School of Data and Computer Science, Sun Yat-sen University

E12 Naive Bayes (C++/Python)

17341189 姚森舰

2019 年 12 月 1 日

目录

1 Datasets

The UCI dataset (http://archive.ics.uci.edu/ml/index.php) is the most widely used dataset for machine learning. If you are interested in other datasets in other areas, you can refer to https://www.zhihu.com/question/63383992/answer/222718972.

Today's experiment is conducted with the **Adult Data Set** which can be found in http://archive.ics.uci.edu/ml/datasets/Adult.

Data Set Characteristics:	Multivariate	**Number of Instances:**	48842	**Area:**	Social
Attribute Characteristics:	Categorical, Integer	**Number of Attributes:**	14	**Date Donated**	1996-05-01
Associated Tasks:	Classification	**Missing Values?**	Yes	**Number of Web Hits:**	1305515

You can also find 3 related files in the current folder, `adult.name` is the description of **Adult Data Set**, `adult.data` is the training set, and `adult.test` is the testing set. There are 14 attributes in this dataset:

>50K, <=50K.

1. age: continuous.
2. workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
3. fnlwgt: continuous.
4. education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 5. 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
5. education-num: continuous.
6. marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
7. occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
8. relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
9. race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
10. sex: Female, Male.
11. capital-gain: continuous.
12. capital-loss: continuous.
13. hours-per-week: continuous.
14. native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras,

Philippines , Italy , Poland , Jamaica , Vietnam , Mexico , Portugal , Ireland , France , Dominican−Republic , Laos , Ecuador , Taiwan , Haiti , Columbia , Hungary , Guatemala , Nicaragua , Scotland , Thailand , Yugoslavia , El−Salvador , Trinadad&Tobago , Peru , Hong , Holand−Netherlands .

Prediction task is to determine whether a person makes over 50K a year.

2 Naive Bayes

Naive Bayes is a simple technique for constructing classifiers: models that assign class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set. It is not a single algorithm for training such classifiers, but a family of algorithms based on a common principle: all naive Bayes classifiers assume that **the value of a particular feature is independent of the value of any other feature**, given the class variable.

For example, a fruit may be considered to be an apple if it is red, round, and about 10 cm in diameter. A naive Bayes classifier considers each of these features to contribute independently to the probability that this fruit is an apple, regardless of any possible correlations between the color, roundness, and diameter features.

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable y and dependent feature vector x_1 through x_n:

$$P(y \mid x_1, ..., x_n) = \frac{P(y)P(x_1, ..., x_n \mid y)}{P(x_1, ..., x_n)}$$

Using the naive conditional independence assumption that

$$P(x_i \mid y, x_1, ..., x_{i-1}, x_{x+1}, ..., x_n) = P(x_i \mid y)$$

, for all i, this relationship is simplified to

$$P(y \mid x_1, ..., x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, ..., x_n)}$$

Since $P(x_1, ..., x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, ..., x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$, the former is then the relative frequency of class y in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$.

- When attribute values are discrete, $P(x_i \mid y)$ can be easily computed according to the training set.
- When attribute values are continuous, an assumption is made that the values associated with each class are distributed according to Gaussian i.e., Normal Distribution. For example, suppose the training data contains a continuous attribute x. We first segment the data by the class, and then compute the mean and variance of x in each class. Let μ_k be the mean of the values in x associated with class y_k, and let σ_k^2 be the variance of the values in x associated with class y_k. Suppose we have collected some observation value x_i. Then, the probability distribution of x_i given a class y_k, $P(x_i \mid y_k)$ can be computed by plugging x_i into the equation for a Normal distribution parameterized by μ_k and σ_k^2. That is,

$$P(x = x_i \mid y = y_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{(x_i - \mu_k)^2}{2\sigma_k^2}}$$

3 Task

- Given the training dataset `adult.data` and the testing dataset `adult.test`, please accomplish the prediction task to determine whether a person makes over 50K a year in `adult.test` by using Naive Bayes algorithm (C++ or Python), and compute the accuracy.
- Note: keep an eye on the discrete and continuous attributes.
- Please finish the experimental report named `E12_YourNumber.pdf`, and send it to `ai_201901@foxmail.com`

4 Codes and Results

4.1 Codes

```python
import math
import numpy as np
from time import *


def load_data(filename):
    """
    加载训练和测试数据集，并做一些初步的处理，删除掉无关的属性
    :param filename: str
    :return: 数据集列表
    """
    with open(filename, 'r') as f:
        dataset = []
        for line in f.readlines()[:-1]:
            record = line.strip().split(', ')
            record[0] = int(record[0])
```

```
            record[2] = int(record[2])
            record[4] = int(record[4])
            record[10] = int(record[10])
            record[11] = int(record[11])
            record[12] = int(record[12])
            # 去除最后的点
            if record[-1][-1] == '.':
                record[-1] = record[-1][:-1].strip()
            dataset.append(record)
        return dataset


def get_miss_attributes(dataset):
    """
    返回数据集中有缺失值的属性，以及对应缺失次数
    :param dataset:
    :return: dict 键值是缺失属性的序号
    """
    miss = {}
    for record in dataset:
        for i in range(len(record)):
            if record[i] == '?':
                miss[i] = miss.get(i, 0) + 1
    return miss


def get_attributes_domain(dataset):
    """
    返回每种属性的取值范围
    :param dataset: 数据集
    :return: dict 每种属性的取值范围
    """
    num = len(dataset[0]) - 1
    attribute_domain = {}
    # 遍历属性，遍历数据集，记录所有取值情况
    for i in range(num):
        domain = set()
        for record in dataset:
            domain.add(record[i])
        attribute_domain[i] = domain
    return attribute_domain


def get_attributes_max_branch(dataset):
    """
    每个属性取值出现次数最多的取值，用于填充
    :param dataset:
```

```
    :return: dict 有缺失的属性的出现最频繁的值
    """
    # 首先找到哪些属性有缺失
    miss_attributes = get_miss_attributes(dataset)
    miss_attributes_max_branches = {}
    # 遍历有缺失值的属性，统计找出有缺失值的属性取值最频繁的值，用于后面填充
    for i in miss_attributes.keys():
        count = {}
        for record in dataset:
            count[record[i]] = count.get(record[i], 0) + 1
        # 出现频率最高
        max_branch = max(count.items(), key=lambda x: x[1])
        miss_attributes_max_branches[i] = max_branch[0]
    return miss_attributes_max_branches

# miss_attributes_max_branches = get_attributes_max_branch(train_data)
# print(miss_attributes_max_branches)


def precondition(dataset):
    """
    预处理，将缺失值填充为该属性出现次数最多的取值
    :param dataset: 数据集，列表
    :return: 处理后的数据集 list， 有用属性序号列表
    """
    # 得到有缺失的属性，以及该属性出现次数最多的取值
    miss_attributes_max_branches = get_attributes_max_branch(dataset)
    # print(miss_attributes_max_branches)
    for record in dataset:
        for i in miss_attributes_max_branches.keys():
            if record[i] == '?':
                record[i] = miss_attributes_max_branches[i]
    return dataset


def get_freq(dataset):
    """
    统计各个属性各个取值在不同分类下出现的频数，对于连续型属性，返回训练集上的均值和方差
    :param dataset:
    :return: list , [ [{attr_value: n (第一类),...} {attr_value: n (第二类),...} ],
        [{}{}], ...]
    """
    total_records_num = len(dataset[0])
    # [ [{attr_value: n (第一类),...} {attr_value: n (第二类),...} ], [{},{}], ...]
    # 每个属性用两个字典记录，第一个是在第一类中计数，第二个是在第二类中计数
    freqs = [[{}, {}] for i in range(total_records_num)]
    for record in dataset:
```

```
        # 离散型属性，直接统计
        for i in [1, 3, 5, 6, 7, 8, 9, 13, 14]:
            value = record[i]
            # 第一类下计数
            if record[-1] == '<=50K':
                freqs[i][0][value] = freqs[i][0].get(value, 0) + 1
            # 第二类下计数
            else:
                freqs[i][1][value] = freqs[i][1].get(value, 0) + 1

    # 处理连续型属性
    continuous_index = [0, 2, 4, 10, 11, 12]
    # 先取出这些属性的取值，方便后面用np求均值和方差
    # 同样，每个属性用两个列表记录，第一个是在第一类中计数，第二个是在第二类中计数
    continuous_attr = [[[], []] for i in range(6)]
    for record in dataset:
        if record[-1] == '<=50K':
            for i in range(6):
                index = continuous_index[i]
                continuous_attr[i][0].append(record[index])
        else:
            for i in range(6):
                index = continuous_index[i]
                continuous_attr[i][1].append(record[index])

    # 计算连续型属性取值的均值和方差
    for i in range(6):
        freqs[continuous_index[i]][0]['mean'] = np.mean(continuous_attr[i][0])
        freqs[continuous_index[i]][0]['var'] = np.var(continuous_attr[i][0])
        freqs[continuous_index[i]][1]['mean'] = np.mean(continuous_attr[i][1])
        freqs[continuous_index[i]][1]['var'] = np.var(continuous_attr[i][1])

    # for freq in freqs:
    #     print('----------------------------')
    #     for k, v in freq[0].items():
    #         print(k,v)
    #     print('============================')
    #     for k, v in freq[1].items():
    #         print(k, v)
    #     print('')
    # print(len(freqs))

    return freqs


def gaussian(x, u, var):
    """
```

```
    输入均值方差，得到高斯函数，用它当作概率密度函数计算变量为x时的概率
    :param x: float,取值
    :param u: float，均值
    :param var: float，方差
    :return: float，概率
    """
    y = np.exp(-(x - u) ** 2 / (2 * var)) / (math.sqrt(2 * math.pi * var))
    return y


def test(test_dataset, freqs):
    """
    在生成的决策树上测试，返回正确率
    :param test_dataset:
    :param probs:
    :return: 正确率
    """
    right = 0
    less_50 = freqs[-1][0]['<=50K']
    more_50 = freqs[-1][1]['>50K']
    py1 = less_50 / (less_50 + more_50)
    py2 = more_50 / (less_50 + more_50)
    for record in test_dataset:
        label = '<=50K'
        prob1 = py1
        prob2 = py2
        # 计算 P(xi|y)
        for i in range(len(test_dataset[0])-1):
            # 连续型属性
            if i in [0, 2, 4, 10, 11, 12]:
                m1 = freqs[i][0]['mean']
                v1 = freqs[i][0]['var']
                p1 = gaussian(record[i], m1, v1)
                prob1 = prob1 * p1

                m2 = freqs[i][1]['mean']
                v2 = freqs[i][1]['var']
                p2 = gaussian(record[i], m2, v2)
                prob2 = prob2 * p2
            # 离散型属性
            else:
                # 加1法处理，注意没出现的算一次，其他的每个取值频数增加1次
                p1 = (freqs[i][0].get(record[i], 0) + 1) / (less_50 + len(freqs[i][0].
                    keys()))
                prob1 = prob1 * p1

                p2 = (freqs[i][1].get(record[i], 0) + 1) / (more_50 + len(freqs[i][1].
```

```
                    keys()))
                prob2 = prob2 * p2
            # print('##', prob1, prob2)

        if prob2 > prob1:
            label = '>50K'
        if label == record[-1].strip():
            right += 1
    # print(right, len(test_dataset))
    return right/len(test_dataset)


if __name__ == '__main__':
    train_data = load_data('adult.data')
    test_data = load_data('adult.test')       # 含标签

    train_data = precondition(train_data)
    test_data = precondition(test_data)
    attributes_domain = get_attributes_domain(train_data)
    frequencies = get_freq(train_data)
    t2 = time()
    accuracy_train = test(train_data, frequencies)
    accuracy_test = test(test_data, frequencies)
    # print(test_data[0][-1] == '<=50K', test_data[2][-1] == '>50K')
    t3 = time()
    print('Accuracy on training data set: {}%'.format(accuracy_train*100))
    print('Accuracy on testing data set: {}%'.format(accuracy_test*100))
    print('Testing time cost: {:.4}s'.format(t3 - t2))
```

4.2 Results

运行结果如下：

```
D:\Anaconda\python.exe F:/桌面/人工智能/E12_20191127_NB/E12.PY
Accuracy on training data set: 83.26218482233347%
Accuracy on testing data set: 83.14599840304649%
Testing time cost: 3.222s
```

读文件，预处理都是用之前决策树的代码，但是这次没有去掉任何属性 (哪怕他们的含义重复了)。这次主要是增加了 get_freq() 函数，统计各个属性取值在各种分类下出现的频数，用于计算概率。统计时，返回一个 freqs 列表，每一个元素对应一个属性，列表元素也是列表，该列表的元素是两个字典，对于离散属性，第一个字典记录了在分类为第一类的情况下，各个属性取值出现的频数，第二个字典记录了在分类为第二类的情况下，各个属性取值出现的频数。对于连续属性，两个字典分

别记录了在两种分类下，属性取值的均值和方差。具体细节见函数实现。测试时在 test() 函数中计算各种概率，详见代码。

此外，实现时还是用到了加 1 法来平滑数据，因为训练数据集上，在 education 这个属性上取值为 Preschool 的都是"<=50K" 这一类的，所以在另一类上出现的频数为 0，同样还有其他的少数属性取值，所以还是稍微处理了一下。另外填充缺失属性"?" 还是用的和之前一样的方法，但是不知道对于连续属性，用均值填充效果是不是会好一些，也没做测试了。83% 的准确率和决策树的结果差不多，但是朴素贝叶斯的实现的确简单许多，而且运行时间更短。