Naive Bayes

In GDA, the feature vectors x were continuous, real-valued vectors. Lets now talk about a different learning algorithm in which the x's are discrete-valued.

---

1. Multi-variate Bernoulli Event Model

For our motivating example, consider building an email spam filter using machine learning. Here, we wish to classify messages according to whether they are unsolicited commercial (spam) email, or non-spam email. After learning to do this, we can then have our mail reader automatically filter out the spam messages and perhaps place them in a separate mail folder. Classifying emails is one example of a broader set of problems called **text classification**.

We will represent an email via a feature vector whose length is equal to the number of words in the dictionary. Specifically, if an email contains the i-th word of the dictionary, then we will set $x_i = 1$; otherwise, we let $x_i = 0$. For instance, the vector

$$x = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \begin{matrix} a \\ aardvark \\ aardwolf \\ \vdots \\ buy \\ \vdots \\ zygmurgy \end{matrix}$$

is used to represent an email that contains the words "a" and "buy", but not "aardvark", "aardwolf" or "zygmurgy". The set of words encoded into the feature is called the **vocabulary**, so the dimension of x is equal to the size of the vocabulary.

Actually, rather than looking through an english dictionary for the list of all english words, in practice it is more common to look through our training set and encode in our feature vector only the words that occur at least once there. Apart from reducing the number of words modeled and hence reducing our computational and space requirements, this also has the advantage of allowing us to model/include as a feature many words that may appear in your email (such as "cs229") but that you won't find in dictionary. Sometimes (as in the homework), we also exclude the very high frequency words (which will be words like "the", "of", "and"; these high frequency, "content free" words are called **stop words**) since they occur in so many documents and do little to indicate whether an email is spam or non-spam.

Having chosen our feature vector, we now want to build a discriminative model. So, we have to model $p(x \mid y)$. But if we have, say, a vocabulary of 50000 words, then $x \in \{0, 1\}^{50000}$ (x is a 50000-dimensional vector of 0's and 1's), and if we were to model x explicitly with a multinomial distribution over the 2^{50000} possible outcomes, then we'd end up with a $(2^{50000} - 1)$-dimensional parameter vector. This is clearly too many parameters.

To model $p(x \mid y)$, we will therefore make a very strong assumption. We will assume that the x_i's are conditionally independent given y. This assumption is called the **Naive Bayes (NB) assumption**, and the resulting algorithm is called the **Naive Bayes classifier**.

We now have:

$$\begin{aligned} p(x_1, \cdots, x_{50000} \mid y) &= p(x_1 \mid y)p(x_2 \mid y, x_1)p(x_3 \mid y, x_1, x_2)\cdots p(x_{50000} \mid y, x_1, \cdots, x_{49999}) \\ &= p(x_1 \mid y)p(x_2 \mid y)p(x_3 \mid y)\cdots p(x_{50000} \mid y) \\ &= \prod_{i=1}^{n} p(x_i \mid y) \end{aligned}$$

The first equality simply follows from the usual properties of probabilities, and the second equality used the NB assumption. We note that even though the Naive Bayes assumption is an extremely strong assumptions, the resulting algorithm works well on many problems.

Our model is parameterized by

$$\begin{aligned} \phi_{i \mid y=1} &= p(x_i = 1 \mid y = 1) \\ \phi_{i \mid y=0} &= p(x_i = 1 \mid y = 0) \\ \phi_y &= p(y = 1) \end{aligned}$$

As usual, given a training set $\{(x^{(i)}, y^{(i)}); i = 1, \cdots, m\}$, we can write down the joint likelihood of the data:

$$\mathrm{L}(\phi_y, \phi_{i \mid y=0}, \phi_{i \mid y=1}) = \prod_{i=1}^{m} p(x^{(i)}, y^{(i)})$$

Maximizing this with respect to $\phi_y, \phi_{i \mid y=0}, \phi_{i \mid y=1}$ gives the maximum likelihood estimates:

$$\phi_{j \mid y=1} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}$$

$$\phi_{j \mid y=0} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}}$$

$$\phi_y = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}{m}$$

In the equations above, the " $\wedge$ " symbol means "and". The parameters have a very natural interpretation. For instance, $\phi_{j \mid y=1}$ is just the fraction of the spam ($y = 1$) emails in which word j does appear.

Having fit all these parameters, to make a prediction on a new example with features x, we then simply calculate:

$$p(y=1 \mid x) = \frac{p(x \mid y=1)p(y=1)}{p(x)}$$
$$= \frac{(\prod_{i=1}^{n} p(x_i \mid y=1))\, p(y=1)}{(\prod_{i=1}^{n} p(x_i \mid y=1))\, p(y=1) + (\prod_{i=1}^{n} p(x_i \mid y=0))\, p(y=0)}$$

and pick whichever class has the higher posterior probability.

详细推导：

已知

$$y \sim Bernoulli(\phi_y)$$
$$x_j \mid y=0 \sim Bernoulli(\phi_{j \mid y=0})$$
$$x_j \mid y=1 \sim Bernoulli(\phi_{j \mid y=1})$$

对数似然函数

$$
\begin{aligned}
l(\phi_y, \phi_{j|y=0}, \phi_{j|y=1}) &= log \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}) \\
&= log \prod_{i=1}^{m} p(x^{(i)} \mid y^{(i)}) p(y^{(i)}) \\
&= \sum_{i=1}^{m} log p(x^{(i)} \mid y^{(i)}) + \sum_{i=1}^{m} log\, p(y^{(i)}) \\
&= \sum_{i=1}^{m} log \prod_{j=1}^{n} p(x_j^{(i)} \mid y^{(i)}) + \sum_{i=1}^{m} log\, p(y^{(i)}) \\
&= \sum_{i=1}^{m} log \prod_{j=1}^{n} p(x_j^{(i)} \mid y^{(i)} = 0)^{1-y^{(i)}} \cdot p(x_j^{(i)} \mid y^{(i)} = 1)^{y^{(i)}} + \sum_{i=1}^{m} log\, p(y^{(i)}) \\
&= \sum_{i=1}^{m} \sum_{j=1}^{n} \left((1 - y^{(i)}) log\, p(x_j^{(i)} \mid y^{(i)} = 0) + y^{(i)} log\, p(x_j^{(i)} \mid y^{(i)} = 1) \right) + \sum_{i=1}^{m} log\, p(y^{(i)}) \\
&= \sum_{i=1}^{m} \sum_{j=1}^{n} (1 - y^{(i)}) log\, p(x_j^{(i)} \mid y^{(i)} = 0) + \sum_{i=1}^{m} \sum_{j=1}^{n} y^{(i)} log\, p(x_j^{(i)} \mid y^{(i)} = 1) + \sum_{i=1}^{m} log\, p(y^{(i)})
\end{aligned}
$$

注意，此函数分为三个部分，其中，$\phi_{j|y=0}$ 只与第一部分有关，$\phi_{j|y=1}$ 只与第二部分有关，ϕ_y 只与第三部分有关。

首先，求 $\phi_{j|y=0}$

$$
\begin{aligned}
\bigtriangledown_{\phi_{j\mid y=0}} l(\phi_y, \phi_{j\mid y=0}, \phi_{j\mid y=1}) &= \bigtriangledown_{\phi_{j\mid y=0}} \sum_{i=1}^{m}\sum_{j=1}^{n} (1-y^{(i)}) \log p(x_j^{(i)} \mid y^{(i)}=0) \\
&= \bigtriangledown_{\phi_{j\mid y=0}} \sum_{i=1}^{m}\sum_{j=1}^{n} (1-y^{(i)}) \log \phi_{j\mid y=0}^{x_j^{(i)}} \cdot (1-\phi_{j\mid y=0})^{(1-x_j^{(i)})} \\
&= \bigtriangledown_{\phi_{j\mid y=0}} \sum_{i=1}^{m}\sum_{j=1}^{n} 1\{y^{(i)}=0\} \left(x_j^{(i)} \log \phi_{j\mid y=0} + (1-x_j^{(i)}) \log (1-\phi_{j\mid y=0}) \right) \\
&= \sum_{i=1}^{m} 1\{y^{(i)}=0\} \left(x_j^{(i)} \frac{1}{\phi_{j\mid y=0}} - (1-x_j^{(i)}) \frac{1}{1-\phi_{j\mid y=0}} \right) \\
&= \frac{\sum_{i=1}^{m} 1\{y^{(i)}=0\} x_j^{(i)}}{\phi_{j\mid y=0}} - \frac{\sum_{i=1}^{m} 1\{y^{(i)}=0\}(1-x_j^{(i)})}{1-\phi_{j\mid y=0}}
\end{aligned}
$$

令其为零，即

$$\frac{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} x_j^{(i)}}{\phi_{j \mid y=0}} = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}(1 - x_j^{(i)})}{1 - \phi_{j \mid y=0}}$$

$$\sum_{i=1}^{m} 1\{y^{(i)} = 0\} x_j^{(i)} - \phi_{j \mid y=0} \sum_{i=1}^{m} 1\{y^{(i)} = 0\} x_j^{(i)} = \phi_{j \mid y=0} \sum_{i=1}^{m} 1\{y^{(i)} = 0\}(1 - x_j^{(i)})$$

$$\sum_{i=1}^{m} 1\{y^{(i)} = 0\} 1\{x_j^{(i)} = 1\} = \phi_{j \mid y=0} \sum_{i=1}^{m} 1\{y^{(i)} = 0\}$$

$$\phi_{j \mid y=0} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}}$$

$\phi_{j \mid y=1}$，同理可得。

最后，求 ϕ_y

$$
\begin{aligned}
\nabla_{\phi_y} l(\phi_y, \phi_{j|y=0}, \phi_{j|y=1}) &= \nabla_{\phi_y} \sum_{i=1}^{m} log\, p(y^{(i)}) \\
&= \nabla_{\phi_y} \sum_{i=1}^{m} log\, \phi_y^{y^{(i)}} (1-\phi_y)^{(1-y^{(i)})} \\
&= \nabla_{\phi_y} \sum_{i=1}^{m} \left(y^{(i)} log \phi_y + (1-y^{(i)}) log(1-\phi_y) \right) \\
&= \sum_{i=1}^{m} y^{(i)} \frac{1}{\phi_y} - \sum_{i=1}^{m} (1-y^{(i)}) \frac{1}{1-\phi_y}
\end{aligned}
$$

令其为零，可得

$$\sum_{i=1}^{m} y^{(i)} \frac{1}{\phi_y} = \sum_{i=1}^{m} (1 - y^{(i)}) \frac{1}{1 - \phi_y}$$

$$\sum_{i=1}^{m} y^{(i)} - \phi_y \sum_{i=1}^{m} y^{(i)} = \phi_y \sum_{i=1}^{m} (1 - y^{(i)})$$

$$\sum_{i=1}^{m} 1\{y^{(i)} = 1\} - \phi_y \sum_{i=1}^{m} 1\{y^{(i)} = 1\} = \phi_y \sum_{i=1}^{m} 1\{y^{(i)} = 0\}$$

$$\sum_{i=1}^{m} 1\{y^{(i)} = 1\} = m\phi_y$$

$$\phi_y = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}{m}$$

Lastly, we note that while we have developed the Naive Bayes algorithm mainly for the case of problems where the features x_i are binary-valued, the generalization to where x_i can take values in $\{1, 2, \cdots, k\}$ is straightforward. Here, we would simply model $p(x_i \mid y)$ as multinomial rather than as Bernoulli. Indeed, even if some original inpute attribute (say, the living area of a house, as in our earlier example) were continuous valued, it is quite common to **discretize** it——that is turn it into a small set of discrete values——and apply Naive Bayes.

When the original, continuous-valued attributes are well-modeled by a multivariate normal distribution, discretizing the features and using Naive Bayes (instead of GDA) will often result in a better classifier.

2. Laplace Smoothing

The Naive Bayes alogorithm as we have described it will work fairly well for many problems, but there is a simple change that makes it work much better, especially for text classification. Lets briefly discuss a problem with the algorithm in its current form, and then talk about how we can fix it.

Consider spam/email classification, and lets suppose that, after completing CS229 and having done excellent work on the project, you decide around June 2003 to submit the work you did to the NIPS conference for publication.(NIPS is one of the top machine learning conferences, and the deadline for submitting a paper is typically in late June or early July.) Because you end up discussing the conference in your emails, you also start getting messages with the word "nips" in it. But this is your first NIPS paper, and until this time, you had not previously seem any emails containing the word "nips"; in particular "nips" did not ever appear in your training set of spam/non-spam emails. Assuming that "nips" was the 35000th word in the dictionary, your Naive Bayes spam filter therefore had picked its maximum likelihood estimates of the parameters $\phi_{35000 \mid y}$ to be

$$\phi_{35000 \mid y=1} = \frac{\sum_{i=1}^{m} 1\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} = 0$$

$$\phi_{35000 \mid y=0} = \frac{\sum_{i=1}^{m} 1\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}} = 0$$

I.e., because it has never seen "nips" before in either spam or non-spam training examples, it thinks the probability of seeing it in either type of email is zero. Hence, when trying to decide if one of these messages containing "nips" is spam, it calculates the class posterior probailities, and obtains

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

This is because each of the terms "$\prod_{i=1}^{n} p(x_i \mid y)$" includes a term $p(x_{35000} \mid y) = 0$ that is multiplied into it. Hence, our algorithm obtains $0/0$, and doesn't know how to make a prediction.

Stating the problem more broadly, it is statistically a bad idea to estimate the probability of some event to be zero just because you haven't seen it before in your finite training set. Take the problem of estimating the mean of a multinomial random variable z taking values in $\{1, \cdots, k\}$. We can parameterize our multinomial with $\phi_i = p(z = i)$. Given a set of m independent observations $\{z^{(1)}, \cdots, z^{(m)}\}$, the maximum likelihood estimates are given by

$$\phi_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\}}{m}$$

As we saw previously, if we were to use these maximum likelihood estimates, then some of the ϕ_j's might end up as zero, which was a problem. To avoid this, we can use **Laplace smoothing**, which replaces the above estimate with

$$\phi_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\} + 1}{m + k}$$

Here we've added 1 to the numerator, and k to the denominator. Note that $\sum_{j=1}^{k} \phi_j = 1$ still holds, which is a desirable property since the ϕ_j's are estimates for probabilities that we know must sum to 1. Also, $\phi_j \neq 0$ for all values of j, sovling our problem of probabilities being estimated as zero. Under certain (arguably quite strong) conditions, it can be shown that the Laplace smoothing actually gives the optimal estimator of the ϕ_j's.

Returning to our Naive Bayes classifier, with Laplace smoothing, we therefore obtain the following estimates of the parameters:

$$\phi_{j \mid y=1} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} + 2}$$

$$\phi_{j \mid y=0} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} + 2}$$

In practice, it usually doesn't matter much whether we apply Laplace smoothing to ϕ_y or not, since we will typically have a fair fraction each of spam and non-spam messages, so ϕ_y will be a reasonable estimate of $p(y = 1)$ and will be quite far from 0 anyway.

3. Demo

In [1]:
```
import numpy as np
```

建立一个朴素贝叶斯分类器，来判断某个句子是否是侮辱性句子。

In [2]:
```
sentences = [['my','dog','has','flea','problems','help','please'],
        ['maybe','not','take','him','to','dog','park','stupid'],
        ['my','dalmation','is','so','cute','I','love','him'],
        ['stop','posting','stupid','worthless','garbage'],
        ['mr','licks','ate','my','steak','how','to','stop','him'],
        ['quit','buying','worthless','dog','food','stupid']]

labels = [0, 1, 0, 1, 0, 1]
```

In [3]:
```
# 建立词汇表
def createVocab(sentences):

    vocab = set()
    for sentence in sentences:
        # 取并集
        vocab = vocab | set(sentence)

    return list(vocab)
```

In [4]:
```
# 对输入的句子进行向量化
def sentence2Vec(vocab, sentence):

    vec = np.zeros(len(vocab))
    for word in sentence:
        if word in vocab:
            vec[vocab.index(word)] = 1
        else:
            print("{}不在词汇表中".format(word))
    return vec
```

In [5]:
```
# 准备数据
def loadData(sentences, labels, vocab):

    # 准备 X_train，y_train，其中 X_train 是一个 (m, len(vocab)) 的矩阵
    # m 为训练数据的个数，即句子的个数
    X_train = np.zeros((len(sentences), len(vocab)))
    for index, sentence in enumerate(sentences):
        X_train[index] = sentence2Vec(vocab, sentence)

    y_train = np.array(labels)

    return X_train, y_train
```

In [6]:
```
def trainNB(X_train, y_train):

    # m 为训练数据的个数
    m = X_train.shape[0]

    num_pos = y_train.sum() # 标签为1的样本数
    num_neg = m - num_pos   # 标签为0的样本数

    phi_pos = (X_train[y_train == 1, :].sum(axis=0) + 1) / (num_pos + 2) # 记录每个词在正样本出现的概率
    phi_neg = (X_train[y_train == 0, :].sum(axis=0) + 1) / (num_neg + 2) # 记录每个词在负样本出现的概率

    phi = num_pos / m # 记录phi_y

    return phi_pos, phi_neg, phi
```

在利用训练得到的参数进行预测的时候，会出现下溢出的问题。这是由于太多很小的数相乘导致的。例如，在计算 $\prod_{i=1}^{n} p(x_i \mid y = 1)$ 的时候，由于大部分因子都非常小，所以程序会出现下溢出或者得到不正确的答案。其中一种解决办法是对乘积取自然对数。

因此，可得

$$
\begin{aligned}
log\, p(y=1 \mid x) &= log \frac{p(x \mid y=1)p(y=1)}{p(x)} \\
&= log\, p(x \mid y=1) + log\, p(y=1) - log\, p(x)
\end{aligned}
$$

下溢出的问题出现在 $log\, p(x \mid y=1)$，所以下面只关注 $log\, p(x \mid y=1)$ 的计算，即

$$
\begin{aligned}
log\, p(x \mid y=1) &= log \prod_{i=1}^{n} p(x_i \mid y=1) \\
&= \sum_{i=1}^{n} log\, p(x_i \mid y=1) \\
&= \sum_{i=1}^{n} log\, \phi_{i \mid y=1}^{x_i} \cdot (1-\phi_{i \mid y=1})^{(1-x_i)} \\
&= \sum_{i=1}^{n} \left(x_i log\, \phi_{i \mid y=1} + (1-x_i) log\, (1-\phi_{i \mid y=1}) \right)
\end{aligned}
$$

In [7]:
```
def predictNB(matrix, state):

    predictions = np.zeros(matrix.shape[0])

    phi_pos, phi_neg, phi = state

    pos = matrix * phi_pos + (1 - matrix) * (1 - phi_pos)
    p_pos = np.sum(np.log(pos), axis=1) + np.log(phi)

    neg = matrix * phi_neg + (1 - matrix) * (1 - phi_neg)
    p_neg = np.sum(np.log(neg), axis=1) + np.log(1-phi)

    # p(y=1|x) 与 p(y=0|x) 的分母是一样的，因此只比较分子
    predictions[p_pos > p_neg] = 1

    return predictions
```

In [8]:
```
vocab = createVocab(sentences)
X_train, y_train = loadData(sentences, labels, vocab)
state = trainNB(X_train, y_train)
predictions = predictNB(X_train, state)
predictions
```

Out[8]:
```
array([0., 1., 0., 1., 0., 1.])
```

In [9]:
```
# 测试的例子
sentences = [['love', 'my', 'dalmation'],
             ['stupid', 'garbage']]
labels = [0, 1]

X_train, y_train = loadData(sentences, labels, vocab)
predictions = predictNB(X_train, state)
predictions
```

Out[9]:
```
array([0., 1.])
```

In [ ]: