Lab 8: Naive Bayes

Simple Probabilistic Classifiers

Based on Neil D. Lawrence MLAI2015 version

Modified by Haiping Lu on 9 November 2018

Machine learning problems normally involve a prediction function and an objective function. So far in the course we've mainly focussed on the case where the prediction function was over the real numbers, so the codomain of the functions, $f(\mathbf{X})$ was the real numbers or sometimes real vectors. The classification problem consists of predicting whether or not a particular example is a member of a particular class. So we may want to know if a particular image represents a digit 6 or if a particular user will click on a given advert. These are classification problems, and they require us to map to *yes* or *no* answers. That makes them naturally discrete mappings.

In binary classification, we are given an input vector, $\mathbf{x}$, and an associated label, y which either takes the value 0 or 1.

Our focus has been on models where the objective function is inspired by a probabilistic analysis of the problem. In particular we've argued that we answer questions about the data set by placing probability distributions over the various quantities of interest. For the case of binary classification this will normally involve introducing probability distributions for discrete variables. Such probability distributions, are in some senses easier than those for continuous variables, in particular we can represent a probability distribution over y, where y is binary, with one value. If we specify the probability that $y = 1$ with a number that is between 0 and 1, i.e. let's say that $P(y = 1) = \pi$ (here we don't mean π the number, we are setting π to be a variable) then we can specify the probability distribution through a table.

The class label y	Probability of 0	Probability of 1
$P(y)$	$(1 - \pi)$	π

Mathematically we can use a trick to implement this same table. We can use the value y as a mathematical switch and write that

$$P(y) = \pi^y (1 - \pi)^{(1-y)}$$

where our probability distribution is now written as a function of y. This probability distribution is known as the Bernoulli distribution (http://en.wikipedia.org/wiki/Bernoulli_distribution). The Bernoulli distribution is a clever trick for mathematically switching between two probabilities if we were to write it as code it would be better described as

```
def bernoulli(x, pi):
    if y_i == 1:
        return pi(x)
    else:
        return 1-pi(x)
```

If we insert $y = 1$ then the function is equal to π, and if we insert $y = 0$ then the function is equal to $1 - \pi$. So the function recreates the table for the distribution given above.

The probability distribution is named for Jacob Bernoulli (http://en.wikipedia.org/wiki/Jacob_Bernoulli), the swiss mathematician. In his book Ars Conjectandi he considered the distribution and the result of a number of 'trials' under the Bernoulli distribution to form the *binomial* distribution. Below is the page where he considers Pascal's triangle in forming combinations of the Bernoulli distribution to realise the binomial distribution for the outcome of positive trials.

In [1]:

```
import pods
pods.notebook.display_google_book('CF4UAAAAQAAJ', 87)
```

```
/home/haiping/anaconda3/lib/python3.6/importlib/_bootstrap.py:219: RuntimeWa
rning: numpy.dtype size changed, may indicate binary incompatibility. Expect
ed 96, got 88
  return f(*args, **kwds)
```

Maximum Likelihood in the Bernoulli Distribution

Maximum likelihood in the Bernoulli distribution is straightforward. Let's assume we have data, $\mathbf{y}$ which consists of a vector of binary values of length n. If we assume each value was sampled independently from the Bernoulli distribution, conditioned on the parameter π then our joint probability density has the form

$$p(\mathbf{y}|\pi) = \prod_{i=1}^{n} \pi^{y_i} (1 - \pi)^{1-y_i}.$$

As normal in maximum likelihood we consider the negative log likelihood as our objective,

$$E(\pi) = -\log p(\mathbf{y}|\pi) = -\sum_{i=1}^{n} y_i \log \pi - \sum_{i=1}^{n} (1 - y_i) \log(1 - \pi),$$

and we seek the gradient with respect to the parameter π.

$$\frac{\mathrm{d}E(\pi)}{\mathrm{d}\pi} = -\frac{\sum_{i=1}^{n} y_i}{\pi} + \frac{\sum_{i=1}^{n}(1 - y_i)}{1 - \pi},$$

and as normal we look for a stationary point for the log likelihood by setting this derivative to zero,

$$0 = -\frac{\sum_{i=1}^{n} y_i}{\pi} + \frac{\sum_{i=1}^{n}(1 - y_i)}{1 - \pi},$$

rearranging we form

$$(1 - \pi) \sum_{i=1}^{n} y_i = \pi \sum_{i=1}^{n} (1 - y_i),$$

which implies

$$\sum_{i=1}^{n} y_i = \pi \left(\sum_{i=1}^{n} (1 - y_i) + \sum_{i=1}^{n} y_i \right),$$

and now we recognise that $\sum_{i=1}^{n}(1 - y_i) + \sum_{i=1}^{n} y_i = n$ so we have

$$\pi = \frac{\sum_{i=1}^{n} y_i}{n}$$

so in other words we estimate the probability associated with the Bernoulli by setting it to the number of observed positives, divided by the total length of y. This makes intiutive sense. If I asked you to estimate the probability of a coin being heads, and you tossed the coin 100 times, and recovered 47 heads, then the estimate of the probability of heads should be $\frac{47}{100}$.

Exercise 1

Show that the maximume likelihood solution we have found is a *minimum* for our objective.

Naive Bayes Classifiers

In the first lecture in this course we talked about placing probability distributions (or densities) over all the variables of interest, our first classification algorithm will do just that. We will consider how to form a classification by making assumptions about the *joint* density of our observations. We need to make assumptions to reduce the number of parameters we need to optimise. In the ideal world, given label data $\mathbf{y}$ and the inputs $\mathbf{X}$ we should be able to specify the joint density of all potential values of $\mathbf{y}$ and $\mathbf{X}$, $p(\mathbf{y}, \mathbf{X})$. If $\mathbf{X}$ and $\mathbf{y}$ are our training data, and we can somehow extend our density to incorporate future test data (by augmenting $\mathbf{y}$ with a new observation y^* and $\mathbf{X}$ with the corresponding inputs, $\mathbf{x}^*$, then we can answer any given question about a future test point y^* given its covariates $\mathbf{x}^*$ by conditioning on the training variables to recover,

$$p(y^*|\mathbf{X}, \mathbf{y}, \mathbf{x}^*),$$

We can compute this distribution using the product and sum rules. However, to specify this density we must give the probability associated with all possible combinations of $\mathbf{y}$ and $\mathbf{X}$. There are 2^n possible combinations for the vector $\mathbf{y}$ and the probability for each of these combinations must be jointly specified along with the joint density of the matrix $\mathbf{X}$, as well as being able to *extend* the density for any chosen test location $\mathbf{x}^*$.

In naive Bayes we make certain simplifying assumptions that allow us to perform all of the above in practice.

Data Conditional Independence

If we are given model parameters $\boldsymbol{\theta}$ we assume that conditioned on all these parameters that all data points in the model are independent. In other words we have,

$$p(y^*, \mathbf{x}^*, \mathbf{y}, \mathbf{X}|\boldsymbol{\theta}) = p(y^*, \mathbf{x}^*|\boldsymbol{\theta}) \prod_{i=1}^{n} p(y_i, \mathbf{x}_i|\boldsymbol{\theta}).$$

This is a conditional independence assumption because we are not assuming our data are purely independent. If we were to assume that, then there would be nothing to learn about our test data given our training data. We are assuming that they are independent *given* our parameters, $\boldsymbol{\theta}$. We made similar assumptions for regression, where our parameter set included $\mathbf{w}$ and σ^2. Given those parameters we assumed that the density over $\mathbf{y}, y^*$ was *independent*. Here we are going a little further with that assumption because we are assuming the *joint* density of $\mathbf{y}$ and $\mathbf{X}$ is independent across the data given the parameters.

Feature Conditional Independence

The assumption that is particular to naive Bayes is to now consider that the *features* are also conditionally independent, but not only given the parameters. We assume that the features are independent given the parameters *and* the label. So for each data point we have

$$p(\mathbf{x}_i|y_i, \boldsymbol{\theta}) = \prod_{j=1}^{p} p(x_{i,j}|y_i, \boldsymbol{\theta})$$

where p is the dimensionality of our inputs.

Marginal Density for y_i

We now have nearly all of the components we need to specify the full joint density. However, the feature conditional independence doesn't yet give us the joint density over $p(y_i, \mathbf{x}_i)$ which is required to subsitute in to our data conditional independence to give us the full density. To recover the joint density given the conditional distribution of each feature, $p(x_{i,j}|y_i, \boldsymbol{\theta})$, we need to make use of the product rule and combine it with a marginal density for y_i,

$$p(x_{i,j}, y_i|\boldsymbol{\theta}) = p(x_{i,j}|y_i, \boldsymbol{\theta})p(y_i).$$

Because y_i is binary the *Bernoulli* density makes a suitable choice for our prior over y_i,

$$p(y_i|\pi) = \pi^{y_i}(1 - \pi)^{1-y_i}$$

where π now has the interpretation as being the *prior* probability that the classification should be positive.

Joint Density for Naive Bayes

This allows us to write down the full joint density of the training data,

$$p(\mathbf{y}, \mathbf{X}|\boldsymbol{\theta}, \pi) = \prod_{i=1}^{n} \prod_{j=1}^{p} p(x_{i,j}|y_i, \boldsymbol{\theta})p(y_i|\pi)$$

which can now be fit by maximum likelihood. As normal we form our objective as the negative log likelihood,

$$E(\boldsymbol{\theta}, \pi) = -\log p(\mathbf{y}, \mathbf{X}|\boldsymbol{\theta}, \pi) = -\sum_{i=1}^{n} \sum_{j=1}^{p} \log p(x_{i,j}|y_i, \boldsymbol{\theta}) - \sum_{i=1}^{n} \log p(y_i|\pi),$$

which we note *decomposes* into two objective functions, one which is dependent on π alone and one which is dependent on $\boldsymbol{\theta}$ alone so we have,

$$E(\pi, \boldsymbol{\theta}) = E(\boldsymbol{\theta}) + E(\pi).$$

Since the two objective functions are separately dependent on the parameters π and $\boldsymbol{\theta}$ we can minimize them independently. Firstly, minimizing the Bernoulli likelihood over the labels we have,

$$E(\pi) = -\sum_{i=1}^{n} \log p(y_i|\pi) = -\sum_{i=1}^{n} y_i \log \pi - \sum_{i=1}^{n} (1 - y_i) \log(1 - \pi)$$

which we already minimized above recovering

$$\pi = \frac{\sum_{i=1}^{n} y_i}{n}.$$

We now need to minimize the objective associated with the conditional distributions for the features,

$$E(\boldsymbol{\theta}) = -\sum_{i=1}^{n}\sum_{j=1}^{p} \log p(x_{i,j}|y_i, \boldsymbol{\theta}),$$

which necessarily implies making some assumptions about the form of the conditional distributions. The right assumption will depend on the nature of our input data. For example, if we have an input which is real valued, we could use a Gaussian density and we could allow the mean and variance of the Gaussian to be different according to whether the class was positive or negative and according to which feature we were measuring. That would give us the form,

$$p(x_{i,j}|y_i, \boldsymbol{\theta}) = \frac{1}{\sqrt{2\pi\sigma_{y_i,j}^2}} \exp\left(-\frac{(x_{i,j} - \mu_{y_i,j})^2}{\sigma_{y_i,j}^2}\right),$$

where $\sigma_{1,j}^2$ is the variance of the density for the jth output and the class $y_i = 1$ and $\sigma_{0,j}^2$ is the variance if the class is 0. The means can vary similarly. Our parameters, $\boldsymbol{\theta}$ would consist of all the means and all the variances for the different dimensions.

Exercise Question 2

Write down the negative log likelihood of the Gaussian density over a vector of variables $\mathbf{x}$. Assume independence between each variable. Minimize this objective to obtain the maximum likelihood solution of the form.

$$\mu = \frac{\sum_{i=1}^n x_i}{n}$$
$$\sigma^2 = \frac{\sum_{i=1}^n (x_i - \mu)^2}{n}$$

If the input data was *binary* then we could also make use of the Bernoulli distribution for the features. For that case we would have the form,

$$p(x_{i,j}|y_i, \boldsymbol{\theta}) = \theta_{y_i,j}^{x_{i,j}}(1 - \theta_{y_i,j})^{(1-x_{i,j})},$$

where $\theta_{1,j}$ is the probability that the jth feature is on if y_i is 1.

In either case, maximum likelihood fitting would proceed in the same way. The objective has the form,

$$E(\boldsymbol{\theta}) = -\sum_{j=1}^{p}\sum_{i=1}^{n} \log p(x_{i,j}|y_i, \boldsymbol{\theta}),$$

and if, as above, the parameters of the distributions are specific to each feature vector (we had means and variances for each continuous feature, and a probability for each binary feature) then we can use the fact that these parameters separate into disjoint subsets across the features to write,

$$E(\boldsymbol{\theta}) = -\sum_{j=1}^{p}\sum_{i=1}^{n} \log p(x_{i,j}|y_i, \boldsymbol{\theta}_j)$$
$$\sum_{j=1}^{p} E(\boldsymbol{\theta}_j),$$

which means we can minimize our objective on each feature independently.

These characteristics mean that naive Bayes scales very well with big data. To fit the model we consider each feature in turn, we select the positive class and fit parameters for that class, then we select each negative class and fit features for that class. We have code below.

Movie Body Count Data

First we will load in the movie body count data. Our aim will be to predict whether a movie is rated R or not given the attributes in the data. We will predict on the basis of year, body count and movie genre. The genres in the CSV file are stored as a list in the following form:

```
Biography|Action|Sci-Fi
```

First we have to do a little work to extract this form and turn it into a vector of binary values. Let's first load in and remind ourselves of the data.

In [2]:

```
data = pods.datasets.movie_body_count()['Y']
data.head()
```

```
Acquiring resource: movie_body_count

Details of data:
Data scraped from www.MovieBodyCounts.com and www.imdb.com using scripts pro
vided on a github repository (in both Python and R) at https://github.com/mo
rpionZ/R-vs-Python/tree/master/Deadliest%20movies%20scrape/code. (https://gi
thub.com/morpionZ/R-vs-Python/tree/master/Deadliest%20movies%20scrape/code.)
 This script pulls down the scraped data.

Please cite:
Simon Garnier and Randy Olson, Blog Post: R vs Python Round 2, February 2nd
 2014 (http://www.theswarmlab.com/r-vs-python-round-2-22/)

After downloading the data will take up 536272 bytes of space.

Data will be stored in /home/haiping/ods_data_cache/movie_body_count.

Do you wish to proceed with the download? [yes/no]
yes
Downloading  https://github.com/sjmgarnier/R-vs-Python/raw/master/Deadliest%
20movies%20scrape/code/film-death-counts-Python.csv (https://github.com/sjmg
arnier/R-vs-Python/raw/master/Deadliest%20movies%20scrape/code/film-death-co
unts-Python.csv) -> /home/haiping/ods_data_cache/movie_body_count/film-death
-counts-Python.csv
[==============================]   0.366/0.366MB
```

Out[2]:

	Film	Year	Body_Count	MPAA_Rating	Genre	Director	
0	24 Hour Party People	2002	7	R	Biography|Comedy|Drama|Music	Michael Winterbottom	P
1	3:10 to Yuma	2007	45	R	Adventure|Crime|Drama|Western	James Mangold	
2	300	2006	0	R	Action|Fantasy|History|War	Zack Snyder	
3	8MM	1999	7	R	Crime|Mystery|Thriller	Joel Schumacher	
4	The Abominable Dr. Phibes	1971	10	PG-13	Fantasy|Horror	Robert Fuest	

Now we will convert this data into a form which we can use as inputs `X`, and labels `y`.

In [3]:

```
import pandas as pd
import numpy as np
X = data[['Year', 'Body_Count']]
y = data['MPAA_Rating']=='R' # set label to be positive for R rated films.

# Create series of movie genres with the relevant index
s = data['Genre'].str.split('|').apply(pd.Series, 1).stack()
s.index = s.index.droplevel(-1) # to line up with df's index

# Extract from the series the unique list of genres.
genres = s.unique()

# For each genre extract the indices where it is present and add a column to X
for genre in genres:
    index = s[s==genre].index.tolist()
    X[genre] = np.zeros(X.shape[0])
    X[genre][index] = np.ones(len(index))
```

```
/home/haiping/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1
6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/s
table/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pand
as-docs/stable/indexing.html#indexing-view-versus-copy)
  app.launch_new_instance()
/home/haiping/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1
7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/s
table/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pand
as-docs/stable/indexing.html#indexing-view-versus-copy)
/home/haiping/anaconda3/lib/python3.6/site-packages/pandas/core/series.py:83
1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/s
table/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pand
as-docs/stable/indexing.html#indexing-view-versus-copy)
  self._set_labels(key, value)
/home/haiping/anaconda3/lib/python3.6/site-packages/IPython/core/interactive
shell.py:2910: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/s
table/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pand
as-docs/stable/indexing.html#indexing-view-versus-copy)
  exec(code_obj, self.user_global_ns, self.user_ns)
```

This has given us a new data frame `X` which contains the different genres in different columns.

In [4]:

```
X.describe()
```

Out[4]:

	Year	Body_Count	Biography	Comedy	Drama	Music	Adventure
count	421.000000	421.000000	421.000000	421.000000	421.000000	421.000000	421.000000
mean	1996.491686	53.287411	0.026128	0.152019	0.384798	0.011876	0.275534
std	10.913210	82.068035	0.159706	0.359466	0.487126	0.108459	0.447315
min	1949.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	1991.000000	11.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	2000.000000	28.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	2005.000000	61.000000	0.000000	0.000000	1.000000	0.000000	1.000000
max	2009.000000	836.000000	1.000000	1.000000	1.000000	1.000000	1.000000

8 rows × 23 columns

We can now specify the naive Bayes model. For the genres we want to model the data as Bernoulli distributed, and for the year and body count we want to model the data as Gaussian distributed. We set up two data frames to contain the parameters for the rows and the columns below.

In [5]:

```

# assume data is binary or real.
# this list encodes whether it is binary or real (1 for binary, 0 for real)
binary_columns = genres
real_columns = ['Year', 'Body_Count']
Bernoulli = pd.DataFrame(data=np.zeros((2,len(binary_columns))), columns=binary_columns, i
Gaussian = pd.DataFrame(data=np.zeros((4,len(real_columns))), columns=real_columns, index=
```

Now we have the data in a form ready for analysis, let's construct our data matrix.

In [6]:

```
num_train = 200
indices = np.random.permutation(X.shape[0])
train_indices = indices[:num_train]
test_indices = indices[num_train:]
X_train = X.loc[train_indices]
y_train = y.loc[train_indices]
X_test = X.loc[test_indices]
y_test = y.loc[test_indices]
```

And we can now train the model. For each feature we can make the fit independently. The fit is given by either counting the number of positives (for binary data) which gives us the maximum likelihood solution for the Bernoulli. Or by computing the empirical mean and variance of the data for the Gaussian, which also gives us the maximum likelihood solution.

In [7]:

```
for column in X_train:
    if column in Gaussian:
        Gaussian[column]['mu_0'] = X_train[column][~y].mean()
        Gaussian[column]['mu_1'] = X_train[column][y].mean()
        Gaussian[column]['sigma2_0'] = X_train[column][~y].var(ddof=0)
        Gaussian[column]['sigma2_1'] = X_train[column][y].var(ddof=0)
    if column in Bernoulli:
        Bernoulli[column]['theta_0'] = X_train[column][~y].sum()/(~y).sum()
        Bernoulli[column]['theta_1'] = X_train[column][y].sum()/(y).sum()
```

We can examine the nature of the distributions we've fitted to the model by looking at the entries in these data frames.

In [8]:

```
Bernoulli
```

Out[8]:

	Biography	Comedy	Drama	Music	Adventure	Crime	Western	Action	Fant
theta_0	0.005128	0.076923	0.102564	0.0	0.241026	0.112821	0.025641	0.353846	0.097
theta_1	0.022124	0.070796	0.252212	0.0	0.044248	0.278761	0.008850	0.261062	0.030

2 rows × 21 columns

In [9]:

```
Gaussian
```

Out[9]:

	Year	Body_Count
mu_0	1992.677419	49.591398
sigma2_0	191.014221	4056.263152
mu_1	2000.672897	49.177570
sigma2_1	28.930387	7641.809590

The final model parameter is the prior probability of the positive class, π, which is computed by maximum likelihood.

In [10]:

```
prior = float(y_train.sum())/len(y_train)
```

Making Predictions

Naive Bayes has given us the class conditional densities: $p(\mathbf{x}_i|y_i, \boldsymbol{\theta})$. To make predictions with these densities we need to form the distribution given by

$$P(y^*|\mathbf{y}, \mathbf{X}, \mathbf{x}^*, \boldsymbol{\theta})$$

This can be computed by using the product rule. We know that

$$P(y^*|\mathbf{y}, \mathbf{X}, \mathbf{x}^*, \boldsymbol{\theta})p(\mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta}) = p(y*, \mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta})$$

implying that

$$P(y^*|\mathbf{y}, \mathbf{X}, \mathbf{x}^*, \boldsymbol{\theta}) = \frac{p(y*, \mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta})}{p(\mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta})}$$

and we've already defined $p(y^*, \mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta})$ using our conditional independence assumptions above

$$p(y^*, \mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta}) = \prod_{j=1}^{p} p(x_j^*|y^*, \boldsymbol{\theta})p(y^*|\pi) \prod_{i=1}^{n} \prod_{j=1}^{p} p(x_{i,j}|y_i, \boldsymbol{\theta})p(y_i|\pi)$$

The other required density is

$$p(\mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta})$$

which can be found from

$$p(y^*, \mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta})$$

using the *sum rule* of probability,

$$p(\mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta}) = \sum_{y^*=0}^{1} p(y^*, \mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta}).$$

Because of our independence assumptions that is simply equal to

$$p(\mathbf{y}, \mathbf{X}, \mathbf{x}^*|\boldsymbol{\theta}) = \sum_{y^*=0}^{1} \prod_{j=1}^{p} p(x_j^*|y_i^*, \boldsymbol{\theta})p(y^*|\pi) \prod_{i=1}^{n} \prod_{j=1}^{p} p(x_{i,j}|y_i, \boldsymbol{\theta})p(y_i|\pi).$$

Substituting both forms in to recover our distribution over the test label conditioned on the training data we have,

$$P(y^*|\mathbf{y}, \mathbf{X}, \mathbf{x}^*, \boldsymbol{\theta}) = \frac{\prod_{j=1}^{p} p(x_j^*|y_i^*, \boldsymbol{\theta})p(y^*|\pi)\prod_{i=1}^{n}\prod_{j=1}^{p} p(x_{i,j}|y_i, \boldsymbol{\theta})p(y_i|\pi)}{\sum_{y^*=0}^{1}\prod_{j=1}^{p} p(x_j^*|y_i^*, \boldsymbol{\theta})p(y^*|\pi)\prod_{i=1}^{n}\prod_{j=1}^{p} p(x_{i,j}|y_i, \boldsymbol{\theta})p(y_i|\pi)}$$

and we notice that all the terms associated with the training data actually cancel, the test prediction is *conditionally independent* of the training data *given* the parameters. This is a result of our conditional independence assumptions over the data points.

$$p(y^*|\mathbf{x}^*, \boldsymbol{\theta}) = \frac{\prod_{j=1}^{p} p(x_j^*|y_i^*, \boldsymbol{\theta})p(y^*|\pi)}{\sum_{y^*=0}^{1}\prod_{j=1}^{p} p(x_j^*|y_i^*, \boldsymbol{\theta})p(y^*|\pi)}$$

This formula is also fairly straightforward to implement. First we implement the log probabilities for the Gaussian density.

In [11]:

```
def log_gaussian(x, mu, sigma2):
    return -0.5* np.log(2*np.pi*sigma2)-((x-mu)**2)/(2*sigma2)
```

Now for any test point we compute the joint distribution of the Gaussian features by *summing* their log probabilities. Working in log space can be a considerable advantage over computing the probabilities directly: as the number of features we include goes up, because all the probabilities are less than 1, the joint probability will become smaller and smaller, and may be difficult to represent accurately (or even underflow). Working in log space can ameliorate this problem. We can also compute the log probability for the Bernoulli distribution.

In [12]:

```
def log_bernoulli(x, theta):
    return x*np.log(theta) + (1-x)*np.log(1-theta)
```

Before we proceed, let's just pause and think for a moment what will happen if `theta` here is either zero or one. This will result in $\log 0 = -\infty$ and cause numerical problems. This definitely can happen in practice. If some of the features are rare or very common across the data set then the maximum likelihood solution could find values of zero or one respectively. Such values are problematic because they cause posterior probabilities of class membership of either one or zero. In practice we deal with this using *Laplace smoothing* (which actually has an interpretation as a Bayesian fit of the Bernoulli distribution. Laplace used an example of the sun rising each day, and a wish to predict the sun rise the following day to describe his idea of smoothing, which can be found at the bottom of following page from Laplace's 'Essai Philosophique ...'

In [13]:

```
pods.notebook.display_google_book('1YQPAAAAQAAJ', page='PA16')
```

Laplace suggests that when computing the probability of an event where a success or failure is rare (he uses an example of the sun rising across the last 5,000 years or 1,826,213 days) that even though only successes have been observed (in the sun rising case) that the odds for tomorrow shouldn't be given as

$$\frac{1,826,213}{1,826,213} = 1$$

but rather by adding one to the numerator and two to the denominator,

$$\frac{1,826,213+1}{1,826,213+2} = 0.99999945.$$

This technique is sometimes called a 'pseudocount technique' because it has an intepretation of assuming some observations before you start, it's as if instead of observing $\sum_i y_i$ successes you have an additional success, $\sum_i y_i + 1$ and instead of having observed n events you've observed $n+2$. So we can think of Laplace's idea saying (before we start) that we have 'two observations worth of belief, that the odds are 50/50', because before we start (i.e. when $n=0$) our estimate is 0.5, yet because the effective n is only 2, this estimate is quickly overwhelmed by data. Laplace used ideas like this a lot, and it is known as his 'principle of insufficient reason'. His idea was that in the absence of knowledge (i.e. before we start) we should assume that all possible outcomes are equally likely. This idea has a modern counterpart, known as the principle of maximum entropy (http://en.wikipedia.org/wiki/Principle_of_maximum_entropy). A lot of the theory of this approach was developed by Ed Jaynes (http://en.wikipedia.org/wiki/Edwin_Thompson_Jaynes), who according to his erstwhile collaborator and friend, John Skilling, learnt French as an undergraduate by reading the works of Laplace. Although John also related that Jaynes's spoken French was not up to the standard of his scientific French. For me Ed Jaynes's work very much carries on the tradition of Laplace into the modern era, in particular his focus on Bayesian approaches. I'm very proud to have met those that knew and worked with him. It turns out that Laplace's idea also has a Bayesian interpretation (as Laplace understood), it comes from assuming a particular prior density for the parameter π, but we won't explore that interpretation for the moment, and merely choose to estimate the probability as,

$$\pi = \frac{\sum_{i=1}^{n} y_i + 1}{n+2}$$

to prevent problems with certainty causing numerical issues and misclassifications. Let's refit the Bernoulli features now.

In [14]:

```
# fit the Bernoulli with Laplace smoothing.
for column in X_train:
    if column in Bernoulli:
        Bernoulli[column]['theta_0'] = (X_train[column][~y].sum() + 1)/((~y).sum() + 2)
        Bernoulli[column]['theta_1'] = (X_train[column][y].sum() + 1)/((y).sum() + 2)
```

That places us in a position to write the prediction function.

In [15]:

```
def predict(X_test, Gaussian, Bernoulli, prior):
    log_positive = pd.Series(data = np.zeros(X_test.shape[0]), index=X_test.index)
    log_negative = pd.Series(data = np.zeros(X_test.shape[0]), index=X_test.index)
    for column in X_test.columns:
        if column in Gaussian:
            log_positive += log_gaussian(X_test[column], Gaussian[column]['mu_1'], Gaussia
            log_negative += log_gaussian(X_test[column], Gaussian[column]['mu_0'], Gaussia
        elif column in Bernoulli:
            log_positive += log_bernoulli(X_test[column], Bernoulli[column]['theta_1'])
            log_negative += log_bernoulli(X_test[column], Bernoulli[column]['theta_0'])

    return np.exp(log_positive + np.log(prior))/(np.exp(log_positive + np.log(prior)) + np
```

Now we are in a position to make the predictions for the test data.

In [16]:

```
p_y = predict(X_test, Gaussian, Bernoulli, prior)
```

We can test the quality of the predictions in the following way. Firstly, we can threshold our probabilities at 0.5, allocating points with greater than 50% probability of membership of the positive class to the positive class. We can then compare to the true values, and see how many of these values we got correct. This is our total number correct.

In [17]:

```
correct = y_test & p_y>0.5
total_correct = sum(correct)
print("Total correct", total_correct, " out of ", len(y_test), "which is", float(total_cor
```

```
Total correct 119  out of  221 which is 0.5384615384615384 %
```

We can also now plot the confusion matrix (http://en.wikipedia.org/wiki/Confusion_matrix). A confusion matrix tells us where we are making mistakes. Along the diagonal it stores the *true positives*, the points that were positive class that we classified correctly, and the *true negatives*, the points that were negative class and that we classified correctly. The off diagonal terms contain the false positives and the false negatives. Along the rows of the matrix we place the actual class, and along the columns we place our predicted class.

In [18]:

```
confusion_matrix = pd.DataFrame(data=np.zeros((2,2)),
                                columns=['predicted R-rated', 'predicted not R-rated'],
                                index =['actual R-rated', 'actual not R-rated'])
confusion_matrix['predicted R-rated']['actual R-rated'] = (y_test & p_y>0.5).sum()
confusion_matrix['predicted R-rated']['actual not R-rated'] = (~y_test & p_y>0.5).sum()
confusion_matrix['predicted not R-rated']['actual R-rated'] = (y_test & ~(p_y>0.5)).sum()
confusion_matrix['predicted not R-rated']['actual not R-rated'] = (~y_test & ~(p_y>0.5)).s
confusion_matrix
```

Out[18]:

	predicted R-rated	predicted not R-rated
actual R-rated	119.0	28.0
actual not R-rated	102.0	71.0

Exercise 3

How can you improve your classification, are all the features equally valid? Are some features more helpful than others? What happens if you remove features that appear to be less helpful. How might you select such features?

Exercise 4

We have decided to classify positive if probability of R rating is greater than 0.5. This has led us to accidentally classify some films as 'safe for children' when the aren't in actuallity. Imagine you wish to ensure that the film is safe for children. With your test set how low do you have to set the threshold to avoid all the false negatives (i.e. films where you said it wasn't R-rated, but in actuality it was?

Naive Bayes Summary

Naive Bayes is making very simple assumptions about the data, in particular it is modeling the full *joint* probability of the data set, $p(\mathbf{y}, \mathbf{X}|\boldsymbol{\theta}, \pi)$ by very strong assumptions about factorizations that are unlikely to be true in practice. The data conditional independence assumption is common, and relies on a rich parameter vector to absorb all the information in the training data. The additional assumption of naive Bayes is that features are conditional indpenenent given the class label y_i (and the parameter vector, $\boldsymbol{\theta}$. This is quite a strong assumption. However, it causes the objective function to decompose into parts which can be independently fitted to the different feature vectors, meaning it is very easy to fit the model to large data. It is also clear how we should handle *streaming* data and *missing* data. This means that the model can be run 'live', adapting parameters and information as it arrives. Indeed, the model is even capable of dealing with new *features* that might arrive at run time. Such is the strength of the modeling the joint probability density. However, the factorization assumption that allows us to do this efficiently is very strong and may lead to poor decision boundaries in practice.

Reading

- Chapter 5 of @Rogers:book11 up to pg 179 (Section 5.1, and 5.2 up to 5.2.2).