 是
4: 真实分类: 是, 预测结果: 是
5: 真实分类: 是, 预测结果: 是
6: 真实分类: 是, 预测结果: 是
7: 真实分类: 是, 预测结果: 否
8: 真实分类: 是, 预测结果: 是
9: 真实分类: 否, 预测结果: 否
10: 真实分类: 否, 预测结果: 否
11: 真实分类: 否, 预测结果: 否
12: 真实分类: 否, 预测结果: 否
13: 真实分类: 否, 预测结果: 是
14: 真实分类: 否, 预测结果: 否
15: 真实分类: 否, 预测结果: 是
16: 真实分类: 否, 预测结果: 否
17: 真实分类: 否, 预测结果: 否
准确率: 0.778
预测消耗时间: 0.015S

统计消耗时间: 0.035S
1: 真实分类: 0.0, 预测结果: 0.0
2: 真实分类: 0.0, 预测结果: 0.0
3: 真实分类: 0.0, 预测结果: 0.0
4: 真实分类: 0.0, 预测结果: 0.0
5: 真实分类: 0.0, 预测结果: 0.0
6: 真实分类: 0.0, 预测结果: 0.0
7: 真实分类: 0.0, 预测结果: 0.0
8: 真实分类: 0.0, 预测结果: 0.0
9: 真实分类: 0.0, 预测结果: 0.0
10: 真实分类: 0.0, 预测结果: 0.0
```

```
11: 真实分类: 0.0, 预测结果: 0.0
12: 真实分类: 0.0, 预测结果: 0.0
13: 真实分类: 0.0, 预测结果: 0.0
14: 真实分类: 0.0, 预测结果: 0.0
15: 真实分类: 0.0, 预测结果: 0.0
16: 真实分类: 0.0, 预测结果: 0.0
17: 真实分类: 0.0, 预测结果: 0.0
18: 真实分类: 0.0, 预测结果: 0.0
19: 真实分类: 0.0, 预测结果: 0.0
20: 真实分类: 0.0, 预测结果: 0.0
21: 真实分类: 0.0, 预测结果: 0.0
22: 真实分类: 0.0, 预测结果: 0.0
23: 真实分类: 0.0, 预测结果: 0.0
24: 真实分类: 0.0, 预测结果: 0.0
25: 真实分类: 0.0, 预测结果: 0.0
26: 真实分类: 0.0, 预测结果: 0.0
27: 真实分类: 0.0, 预测结果: 0.0
28: 真实分类: 0.0, 预测结果: 0.0
29: 真实分类: 0.0, 预测结果: 0.0
30: 真实分类: 0.0, 预测结果: 0.0
31: 真实分类: 0.0, 预测结果: 0.0
32: 真实分类: 0.0, 预测结果: 0.0
33: 真实分类: 0.0, 预测结果: 0.0
34: 真实分类: 0.0, 预测结果: 0.0
35: 真实分类: 0.0, 预测结果: 0.0
36: 真实分类: 0.0, 预测结果: 0.0
37: 真实分类: 0.0, 预测结果: 0.0
38: 真实分类: 0.0, 预测结果: 0.0
39: 真实分类: 0.0, 预测结果: 0.0
40: 真实分类: 0.0, 预测结果: 0.0
41: 真实分类: 0.0, 预测结果: 0.0
42: 真实分类: 0.0, 预测结果: 0.0
43: 真实分类: 0.0, 预测结果: 0.0
```

```
44: 真实分类: 0.0, 预测结果: 0.0
45: 真实分类: 0.0, 预测结果: 0.0
46: 真实分类: 0.0, 预测结果: 0.0
47: 真实分类: 0.0, 预测结果: 0.0
48: 真实分类: 0.0, 预测结果: 0.0
49: 真实分类: 0.0, 预测结果: 0.0
50: 真实分类: 0.0, 预测结果: 0.0
51: 真实分类: 1.0, 预测结果: 1.0
52: 真实分类: 1.0, 预测结果: 1.0
53: 真实分类: 1.0, 预测结果: 2.0
54: 真实分类: 1.0, 预测结果: 1.0
55: 真实分类: 1.0, 预测结果: 1.0
56: 真实分类: 1.0, 预测结果: 1.0
57: 真实分类: 1.0, 预测结果: 1.0
58: 真实分类: 1.0, 预测结果: 1.0
59: 真实分类: 1.0, 预测结果: 1.0
60: 真实分类: 1.0, 预测结果: 1.0
61: 真实分类: 1.0, 预测结果: 1.0
62: 真实分类: 1.0, 预测结果: 1.0
63: 真实分类: 1.0, 预测结果: 1.0
64: 真实分类: 1.0, 预测结果: 1.0
65: 真实分类: 1.0, 预测结果: 1.0
66: 真实分类: 1.0, 预测结果: 1.0
67: 真实分类: 1.0, 预测结果: 1.0
68: 真实分类: 1.0, 预测结果: 1.0
69: 真实分类: 1.0, 预测结果: 1.0
70: 真实分类: 1.0, 预测结果: 1.0
71: 真实分类: 1.0, 预测结果: 2.0
72: 真实分类: 1.0, 预测结果: 1.0
73: 真实分类: 1.0, 预测结果: 1.0
74: 真实分类: 1.0, 预测结果: 1.0
75: 真实分类: 1.0, 预测结果: 1.0
76: 真实分类: 1.0, 预测结果: 1.0
```

```
77: 真实分类: 1.0, 预测结果: 1.0
78: 真实分类: 1.0, 预测结果: 2.0
79: 真实分类: 1.0, 预测结果: 1.0
80: 真实分类: 1.0, 预测结果: 1.0
81: 真实分类: 1.0, 预测结果: 1.0
82: 真实分类: 1.0, 预测结果: 1.0
83: 真实分类: 1.0, 预测结果: 1.0
84: 真实分类: 1.0, 预测结果: 1.0
85: 真实分类: 1.0, 预测结果: 1.0
86: 真实分类: 1.0, 预测结果: 1.0
87: 真实分类: 1.0, 预测结果: 1.0
88: 真实分类: 1.0, 预测结果: 1.0
89: 真实分类: 1.0, 预测结果: 1.0
90: 真实分类: 1.0, 预测结果: 1.0
91: 真实分类: 1.0, 预测结果: 1.0
92: 真实分类: 1.0, 预测结果: 1.0
93: 真实分类: 1.0, 预测结果: 1.0
94: 真实分类: 1.0, 预测结果: 1.0
95: 真实分类: 1.0, 预测结果: 1.0
96: 真实分类: 1.0, 预测结果: 1.0
97: 真实分类: 1.0, 预测结果: 1.0
98: 真实分类: 1.0, 预测结果: 1.0
99: 真实分类: 1.0, 预测结果: 1.0
100: 真实分类: 1.0, 预测结果: 1.0
101: 真实分类: 2.0, 预测结果: 2.0
102: 真实分类: 2.0, 预测结果: 2.0
103: 真实分类: 2.0, 预测结果: 2.0
104: 真实分类: 2.0, 预测结果: 2.0
105: 真实分类: 2.0, 预测结果: 2.0
143: 真实分类: 2.0, 预测结果: 2.0
144: 真实分类: 2.0, 预测结果: 2.0
145: 真实分类: 2.0, 预测结果: 2.0
146: 真实分类: 2.0, 预测结果: 2.0
```

```
147: 真实分类: 2.0, 预测结果: 2.0
148: 真实分类: 2.0, 预测结果: 2.0
149: 真实分类: 2.0, 预测结果: 2.0
150: 真实分类: 2.0, 预测结果: 2.0
准确率: 0.954
预测消耗时间: 0.269S
```

4 实验总结

4.1 拉普拉斯修正的朴素贝叶斯分类器思想

贝叶斯分类器利用了贪心思想，将后验概率 $P(c|\mathbf{x})$ 最大的分类 c 作为分类结果，而后验概率则根据数据集估计先验概率 $P(c)$ 和条件概率 $P(\mathbf{x}|c)$，再通过贝叶斯公式计算而来。

但是样本空间的大小随属性个数指数增长，在有限的数据集上，很难对以上两个统计量进行有效估计，朴素贝叶斯和拉普拉斯修正都是为了解决这个问题而引入的方法。

4.2 程序总结

4.2.1 程序特性

可以处理西瓜分类之外的其它分类任务,训练集以 Pandas 库 DataFrame 格式在 NaiveBayes 类实例化的时候输入，待预测样本以 Python 列表形式输入。训练集中 float64 类型会被识别为连续属性，其它类型会被识别为离散属性。

4.2.2 待改进

运行效率偏低；输入输出的普适性还不足够；运算样本方差的时候如果样本数量 $|D_c| = 0$ 还会出现问题，不可处理极端小的数据集。