原理

贝叶斯定理：$P(Y|X)=\frac{P(XY)}{P(X)}=\frac{P(X|Y)P(Y)}{P(X)}$

其中，

$P(X|Y)=P(X^{(1)}=x^{(1)},\cdots,X^{(n)}=x^{(n)}|Y=c_k)\xrightarrow{条件独立假设}=\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k)$

那么分子$=P(Y=c_k)\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k),\quad k=1,2\ldots K$

分母

$=\sum_Y P(XY)=\sum_Y P(X|Y)P(Y)=\sum_k P(Y=c_k)\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k),\quad k=1,2\ldots K$

由此得到朴素贝叶斯分类器可表示为：

$y=f(x)=\underset{c_k}{argmax}\frac{P(Y=c_k)\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k)}{\sum_k P(Y=c_k)\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k)}\propto\underset{c_k}{argmax}P(Y=c_k)\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k)$

三种模型

根据概率的不同计算形式，有以下三种常用的模型：

- 多项式模型：数据服从多项式分布，适用于特征离散的情况。

  先验概率 $P(Y=c_k)=\frac{\sum_{i=1}^{N}I(y_i=c_k)+\lambda}{N+K\lambda},\quad k=1,2,\cdots,K$

  条件概率

  $P(X^{(j)}=a_{jl}|Y=c_k)=\frac{\sum_{i=1}^{N}I\left(x_i^{(j)}=a_{jl},y_i=c_k\right)+\lambda}{\sum_{i=1}^{N}I(y_i=c_k)+S_j\lambda},\quad j=1,2,\cdots,n;\quad l=1,2,\cdots,S_j;\quad k=1,2,\cdots,K$

  其中，$x_i^{(j)}$是第i个样本的第j个特征，a_{jl}是第j个特征的可能取的第l个值，k是样本类别。
- 伯努利模型：数据服从多元伯努利分布，每个特征的取值只能是1和0，适用于特征离散的情况。
- 高斯模型：数据服从高斯分布（正态分布），适用于特征连续的情况。

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

  其中，$\mu_{y_k,i}$表示类别为y_k的样本中，第i维特征的均值，$\sigma_{y_k,i}$表示类别为y_k的样本中，第i维特征的方差。

模型实现

多项式模型

```
import numpy as np


class NB:
    def __init__(self, lambda_):
```

```python
        self.lambda_ = lambda_  # 拉普拉斯平滑
        self.feat_val = [0, 1]  # 文本分类中每一列特征的取值
        self.py = {}
        self.pxy = {}

    def fit(self, X, y):
        N, M = X.shape
        data = np.hstack((X, y.reshape(N, 1)))

        # 统计标签中的类别及其个数，即\sum{I(y_i=c_k)}
        unique_y, counts_y = np.unique(y, return_counts=True)
        y_info = dict(zip(unique_y, counts_y))

        # 对每一个类别进行遍历
        for ck, ck_count in y_info.items():
            # 计算P(Y=c_k)
            self.py['P(Y={})'.format(ck)] = (ck_count + self.lambda_) / (N + len(unique_y) * self.lambda_)

            # 取出标签=ck的所有行
            tmp_data = data[data[:, -1] == ck]

            # 对每一个特征遍历
            for col in range(M):
                # 统计类别为ck且该列特征下每个取值的个数，即\sum{I(x_ij=a_jl,y_i=c_k)}
                unique_feat, counts_feat = np.unique(tmp_data[:, col], return_counts=True)
                feat_info = dict(zip(unique_feat, counts_feat))
                # 如果该类别下的特征的取值全相等，那也需要把其它取值也加入到feat_info中
                if len(feat_info) != len(self.feat_val):
                    for v in self.feat_val:
                        feat_info[v] = feat_info.get(v, 0)
                # 对该特征下的每一个不同取值进行遍历
                for feat_val, feat_count in feat_info.items():
                    # 计算P(X^{j}=a_{j_l}|Y=c_k)
                    self.pxy['P(X({})={}|Y={})'.format(col + 1, feat_val, ck)] = (feat_count + self.lambda_) / (
                        (ck_count + len(feat_info) * self.lambda_))

    def predict(self, x):
        res = {}
        for k, v in self.py.items():
            p = np.log(v)
            ck = k.split('=')[-1][:-1]
            for i in range(len(x)):
                # 计算P(Y=c_k)\prod{P(X^{(j)}=x^{(j)}|Y=c_{k})}
                p = p + np.log(self.pxy['P(X({})={}|Y={})'.format(i + 1, x[i], ck)])
```

```python
                res[ck] = p
        # print(res)

        max_p = float('-inf')
        max_cate = float('-inf')
        for cate, p in res.items():
            if p > max_p:
                max_p = p
                max_cate = cate

        return max_cate, max_p

    def score(self, Xtest, ytest):
        c = 0
        for x, y in zip(Xtest, ytest):
            cate, p = self.predict(x)
            if int(cate) == int(y):
                c += 1
        return c / len(Xtest)

# 垃圾邮件检测
def spam_test():
    # 读入数据
    doc_list = []
    class_list = []
    for filename in ['ham', 'spam']:
        for i in range(1, 26):
            with open("./email/" + filename + '/' + str(i) + '.txt') as f:
                words = f.read()
                words = text_parse(words)
            doc_list.append(' '.join(words))
            if filename == 'ham':
                class_list.append(1)
            else:
                class_list.append(-1)

    # 单词计数
    vec = CountVectorizer()
    words = vec.fit_transform(doc_list)
    words = pd.DataFrame(words.toarray(), columns=vec.get_feature_names())
    # 转为二值，单词出现为0，没出现为1
    words[words > 0] = 1

    # 构建数据集
    X = words.values
    y = np.array(class_list)
    Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2)

    # 训练
    model = NB(lambda_=0.2)
```

```
    model.fit(np.array(Xtrain), np.array(ytrain))

    # 测试
    score = model.score(Xtest, ytest)
    print('正确率: {}'.format(score))
```

高斯模型

```
import numpy as np

class GNB:
    def __init__(self):
        self.parameters = {}
        self.classes = []

    def fit(self, X, y):
        self.classes = list(np.unique(y))

        for c in self.classes:
            # 计算每个类别的平均值,方差,先验概率
            X_Index_c = X[np.where(y == c)]
            X_index_c_mean = np.mean(X_Index_c, axis=0, keepdims=True)
            X_index_c_var = np.var(X_Index_c, axis=0, keepdims=True)
            prior = X_Index_c.shape[0] / X.shape[0]
            self.parameters["class" + str(c)] = {"mean": X_index_c_mean, "var": X_index_c_var, "prior": prior}
        # print(self.parameters)

    def predict(self, X):
        # 取概率最大的类别返回预测值
        output = []
        for y in self.classes:
            # 先验概率
            prior = np.log(self.parameters["class" + str(y)]["prior"])

            # 后验概率:一维高斯分布的概率密度函数
            mean = self.parameters["class" + str(y)]["mean"]
            var = self.parameters["class" + str(y)]["var"]

            eps = 1e-4
            numerator = np.exp(-(X - mean) ** 2 / (2 * var + eps))
            denominator = np.sqrt(2 * np.pi * var + eps)

            # 取对数防止数值溢出
            posterior = np.sum(np.log(numerator / denominator), axis=1, keepdims=True).T
            prediction = prior + posterior
            output.append(prediction)

```

```python
        output = np.reshape(output, (len(self.classes), X.shape[0]))
        prediction = np.argmax(output, axis=0)
        return prediction

    def score(self, X_test, y_test):
        pred = self.predict(X_test)
        right = (y_test - pred == 0.0).sum()

        return right / float(len(X_test))

# 鸢尾花分类
def iris_test():
    iris = load_iris()  # 鸢尾花数据集
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    # 构建数据集
    X, y = data[:, :-1], data[:, -1]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

    # 训练
    model = GNB()
    model.fit(X_train, y_train)

    # 测试
    ck = model.predict(np.array([4.4, 3.2, 1.3, 0.2]).reshape(1, -1))
    print("预测的类别是：{}".format(ck))

    score = model.score(X_test, y_test)
    print('正确率：{}'.format(score))
```

常见面试题

1. 朴素贝叶斯中的“朴素”是什么含义?

   假定所有的特征在数据集中是独立同分布的，但这个假设在现实生活中很不真实，因此很“naive”。

2. 实际项目中，概率值往往是很小的小数，连续微小小数相乘容易造成下溢出使乘积为0，因此可以取自然对数，将连乘变为连加。
3. 朴素贝叶斯的优缺点
   - 优点：朴素贝叶斯模型有着坚实的数学基础，以及稳定的分类效率；对小规模数据表现很好，能够处理多分类任务，适合增量式训练；适用于分类任务，对预测样本进行预测时，过程简单速度快。
   - 缺点：特征条件独立性假设在实际应用中往往是不成立，在特征的属性相关性较小时性能较好；先验概率很多时候是基于假设或者已有的训练数据所得的；对输入数据的表达形式很敏感（离散、连续、值极大极小）。

4. 为什么引入条件独立性假设?

   为了避免贝叶斯定理求解时面临的组合爆炸问题。

   $P(X|Y)=P(X^{(1)}=x^{(1)},\cdots,X^{(n)}=x^{(n)}|Y=c_k)$

   参数个数为$K\prod_{j=1}^{n}S_j$个，这就导致条件概率分布的参数数量为指数级别。

5. 朴素贝叶斯是一个生成模型，它通过学习已知样本，计算出联合概率，再求条件概率。

   - 生成模式：由数据学得联合概率分布，再求出条件概率分布$P(Y|X)$的预测模型；

     常见的生成模型有：朴素贝叶斯、隐马尔可夫模型、高斯混合模型、文档主题生成模型（LDA）、限制玻尔兹曼机

   - 判别模式：由数据学得决策函数或条件概率分布的预测模型

     常见的判别模型有：K近邻、SVM、决策树、感知机、线性判别分析（LDA）、线性回归、传统的神经网络、逻辑斯蒂回归、boosting、条件随机场