Albíter Alpízar Vicente
Aprendizaje Computacional: Tarea 2
13/03/2019

Problema 1

EMV

Primero, construiremos el clasificador usando el estimador de máxima verosimilitud. Asumimos que nuestro modelo sigue una distribución de probabilidad de Bernoulli.

Comenzaremos por calcular, dada la tabla descrita en el problema, las probabilidades a priori siguientes:

- $P(JA = Joven) = \frac{6}{13}$
- $P(JA = Adulto) = \frac{7}{13}$

Ahora bien, sea X = (Pink Floyd, The Beatles, REM, Nirvana, Queen, Oasis), asumiendo que las variables son condicionalmente independientes, calculamos las probabilidades de verosimilitud para la clase Joven:

- $P(PinkFloyd = 1|JA = Joven) = \frac{6}{6}$
- $P(TheBeatles = 1|JA = Joven) = \frac{3}{6}$
- $P(REM = 1|JA = Joven) = \frac{3}{6}$
- $P(Nirvana = 1|JA = Joven) = \frac{2}{6}$
- $P(Queen = 1|JA = Joven) = \frac{3}{6}$
- $P(Oasis = 1|JA = Joven) = \frac{3}{6}$

Hacemos lo mismo para la clase Adulto:

- $P(PinkFloyd = 1|JA = Adulto) = \frac{7}{7}$
- $P(TheBeatles = 1|JA = Adulto) = \frac{7}{7}$
- $P(REM = 1|JA = Adulto) = \frac{4}{7}$
- $P(Nirvana = 1|JA = Adulto) = \frac{3}{7}$
- $P(Queen = 1|JA = Adulto) = \frac{5}{7}$
- $P(Oasis = 1|JA = Adulto) = \frac{3}{7}$

Con los parámetros anteriores, clasificamos a cada uno de los vectores dados:

- $x_1 = (1, 1, 0, 1, 1, 0)$

$$P(Joven|x_1)\ \alpha\ P(Joven) \times\ (P(PinkFloyd = 1|Joven) \times P(TheBeatles = 1|Joven) \times (1 - P(REM = 1|J$$
$$\times P(Queen = 1|Joven) \times (1 - P(Oasis = 1|Joven)))$$
$$P(Joven|x_1)\ \alpha\ \frac{6}{13} \times\ (\frac{6}{6} \times \frac{3}{6} \times (1 - \frac{3}{6}) \times \frac{2}{6} \times \frac{3}{6} \times (1 - \frac{3}{6}))$$

In [1]:
```
(6.0/13)*(6.0/6)*(3.0/6)*(1 - (3.0/6))*(2.0/6)*(3.0/6)*(1-(3.0/6))
```

Out[1]: 0.009615384615384616

$$P(Joven|x_1) = 0.0096$$

$$P(Adulto|x_1)\ \alpha\ P(Adulto) \times\ (P(PinkFloyd = 1|Adulto) \times P(TheBeatles = 1|Adulto) \times (1 - P(REM =$$
$$\times P(Queen = 1|Adulto) \times (1 - P(Oasis = 1|Adulto)))$$
$$P(Adulto|x_1)\ \alpha\ \frac{7}{13} \times\ (\frac{7}{7} \times \frac{7}{7} \times (1 - \frac{4}{7}) \times \frac{3}{7} \times \frac{5}{7} \times (1 - \frac{3}{7}))$$

In [2]:
```
(7.0/13)*(7.0/7)*(7.0/7)*(1 - (4.0/7))*(3.0/7)*(5.0/7)*(1-(3.0/7))
```

Out[2]: 0.040367795469836286

$$P(Adulto|x_1) = 0.0404$$

Ya que $P(Adulto|x_1)\ >\ P(Joven|x_1)$, concluimos que el vector x_1 corresponde a la clase Adulto.

- $x_2 = (1, 0, 1, 1, 1, 1)$

$$P(Joven|x_2)\ \alpha\ \frac{6}{13} \times\ (\frac{6}{6} \times (1 - \frac{3}{6}) \times \frac{3}{6} \times \frac{2}{6} \times \frac{3}{6} \times \frac{3}{6})$$

In [3]:
```
(6.0/13)*(6.0/6)*(1 - (3.0/6))*(3.0/6)*(2.0/6)*(3.0/6)*(3.0/6)
```

Out[3]: 0.009615384615384616

$$P(Joven|x_2) = 0.0096$$

$$P(Adulto|x_2)\ \alpha\ \frac{7}{13} \times\ (\frac{7}{7} \times (1 - \frac{7}{7}) \times \frac{4}{7} \times \frac{3}{7} \times \frac{5}{7} \times \frac{3}{7})$$

$$P(Adulto|x_2) = 0$$

Ya que $P(Joven|x_2) > P(Adulto|x_2)$, concluimos que el vector x_2 corresponde a la clase Joven.

- $x_3 = (1, 1, 0, 0, 0, 0)$

$$P(Joven|x_3)\ \alpha\ \frac{6}{13} \times\ (\frac{6}{6} \times \frac{3}{6} \times (1 - \frac{3}{6}) \times (1 - \frac{2}{6}) \times (1 - \frac{3}{6}) \times (1 - \frac{3}{6}))$$

In [4]:
```
(6.0/13)*(6.0/6)*(3.0/6)*(1 - (3.0/6))*(1-(2.0/6))*(1-(3.0/6))*(1-(3.
0/6))
```

Out[4]:
```
0.019230769230769235
```

$$P(Joven|x_3) = 0.0192$$

$$P(Adulto|x_3)\ \alpha\ \frac{7}{13} \times\ (\frac{7}{7} \times \frac{7}{7} \times (1 - \frac{4}{7}) \times (1 - \frac{3}{7}) \times (1 - \frac{5}{7}) \times (1 - \frac{3}{7}))$$

In [5]:
```
(7.0/13)*(7.0/7)*(7.0/7)*(1 - (4.0/7))*(1-(3.0/7))*(1-(5.0/7))*(1-(3.
0/7))
```

Out[5]:
```
0.021529490917246017
```

$$P(Adulto|x_3) = 0.0215$$

Ya que $P(Adulto|x_3) > P(Joven|x_3)$, concluimos que el vector x_3 corresponde a la clase Adulto.

- $x_4 = (1, 1, 1, 1, 1, 1)$

$$P(Joven|x_4)\ \alpha\ \frac{6}{13} \times\ (\frac{6}{6} \times \frac{3}{6} \times \frac{3}{6} \times \frac{2}{6} \times \frac{3}{6} \times \frac{3}{6})$$

In [6]:
```
(6.0/13)*(6.0/6)*(3.0/6)*(3.0/6)*(2.0/6)*(3.0/6)*(3.0/6)
```

Out[6]: 0.009615384615384616

$$P(Joven|x_4) = 0.0096$$

$$P(Adulto|x_4)\ \alpha\ \frac{7}{13} \times (\frac{7}{7} \times \frac{7}{7} \times \frac{4}{7} \times \frac{3}{7} \times \frac{5}{7} \times \frac{3}{7})$$

In [7]:
```
(7.0/13)*(7.0/7)*(7.0/7)*(4.0/7)*(3.0/7)*(5.0/7)*(3.0/7)
```

Out[7]: 0.04036779546983627

$$P(Adulto|x_4) = 0.0404$$

Ya que $P(Adulto|x_4) > P(Joven|x_4)$, concluimos que el vector x_4 corresponde a la clase Adulto.

- $x_5 = (0, 1, 1, 1, 1, 1)$

$$P(Joven|x_5)\ \alpha\ \frac{6}{13} \times ((1 - \frac{6}{6}) \times \frac{3}{6} \times \frac{3}{6} \times \frac{2}{6} \times \frac{3}{6} \times \frac{3}{6})$$

$$P(Joven|x_5) = 0$$

$$P(Adulto|x_5)\ \alpha\ \frac{7}{13} \times ((1 - \frac{7}{7}) \times \frac{7}{7} \times \frac{4}{7} \times \frac{3}{7} \times \frac{5}{7} \times \frac{3}{7})$$

$$P(Adulto|x_5) = 0$$

Ya que $P(Joven|x_5) = P(Adulto|x_5)$ no es posible clasificar al vector x_5 usando el estimador de máxima verosimilitud planteado.

MAP

A continuación, realizaremos la clasificación de los vectores anteriores usando un estimador de máximo a posteriori para calcular todos los parámetros de los atributos definidos por X y la clase C. Nuevamente asumiremos una distribución de probabilidad de Bernoulli, y para calcular los parámetros con MAP, usaremos usaremos:

$$q_C = \frac{N_C + \alpha - 1}{N + \beta + \alpha - 2}$$
$$q_{x|C} = \frac{n_c(x) + \alpha - 1}{N + \beta + \alpha - 2}$$

Donde asumiremos que:

- Para los parámetros de los atributos, $\alpha = 2$ y $\beta = |V|$ (donde |V| representa el número de valores que puede tomar el atributo en cuestión). En todos los casos $\beta = 2$.
- Para los parámetros de la clase, $\alpha = 2$ y $\beta = |C|$ (donde C representa el número de valores que puede tomar la clase en cuestión). En todos los casos $\beta = 2$.

Calculamos pues los parámetros de clase:

- $q_{Joven} = \frac{6+2-1}{13+2+2-2} = \frac{7}{15}$
- $q_{Adulto} = \frac{7+2-1}{13+2+2-2} = \frac{8}{15}$

Para la parte restante de este inciso, y en adelante, automatizaremos el cálculo de cada uno de los parámetros de clase y de atributos. Para hacerlo, importaremos la tabla mostrada en el problema a un arreglo que pueda ser manipulado por NumPy.

In [8]:
```
qj = 7.0/15
qa = 8.0/15
```

Para la parte restante de este inciso, y en adelante, automatizaremos el cálculo de cada uno de los parámetros de clase y de atributos. Para hacerlo, importaremos la tabla mostrada en el problema a un arreglo que pueda ser manipulado por NumPy.

In [9]:
```
import pandas
```

In [10]:
```
bandas = pandas.read_csv('bandas.csv')
```

In [11]:
```
bandas
```

Out[11]:

	Pink Floyd	The Beatles	REM	Nirvana	Queen	Oasis	JA
0	1	0	0	1	1	1	J
1	1	1	0	1	1	0	J
2	1	1	1	0	0	1	J
3	1	0	1	0	0	1	J
4	1	0	0	0	1	0	J
5	1	1	1	0	0	0	J
6	1	1	0	0	1	1	A
7	1	1	1	0	0	1	A
8	1	1	1	1	1	0	A
9	1	1	1	0	1	0	A
10	1	1	1	0	1	1	A
11	1	1	0	1	1	0	A
12	1	1	0	1	0	0	A

Primero, dividimos a la tabla anterior en un arreglo "v" donde esté contenida la información de las bandas, y un vector "c" que tenga la información de las clases. Esto con la intención de poder manejar los datos más fácilmente en un programa.

In [12]:
```
b = bandas.values
v = b[:,:-1]
c = b[:,-1]
```

Ahora, para calcular los parámetros $q_{a|C}$, usamos la siguiente función, que toma como entradas el nombre de la clase en cuestión (*C*), y los valores de α y β definidos previamente.

In [13]:
```
def getParamsForClass(name, alpha, beta):
    n = 0
    nc = 0
    params = []
    for i in range(len(c)):
        if c[i] == name:
            n = n + 1
    for i in range(len(v[0])):
        for j in range(len(v)):
            if c[j] == name:
                nc = nc + v[j][i]
        params.append(1.0*(nc + alpha - 1)/(n + beta + alpha - 2))
        nc = 0

    return params
```

In [14]:
```
jparams = getParamsForClass('J', 2, 2)
jparams
```

Out[14]:
```
[0.875, 0.5, 0.5, 0.375, 0.5, 0.5]
```

Esto es, para la clase Joven, tenemos:

- $P_{PinkFloyd|Joven} = 0.875$
- $P_{TheBeatles|Joven} = 0.5$
- $P_{REM|Joven} = 0.5$
- $P_{Nirvana|Joven} = 0.375$
- $P_{Queen|Joven} = 0.5$
- $P_{Oasis|Joven} = 0.5$

In [15]:
```
aparams = getParamsForClass('A', 2, 2)
aparams
```

Out[15]:
```
[0.8888888888888888,
 0.8888888888888888,
 0.5555555555555556,
 0.4444444444444444,
 0.6666666666666666,
 0.4444444444444444]
```

Por su parte, para la clase Adulto, tenemos:

- $P_{PinkFloyd|Adulto} = 0.8889$
- $P_{TheBeatles|Adulto} = 0.8889$
- $P_{REM|Adulto} = 0.5556$
- $P_{Nirvana|Adulto} = 0.4444$
- $P_{Queen|Adulto} = 0.6667$
- $P_{Oasis|Adulto} = 0.4444$

Con los parámetros obtenidos, obtenemos las probabilidades de pertenencia a cada una de las clases para cada uno de los vectores de prueba, usando la siguiente función. Esta función toma como entradas el parámetro de la clase en cuestión ("cls"), el vector de prueba ("testvector") y los parámetros asociados a la clase ("params"), y regresa la probabilidad estimada $P(C|X)$.

In [16]:
```
def getProbabilities(cls, testvector, params):
    prob = 1
    for i in range(len(testvector)):
        if testvector[i] == 1:
            prob = prob * params[i]
        else:
            prob = prob * (1 - params[i])
    return cls * prob
```

Así pues, clasificamos a cada vector según la máxima probabilidad obtenida.

- $x_1 = (1, 1, 0, 1, 1, 0)$

```
In [17]: getProbabilities(qj, [1,1,0,1,1,0], jparams)
Out[17]: 0.0095703125

In [18]: getProbabilities(qa, [1,1,0,1,1,0], aparams)
Out[18]: 0.030829386517035755
```

Tenemos entonces que:

$$P(Joven|x_1) = 0.0096$$
$$P(Adulto|x_1) = 0.0309$$

Por lo tanto, concluimos que x_1 pertenece a la clase Adulto.

- $x_2 = (1, 0, 1, 1, 1, 1)$

```
In [19]: getProbabilities(qj, [1,0,1,1,1,1], jparams)
Out[19]: 0.0095703125

In [20]: getProbabilities(qa, [1,0,1,1,1,1], aparams)
Out[20]: 0.003853673314629471
```

Tenemos entonces que :

$$P(Joven|x_2) = 0.0096$$
$$P(Adulto|x_2) = 0.0038$$

Por lo tanto, concluimos que x_2 pertenece a la clase Joven.

- $x_3 = (1, 1, 0, 0, 0, 0)$

```
In [21]: getProbabilities(qj, [1,1,0,0,0,0], jparams)
Out[21]: 0.015950520833333332

In [22]: getProbabilities(qa, [1,1,0,0,0,0], aparams)
Out[22]: 0.01926836657314735
```

Tenemos entonces que :

$$P(Joven|x_3) = 0.0159$$
$$P(Adulto|x_3) = 0.0192$$

Por lo tanto, concluimos que x_3 pertenece a la clase Adulto.

- $x_4 = (1, 1, 1, 1, 1, 1)$

```
In [23]: getProbabilities(qj, [1,1,1,1,1,1], jparams)
Out[23]: 0.0095703125

In [24]: getProbabilities(qa, [1,1,1,1,1,1], aparams)
Out[24]: 0.03082938651703575
```

Tenemos entonces que :

$$P(Joven|x_4) = 0.0096$$
$$P(Adulto|x_4) = 0.0308$$

Por lo tanto, concluimos que x_4 pertenece a la clase Adulto.

- $x_5 = (0, 1, 1, 1, 1, 1)$

```
In [25]: getProbabilities(qj, [0,1,1,1,1,1], jparams)
Out[25]: 0.0013671875

In [26]: getProbabilities(qa, [0,1,1,1,1,1], aparams)
Out[26]: 0.003853673314629471
```

Tenemos entonces que :

$$P(Joven|x_5) = 0.0014$$
$$P(Adulto|x_5) = 0.0039$$

Por lo tanto, concluimos que x_5 pertenece a la clase Adulto.

Problema 2

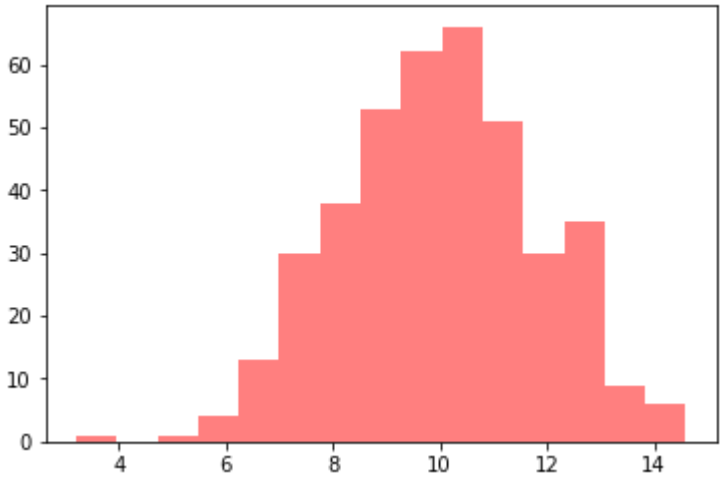
Preparación de los datos y elección del modelo

Para construir a nuestro clasificador bayesiano ingenuo (más adelante se explicará que se usará un estimador de máxima verosimilitud para hacerlo), primero debemos cómo están distribuidas nuestras variables de entrenamiento. Esto con la intención de determinar el modelo a utilizar a la hora de entrenar al clasificador.

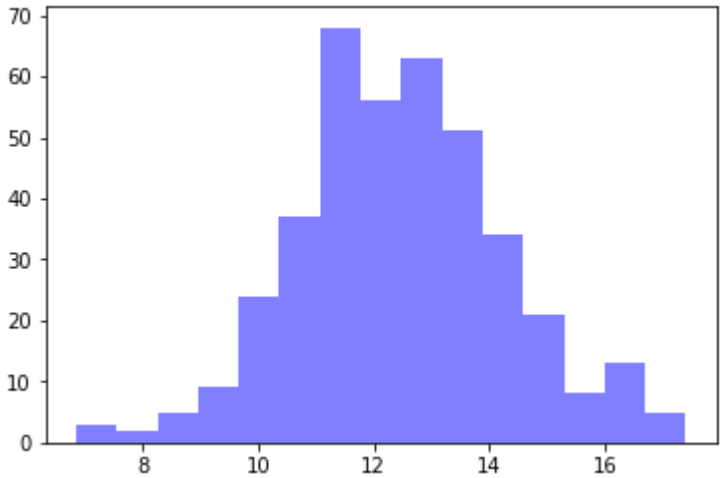
In [27]:
```
import matplotlib.pyplot as plt
bact = pandas.read_csv('bacterias.csv')
```

In [28]:
```
b = bact.values
```

In [29]:
```
n, bins, patches = plt.hist(b[:,0], 15, facecolor='red', alpha=0.5)
```



In [30]:
```
n, bins, patches = plt.hist(b[:,1], 15, facecolor='blue', alpha=0.5)
```



Observamos que tanto las variables X_1 y X_2 parecen estar normalmente distribuidas. Por lo tanto, asumiremos para nuestro modelo que $P(X_1|C)$ y $P(X_2|C)$ siguen distribuciones Gaussianas. Ya que la clase C sólo puede tomar valores de 0 y 1, asumiremos que $P(C)$ sigue una distribución binomial.

Para construir el clasificador, haremos uso de sci-kitlearn. Específicamente, utilizaremos la clase *GaussianNB* para entrenar al clasificador. Según la documentación de la clase *GaussianNB*, los parámetros de los atributos son estimados usando un estimador de máxima verosimilitud (EMP).

Así pues, importamos la clase mencionada, y le damos como entrada los datos de las variables X_1 y X_2, y su clase asociada C.

In [31]:
```
import numpy as np
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
data = b[:, :-1]
labels = np.transpose(b[:, 2:].flatten())
```

Predicciones

Con lo anterior, entrenamos a nuestro clasificador (usando el método *fit* de la clase GaussianNB), y realizamos las predicciones de clase sobre los mismos datos de entrenamiento (usando el método *predict*):

In [32]:
```
predictions = gnb.fit(data, labels).predict(data)
```

Observamos que las predicciones sobre los datos de entrenamiento son las siguientes:

In [33]:
```
predictions
```

Out[33]:
```
array([1., 1., 1., 0., 0., 1., 0., 0., 1., 1., 0., 1., 1., 0., 1.,
       0., 1.,
       1., 1., 1., 1., 1., 0., 1., 0., 1., 0., 1., 1., 1., 1., 1.,
       0., 0.,
       1., 1., 1., 1., 0., 1., 0., 0., 1., 0., 1., 1., 1., 1., 0.,
       0., 1.,
       0., 1., 0., 1., 1., 0., 0., 1., 0., 1., 0., 1., 1., 0., 0.,
       1., 0.,
       1., 0., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 0., 1.,
       1., 1.,
       0., 1., 0., 0., 1., 0., 1., 0., 1., 1., 1., 1., 1., 0., 0.,
       1., 1.,
       0., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 0., 0., 1.,
       1., 0.,
       0., 1., 0., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 0.,
       1., 1.,
       1., 0., 1., 1., 1., 0., 0., 0., 0., 0., 1., 1., 1., 1., 0.,
       0., 0.,
       1., 1., 1., 1., 1., 1., 0., 1., 0., 1., 0., 1., 0., 1., 0.,
       1., 1.,
       0., 0., 1., 1., 1., 0., 1., 1., 0., 0., 0., 0., 1., 1., 1.,
       1., 1.,
       0., 1., 1., 1., 0., 0., 0., 1., 1., 0., 1., 1., 1., 0., 0.,
       0., 0.,
       1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 1., 1.,
       0., 0.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0.,
       1., 1.,
       1., 0., 0., 1., 0., 1., 1., 1., 0., 1., 1., 0., 1., 1., 0.,
       1., 0.,
       1., 1., 1., 0., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 0.,
       0., 0.,
       1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1.,
       1., 1., 0., 1., 1., 1., 1., 1., 1., 0., 1., 0., 1., 1., 1.,
       0., 0.,
       1., 1., 1., 0., 1., 1., 0., 0., 0., 1., 1., 1., 1., 1., 0.,
       1., 1.,
       0., 0., 0., 1., 1., 0., 1., 0., 1., 0., 1., 1., 1., 1., 1.,
       1., 1.,
       1., 0., 0., 1., 0., 0., 1., 0., 1., 1., 1., 1., 0., 1., 0.,
       1., 0.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 1.,
       0., 0.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 0., 1., 1., 1.,
       1., 1.,
       0., 1., 1., 1., 1., 1., 0., 1.])
```

Para calcular el número de errores que nuestro clasificador cometió a la hora de predecir las etiquedas de los datos que le dimos, usamos la siguiente función:

In [34]:
```
def countCorrectPredictions(expected_labels, predicted_labels):
    count = 0
    for i in range(len(predicted_labels)):
        if expected_labels[i] == predicted_labels[i]:
            count = count + 1
    return count
```

Así pues, le damos como entrada a nuestra función las etiquetas esperadas (o sea, las etiquetas asociadas a los datos de entrenamiento) y las etiquedas arrojadas por el clasificador. Ya que hay 400 datos de entrenamiento, calculamos también el porcentaje de predicciones correctas:

In [35]:
```
count = countCorrectPredictions(labels, predictions)
print 'Predicciones correctas: ' + str(count)
print '% de predicciones correctas: ' + str(count / 400.0)
```

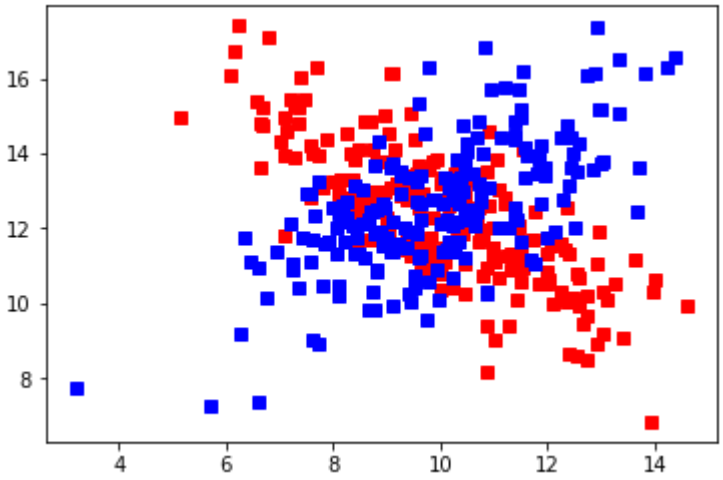
```
Predicciones correctas: 220
% de predicciones correctas: 0.55
```

Es decir, se realizaron **220** predicciones correctas sobre los datos de entrenamiento, o sea, un **55%** de predicciones correctas (sólo 5% arriba del valor esperado respecto a si se estuvieran haciendo las predicciones de manera aleatoria).

Conclusiones

Para entender por qué a nuestro clasificador le cuesta tanto trabajo predecir correctamente los datos, observemos la distribución de los datos de entrenamiento, vistos de manera gráfica, donde los puntos rojos corresponden a los datos clasificados como '0', y los puntos azules corresponden a los datos clasificados como '1'.

In [36]:
```
x0 = b[:-200, 0]
y0 = b[:-200, 1]
x1 = b[199:,0]
y1 = b[199:,1]
plt.plot(x0, y0, 'rs', x1, y1, 'bs')
plt.show()
```



Se observa claramente cómo, ya que tanto X_1 y X_2 están distribuidos normalmente, la mayor parte de las concentraciones de químicos usadas en el cultivo de bacterias se encuetran entre los valores intermedios de X_1 y X_2 (aproximadamente entre 8 y 12 unidades de concentración). Para estos valores (que representan a la mayoría), no es claro cómo las concentraciones de los químicos determinan si una bacteria vive o muere. Por otro lado, se observa que para los valores extremos de las distribuciones, sí hay un mecanismo claro de supervivencia de las bacterias. Esto es, las bacterias viven más cuando:

- Hay una alta concentración del químico X_1 combinada con una baja concentración del químico X_2.
- Hay una alta concentración del químico X_2, combinada con una baja concentración del químico X_1.

Podemos suponer que nuestro clasificador puede funcionar bien para estos casos extremos (de ahí que el número de predicciones correctas sea ligeramente superior a una predicción aleatoria). No obstante, para el resto de ellos, donde no hay una clara influencia de las variables sobre la clase de pertenencia, el clasificador bayesiano ingenuo parece funcionar sólo tan bien como una predicción aleatoria.

Problema 3

Preparación de los datos y elección del modelo

En primer lugar, sacamos los datos del archivo proporcionado:

In [37]:
```
spam = pandas.read_csv('spam.csv', header=None, delimiter=" ")
```

In [38]:
```
spam.head()
```

Out[38]:

	0	1	2	3	4	5	6	7	8	9	...	1991	1992	1993	1994	1995	1996	1997	1998	1999	2
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	
1	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	1	
2	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	
4	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	

5 rows × 2001 columns

Ahora, con la ayuda de la función *traintestsplit*, separamos los datos en datos de entrenamiento (*trainingdata*) y de prueba (*testdata*) a razón de 70-30.

In [39]:
```
from sklearn.model_selection import train_test_split as tts
sval = spam.values
training_data, test_data = tts(sval, test_size=0.30)
```

Ahora bien, de manera que las 2000 columnas con información de las palabras y la columna con la información de las etiquetas queden almacenadas en dos arreglos de NumPy distintos (a los que pondremos el sufijo de *words* y *labels*, respectivamente), aplicamos los siguientes operadores de slice sobre los arreglos de NumPy:

In [237]:
```
training_data_words = training_data[:, :-1]
training_data_labels = np.transpose(training_data[:, 2000:]).flatten
()
test_data_words = test_data[:, :-1]
test_data_labels = np.transpose(test_data[:, 2000:]).flatten()
```

Entrenamiento

Con los datos anteriores, entrenaremos dos clasificadores: uno asumiendo distribución Bernoulli de las palabras, y otro asumiendo distribución multinomial de las mismas. Nuevamente, para hacer esto, haremos uso de sci-kitlearn. En particular, usaremos las clases *BernoulliNB* y *MultinomialNB*. Los clasificadores arrojados por ambas funciones son entrenados con un estimador de máxima verosimilitud.

In [41]:
```
from sklearn.naive_bayes import BernoulliNB
from sklearn.naive_bayes import MultinomialNB

bnb = BernoulliNB()
mnb = MultinomialNB()
```

La distribución de Bernoulli trabaja con variables binarias. En el caso específico donde la variable es la ocurrencia de una palabra, "0" significa que la palabra no ocurre en un correo dado, y "1" que sí aparece. Ya que la información contenida en nuestros datos actuales es de frecuencia de palabras, y no de ocurrencia, antes de entrenar a nuestro clasificador de Bernoulli, debemos "binarizar" nuestro arreglo de variables de entrenamiento. Es decir, transformaremos dicho arreglo de manera que, si una variable tiene un valor superior a "0", lo convertiremos en "1".

In [42]:
```
from sklearn.preprocessing import binarize
training_data_words_binary = binarize(training_data_words)
test_data_words_binary = binarize(test_data_words)
```

Procedemos pues al entrenamiento de ambos clasificadores.

In [43]:
```
bernoulli_nb = bnb.fit(training_data_words_binary, training_data_labe
ls)
multinomial_nb = mnb.fit(training_data_words, training_data_labels)
```

Predicciones

A continuación, realizamos las predicciones usando nuestros clasificadores sobre los datos de entrenamiento y los datos de prueba. Para contar el número de predicciones correctas, hacemos uso de la función *countCorrectPredictions* definida en el Problema 2.

Bernoulli NB

In [44]:
```
pred_bernoulli_training = bernoulli_nb.predict(training_data_words_bi
nary)
reference = len(training_data_words_binary)
count = countCorrectPredictions(training_data_labels, pred_bernoulli_
training)
print 'Bernoulli NB - Predicción sobre datos de Entrenamiento'
print 'Predicciones correctas: ' + str(count)
print '% de predicciones correctas: ' + str(1.0*count / reference)

print ''

pred_bernoulli_test = bernoulli_nb.predict(test_data_words_binary)
reference = len(test_data_words_binary)
count = countCorrectPredictions(test_data_labels, pred_bernoulli_test
)
print 'Bernoulli NB - Predicción sobre datos de Prueba'
print 'Predicciones correctas: ' + str(count)
print '% de predicciones correctas: ' + str(1.0*count / reference)
```

```
Bernoulli NB - Predicción sobre datos de Entrenamiento
Predicciones correctas: 3278
% de predicciones correctas: 0.905524861878

Bernoulli NB - Predicción sobre datos de Prueba
Predicciones correctas: 1400
% de predicciones correctas: 0.90206185567
```

Esto es, con el clasificador NB que asume distribución de Bernoulli:

- Se hicieron **3276** predicciones correctas sobre los datos de entrenamiento, o bien, un **90.49%** de predicciones correctas sobre de los datos de entrenamiento totales.
- Se hicieron **1408** predicciones correctas sobre los datos de prueba, o bien, un **90.72%** de predicciones correctas sobre de los datos de prueba totales.

Multinomial NB

In [45]:
```
pred_multinomial_training = multinomial_nb.predict(training_data_word
s)
reference = len(training_data_words)
count = countCorrectPredictions(training_data_labels, pred_multinomia
l_training)
print 'Multinomial NB - Predicción sobre datos de Entrenamiento'
print 'Predicciones correctas: ' + str(count)
print '% de predicciones correctas: ' + str(1.0*count / reference)

print ''

pred_multinomial_test = multinomial_nb.predict(test_data_words)
reference = len(test_data_words)
count = countCorrectPredictions(test_data_labels, pred_multinomial_te
st)
print 'Multinomial NB - Predicción sobre datos de Prueba'
print 'Predicciones correctas: ' + str(count)
print '% de predicciones correctas: ' + str(1.0*count / reference)
```

```
Multinomial NB - Predicción sobre datos de Entrenamiento
Predicciones correctas: 3460
% de predicciones correctas: 0.955801104972

Multinomial NB - Predicción sobre datos de Prueba
Predicciones correctas: 1471
% de predicciones correctas: 0.947809278351
```

Esto es, con el clasificador NB que asume distribución Multinomial:

- Se hicieron **3453** predicciones correctas sobre los datos de entrenamiento, o bien, un **95.39%** de predicciones correctas sobre de los datos de entrenamiento totales.
- Se hicieron **1477** predicciones correctas sobre los datos de prueba, o bien, un **95.17%** de predicciones correctas sobre de los datos de prueba totales.

Conclusiones

En lo general, se observa que ambos clasificadores hicieron más del 90% de predicciones correctas sobre los datos de entrenamiento y de prueba. No obstante, el clasificador que asumió distribución multinomial tuvo un mejor desempeño en ambos casos. Para clasificación de textos, este resultado concuerda con la intuición: existen ciertas palabras que no sólo tienden a "aparecer", sino también a "predominar" en correos de spam (pues muchos de estos correos son de índole de ventas, donde ciertas *buzzwords* se repiten con mucha frecuencia).

Problema 4

Preparación de los datos y elección del modelo

Primero, extraemos los datos de la base de datos.

In [218]:
```
cancer = pandas.read_csv('cancer.csv', header=None)
c = cancer.values
cancer.head()
```

Out[218]:

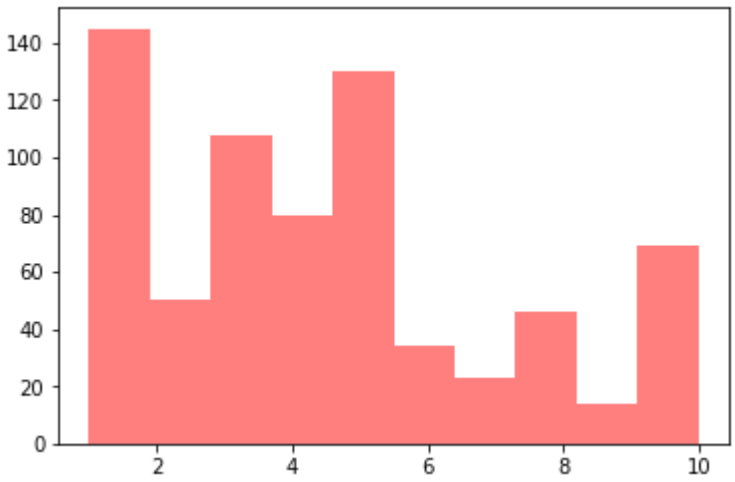
	0	1	2	3	4	5	6	7	8	9	10
0	1000025	5	1	1	1	2	1	3	1	1	2
1	1002945	5	4	4	5	7	10	3	2	1	2
2	1015425	3	1	1	1	2	2	3	1	1	2
3	1016277	6	8	8	1	3	4	3	7	1	2
4	1017023	4	1	1	3	2	1	3	1	1	2

Ya que tenemos algunos datos faltantes dentro de la base de datos, utilizaremos la función *fixMissingValues*, mostrada a continuación, para tratar de "arreglar" dichos datos faltantes, sustituyéndolos por el promedio de los otros valores de la misma columna. Alternativamente, la función tiene incorporado un método distinto de "arreglo" de los datos: si se pone como parámetro *method* la cadena 'zero', se sustituirá a los datos faltantes con un 0. Otro método no explorado fue el de la eliminación completa de la muestra. No obstante, se utilizó el primer método mencionado para arreglar los datos.

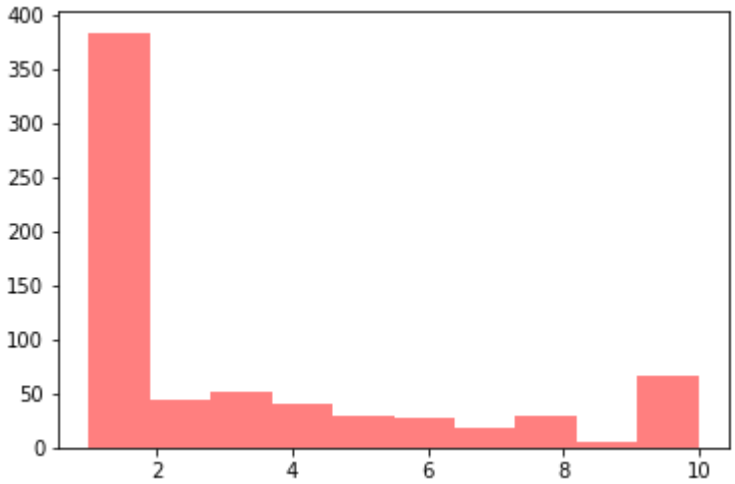
In [219]:
```
def fixMissingValues(array, method):

    arr = np.copy(array)

    if method == 'mean':
        means = []

        sumv = 0
        count = 0
        for j in range(len(arr[0])):
            for i in range(len(arr)):
                if arr[i][j] == '?':
                    continue
                else:
                    sumv = sumv + int(arr[i][j])
                    count = count + 1
            means.append(int(1.0*sumv/count))
            sumv = 0
            count = 0

        for i in range(len(arr)):
            for j in range(len(arr[0])):
                if arr[i][j] == '?':
                    arr[i][j] = means[j]
                else:
                    arr[i][j] = int(arr[i][j])
        return arr

    elif method == 'zero':
        for i in range(len(arr)):
            for j in range(len(arr[0])):
                if arr[i][j] == '?':
                    arr[i][j] = 0
                else:
                    arr[i][j] = int(arr[i][j])
        return arr

    return None
```

A continuación, procedemos a explorar los datos, con el fin de ver cómo están distribuidos los valores de cada una de las variables.

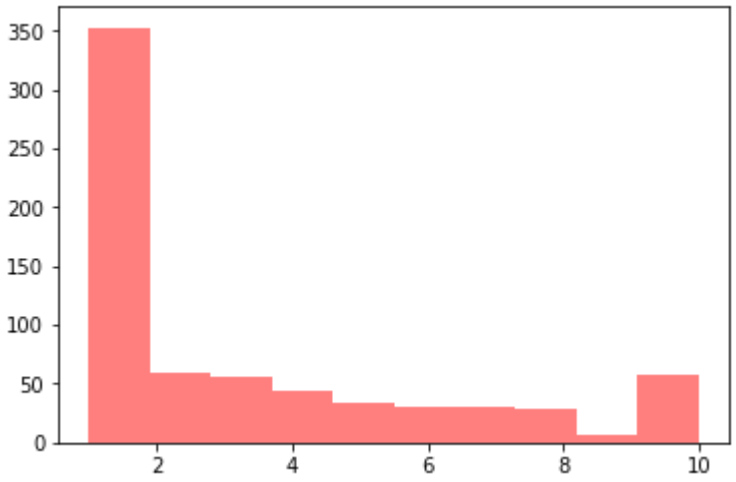
In [224]:
```
cnmv = fixMissingValues(c, 'mean')
ce = np.transpose(list(cnmv[:,1]))
n, bins, patches = plt.hist(ce, 10, facecolor='red', alpha=0.5)
```



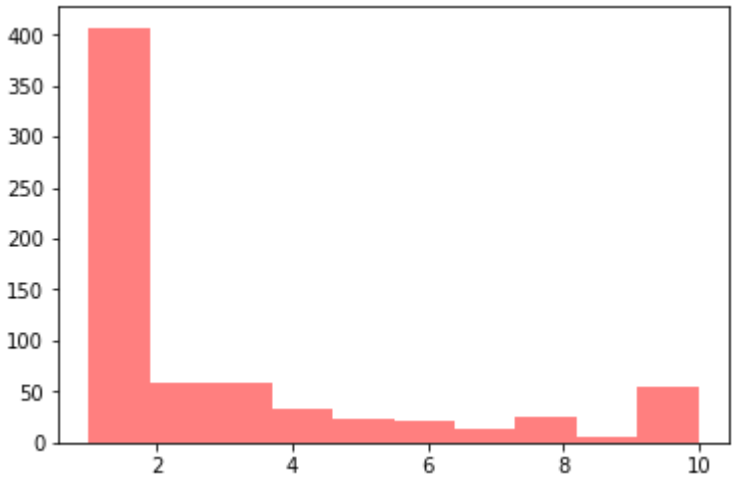
In [225]:
```
ce = np.transpose(list(cnmv[:,2]))
n, bins, patches = plt.hist(ce, 10, facecolor='red', alpha=0.5)
```



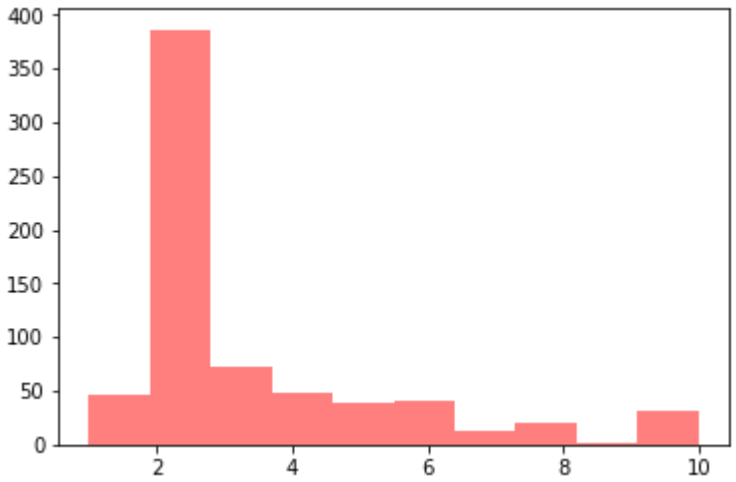
In [226]:
```
ce = np.transpose(list(cnmv[:,3]))
n, bins, patches = plt.hist(ce, 10, facecolor='red', alpha=0.5)
```



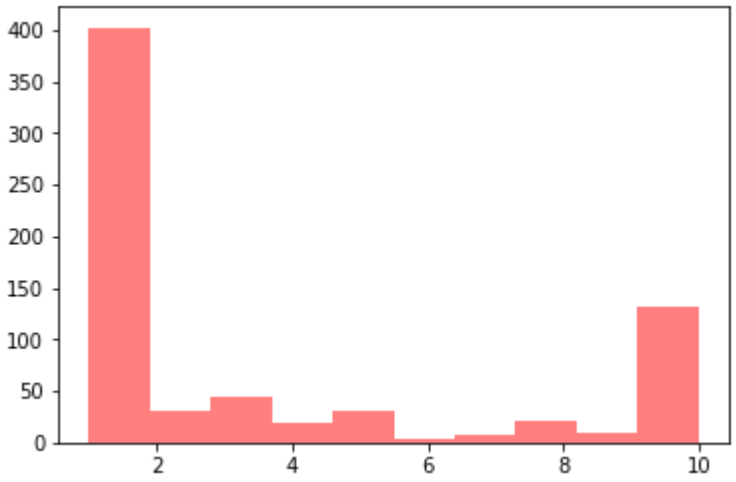
In [227]:
```
ce = np.transpose(list(cnmv[:,4]))
n, bins, patches = plt.hist(ce, 10, facecolor='red', alpha=0.5)
```



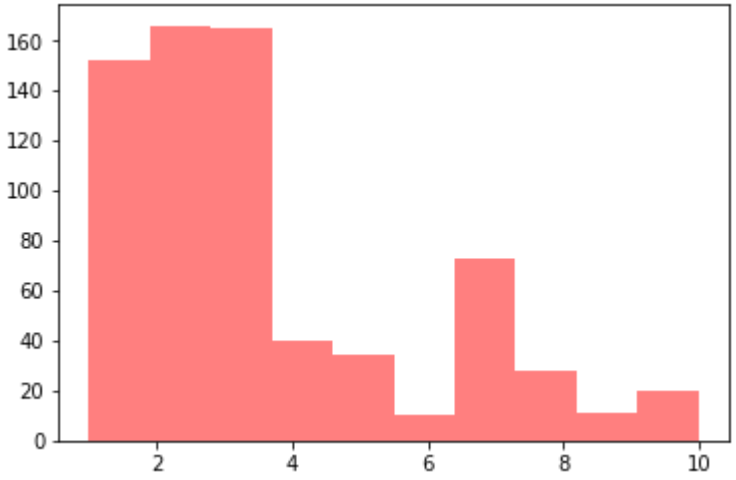
In [228]:
```
ce = np.transpose(list(cnmv[:,5]))
n, bins, patches = plt.hist(ce, 10, facecolor='red', alpha=0.5)
```



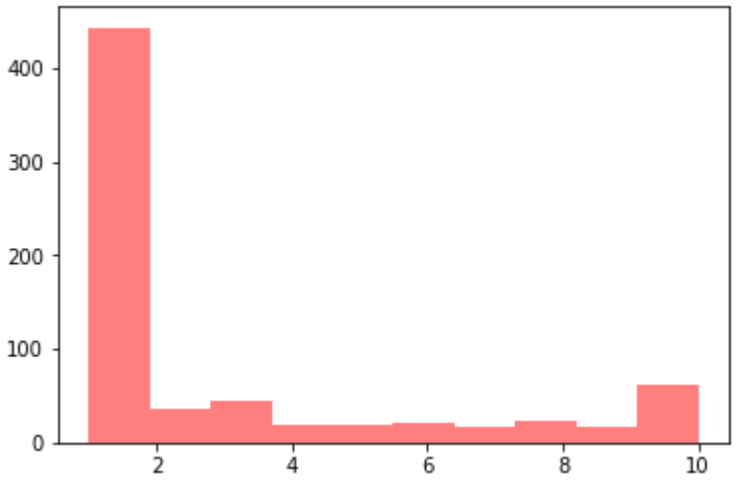
In [229]:
```
ce = np.transpose(list(cnmv[:,6]))
n, bins, patches = plt.hist(ce, 10, facecolor='red', alpha=0.5)
```



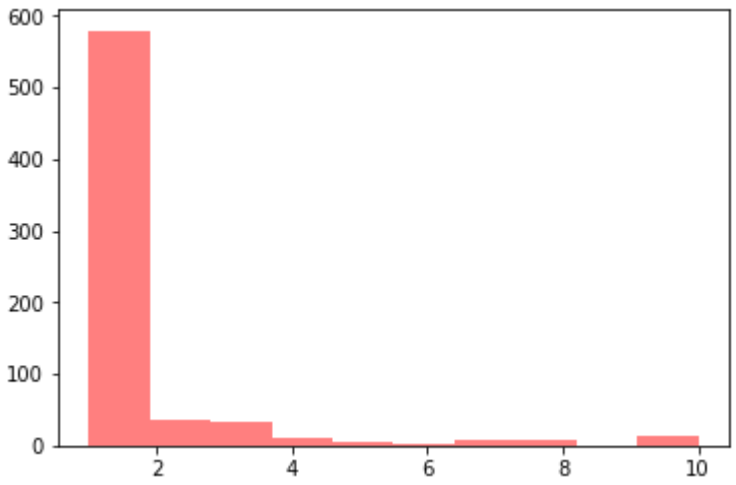
In [230]:
```
ce = np.transpose(list(cnmv[:,7]))
n, bins, patches = plt.hist(ce, 10, facecolor='red', alpha=0.5)
```



In [231]:
```
ce = np.transpose(list(cnmv[:,8]))
n, bins, patches = plt.hist(ce, 10, facecolor='red', alpha=0.5)
```



In [232]:
```
ce = np.transpose(list(cnmv[:,9]))
n, bins, patches = plt.hist(ce, 10, facecolor='red', alpha=0.5)
```



Se observa que los datos pertenecen a múltiples categorías, y que posiblemente algunas de éstas sigan distribuciones Gaussianas (la mayoría de ellas con mucho sesgo hacia la izquierda o derecha de la distribución). Para efectos de comparación, entrenaremos a un clasificador NB asumiendo **distribución multinomial** de las variables, y otro asumiendo **distribución Gaussiana**.

Antes de proceder al entrenamiento de los clasificadores, dividimos los datos entre datos de entrenamiento y datos de prueba, usando un procedimiento similar al del Problema 3. Ya que los números de la primera columna son sólo identificadores de las muestras (y no un atributo del cáncer en sí), descartaremos a toda la primera columna de los datos de entrenamiento y prueba.

In [247]:
```
training_data, test_data = tts(cnmv, test_size=0.30)
```

In [288]:
```
training_data_var = training_data[:, 1:-1]
training_data_labels = np.transpose(training_data[:, 10:]).flatten()
test_data_var = test_data[:, 1:-1]
test_data_labels = np.transpose(test_data[:, 10:]).flatten()
```

Además, ya que el tumor sólo puede ser benigno o maligno, reemplazaremos los valores de la última columna, 2 y 4, con 0 y 1, respectivamente.

In [289]:
```
training_data_labels_binary = list((training_data_labels - 2)/2)
test_data_labels_binary = list((test_data_labels - 2)/2)
```

Entrenamiento

Con los datos anteriores, entrenaremos dos clasificadores: uno asumiendo distribución multinomial de las palabras, y otro asumiendo distribución Gaussiana de las mismas. Nuevamente, para hacer esto, haremos uso de sci-kitlearn. En particular, usaremos las clases *MultinomialNB* y *GaussianNB*. Los clasificadores arrojados por ambas funciones son entrenados con un estimador de máxima verosimilitud.

In [300]:
```
from sklearn.naive_bayes import GaussianNB

mnb = MultinomialNB()
gnb = GaussianNB()
multinomial_nb = mnb.fit(training_data_var, training_data_labels_bina
ry)
gaussian_nb = gnb.fit(training_data_var, training_data_labels_binary)
```

Predicciones

A continuación, realizamos las predicciones usando nuestros clasificadores sobre los datos de entrenamiento y los datos de prueba. Para contar el número de predicciones correctas, hacemos uso de la función *countCorrectPredictions* definida en el Problema 2.

Multinomial NB

In [301]:
```
pred_training = multinomial_nb.predict(training_data_var)
reference = len(training_data_var)
count = countCorrectPredictions(training_data_labels_binary, pred_tra
ining)
print 'Multinomial NB - Predicción sobre datos de Entrenamiento'
print 'Predicciones correctas: ' + str(count)
print '% de predicciones correctas: ' + str(1.0*count / reference)

print ''

pred_test = multinomial_nb.predict(test_data_var)
reference = len(test_data_var)
count = countCorrectPredictions(test_data_labels_binary, pred_test)
print 'Multinomial NB - Predicción sobre datos de Prueba'
print 'Predicciones correctas: ' + str(count)
print '% de predicciones correctas: ' + str(1.0*count / reference)

print ''
```

```
Multinomial NB - Predicción sobre datos de Entrenamiento
Predicciones correctas: 434
% de predicciones correctas: 0.887525562372

Multinomial NB - Predicción sobre datos de Prueba
Predicciones correctas: 194
% de predicciones correctas: 0.92380952381
```

Esto es, con el clasificador NB que asume distribución Multinomial:

- Se hicieron **434** predicciones correctas sobre los datos de entrenamiento, o bien, un **88.75%** de predicciones correctas sobre de los datos de entrenamiento totales.
- Se hicieron **194** predicciones correctas sobre los datos de prueba, o bien, un **92.38%** de predicciones correctas sobre de los datos de prueba totales.

Gaussian NB

In [304]:
```
pred_training = gaussian_nb.predict(training_data_var)
reference = len(training_data_var)
count = countCorrectPredictions(training_data_labels_binary, pred_tra
ining)
print 'Gaussian NB - Predicción sobre datos de Entrenamiento'
print 'Predicciones correctas: ' + str(count)
print '% de predicciones correctas: ' + str(1.0*count / reference)

print ''

pred_test = gaussian_nb.predict(test_data_var)
reference = len(test_data_var)
count = countCorrectPredictions(test_data_labels_binary, pred_test)
print 'Gaussian NB - Predicción sobre datos de Prueba'
print 'Predicciones correctas: ' + str(count)
print '% de predicciones correctas: ' + str(1.0*count / reference)

print ''
```

```
Gaussian NB - Predicción sobre datos de Entrenamiento
Predicciones correctas: 470
% de predicciones correctas: 0.961145194274

Gaussian NB - Predicción sobre datos de Prueba
Predicciones correctas: 202
% de predicciones correctas: 0.961904761905
```

Esto es, con el clasificador NB que asume distribución Multinomial:

- Se hicieron **470** predicciones correctas sobre los datos de entrenamiento, o bien, un **96.11%** de predicciones correctas sobre de los datos de entrenamiento totales.
- Se hicieron **202** predicciones correctas sobre los datos de prueba, o bien, un **96.19%** de predicciones correctas sobre de los datos de prueba totales.

Conclusiones

Se observa que el clasificador NB que asumió una distribución Gaussiana es el que mejor realizó las predicciones de tumores, tanto para los datos de entrenamiento como para los datos de prueba. De cierto modo, esto concuerda con la intuición que habíamos generado al explorar los datos: parece que éstos siguen distribuciones normales con sesgos hacia la izquierda o hacia la derecha.

Por el lado de los datos faltantes, se decidió no usar los otros métodos que se habían planteado (sustituir los datos faltantes por 0, o eliminar la muestra completa de los datos), ya que por la forma en que estaban distribuidos los datos, en ningún caso la sustitución que utilizamos pareció sesgar las distribuciones más de lo que ya estaban.
