Problema 3

In [12]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder
from sklearn import metrics
from sklearn.utils.fixes import signature
```

Comenzamos importando los datos pre-procesados en las matrices de atributos X_t (entrenamiento) y X_v (validación) y los vectores de etiquetas Y_t (entrenamiento) y Y_v (validación).

In [13]:
```
training = pd.read_csv('juegos_entrenamiento.csv', header=None)
validation = pd.read_csv('juegos_validacion.csv', header=None)
training_values = training.values
validation_values = validation.values
Xt = training_values[:,:-1]
Yt = training_values[:, -1]
Xv = validation_values[:, :-1]
Yv = validation_values[:, -1]
```

Reparametrización de los vectores de entrada

A continuación realizaremos la reparametrización de los vectores de entrada, pues, como se explicará más adelante, esta reparametrización es necesaria tanto para el clasificador bayesiano ingenuo de distribución multinomial como para el clasificador por regresión logística que se entrenarán.

Por la forma en que están dados los vectores de entrada (dos columnas de números enteros), si se entrenara un clasificador por regresión lógistica, o un clasificador bayesiano ingenuo de distribución multimonial con base en estos datos crudos, los clasificadores interpretarían un orden entre los datos que realmente no es de relavancia para nuestro problema de clasificación: es decir, tener un ID de jugador más alto que otro, no debería afectar en los resultados que dicho jugador tenga a la hora de medirse contra otros jugadores.

Por este motivo, debemos reorganizar nuestros atributos de manera que no se implique ningún orden entre ellos. Ya que tenemos 143 identificadores de jugadores posibles por columna, proponemos el hacer a cada uno (por cada columna) un atributo por sí solo. Bajo esta transformación, tendremos 2 x 143 = 286 atributos dentro de nuestra matriz de entradas, cada uno representado por su propia columna. Cuando el jugador en cuestión esté presente en un determinado juego, la matriz de entrada tendrá un "1" en el atributo correspondiente a dicho jugador. Esta columna tendrá 0 en todos los juegos donde dicho jugador no participe.

Este proceso, llamado *One Hot Encoding* se llevará a cabo con la ayuda de la clase *OneHotEncoder* de *sklearn*, tal y como se muestra a continuación.

In [4]:
```
club_categories = [np.arange(1, 143), np.arange(1, 143)]
enct = OneHotEncoder(categories=club_categories)
# Training set
enct.fit(Xt)
Xte = enct.transform(Xt).toarray()
# Validation set
encv = OneHotEncoder(categories=club_categories)
encv.fit(Xv)
Xve = encv.transform(Xv).toarray()
```

Clasificador bayesiano ingenuo

Ya que yo no construí ningún clasificador bayesiano ingenuo en la tarea 2, para este problema, utilizaré un clasificador bayesiano ingenuo de distribución multinomial basado en la clase *MultinomialNB* de sklearn. Usaremos, por las razones presentadas arriba, la matriz de entrada modificada (286 atributos) para este fin.

In [23]:
```
from sklearn.naive_bayes import MultinomialNB
mnb = MultinomialNB()
mnb.fit(Xte, Yt)
print 'Tasa de predicciones correctas para:'
print 'Datos de entrenamiento: %0.04f' % mnb.score(Xte, Yt)
print 'Datos de validación: %0.04f' % mnb.score(Xve, Yv)
```

```
Tasa de predicciones correctas para:
Datos de entrenamiento: 0.8464
Datos de validación: 0.6957
```

Generación de partidas

Para generar partidas, en primer lugar, generamos dos números, ID1 e ID2, entre 1 y 146 aleatoriamente (cuidando sólo que no se elija al mismo jugador). Esta elección nos da a nuestros datos de entrada X. Queremos, dado este X, dar pues las probabilidades de que el resultado de que estos dos jugadores específicos se enfrenten entre ellos sean 1 ó 0. Es decir, queremos saber P(Y=0|X=ID1,ID2) y P(Y=1|X=ID1,ID2). Como conocemos P(Y=1) y P(Y=0), y conocemos P(X=ID1, ID2|Y) (de las partidas que ambos hayan jugado en el pasado), podemos generar P(X, Y), y por tanto, podemos generar P(Y=0|X=ID1,ID2) y P(Y=1|X=ID1,ID2). Si los jugadores han jugado en el pasado, estas probabilidades deberían sumar 1, por lo que basta con generar un número aleatorio entre 0 y 1, y verificar si este número es mayor que la menor probabilidad de P(Y=0|X=ID1,ID2) y P(Y=1|X=ID1,ID2). Si es mayor, el resultado es 1 (gana ID1), si es menor, el resultado es asignado a 0 (gana ID2).

Clasificador por regresión logística

De manera similar a como ocurre con las regresiones lineales, el clasificador por regresión logística se encarga de minimizar el error existente entre las predicciones hechas y las etiquetas reales de un modelo. De la misma forma a como se hizo anteriormente, queremos obtener un vector de parámetros θ que minimice pues esta diferencia. No obstante, no existe una forma cerrada para obtener el vector θ, por lo que tenemos que hacer un descenso de gradiente que nos lleve, desde una inicialización dada de θ hacia un vector θ que optimice nuestro problema planteado (i.e. que minimice el error de predicciones vs. etiquetas).

El gradiente en cuestión es el siguiente:

$$g(\theta) = X^T(sigm(\theta^T X) - Y)$$

Donde:

$$sigm(z) = \frac{1}{1 + exp(-z)}$$

El trabajo anterior es realizado por la funcion *logistic-regression* mostrada abajo, misma que utiliza las funciones auxiliares *fit*, *update-theta*, *predict* y *sigmoid*. Todas estas funciones están mostradas a continuación.

In [11]:
```
def logistic_regression(Xt, Yt, rate, n):
    # Create design matrix
    phi_Xt = np.column_stack((np.ones(Xt.shape[0]), Xt))
    # Initialize theta
    theta = np.ones(Xt.shape[1] + 1)
    # Return vector of parameters theta
    return fit(phi_Xt, Yt, theta, rate, n)

# Perform a gradient descent in order to find, given a set of inputs
 Xt and labels Yt, the
# parameters theta. This gradient descent uses a learning rate "rat
e", through n iterations
def fit(Xt, Yt, theta, rate, n):
    for i in range(n):
        theta = update_theta(Xt, Yt, theta, rate)
    return theta

# Helper function for gradient descent
def update_theta(Xt, Yt, theta, rate):
    Yh = predict(Xt, theta)
    # Calculate the gradient descent
    gradient = (1.0/Xt.shape[0])*np.dot(Xt.T, Yh - Yt)
    gradient *= rate
    theta -= gradient
    return theta

# Make predictions over X inputs with parameters theta
def predict(X, theta):
    z = np.dot(X, theta)
    return sigmoid(z)

def get_probabilities(X, theta):
    phi_X = np.column_stack((np.ones(X.shape[0]), X))
    z = np.dot(phi_X, theta)
    return sigmoid(z)

# Calculate the sigmoid of z (where z is a vector)
def sigmoid(z):
    return 1 / (1 + np.exp(-1*z))

# Classify the probabilities vector Y (if Y(i)>0.5, then Y'(i)=1)
def classify(Y):
    Yc = []
    for p in Y:
        if p > 0.5:
            Yc.append(1)
        else:
            Yc.append(0)
    return Yc

# Calculate the % of correct predictions given a set of actual labels
Y, and a set of
# predicted labels Yc
def error(Y, Yc):
    diff = Y - Yc
    return 1.0 - np.count_nonzero(diff) / float(len(diff))
```

```
# Calculate the performace of the parameteres theta with the set X/Y
def performance(X, Y, theta):
    Yh = predict(np.column_stack((np.ones(X.shape[0]), X)), theta)
    Yc = classify(Yh)
    return error(Y, Yc)
```

Ya que la función _logistic*regression* regresa un vector de parámetros, podemos usar este vector de parámetros para calcular nuestro vector de predicciones. No obstante, el clasificador no hará la clasificación directamente; en lugar de ser un vector de predicciones donde los valores de una predicción pueden ser 1 ó 0, el vector de predicciones arrojado por el modelo es un *vector de probabilidades*, donde si una predicción tiene una probabilidad mayor que un umbral *u*, dicha predicción se clasifica como "1". Si esta probabilidad, en su lugar, es menor que este umbral, dicha predicción se clasifica como "0". Este trabajo de interpretación/clasificación, es llevado a cabo por la función *classify* mostrada arriba, misma que usa como umbral un valor predefinido de 0.5.

Para facilitar el cálculo del desempeño de un modelo entrenado bajo las funciones anteriores, usamos la función *performance*, misma que calcula el número de predicciones correctas con respecto del vector de etiquetas asociado a la matriz de entradas utilizada.

A continuación, entrenamos y verificamos el desempeño de nuestro modeo usando las funciones definidas arriba. Para el descenso del gradiente, utilizamos una tasa de aprendizaje igual a 0.1 durante 10000 iteraciones.

In [14]:
```
theta = logistic_regression(Xte, Yt, 0.1, 10000)
print 'Tasa de predicciones correctas para:'
print 'Datos de entrenamiento: %0.04f' % performance(Xte, Yt, theta)
print 'Datos de validación: %0.04f' % performance(Xve, Yv, theta)
```

```
Tasa de predicciones correctas para:
Datos de entrenamiento: 0.9216
Datos de validación: 0.7130
```

Curvas ROC y de Precisión/Exhaustividad

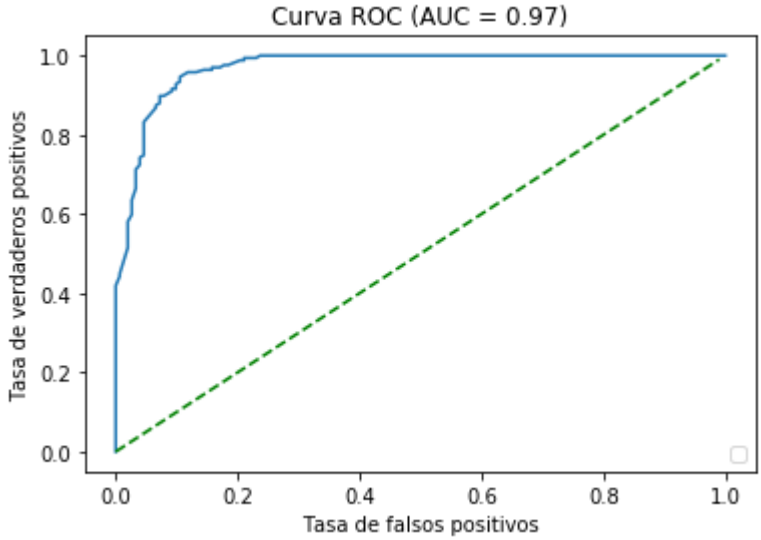
Para calcular las curvas ROC y de Precisión/Exhaustividad, utilizaremos las funciones *roc-curve* y *precision-recall-curve*, de la clase *sklearn.metrics* respectivamente. Asímismo, para calcular las áreas bajo las curvas respectivas, usaremos *roc-auc-score* y *average-precision-score*.

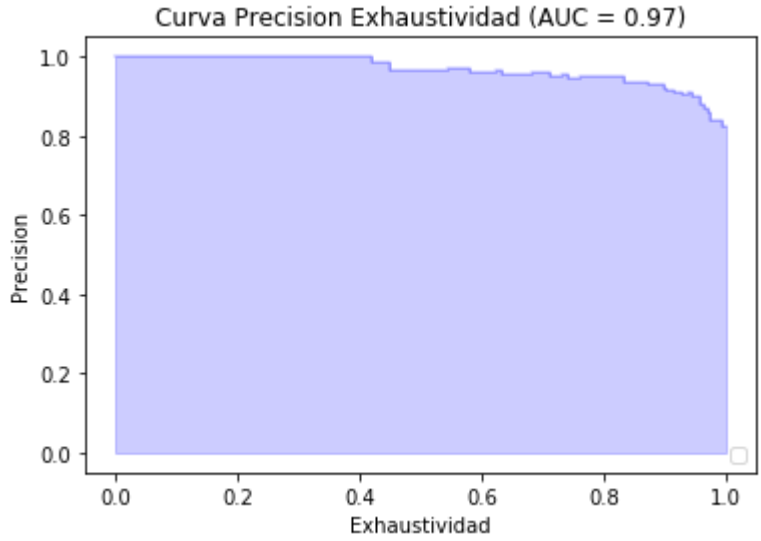
In [17]:
```
def plot_ROC(Y, YProb):
    fpr, tpr, thresholds = metrics.roc_curve(Y, YProb)
    auc = metrics.roc_auc_score(Y, YProb)
    plt.plot(fpr, tpr)
    plt.title('Curva ROC (AUC = %0.2f)' % auc)
    plt.plot(np.arange(0, 1, 0.01), np.arange(0, 1, 0.01),linestyle=
'--', color='g')
    plt.legend(loc="lower right")
    plt.xlabel('Tasa de falsos positivos')
    plt.ylabel('Tasa de verdaderos positivos')
    plt.show()

def plot_PE(Y, YProb):
    pr, ex, thresholds = metrics.precision_recall_curve(Y, YProb)
    aps = metrics.average_precision_score(Y, YProb)
    step_kwargs = ({'step': 'post'}
               if 'step' in signature(plt.fill_between).parameters
               else {})
    plt.step(ex, pr, color='b',alpha=0.2, where='post')
    plt.fill_between(ex, pr, alpha=0.2, color='b', **step_kwargs)
    plt.title('Curva Precision Exhaustividad (AUC = %0.2f)' % aps)
    plt.legend(loc="lower right")
    plt.xlabel('Exhaustividad')
    plt.ylabel('Precision')
    plt.show()
```

Regresión Logística

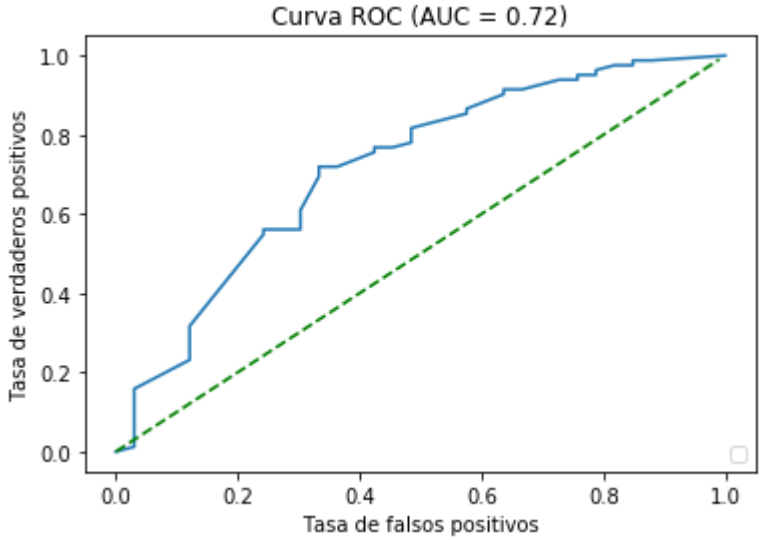
Con las funciones anteriores, hacemos las gráficas requeridas. Se observa que la curva ROC para el modelo de regresión logística tiene un área bajo la curva muy cercana a 1, lo que implica que los valores de las predicciones, en la gran mayoría de los casos, son hechas a partir de probabilidades relativamente extremas (i.e. muy cercanas a 0 o muy cercanas a 1).

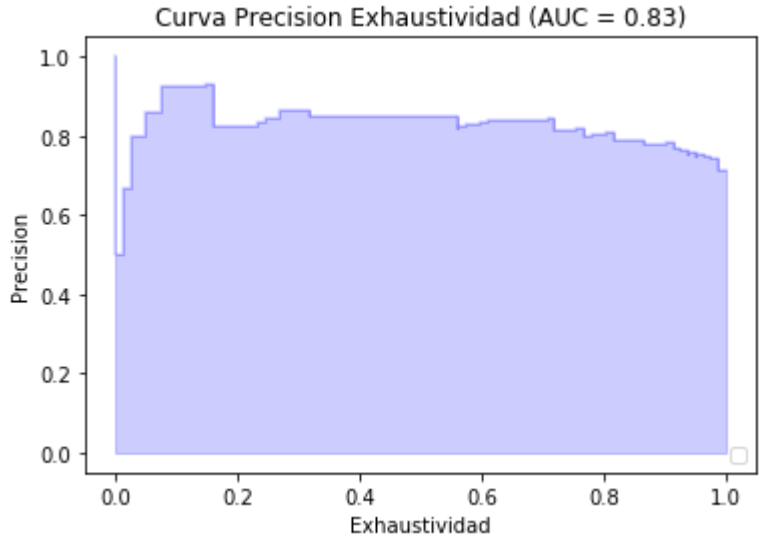
In [18]:
```
YtProb = get_probabilities(Xte, theta)
plot_ROC(Yt, YtProb)
plot_PE(Yt, YtProb)
```





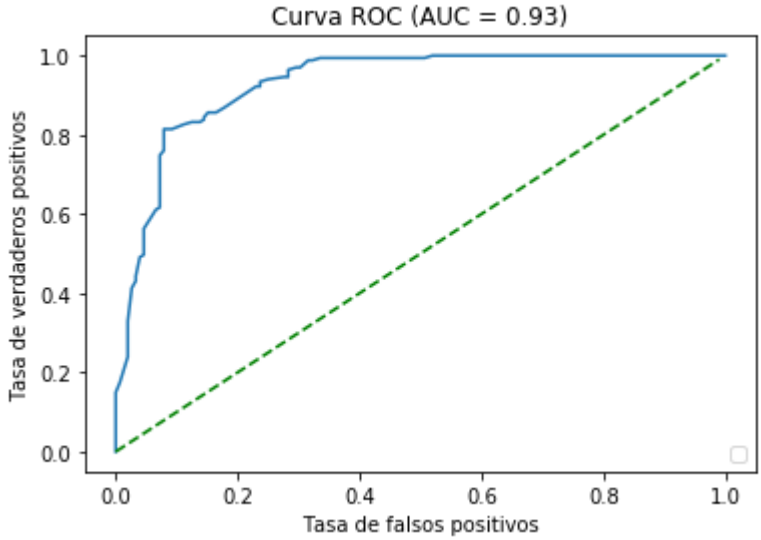
In [20]:
```
YvProb = get_probabilities(Xve, theta)
plot_ROC(Yv, YvProb)
plot_PE(Yv, YvProb)
```

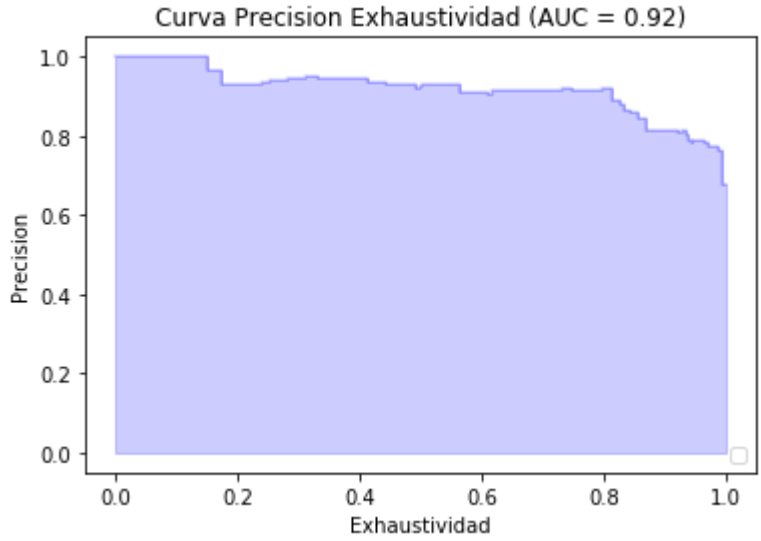




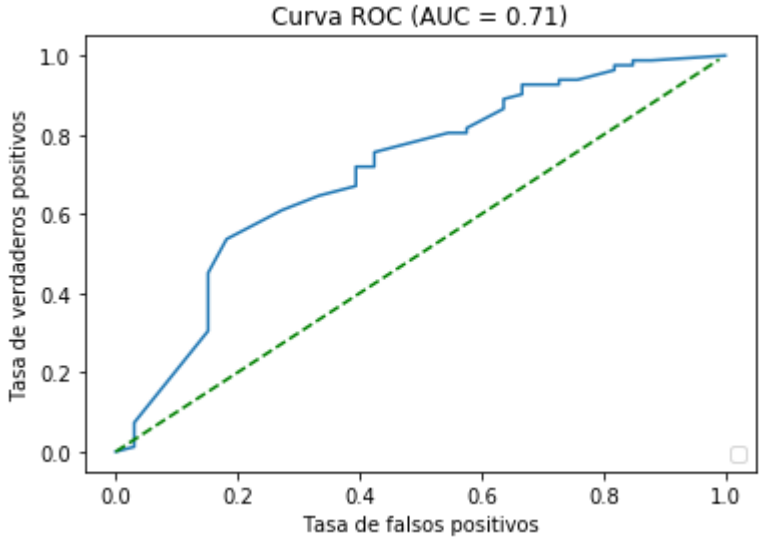
Naive Bayes

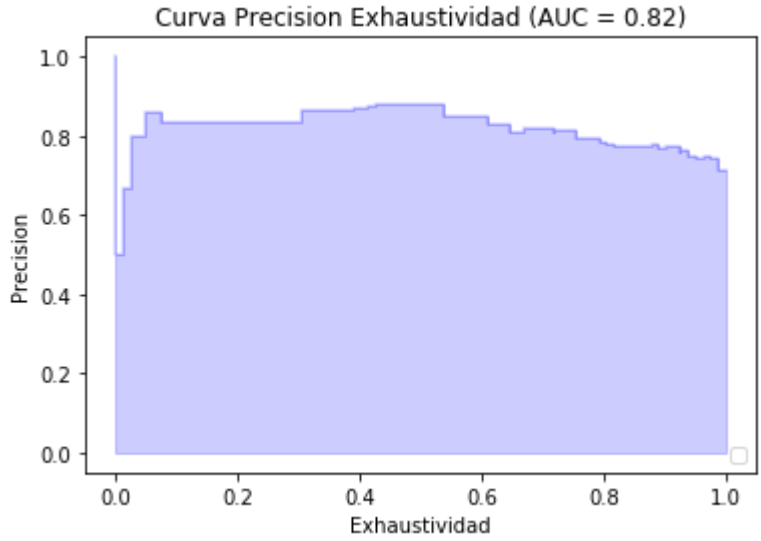
In [21]:
```
YtProb = mnb.predict_proba(Xte)
plot_ROC(Yt, YtProb[:,1])
plot_PE(Yt, YtProb[:,1])
```





In [22]:
```
YvProb = mnb.predict_proba(Xve)
plot_ROC(Yv, YvProb[:,1])
plot_PE(Yv, YvProb[:,1])
```





Comparación de modelos

En general, para este problema, se observó que el clasificador por regresión logística entrenado tuvo un mejor desempeño que el clasificador de bayes ingenuo. Quizá la primera característica que salta a la vista de una ventaja que tiene el clasificador por regresión logística vs NB es que NB asume que las variables de entrada son independientes, mientras que el CRL no. Para este conjunto de datos, asumir que las variables de entrada son independientes posiblemente es una aseveración demasiado fuerte (pues la derrota o victoria de un jugador contra otro bien podría afectar en el desempeño del mismo jugador en sus partidas subsecuentes).

El clasificador bayesiano, por otro lado, si hubiera sido entrenado por medio de una distribución categórica y no una multinomial como en este caso, no habría necesitado del OneHotEncoding. El CLR necesita de este OHE, de otro modo, no podría funcionar (debido al problema del orden explicado anteriormente). Computacionalmente hablando, esto supone una ventaja vs. el CLR en caso de que la base de datos fuera más grande. La velocidad de entrenamiento es en términos generales un punto a favor del clasificador bayesiano ingenuo; al depdender el CLR de un método numérico como el de descenso de gradiente, su entrenamiento siempre toma más tiempo.