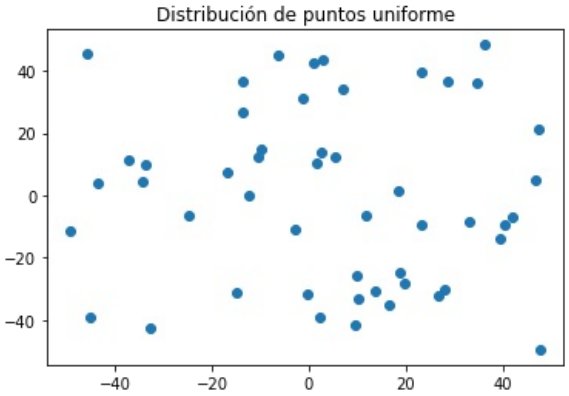
Práctica 2 de AA

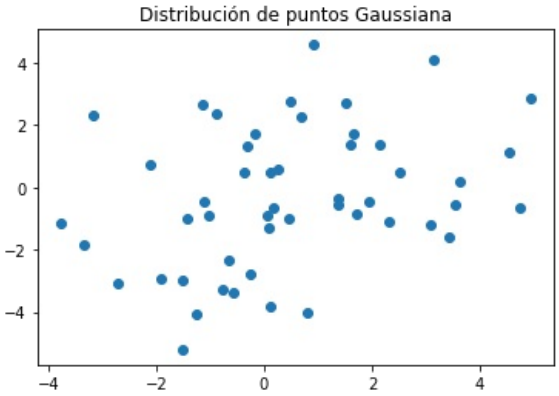
Ejercicio 1

En este ejercicio debemos aprender la dificultad que introduce la aparición de ruido en las etiquetas a la hora de elegir la clase de funciones más adecuada. Haremos uso de tres funciones ya programadas:

- $simula\_unif(N, dim, rango)$, que calcula una lista de N vectores de dimensión dim. Cada vector contiene dim números aleatorios uniformes en el intervalo $rango$.
- $simula\_gaus(N, dim, sigma)$, que calcula una lista de longitud N de vectores de dimensión dim, donde cada posición del vector contiene un número aleatorio extraido de una distribucción Gaussiana de media 0 y varianza dada, para cada dimension, por la posición del vector $sigma$.
- $simula\_recta(intervalo)$, que simula de forma aleatoria los parámetros, $v = (a, b)$ de una recta, $y = ax + b$, que corta al cuadrado $[-50, 50] \times [-50, 50]$.

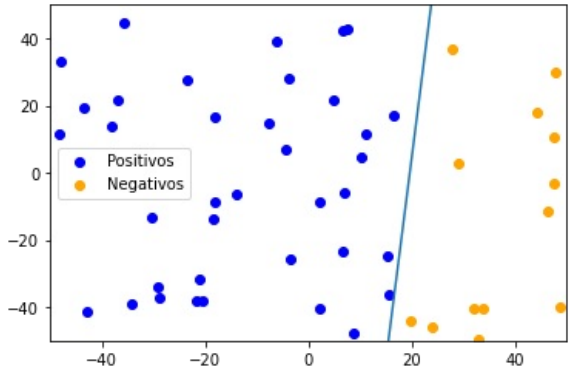
1. (1 punto) Dibujar una gráfica con la nube de puntos de salida correspondiente.
   a) Considere $N = 50$, $dim = 2$, $rango = [-50, +50]$ con $simula\_unif(N, dim, rango)$.
   b) Considere $N = 50$, $dim = 2$ y $sigma = [5, 7]$ con $simula\_gaus(N, dim, sigma)$.





No hay muchos puntos, pero se puede apreciar que en la función que reparte los puntos simulando una distribución Gaussiana, los puntos están más concentrados en torno a lo que sería la media (0) de la distribución.

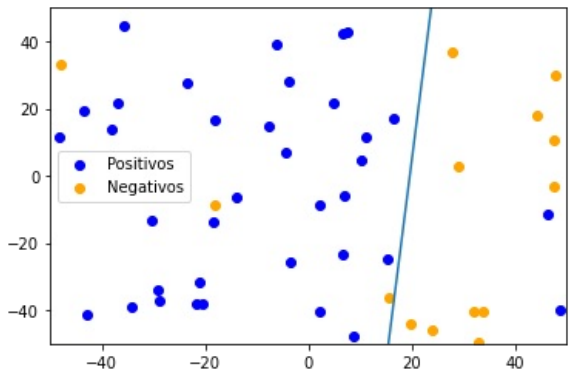
2. (1.5 puntos) Con ayuda de la función *simula_ unif()* generar una muestra de puntos 2D a los que vamos añadir una etiqueta usando el signo de la función $f(x, y) = y - ax - b$, es decir el signo de la distancia de cada punto a la recta simulada con *simula_ recta()*.
   a) Dibujar una gráfica donde los puntos muestren el resultado de su etiqueta, junto con la recta usada para ello. (Observe que todos los puntos están bien clasificados respecto de la recta)
   b) Modifique de forma aleatoria un 10 % etiquetas positivas y otro 10 % de negativas y guarde los puntos con sus nuevas etiquetas. Dibuje de nuevo la gráfica anterior. (Ahora hay puntos mal clasificados respecto de la recta)



La linea separa perfectamente las dos clases.

In [8]:

```
# Para añadir estrictamente el ruido que se pide

def addNoiseToLabels(labels):
  numOfPositives = labels.count(1)
  positivesToDelete = numOfPositives * 0.1
  positivesToBecameNegatives = []

  numOfNegatives = labels.count(-1)
  negativesToDelete = numOfNegatives * 0.1
  negativesToBecamePositives = []

  while (positivesToDelete > 0):
    index = np.random.randint(0, len(labels)-1)
    if labels[index] == 1:
      positivesToBecameNegatives.append(index)
      positivesToDelete-=1

  while (negativesToDelete > 0):
    index = np.random.randint(0, len(labels)-1)
    if labels[index] == -1:
      negativesToBecamePositives.append(index)
      negativesToDelete-=1


  for i in positivesToBecameNegatives:
    labels[i] = -1

  for i in negativesToBecamePositives:
    labels[i] = 1
```

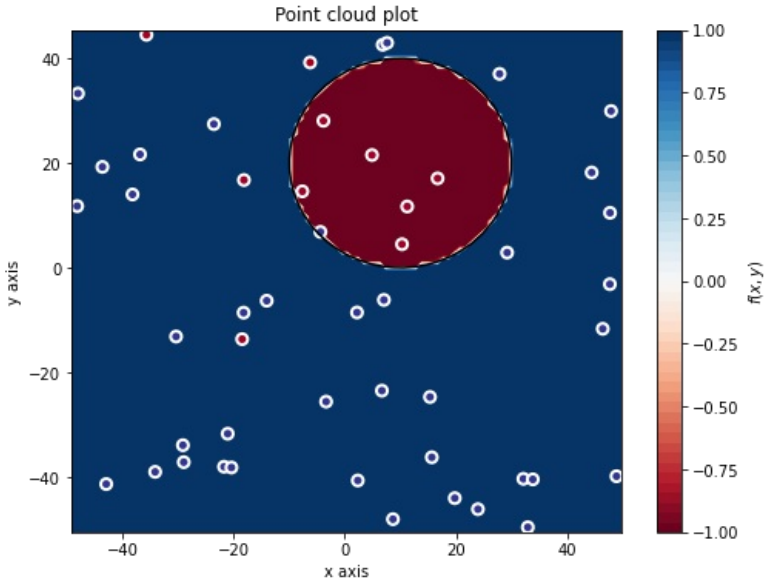


Podemos observar como hay 3 puntos de la clase positiva que han "mutado" y 2 de la negativa que han hecho lo mismo. De la clase positiva había más y había que convertir un 10 %, por eso hay más ruido ahí.

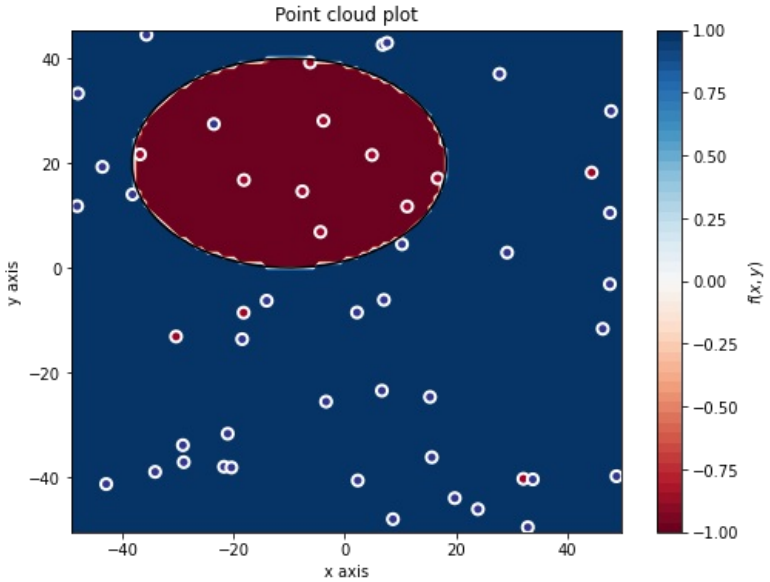
3. (2.5 puntos) Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta

   Visualizar el etiquetado generado en $2b$ junto con cada una de las gráficas de cada una de las funciones. Comparar las formas de las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta ¿Son estas funciones más complejas mejores clasificadores que la función lineal? ¿En que ganan a la función lineal? Explicar el razonamiento.

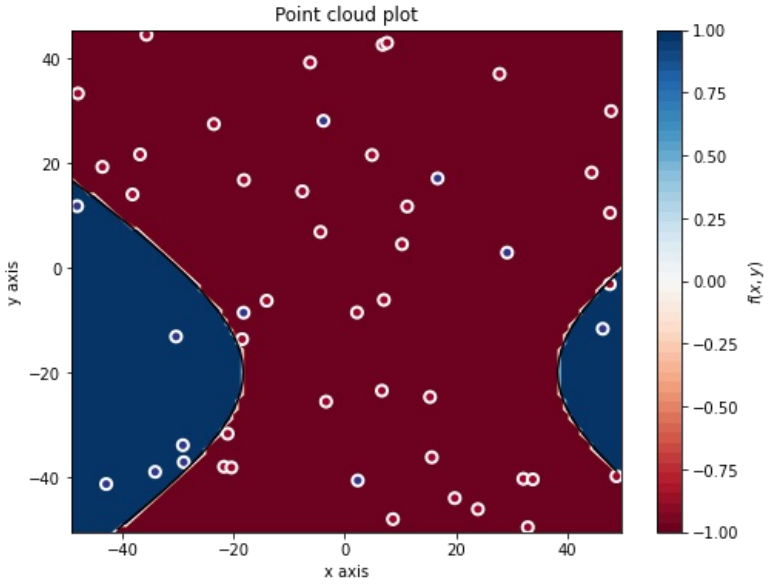
- $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$



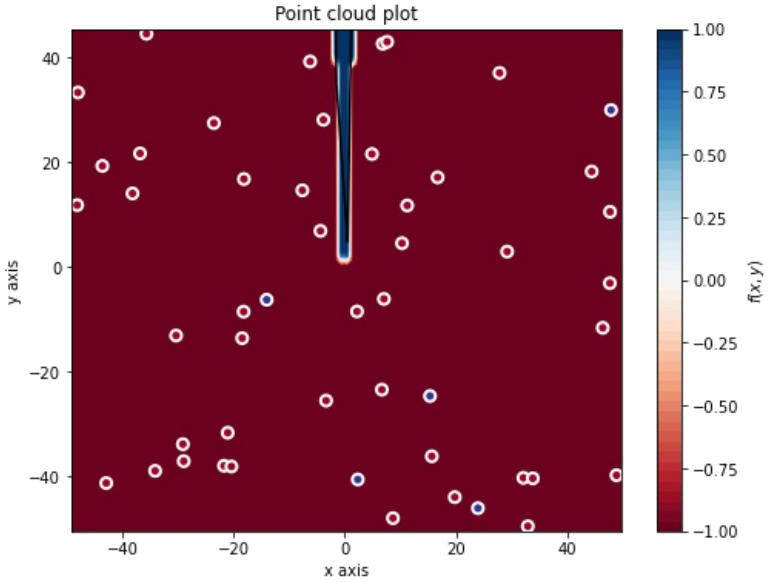
- $f(x, y) = 0{,}5(x + 10)^2 + (y - 20)^2 - 400$



- $f(x, y) = 0{,}5(x - 10)^2 - (y + 20)^2 - 400$



- $f(x, y) = y - 20x^2 - 5x + 3$



Estas funciones, más complejas, no necesariamente son mejores clasificadores. Todo depende del problema que se pretenda resolver, en este caso, el problema lo hemos creado nosotros y es uno en el que una aproximación con funciones lineales es mucho más sensata.

Hablando ya en términos de sesgo y varianza, la función lineal tiene un sesgo mucho más pronunciado y es menos flexible en cuanto a adaptación al problema. Sin embargo, la varianza es menor, la penalización en datos que no conocemos será menor también.

Sucede al contrario con las funciones más complejas, capaces de ajustarse de manera perfecta a todos los datos de la muestra de entrenamiento (bajo sesgo) pero con una varianza mucho mayor, que penalizará en Eout.

Ejercicio 2

1. **(3 puntos) Algoritmo Perceptron:** Implementar la función $ajusta\_PLA(datos, label, max\_iter, vini)$ que calcula el hiperplano solución a un problema de clasificación binaria usando el algoritmo

PLA. La entrada *datos* es una matriz donde cada item con su etiqueta está representado por una fila de la matriz, *label* el vector de etiquetas (cada etiqueta es un valor $+1$ o -1), max_iter es el número máximo de iteraciones permitidas y *vini* el valor inicial del vector. La función devuelve los coeficientes del hiperplano.

   a) Ejecutar el algoritmo PLA con los datos simulados en los apartados $2a$ de la sección.1. Inicializar el algoritmo con: a) el vector cero y, b) con vectores de números aleatorios en $[0, 1]$ (10 veces). Anotar el número medio de iteraciones necesarias en ambos para converger. Valorar el resultado relacionando el punto de inicio con el número de iteraciones.

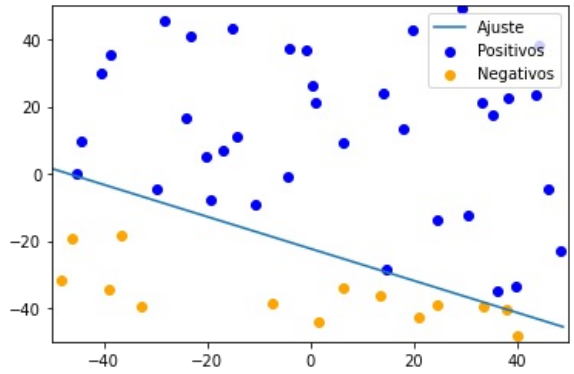
   b) Hacer lo mismo que antes usando ahora los datos del apartado $2b$ de la sección.1. ¿Observa algún comportamiento diferente? En caso afirmativo diga cual y las razones para que ello ocurra.

In [16]:

```
#Función de error para problemas de clasificación

def error(data, labels, w):
  # x (dot) wt
  # n x 3 | 3 x 1 -> n x 1
  guessUnsigned = data.dot(w.reshape(3,-1))

  count = 0
  # Contamos los puntos mal clasificados
  for i in range(len(guessUnsigned)):
    if labels[i] != sign(guessUnsigned[i]):
      count += 1

  # Devolvemos el promedio

  return count / len(data)

# PLA

def pla(data, labels, max_iter, vini):
  # Copiamos la w inicial
  w = vini.copy()

  hasConverged = False
  pasadas = 0

  # Si no hemos llegado al número máximo
  # de pasadas y no ha convergido...
  # SEGUIMOS!

  while pasadas != max_iter and not hasConverged:
    hasConverged = True
    for i in range(len(data)):
      guess = sign(w.dot(data[i].reshape(-1,1)))
      # En caso de fallar la predicción
      # ajustamos los pesos
      if guess != labels[i]:
        hasConverged = False
        w += labels[i] * data[i]

    pasadas += 1

  return w, pasadas
```

El algoritmo perceptrón nos garantiza la convergencia en un número finito de pasos si los datos son linealmente separables. Pasa por todos los datos disponibles en el mismo orden una y otra vez hasta que converge (si no lo hace itera infinitamente, aquí no, porque hemos puesto número máximo de iteraciones). También se podrían barajear los datos antes de cada pasada, aunque en teoría hemos visto el esquema que aquí codifico.

```
PLA - Puntos linealmente separables - Con un vector de ceros
Coordenadas [595.          12.69901395  26.6961201 ]
Error 0.0
Iteraciones 87
```

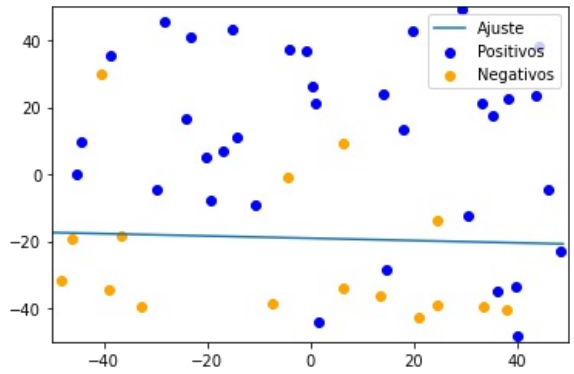


Podemos observar que efectivamente ha convergido en un número finito de pasos.

```
PLA - Puntos linealmente separables - Puntos de arranque estocásticos
Error medio 0.0
Iteraciones de media 111.3
Desviación típica de las iteraciones 17.720327310746832
```

Ahora arrancamos estocásticamente. Podemos ver por la desviación típica que importa el punto inicial. Por la naturaleza del algoritmo tanto el punto inicial como el orden en el que se van visitando los datos influyen en el tiempo de cómputo.

```
PLA - Puntos NO linealmente separables - Con un vector de ceros
Coordenadas [832.           1.49035519  43.54868533]
Error 0.2
Iteraciones 500
```



Ahora la cosa cambia. Los datos ya no son linealmente separables y el algoritmo consume el número máximo de iteraciones especificadas.

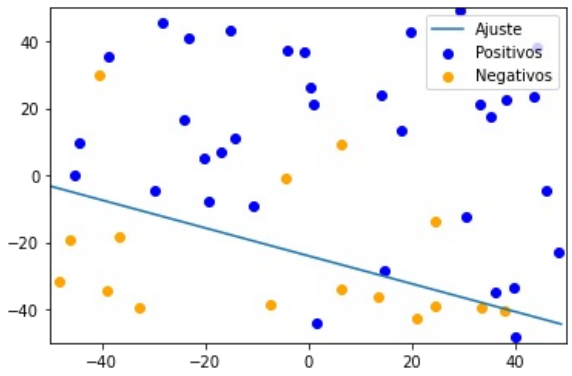
```
PLA - Puntos NO linealmente separables - Arrancando estocásticamente
Error medio 0.21400000000000002
Iteraciones de media 500.0
Desviación típica de las iteraciones 0.0
```

Debido a que hemos introducido ruido en la muestra, los datos dejan de ser linealmente separables y el algoritmo perceptron NUNCA converge. En todos los ajustes gastamos el número máximo de iteraciones. Para este tipo de problemas en los que los datos no son linealmente separables por ruido y no por la naturaleza del problema, es preferible usar algoritmos como el Pocket.

In [22]:

```
# Implemento el POCKET para ver si podemos mejorar en muestras ruidosas

def pocket(data, labels, max_iter, vini):
  # Copiamos la w inicial
  w = vini.copy()

  hasConverged = False
  pasadas = 0

  best_w = w.copy()
  best_error = error(data, labels, w)

  # Si no hemos llegado al número máximo
  # de pasadas y no ha convergido...
  # SEGUIMOS!

  while pasadas != max_iter and not hasConverged:
    hasConverged = True
    for i in range(len(data)):
      guess = sign(w.dot(data[i].reshape(-1,1)))
      # En caso de fallar la predicción
      # ajustamos los pesos
      if guess != labels[i]:
        hasConverged = False
        w += labels[i] * data[i]

    # Nos metemos al "pocket" la mejor
    # solución que hemos encontrado
    current_error = error(data, labels, w)

    if current_error < best_error:
      best_w = w.copy()
      best_error = current_error
    pasadas += 1

  return best_w, pasadas
```

```
PLA-POCKET, datos NO linealmente separables. Arrancando con un vector de ceros
Coordenadas [455.          7.87197122  18.92445095]
Error 0.12
Iteraciones 500
```



Misión conseguida, hemos conseguido bajar el error. Además, la solución se parece mucho a la obtenida por el PLA sin ruido.

2. **(4 puntos) Regresión Logística**: En este ejercicio crearemos nuestra propia función objetivo f (una probabilidad en este caso) y nuestro conjunto de datos $\mathcal{D}$ para ver cómo funciona regresión logística. Supondremos por simplicidad que f es una probabilidad con valores $0/1$ y por tanto que la etiqueta y es una función determinista de $\mathbf{x}$.

   Consideremos $d = 2$ para que los datos sean visualizables, y sea $\mathcal{X} = [0, 2] \times [0, 2]$ con probabilidad uniforme de elegir cada $\mathbf{x} \in \mathcal{X}$. Elegir una línea en el plano que pase por $\mathcal{X}$ como la frontera entre $f(\mathbf{x}) = 1$ (donde y toma valores $+1$) y $f(\mathbf{x}) = 0$ (donde y toma valores -1), para ello seleccionar dos puntos aleatorios del plano y calcular la línea que pasa por ambos. Seleccionar $N = 100$ puntos aleatorios $\{\mathbf{x}_n\}$ de $\mathcal{X}$ y evaluar las respuestas $\{y_n\}$ de todos ellos respecto de la frontera elegida.

   a) Implementar Regresión Logística(RL) con Gradiente Descendente Estocástico (SGD) bajo las siguientes condiciones:

      - Inicializar el vector de pesos con valores 0.
      - Parar el algoritmo cuando $||\mathbf{w}^{(t-1)} - \mathbf{w}^{(t)}|| < 0{,}01$, donde $\mathbf{w}^{(t)}$ denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos.
      - Aplicar una permutación aleatoria, $1, 2, \ldots, N$, en el orden de los datos antes de usarlos en cada época del algoritmo.
      - Usar una tasa de aprendizaje de $\eta = 0{,}01$

Utilizamos en esta ocasión minibatches de tamaño 1. Pero, como dice en el guión, permutamos el orden en el que iteramos por sus datos. Este patrón estocástico lo encontramos frecuentemente en problemas de clasificación.

Además de eso, otra diferencia que nos encontramos con el SGD que ya conocíamos es la nueva función del gradiente, adecuada a la inclusión de la sigmoide en vez de wt*x como hacíamos antes.

$$\nabla E_{\text{in}} = -\frac{1}{N}\sum_{n=1}^{N}\frac{y_n \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^{\intercal}(t)\mathbf{x}_n}}$$

Por último, la condición de parada: En vez de utilizar un número fijado de iteraciones, computamos la normal de la diferencia entre la w de la actual iteración y la anterior. Fijamos 0.01 como incremento mínimo para seguir iterando.

In [25]:

```
def sgd(x, y, eta, cambio_minimo, w=np.zeros(3)):

  cambio_actual = np.inf

  indices = np.arange(0, len(x))

  while cambio_actual >= cambio_minimo:

    # Nos quedamos con el w pre-iteración
    # para poder hacer el incremento después
    w_anterior = w.copy()

    # Permutación
    np.random.shuffle(indices)

    for i in indices:
      xn = x[i]
      yn = y[i]
      gt = -( yn * xn ) / ( 1 + np.exp(yn * np.dot(w,xn)))
      w -= eta * gt

    cambio_actual = np.linalg.norm(w_anterior - w)

  return w
```

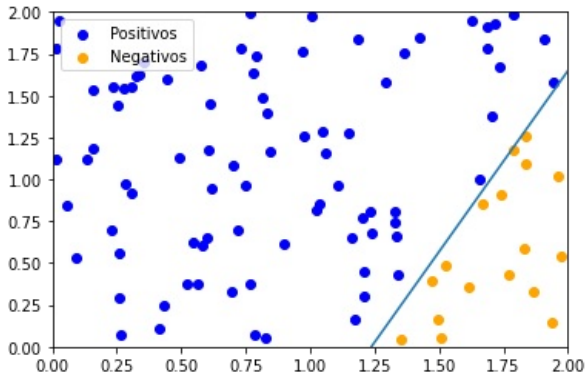
El error también cambia, la fórmula es ahora la siguiente:

$$E_{\text{in}}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \ln\left(e^{-y_i \boldsymbol{\theta}^T \mathbf{x_i}} + 1\right)$$

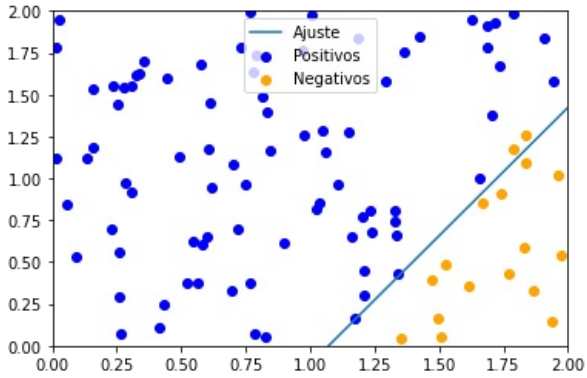
Notar que no utilizamos la función de error de clasificación (contar etiquetas mal clasificadas). La regresión logística tiene como objetivo dar el "how likely" de una etiqueta. Y por tanto las funciones de error darían resultados distintos, aún con las mismas w.

In [26]:

```
def errorSgd(x,y,w):
  error = 0
  for xn, yn in zip(x, y):
    error += np.log(1 + np.exp(-yn * w.dot(xn)))

  return error / len(x)
```

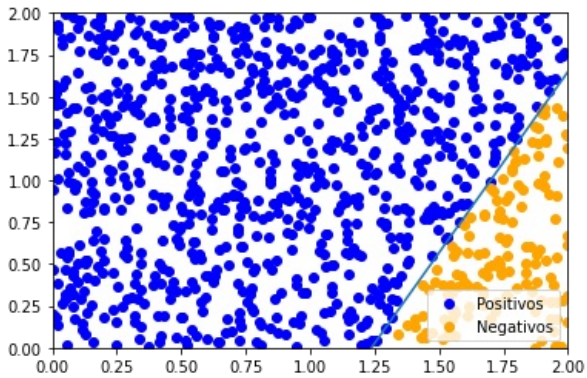



```
Ajuste de regresión logística con datos linealmente separables
Error Ein 0.11094175500242644
```

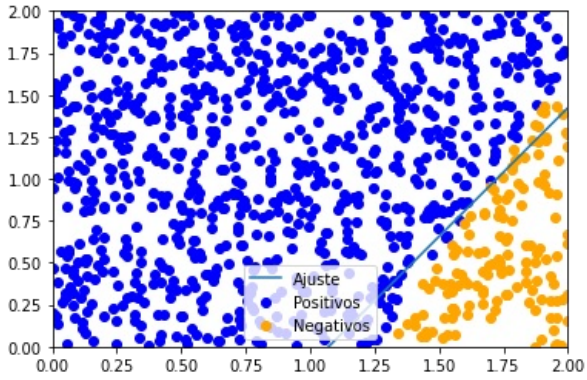


Observamos que el ajuste se parece mucho a la función target.

b) Usar la muestra de datos etiquetada para encontrar nuestra solución g y estimar E_{out} usando para ello un número suficientemente grande de nuevas muestras (>999).



```
Ajuste de regresión logística con datos linealmente separables
Error Eout 0.11310249094439043
```



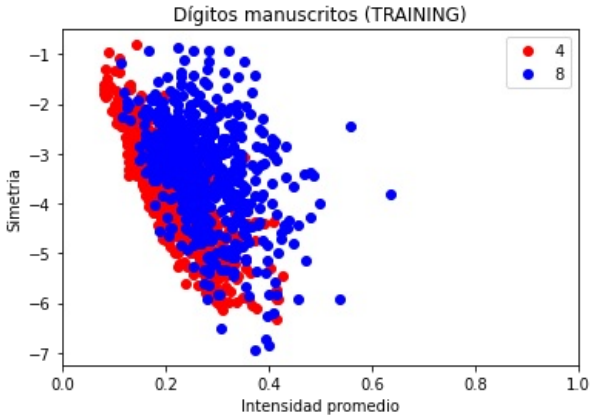
Tenemos un error muy similiar en Ein y en Eout.

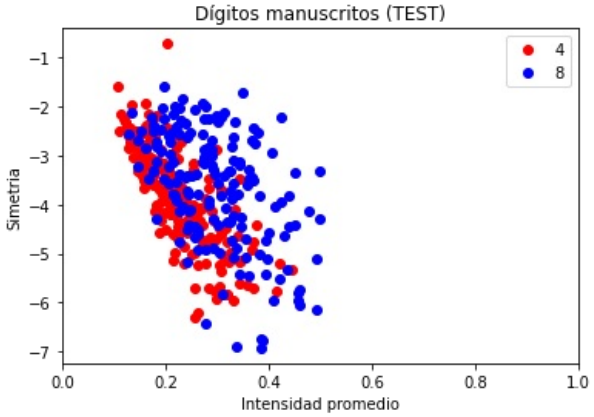
EJERCICIO 3

(1.5 puntos) Clasificación de Dígitos. Considerar el conjunto de datos de los dígitos manuscritos y seleccionar las muestras de los dígitos 4 y 8. Usar los ficheros de entrenamiento (training) y test que se proporcionan. Extraer las características de **intensidad promedio** y **simetría** en la manera que se indicó en el ejercicio 3 del trabajo 1.

1. Plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g.

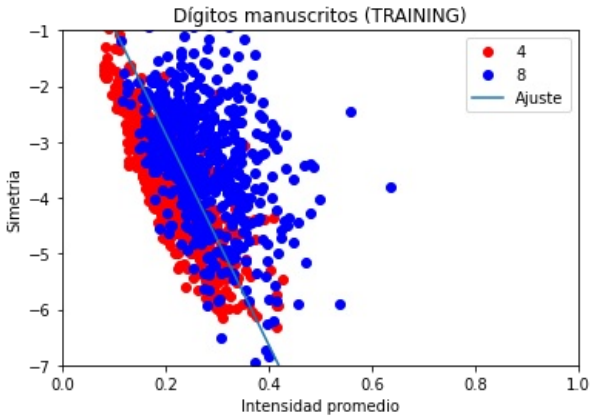
Tenemos unos datos de entrenamiento con características: intensidad promedio y simetría. Tenemos dos clases: 4 y 8. Debajo leemos (gracias a las funciones proporcionadas) los datos.



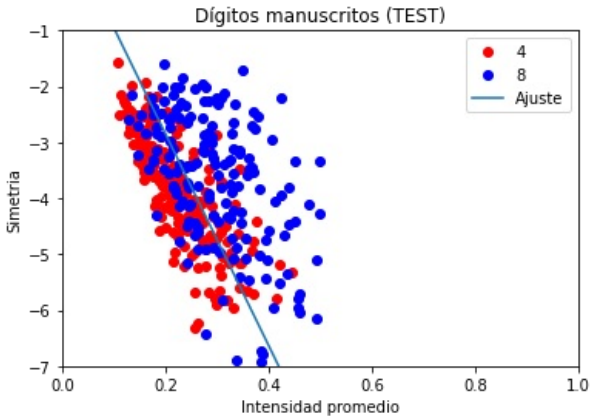


2. Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora. Responder a las siguientes cuestiones.
   a) Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.
   b) Calcular E_{in} y E_{test} (error sobre los datos de test).
   c) Obtener cotas sobre el verdadero valor de E_{out}. Pueden calcularse dos cotas una basada en E_{in} y otra basada en E_{test}. Usar una tolerancia $\delta = 0{,}05$. ¿Que cota es mejor?

```
Aplicando SGD logistic regression
Error Ein de regresión logística 0.465808879851049
Error Ein de clasificación 0.22110552763819097
```

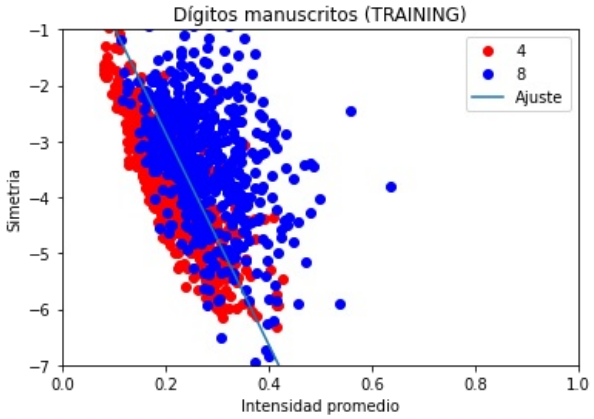



```
Error Eout de regresión logística 0.5323632634427168
Error Eout de clasificación 0.25956284153005466
```

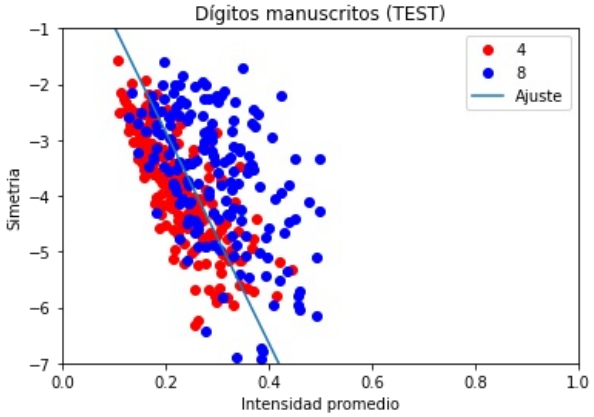


Primero entreno con el algoritmo de regresión logística que he presentado un poco más atrás. Y mido el error de dos formas, ya que como he explicado antes son diferentes y tienen distintos propósitos: El error de regresión logística y el de clasificación. El de regresión logística está orientado a problemas en los que interesa más dar una probabilidad, por ejemplo: ¿Cómo de probable es que este paciente sufra un infarto? Sin embargo, en este lo que queremos saber es si el dígito es un 4 o un 8. Nos interesa más saber cuántos 4's se interpretan como 8's y viceversa.

```
Aplicando PLA-Pocket después de logistic regression
Error Ein  0.22110552763819097
```

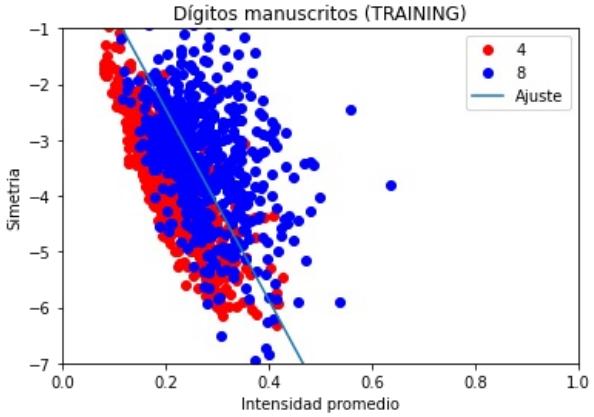



```
Error Eout  0.25956284153005466
```

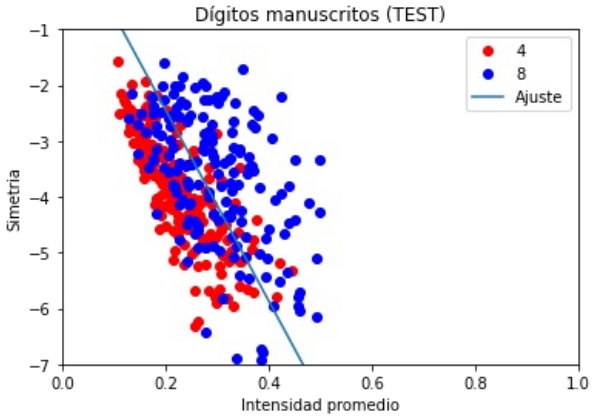


Podemos ver que el pla-pocket no es capaz de obtener un w con menor error en las 1000 iteraciones y por tanto de mejorar la predicción obtenida por el algoritmo de regresión logística. Vamos a probar ahora a utilizar el PLA-Pocket como primer algoritmo.

```
Aplicando solo PLA-POCKET, arrancando con ceros
Error Ein  0.228643216080402
```

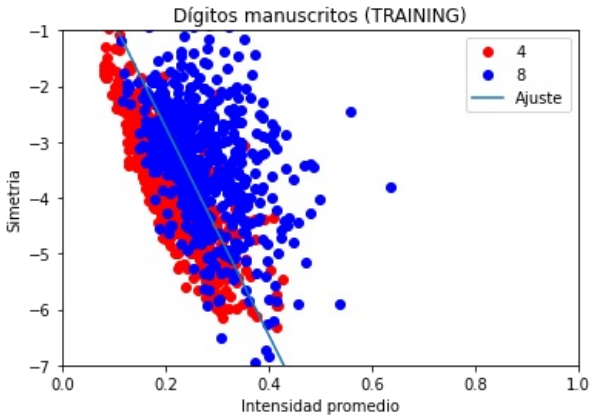



```
Error Eout  0.2459016393442623
```

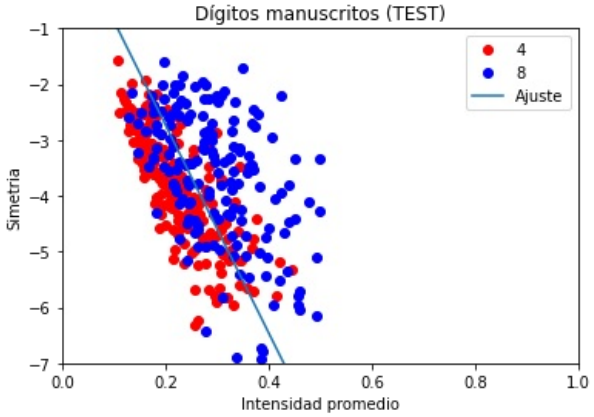


Obtenemos un nuevo error de Ein algo mayor que el obtenido con el Sgd, pero sin embargo, obtenemos un Eout menor con los datos de TEST.

```
Aplicando SGD logistic regression al w obtenido con PLA-Pocket
Error Ein de regresión logística 0.4651368420912251
Error Ein de clasificación 0.2236180904522613
```




```
Error Eout de regresión logística 0.5414545869489784
Error Eout de clasificación 0.25136612021857924
```



Ahora probamos lo contrario, partimos del PLA-Pocket y le pasamos el resultado al Sgd como pesos iniciales. Conseguimos bajar el Ein Respecto al paso anterior pero no conseguimos un Eout mejor. Fuera de la muestra el mejor "pipeline" ha sido utilizar solamente PLA-Pocket. Seguido de utilizar PLA-Pocket y luego Sgd. Y por último Sgd - PLA-Pocket (que es lo mismo que Sgd solo).

```
Cotas para EOUT
Eout para sgd solo (utilizando ein) es:  0.26040892304281815
Eout para sgd solo (utilizando etest) es:  0.33055194496267015


Eout para PLA solo (utilizando ein) es:  0.2679466114850292
Eout para PLA solo (utilizando etest) es:  0.3168907427768778


Eout para PLA-SGD (utilizando ein) es:  0.26292148585688846
Eout para PLA-SGD (utilizando etest) es:  0.32235522365119473
```

Utilizando sgd solo tenemos la cota superior de Eout más baja si usamos Ein como error base. Sin embargo, usando Etest como error base, PLA-Pocket solo nos da la cota superior más baja. Por último, decir que probablmente sea más fiable escoger la cota que utiliza Ein, ya que tenemos más datos que en Etest y el intervalo de estimación será menor para una misma probabilidad.