Práctica 2

Aprendizaje Automático

Alejandro Borrego Megías

Curso 2020-2021

Índice

1. Complejidad de $\mathcal{H}$ y ruido **2**
Ejercicio 1.1 . . . 2
Probar funciones para generar muestras . . . 2
Ejercicio 1.2 . . . 4
Comparación de funciones clasificadoras . . . 4

2.Modelos Lineales **15**
Ejercicio 2.1 PLA . . . 15
Implementación del algoritmo PLA . . . 15
Comportamiento en Muestra Linealmente Separable . . . 19
Comportamiento en Muestra No Linealmente Separable . . . 22
Conclusiones . . . 25
Ejercicio 2.2 Regresión Logística . . . 26
Implementación RL . . . 26
Función f y conjunto de entrenamiento . . . 29
Experimento . . . 30
Conclusiones finales . . . 33

3. Bonus **35**

Notas **43**

1. Complejidad de $\mathcal{H}$ y ruido

Ejercicio 1.1

Probar funciones para generar muestras

Dibujar gráficas con las nubes de puntos simuladas con las siguientes condiciones

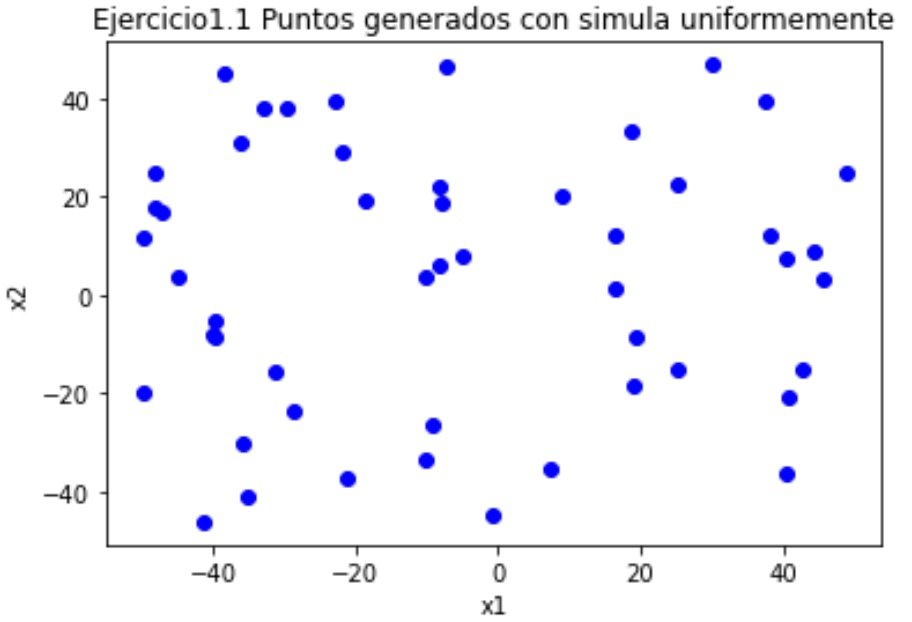
- **Considere N=50, dim=2, rango=[-50,50] con simula_unif(N,dim,rango).**
- **Considere N=50, dim=2, sigma=[5,7] con simula_gaus(N,dim,sigma).**

En este primer ejercicio vamos a probar las funciones que se nos proporcionan en el template para generar nubes de puntos:

En primer lugar, genero con ayuda de la función simula_unif(50,2,[-50,50]) una nube de 50 puntos en 2 dimensiones que toman valores en el intervalo [-50,50] en cada componente y qu que siguen una distribución uniforme de probabilidad, el código sería el siguiente:

```
x = simula_unif(50, 2, [-50,50])
# Dibujo el Scatter Plot de los puntos generados
plt.scatter(x[:,0],x[:,1], c='blue')
plt.title('Ejercicio1.1 Puntos generados con simula uniformemente')
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```

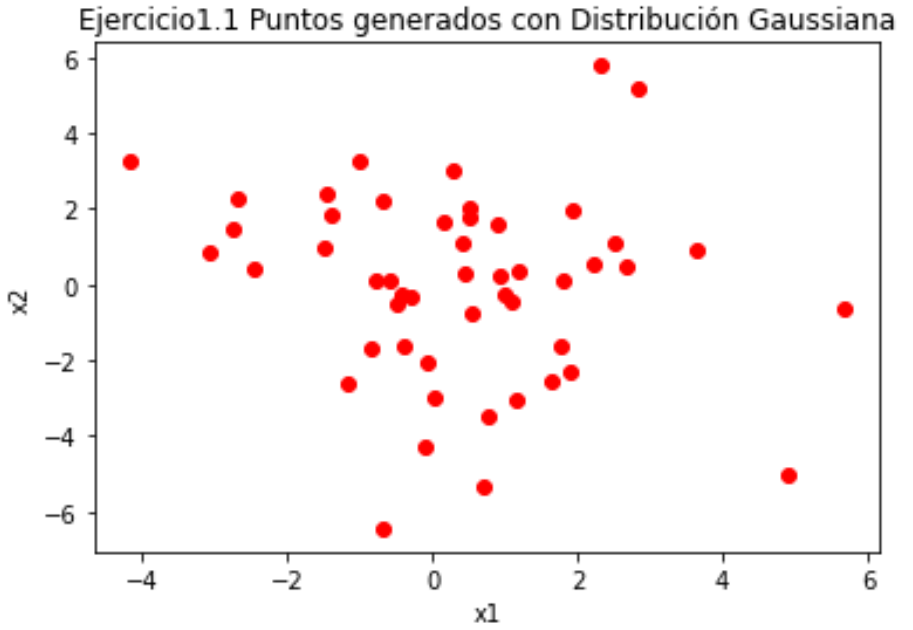
La matriz x contiene los valores de la primera y segunda coordenada de cada punto y con la función scatter de matplotlib.pyplot representamos la nube de puntos, que quedaría así:



En segundo lugar, genero con ayuda de la función simula_gaus(50,2,[5,7]) una nube de 50 puntos en 2 dimensiones que siguen una distribución Normal de parámetros N(0,5) en el eje x y N(0,7) en el eje y, el código sería el siguiente:

```
x = simula_gaus(50, 2, np.array([5,7]))
# Dibujo el Scatter Plot de los puntos generados
plt.scatter(x[:,0],x[:,1], c='red')
plt.title('Ejercicio1.1 Puntos generados con Distribución Gaussiana')
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```

Y el gráfico generado es:



Ejercicio 1.2

Comparación de funciones clasificadoras

Vamos a valorar la influencia del ruido en la selección de la complejidad de la clase de funciones. Con ayuda de la función simula_unif(100,2,[-50,50]) generamos una muestra de puntos 2D a los que vamos a 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()

- **Dibujar un grafico 2D donde los puntos muestren (usen colores) el resultado de su etiqueta. Dibuje también la recta usada para etiquetar. (observe que todos los puntos están bien clasificados respecto de la recta)**

En este apartado vamos a generar de nuevo una nube de puntos en 2 dimensiones que siguen una distribución Normal, pero en este caso, usaremos un hiperplano (una recta pues estamos en 2D) para dar una etiqueta a cada punto del espacio, si un punto se sitúa por encima de la recta, al evaluar dicho punto en $f(x, y)$ obtendrá un valor positivo, y la función signo le pondrá como etiqueta $+1$, en cambio si el punto se sitúa por debajo de la recta, en este caso se le asigna el valor -1.

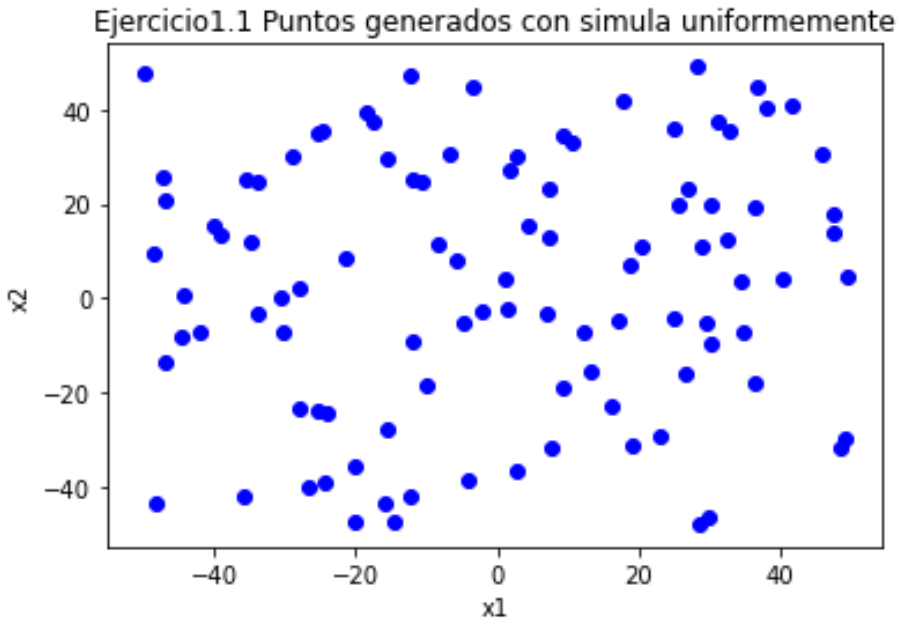
Para obtener los coeficientes a, b de la recta $f(x, y) = y - ax - b$ usaremos la función simula_recta([-50,50]) que se nos proporciona en el template, y pasamos como parámetro el intervalo de defincición, en este caso [-50,50]. Finalmente, representamos en un gráfico la recta junto con los puntos clasificados. El código sería el siguiente:

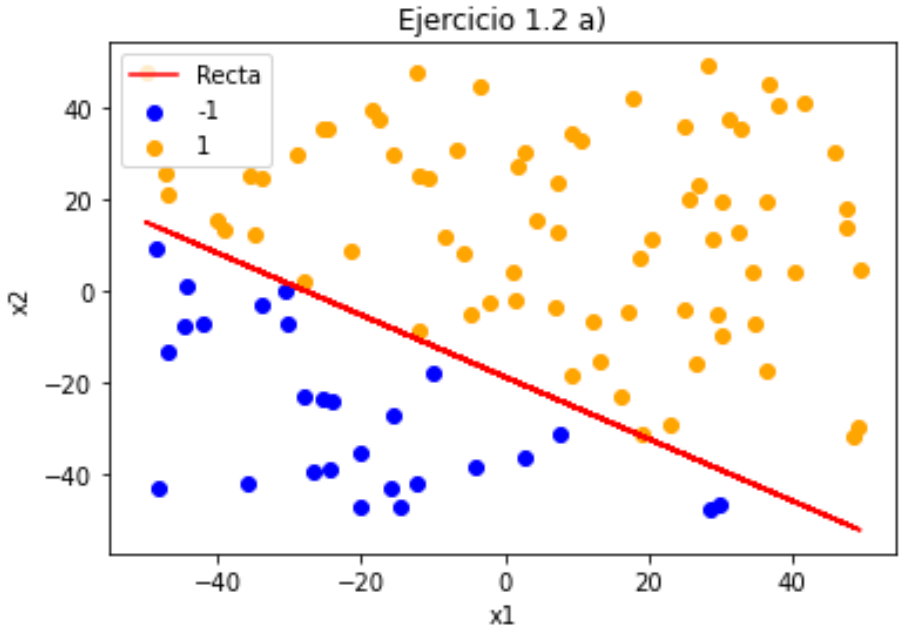
```
#Calculamos los coeficientes de la recta
a,b=simula_recta([-50,50])

#Generamos las etiquetas
y=[]

for i in x :
    y.append(f(i[0],i[1],a,b))

y=np.array(y)
y0 = np.where(y == -1) #capturo los índices de los elementos con -1
y1 = np.where(y == 1) #capturo los índices de los elementos con 1
#x_2 contiene dos arrays, uno en cada componente,
#el primero tiene los valores de x con etiqueta -1
#y la segunda los de etiqueta 1
x_2 = np.array([x[y0[0]],x[y1[0]]])
plt.scatter(x_2[0][:, 0], x_2[0][:, 1],  c = 'blue', label = '-1')
plt.scatter(x_2[1][:, 0], x_2[1][:, 1],  c = 'orange', label = '1')

#Calculamos las imagenes de los puntos
#(sin aplicar la función signo)
#y así dibujar la recta de regresión
imagenes=[]

for i in x :
    imagenes.append(a*i[0]+b)

#Para representarlo, despejo x2 de la ecuación
#y represento la función resultante en 2D
plt.plot( x[:,0], imagenes, c = 'red', label='Recta')
plt.legend()
plt.title("Ejercicio 1.2 a)")
plt.xlabel('x1')
plt.ylabel('x2')
plt.figure()
plt.show()
```

En primer lugar, como hemos dicho obtenemos los coeficientes de la recta, y después obtenemos las etiquetas para cada punto de la matriz x (matriz de los 100 puntos obtenidos por simula_unif()), en segundo lugar pintamos por separado los puntos clasificados con +1 y -1 usando el color naranja para los

primeros y azul para los segundos. Finalmente dibujo la gráfica de la recta y represento todos los datos juntos en un gráfico. Finalmente comentar que la forma en que represento los datos es la misma que en la práctica pasada y por tanto ya se explicó en la práctica anterior cómo se realizaba.

A continuación muestro los gráficos correspondientes a este apartado:





Como comentario final, por la forma en que hemos procedido, el error de clasificación cometido por esta recta será de 0, ya que hemos usado esta recta para clasificar los datos.

- **Modifique de forma aleatoria un 10 % de las etiquetas positivas y otro 10 % de las etiquetas negativas y guarde los puntos con sus nuevas etiquetas. Dibuje de nuevo la gráfica anterior. (Ahora habrá puntos mal clasificados respecto de la recta)**

En este apartado convertimos el vector y_0 de índices que nos indicaba las posiciones de las etiquetas con valor -1 en el vector y (vector de etiquetas) en un DataFrame de Pandas, ya que esta librería nos permite con facilidad obtener el 10 % de los índices y cambiar el valor de su etiqueta. Después hacemos lo mismo con el vector y_1.

A continuación muestro el código:

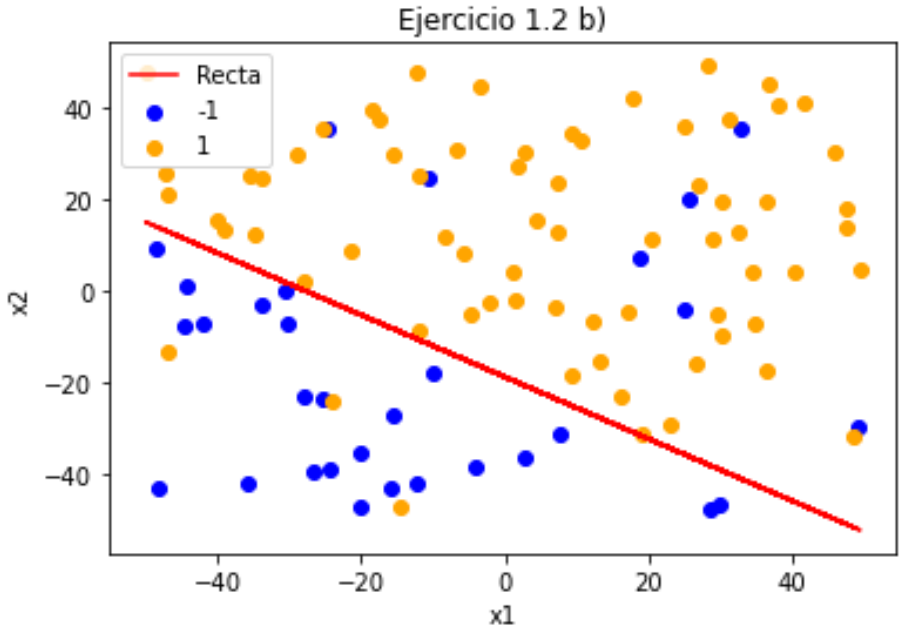
```
TN=len(y0[0])
TP=len(y1[0])

# Array con 10% de indices aleatorios para introducir ruido
y0=pd.DataFrame(data=y0[0]);
y0=y0.sample(frac=0.10,random_state=1);
y0=y0.to_numpy()
```

```
for i in y0:
    y[i]=1

TN=TN-len(y0)
```

Como comentarios al código, en primer lugar las variables TN y TP hacen referencia a los "True negatives" y "True positives", es decir, aquellos puntos cuya etiqueta real coincide con la etiqueta de la función que estamos usando como clasificador. En este caso, antes de modificar las etiquetas, por lo comentado en el apartado anterior, la recta clasifica perfectamente los datos, es por eso que los TP coinciden con los "Positives" de la muestra y los TN coinciden con los "Negatives" de la muestra, y al actualizar el 10 % de las etiquetas , estos puntos pasarán automáticamente a estar mal clasificados por la recta y se debe restar el número de puntos elegido al valor de TP y TN que teníamos al principio. Esto se utilizará en el apartado siguiente, dónde compararemos la precisión (Accuracy) de distintas funciones al utilizarlas como clasificadores.

Finalmente comentar que el código es análogo para las etiquetas con +1.

Tras esta modificación, el gráfico queda de la siguiente forma:



- **Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de la recta. Visualizar el etiquetado generado en 2b) junto con cada una de las gráficas de cada una de las funciones. Compara las**

regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta. Argumente si estas funciones más complejas son mejores clasificadores que la función lineal. Observe las gráficas y diga que consecuencias extrae sobre la influencia del proceso de modificación de etiquetas en el proceso de aprendizaje. Explicar el razonamiento.

- $f(x,y) = (x-10)^2 + (y-20)^2 - 400$
- $f(x,y) = 0{,}5(x-10)^2 + (y-20)^2 - 400$
- $f(x,y) = 0{,}5(x-10)^2 - (y+20)^2 - 400$
- $f(x,y) = y - 20x^2 - 5x + 3$

Este apartado es el más interesante, pues vamos a comparar el clasificador obtenido en los apartados anteriores (recordemos, la recta $f(x,y) = y - ax - b$) con otras funciones que definen distintas fronteras de decisión.

En primer lugar aclarar como ya se comentó en el apartado anterior que para comparar las distintas funciones usaremos la precisión o accuracy ($\frac{TN+TP}{P+N}$) que nos proporciona el porcentaje de puntos bien clasificados por cada función. De hecho comenzamos calculando la precisión de la recta del apartado anterior, que tiene una precisión de 0.9. Tiene lógica que salga tan buena precisión pues es la función que hemos usado para clasificar la nube de puntos y después hemos metido un 10 % de ruido a sus etiquetas positivas y negativas, por lo que no debía bajar mucho la precisión del 100 %.

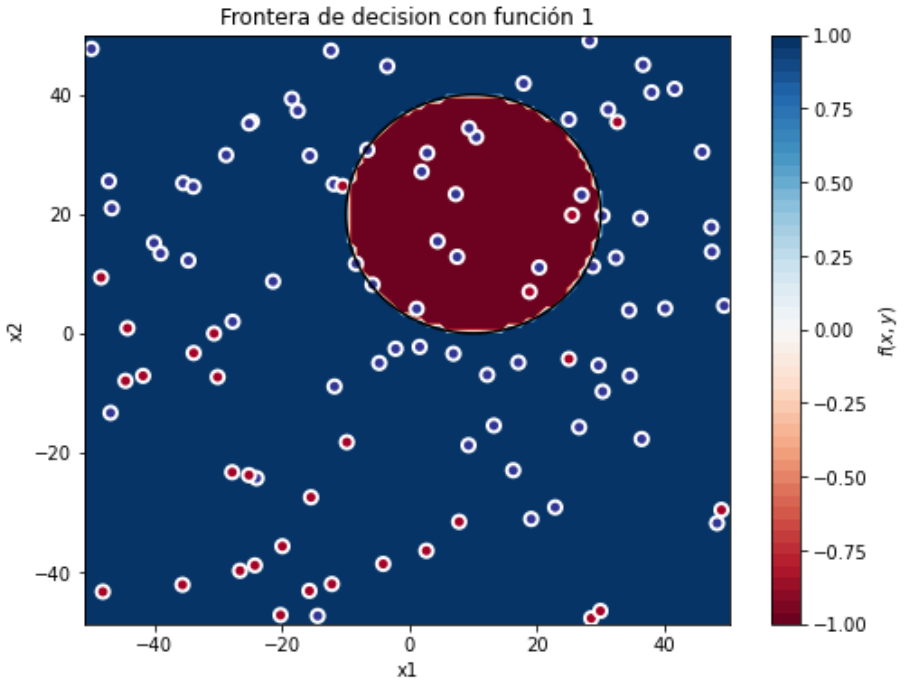
Dicho esto, realizaré una explicación del código usado para la primera función pues para el resto es completamente análogo:

```
def f1(x):
    y=[]
    for i in x:
        y.append((i[0]-10)**2 + (i[1]-20)**2-400)

    return np.asarray(y)

plot_datos_cuad(x,y,f1,'Frontera de decision con función 1', 'x1', 'x2')

imagenes=f1(x) #Capturo las imágenes de cada punto
TN=0
TP=0
cont=0

for i in imagenes:

    if i>0 and y[cont]>0: #Si tienen la misma etiqueta +1
        TP+=1
    if i<0 and y[cont]<0: #Si tienen la misma etiqueta -1
        TN+=1
    cont+=1
```

```
print("Mostramos la accuracy del método f1")
print ("ACCURACY= ", (TN+TP)/100.0)
```

En primer lugar definimos la función que vamos a usar para definir la frontera de decisión. Como comentario, para poder hacer uso posteriormente de la función plot_datos_cuad() que se nos proporciona en el template debemos definir la función de manera que dada la matriz de datos x, nos devuelva directamente el vector y de imágenes.

En segundo lugar, haciendo uso de la función anteriormente mencionada representamos las regiones de puntos con etiqueta +1 y -1 y los puntos del apartado anterior (así veremos visualmente si nuestro clasificador es bueno o no). Dicho gráfico es el siguiente:



La región azul representa la región de puntos con etiqueta +1 según nuestro nuevo clasificador y la región burdeos la región de puntos con etiqueta -1.

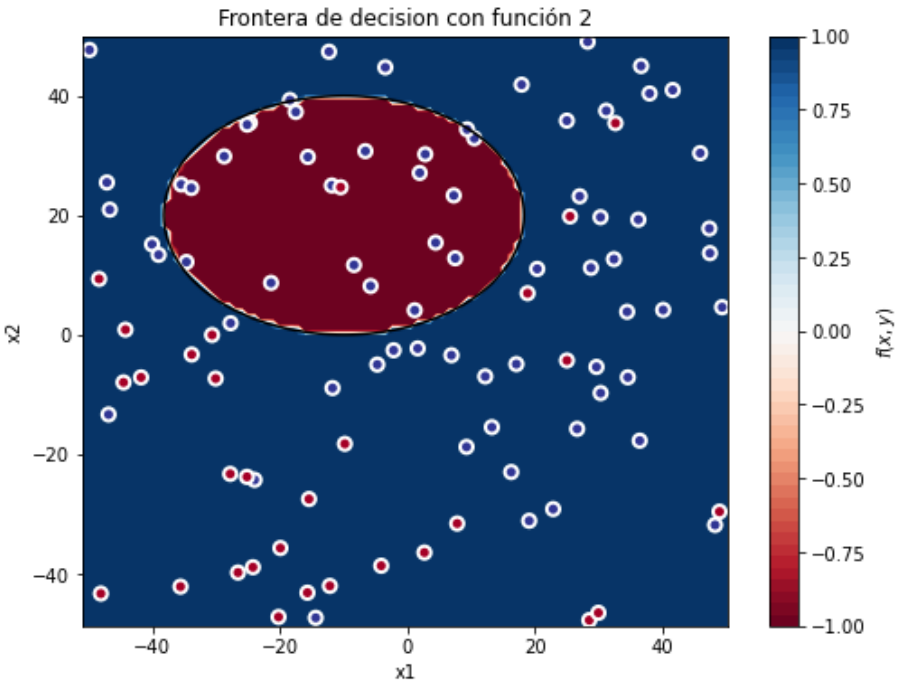
Finalmente, en la útlima parte del código calculamos los TP y TN de nuestra nueva función. Para ello, usando el vector de imágenes generado por la nueva función, vemos en que puntos coincide el signo del punto con el de la etiqueta, y cuando hacemos fracción nos da el siguiente resultado:

```
Mostramos la precisión del método del apartado anterior
ACCURACY=  0.9
Mostramos la accuracy del método f1
ACCURACY=  0.59
```

Como podemos observar, la nueva función no clasifica tan bien los datos como la del apartado anterior, y hacierta tan solo en un 59 % de las etiquetas, luego esto no lleva a pensar que si dicha función no representa una frontera de decisión que explique correctamente la muestra. A pesar de esto se puede pensar que la precisión tampoco es tan baja, dado que el área donde la función asigna etiqueta +1 es tan grande que engloba muchos puntos (que clasifica tanto bien como mal), y además en la región -1 ha dado la casualidad de que caen algunos de los puntos alterados, lo cual hace que los clasifique como correctos también. Pero claro, estas conclusiones las hemos sacado visualizando el gráfico, pues solo con los datos podríamos haber pensado que quizá no era tan mal clasificador.

En las siguientes funciones procedemos de forma análoga.

Para la función $f(x, y) = 0{,}5(x-10)^2 + (y-20)^2 - 400$ obtenemos los siguientes resultados:

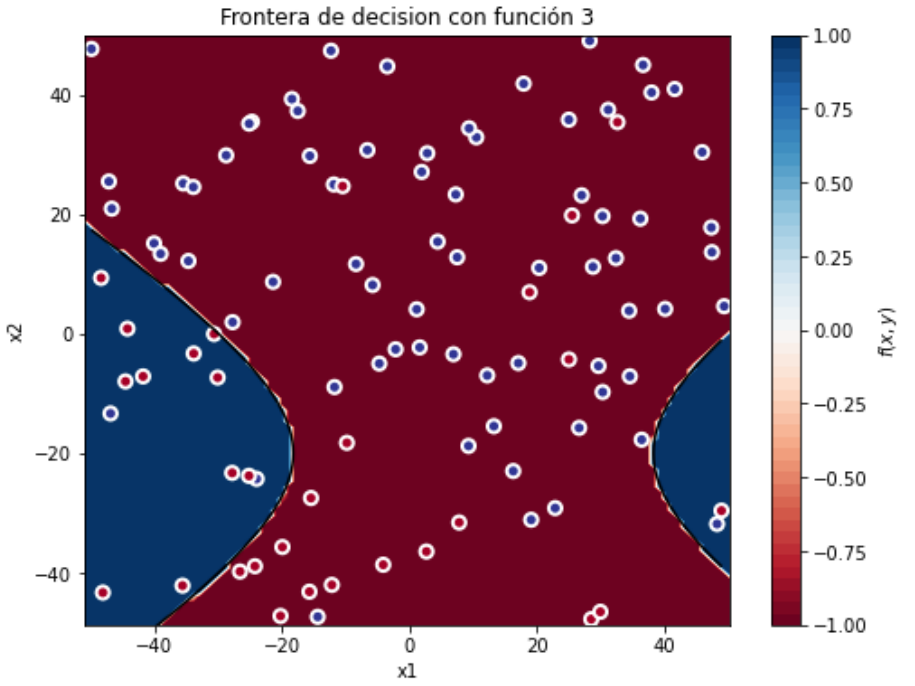



```
Mostramos la precisión del método del apartado anterior
ACCURACY=  0.9
Mostramos la accuracy del método f2
```

```
ACCURACY=  0.51
```

Como podemos observar, en este caso pasa algo similar al anterior, la precisión comparada con la recta empeora bastante, y el porcentaje de aciertos que tiene se debe a lo mismo que pasaba en el apartado anterior, luego la conclusión es que tampoco es una buena función para definir la frontera de decisión.

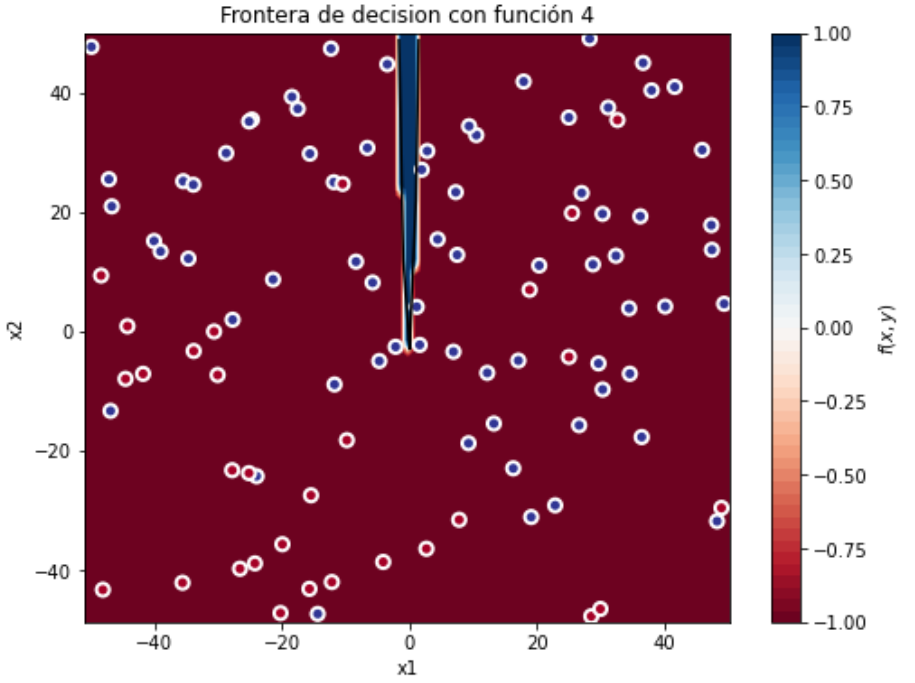
Para $f(x,y) = 0{,}5(x-10)^2 - (y+20)^2 - 400$ obtenemos los siguientes resultados:




```
Mostramos la precisión del método del apartado anterior
ACCURACY=  0.9
Mostramos la accuracy del método f3
ACCURACY=  0.22
```

Claramente esta función no explica la muestra, el porcentaje de acierto es de un 22 % frnte al 90 % de la recta y por la gráfica se da a entender que los puntos correctamente clasificados han sido por suerte. Luego como conclusión no es una función correcta para gneralizar y para definir la frontera de decisión.

Finalmente para $f(x,y) = y - 20x^2 - 5x + 3$ obtenemos los siguientes resultados:




```
Mostramos la precisión del método del apartado anterior
ACCURACY=  0.9
Mostramos la accuracy del método f4
ACCURACY=  0.31
```

De nuevo nos encontramos ante un mal clasificador para nuestra frontera de decisión, las regiones de +1 y -1 no se ajustan a los datos y las coincidencias se deben a que la región -1 es muy amplia y engloba a todos los datos con etiqueta -1, pero como también engloba a los de etiqueta +1 pues se comete un error muy alto.

Por lo tanto, la conclusión que podemos sacar sobre la influencia del ruido en la muestra y las funciones de $\mathcal{H}$ que valoremos es que puede influir positiva o negativamente en las funciones provocando que algunos clasificadores que a priori no explican en absoluto la muestra cometan un porcentaje de acierto mayor de lo esperado o que buenos clasificadores como la recta del apartado anterior cometan un error mayor del esperado.

Por otro lado, como hemos podido comprobar en este ejemplo, añadir funciones de una mayor complejidad para valorarlas como posibles clasificadores no es siempre sinónimo de que vamos a obtener mejores resultados que con otras funciones más simples. De hecho para esta muestra concreta una recta nos proporciona un porcentaje de acierto muy superor al de cualquier otra función.

Finalmente comentar que es preferible en este tipo de situaciones estudiar un poco más la muestra, representar los puntos gráficamente si se puede y valorar que funciones pueden ajustarse mejor a la muestra antes que tomar un conjunto muy grande de funciones complejas y probarlas. No obstante no debe perderse de vista el hecho de que minimizar el error en Training Set no es sinónimo de mejor error E_{out} , luego centrarnos en conseguir un clasificador perfecto para nuestro Training Set añadiendo funciones más complejas no es lo correcto, en su lugar tendríamos que intentar encontrar una solución que generalice bien error cometido en la muestra y que dentro de lo posible este error sea lo más bajo que podamos conseguir.

Esto último se aplica también a los ejercicios que se van a realizar a continuación, aunque se hable del hecho de tratar de mejorar el E_{in}. De hecho no trataremos el tema de la generalización hasta el Bonus.

2.Modelos Lineales

Ejercicio 2.1 PLA

Implementación del algoritmo PLA

Implementar la función ajusta_PLA(datos,label,max_iter,vini) que calcula el algoritmo PLA.

El Perceptrón es un algoritmo empleado generalmente en problemas de clasificación binaria en el cual tratamos de separar con un hiperplano una serie de puntos con etiquetas distintas (en nuestro caso +1 y -1), dicho esto el algoritmo sería el siguiente:

Perceptron Learning Algorithm (PLA):

- Given the data set $(\mathbf{x}_n, y_n), n = 1,2,\cdots,N$
- Step.1: Fix $\mathbf{w}_{\text{ini}}=0$
- Step.2: Iterate on the $\mathcal{D}$-samples improving the solution:
- repeat

  For each $\boldsymbol{x}_i \in \mathcal{D}$ **do**

  if: $sign(\boldsymbol{w}^T \boldsymbol{x}_i) \neq y_i$ **then**

  update **w:** $\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + y_i \mathbf{x}_i$

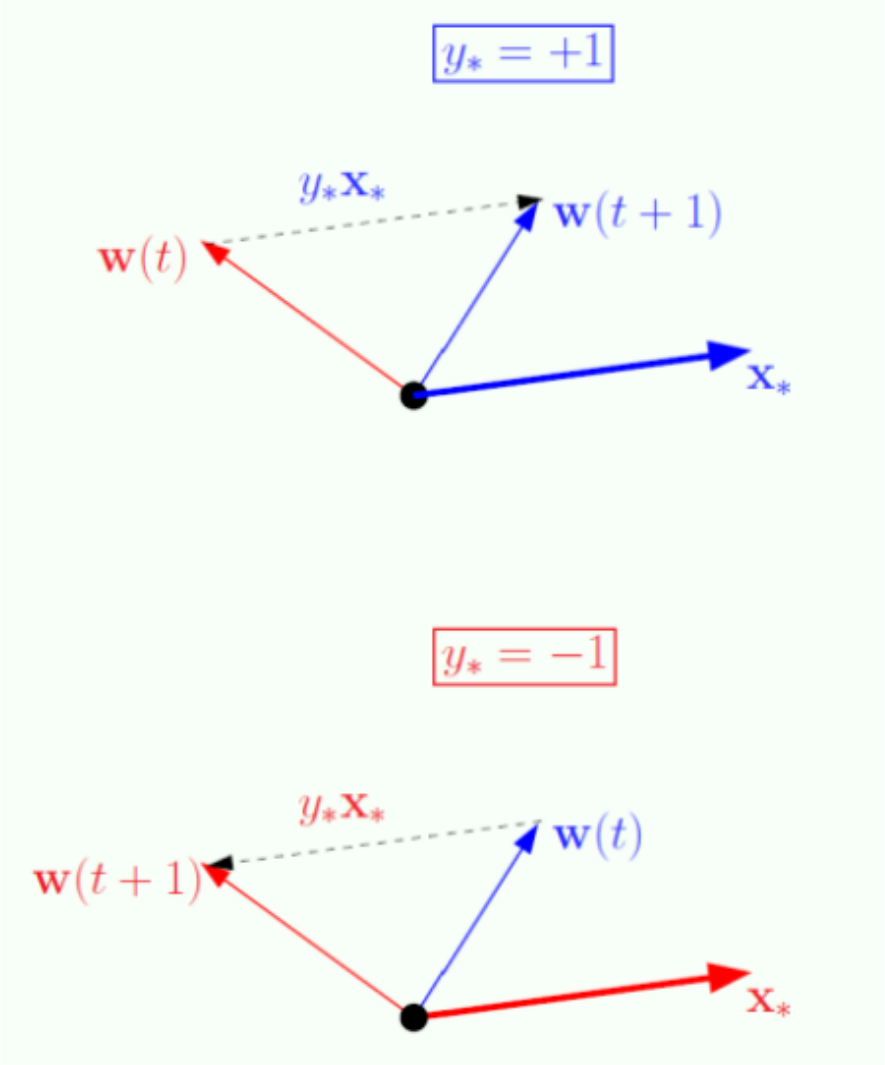
  else continue

  End for
- Until No changes in a full pass on $\mathcal{D}$

7 de 12

Como podemos ver, comienza inicializando el vector de pesos (los coeficientes de nuestro hiperplano) a cero. Tras esto se va iterando sobre los elementos de la muestra ($\mathcal{D}$) de manera que si el signo de evaluar el punto en el hiperplano actual (valor de los coeficientes en ese momento) es distinto a la etiqueta real

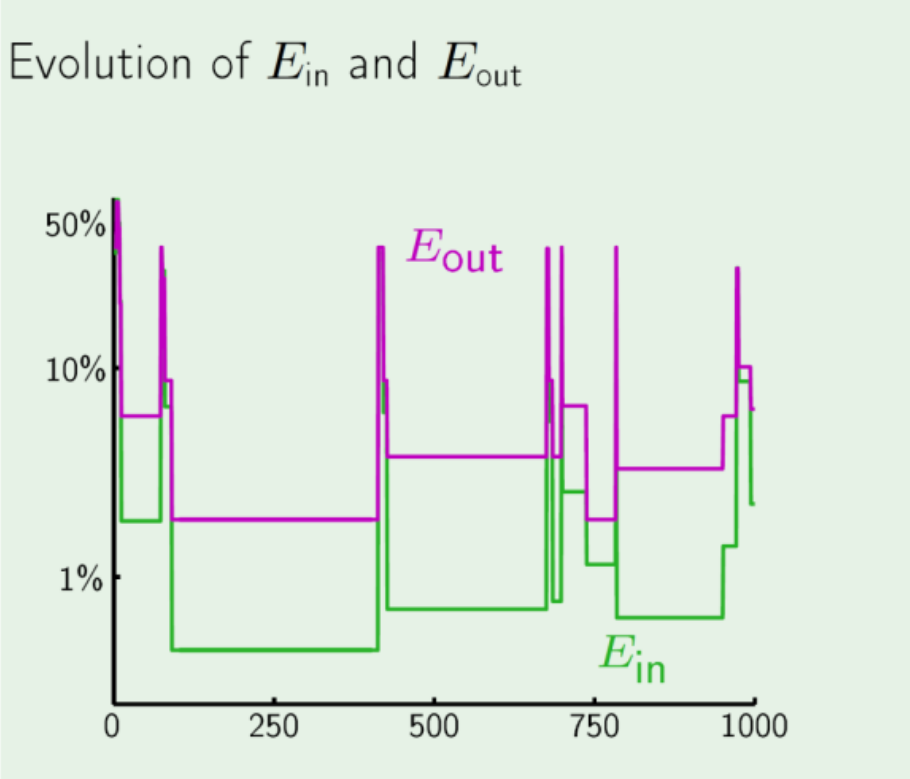
del punto, lo que hacemos es actualizar el valor de los pesos de manera que el nuevo hiperplano clasifique bien este punto, geométricamente, lo que estamos haciendo es lo siguiente:



Como podemos ver, si en la iteración t, el vector de pesos w(t) clasificaba erróneamente el punto, la diferencia w(t)- $y_i x_i$ gira el hiperplano clasificando correctamente el punto. Y si el punto estaba bien clasificado entonces no se actualiza el vector de pesos.

Este proceso se repite hasta que el algoritmo da una vuelta sobre todos los puntos y no encuentra puntos mal clasificados.

Es importante tener en cuenta que este algoritmo para en el caso de que el conjunto de puntos sea linealmente separable, en otro caso el algoritmo cicla indefinidamente por lo que se suele añadir un criterio de parada extra que es un máximo de iteraciones. No obstante conviene saber que en este algoritmo, si los datos no son linealmente separables más iteraciones no es sinónimo de mejor comportamiento, hay que pensar que con cada cambio en el vector de pesos se clasifica correctamente el punto deseado pero a costa de clasificar mal otros puntos quizás. De hecho la gráfica del error cometido con este algoritmo nos lo muestra:



Dicho esto, el algoritmo implementado es el siguiente:

```
def ajusta_PLA(datos, label, max_iter, vini):
    '''
    Algoritmo Perceptrón
    Parameters
    ----------
```

```
    datos : Matriz de datos

    label : Vector de etiquetas

    max_iter : número máximo de iteraciones

    vini : vector inicial de pesos

    Returns
    -------
    w : vector de pesos.

    it : número de iteraciones empleadas

    '''
    mejora=True
    it=0
    w=np.array(vini)
    w=w.reshape(-1,1) #Lo transformo en un vector columna

    while (mejora and it<max_iter): #
        mejora=False
        it+=1
        for i in range(len(datos)):
            valor=datos[i,:].dot(w)
            sign=signo(valor.item())

            if sign!=label[i]:
                actualiza=label[i]*datos[i,:]
                actualiza=np.array(actualiza)
                actualiza=actualiza.reshape(-1,1)
                w=w+actualiza
                mejora=True



    return w, it
```

El booleano *mejora* controla si en una pasada sobre los datos se actualiza el vector de pesos o no y en caso de que no se haya "mejorado" el algoritmo en una pasada, se sale del bucle principal y acaba el algoritmo.

La variable it controla el número de iteraciones y junto con el vector de pesos se devuelve al final.

Finalmente w es el vector de pesos (inicializado al valor que se pase en la variable

vini).

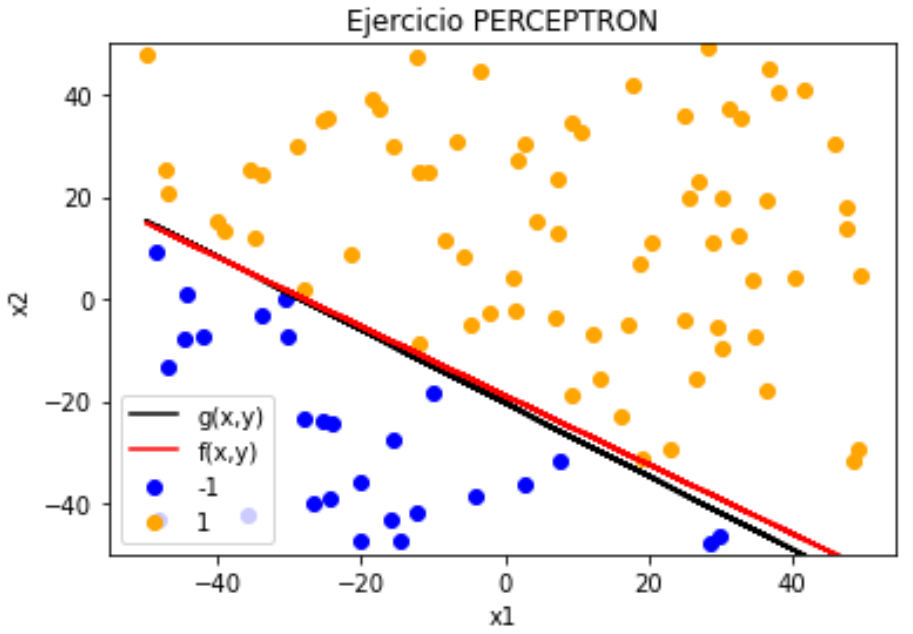
Comportamiento en Muestra Linealmente Separable

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) vectores de números aleatorios en [0,1] (10 veces). Anotar el número medio de iteraciones necesarias en ambos casos para converger. Valorar el resultado relacionando el punto de inicio con el número de iteraciones

Ahora vamos a poner en funcionamiento el algoritmo implementado y vamos a ver cómo se comporta. En primer lugar lo hacemos con una muestra de datos separable (lo que nos garantiza convergencia en un número finito de iteraciones) y con el vector de pesos inicializado a 0. El código sería el siguiente:

```
vini=[0.0,0.0,0.0]
unos=np.ones((x.shape[0],1))
x=np.concatenate((unos,x),axis=1)
w, iteraciones=ajusta_PLA(x,labels,1000,vini)
```

Con esto, los resultados obtenidos son los siguientes:



Y el número de iteraciones empleado es **75** lo cual nos indica un buen comportamiento del algoritmo para la muestra.

Si repetimos el experimento diez veces, pero esta vez inicializando el algoritmo con un vector de pesos aleatorios en el intervalo [0,1]. Los resultados obtenidos son los siguientes:

```
Iteracion:  0
vector inicial=  [0.61851357 0.01036426 0.53862728]
vector obtenido=  [[555.61851357]
 [ 19.40133558]
 [ 29.25372607]]
Iteraciones:  60



Iteracion:  1
vector inicial=  [0.00301796 0.95119379 0.90540203]
vector obtenido=  [[1118.00301796]
 [  39.14140853]
 [  59.61274055]]
Iteraciones:  248



Iteracion:  2
vector inicial=  [0.79596694 0.91527432 0.14555823]
vector obtenido=  [[458.79596694]
 [ 15.33193632]
 [ 23.83811624]]
Iteraciones:  43



Iteracion:  3
vector inicial=  [0.15773007 0.18763167 0.6224959 ]
vector obtenido=  [[609.15773007]
 [ 24.12447153]
 [ 33.26034218]]
Iteraciones:  72



Iteracion:  4
vector inicial=  [0.9058095  0.98995518 0.71112246]
vector obtenido=  [[831.9058095 ]
 [ 32.42230954]
 [ 46.4541019 ]]
Iteraciones:  129
```

```
Iteracion:  5
vector inicial=  [0.73180041 0.9092932  0.40087373]
vector obtenido=  [[1087.73180041]
 [  39.49082489]
 [  53.52108012]]
Iteraciones:  244



Iteracion:  6
vector inicial=  [0.24985068 0.17343017 0.11945705]
vector obtenido=  [[641.24985068]
 [ 22.45196541]
 [ 30.99115595]]
Iteraciones:  70



Iteracion:  7
vector inicial=  [0.81261059 0.14679237 0.26429748]
vector obtenido=  [[663.81261059]
 [ 22.1176698 ]
 [ 30.86402695]]
Iteraciones:  84



Iteracion:  8
vector inicial=  [0.81908918 0.31058725 0.98241745]
vector obtenido=  [[826.81908918]
 [ 31.22040887]
 [ 45.12679496]]
Iteraciones:  122



Iteracion:  9
vector inicial=  [0.2666387  0.53365334 0.31446701]
vector obtenido=  [[383.2666387 ]
 [ 14.43198212]
 [ 17.53050619]]
Iteraciones:  37
```

```
Valor medio de iteraciones necesario para converger con 10 vectores random:  110.9
```

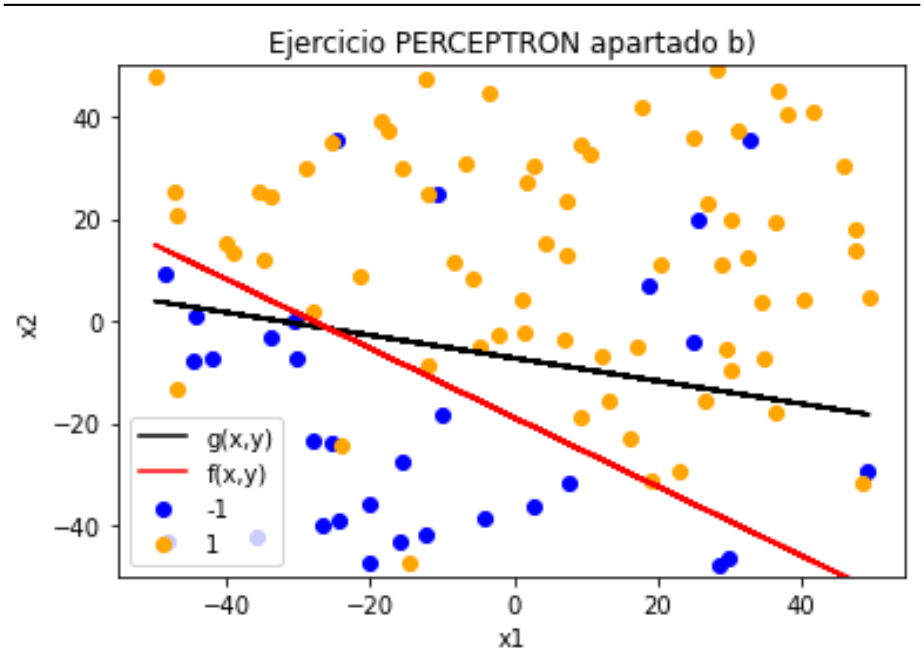
En este caso el número medio de iteraciones es de 110.9, superior al obtenido con el vector inicializado a 0. Esto nos lleva a pensar que existe una relación entre el vector inicial de pesos y el número de iteraciones empleadas por el algoritmo. El problema es que a priori no sabemos con qué vector es mejor inicializar el algoritmo pues tenemos resultados muy dispares como podemos ver, es por eso que se suele optar por inicializar el vector de pesos a 0.

Comportamiento en Muestra No Linealmente Separable

Hacer lo mismo pero con el etiquetado del *2b* de la sección 1. ¿Observa algún comportamiento diferente? En caso afirmativo diga cual y las razones para que ello ocurra.

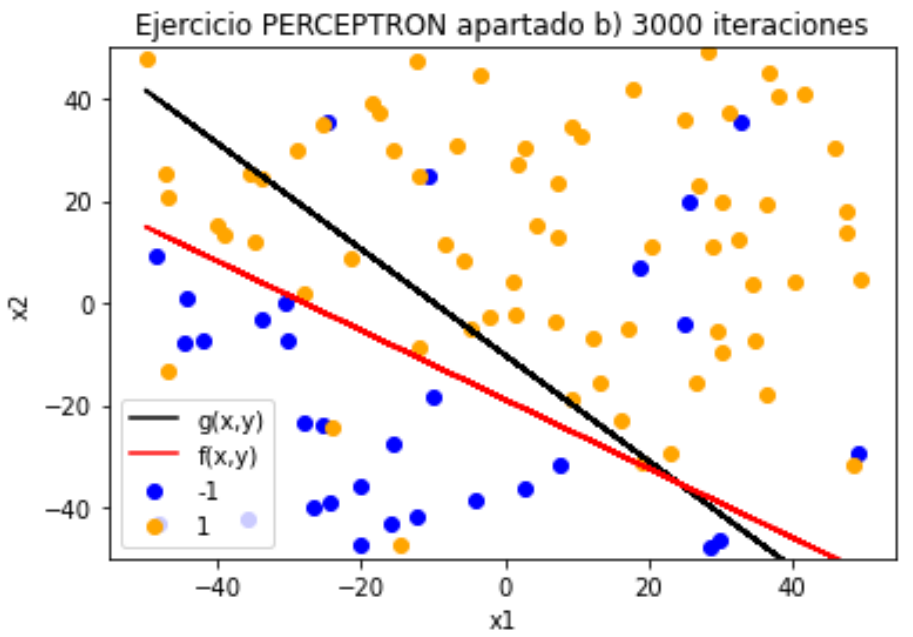
En este caso, la muestra tiene ruido, y por lo tanto no es linealmente separable, es por ello que el algoritmo PLA no convergerá. Veámoslo:

Si inicializamos el vector de pesos a 0 obtenemos el siguiente resultado para PLA:



En este caso, el algoritmo ha finalizado porque ha llegado al número de iteraciones máximas permitidas (1000 en nuestro caso), y como podemos ver, la recta

obtenida no se ajusta a los datos tan bien como la obtenida en el apartado anterior. Como ya se comentó anteriormente, el aumento del número de iteraciones no significa que obtendríamos un resultado mejor, de hecho para 3000 iteraciones el resultado es el siguiente:



Que como podemos observar sigue alejado de la función $f(x, y)$ que buscábamos.

Si ahora repetimos el experimento 10 veces con vector inicial aleatorio, obtenemos los siguientes resultados:

```
Iteracion:  0
vector inicial=  [0.91077283 0.36655664 0.43359233]
vector obtenido=  [[362.91077283]
 [ 20.10049452]
 [  9.08648493]]
Iteraciones:  1000



Iteracion:  1
vector inicial=  [0.51229269 0.93888648 0.03094901]
vector obtenido=  [[354.51229269]
 [ 20.8787443 ]
 [  7.98916437]]
```

```
Iteraciones:  1000



Iteracion:  2
vector inicial=  [0.71687866 0.89101895 0.02728722]
vector obtenido=  [[344.71687866]
 [ 10.04335853]
 [ 46.7802725 ]]
Iteraciones:  1000



Iteracion:  3
vector inicial=  [0.52205125 0.32598981 0.85948932]
vector obtenido=  [[359.52205125]
 [ 26.83227944]
 [ 18.26746968]]
Iteraciones:  1000



Iteracion:  4
vector inicial=  [0.55851655 0.69022787 0.4528535 ]
vector obtenido=  [[348.55851655]
 [ 34.87790248]
 [ 26.09851797]]
Iteraciones:  1000



Iteracion:  5
vector inicial=  [0.62830904 0.29009685 0.00934858]
vector obtenido=  [[347.62830904]
 [ 34.91414172]
 [ 34.56543139]]
Iteraciones:  1000



Iteracion:  6
vector inicial=  [0.57675593 0.31144421 0.5172676 ]
vector obtenido=  [[360.57675593]
 [ 28.60358562]
 [ 27.45930253]]
Iteraciones:  1000
```

```
Iteracion:  7
vector inicial=  [0.91640585 0.42647479 0.24739604]
vector obtenido=  [[341.91640585]
 [ 30.8084188 ]
 [ 19.11802008]]
Iteraciones:  1000
```

```
Iteracion:  8
vector inicial=  [0.37129376 0.93186112 0.93686838]
vector obtenido=  [[351.37129376]
 [ 10.4227821 ]
 [ 44.05950166]]
Iteraciones:  1000
```

```
Iteracion:  9
vector inicial=  [0.84432995 0.92020651 0.22790029]
vector obtenido=  [[341.84432995]
 [ 23.05147752]
 [ 16.79699087]]
Iteraciones:  1000
```

```
Valor medio de iteraciones necesario para converger con 10 vectores random:  1000.0
```

Como es de esperar, en todos y cada uno de los casos, el algoritmo consume las 1000 iteraciones máximas (pues la muestra no es linealmente separable). Y en este caso el punto inicial no influye, o al menos no de la misma forma que en el apartado anterior.

Conclusiones

Luego las conclusiones que podemos sacar una vez realizado el ejercicio serían las siguientes:

El algoritmo funciona muy bien en muestras linealmente separables, aunque si la muestra es muy grande puede ser un algoritmo lento, pero al menos garantiza una solución que separa perfectamente los elementos de la muestra. En cambio, como se ha visto en teoría, la mayoría de casos reales presentan conjuntos de entrenamiento con ruidol, lo cual hace que en la mayoría de ocasiones el algoritmo

no pueda converger a una solución. Es por ello que puede ser una buena idea elegir un máximo de iteraciones a realizar (en nuestro caso hemos elegido 1000), así se reduce el tiempo de ejecución y se pueden llegar a soluciones razonables. No obstante como ya hemos comentado el error cometido por el hiperplano no es decreciente en cada iteración, por lo que elegir un número máximo de iteraciones no es sinónimo de éxito en este tipo de problemas. Ante esta problemática se crea el algoritmo PLA Pocket, cuya diferencia con el PLA es que tras actualizar los pesos, se comprueba si esta actualización tiene una mejora del Ein sobre la totalidad de los datos de la muestra, y solo se actualizan si los nuevos coeficientes mejoran el error. Esto nos asegura que el Error cometido desciende en cada iteración y por lo tanto tiene más sentido elegir un mayor número de iteraciones para el algoritmo, pues esto daría más oportunidades de mejorar los coeficientes (cuantas más iteraciones más actualizaciones de w).

Ejercicio 2.2 Regresión Logística

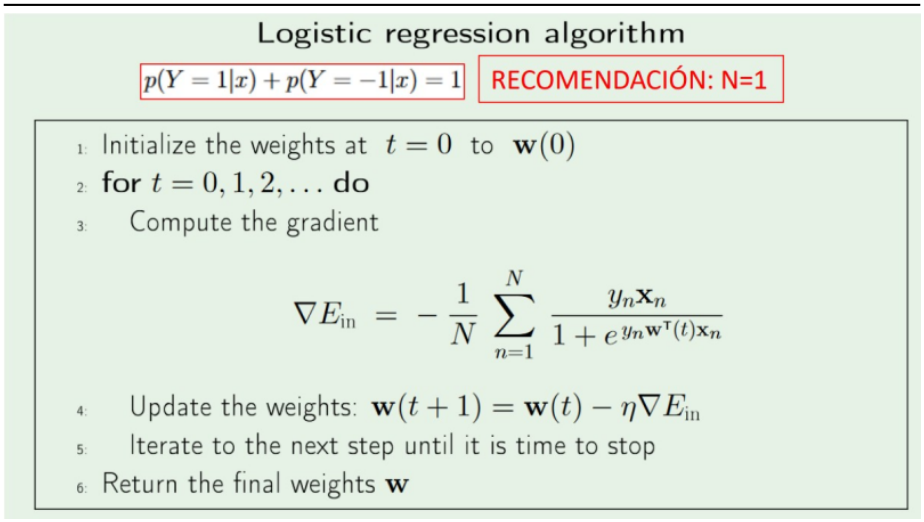
Implementación RL

Implementar Regresión Logística con las siguientes condiciones:

- **Inicializar vector de pesos con valores 0**
- **Parar el algoritmo cuando** $||w^{(t-1)} - w^t|| < 0{,}01$**, donde** w^t **representa el vector de pesos al final de la época t.**
- **Aplicar una permutación aleatoria a los índices de los datos antes de usarlos en cada época del algoritmo**
- **Usar tasa de aprendizaje** $\eta = 0{,}01$

Vamos a implementar el algoritmo de regresión logística, que se usa para problemas de clasificación en los cuales dada una entrada queremos proporcionar la probabilidad de que dicha entrada perteneza a una determinada clase, en lugar de asignar directamente una clase como en un programa de clasificación convencional.

El algoritmo de Regresión Logística es el siguiente:



En dicho algoritmo lo que se realiza es dado un vector de pesos, se sigue el método de gradiente descendente para obtener el vector w que minimiza el error, aunque en nuestro caso, el criterio de parada será que la distancia entre el w de una determinada iteración t y la anterior sea menor que 0.01, esto nos indicara que a partir de ese moemnto el vector de pesos apenas cambia y por lo tanto podemos parar el algoritmo, pues seguir no supondrá mucha diferencia.

Como podemos ver guarda muchas similitudes con el algoritmo de SGD de la práctica anterior, por lo que lo usaremos utilizando como tamaño de minibatch 1, siguiendo la recomendación de la diapositiva.

Dicho esto el código sería el siguiente:

```
def err(x, y, w):
    '''
    Calcula el error en Regresión Logística
    Parameters
    ----------
    X : Matriz de datos.

    y : vector de etiquetas.

    w : vector de pesos.

    '''
    y=y.reshape(-1,1)
    return np.mean(np.log(1 + np.exp(-y * x.dot(w))))
```

```python
def gradiente(x,y,w): #w e y deben ser vectores columna
    '''
    Calcula el gradiente en el caso N=1
    Parameters
    ----------
    x : Matriz de datos

    y : vector de etiqyetas

    w : vector de pesos

    '''
    return -y * np.transpose(x) / (1 + np.exp(y * x.dot(w)))


def sgdRL(x,y,eta, tolerancia,tam_Minibatch=1):
    '''
    Algoritmo SGD aplicado a Regresión Logística

    Parameters
    ----------
    x : Matriz de datos

    y : vector de etiqyetas

    eta : learning rate

    tolerancia : tolerancia

    tam_Minibatch : Tamaño del minibatch, 1 por defecto

    Returns
    -------
    w : vector de pesos

    it: número de épocas


    '''
    y=y.reshape(-1,1)
    N=len(y)
    iterations=0
    dif=1000.0
    w=np.zeros((x.shape[1],1))
    w=w.reshape(-1,1)
```

```
    xy=np.c_[x.copy(),y.copy()]

    while dif>tolerancia:
        w_anterior=w.copy()
        np.random.shuffle(xy) #Mezclo los datos

        for i in range(0,N,tam_Minibatch):
            parada= i + tam_Minibatch
            x_mini,y_mini=xy[i:parada, :-1], xy[i:parada,-1:]
            grad=gradiente(x_mini,y_mini,w)
            w=w - eta*grad

        iterations=iterations + 1
        dif= np.linalg.norm(w_anterior - w)

    return w, iterations
```

En primer lugar definimos la función que nos da el error de regresión logística:

$$E(w) = \frac{1}{N}\sum_{i=0}^{N} ln(1 + e^{-y_i w^T x_i})$$

Este error lo usaremos más adelante, pero como vemos gracias a las funciones de numpy, la traducción de la expresión matemática al código es muy sencilla.

En segundo lugar calculamos el gradiente, que nos hará falta para el algoritmo de regresión logística. De nuevo, la traducción de la expresión matemática del gradiente al código es directa aprovechando la librería numpy.

Finalmente definimos el algoritmo principal, como se puede observar es idéntico al algoritmo SGD de la práctica anterior, las únicas diferencia es que la exrpresión del gradiente cambia con respecto al de la práctica pasada, y que el criterio de parada en este caso es la tolerancia introducida por el usuario (la distancia mínima entre w en la iteración t y t-1), que será 0.01, al igual que el learning rate. También usaremos siempre tamaño de minibatch 1.

Función f y conjunto de entrenamiento

En este ejercicio usaremos nuestra propia función objetivo f 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 lo tanto la etiqueta y es una función determinista de x

Consideremos $d = 2$ para que los datos sean visualizables, y sea $\mathcal{X}$ =[0,2]x[0,2] con probabilidad uniforme de elegir cada $x \in \mathcal{X}$. Elegir una línea en el plano que pase por $\mathcal{X}$ como la frontera entre $f(x) = 1$ (donde toma valores +1) y $f(x) = 0$ (donde toma valores -1),para ello

seleccionar dos puntos aleatorios de X y calcular la línea que pasa por ambos

La elección de la función clasificadora f se hará según los coeficientes que devuelva la función simula_recta(0,2) proporcionada en el template, y la forma en que clasificaremos los puntos es idéntica al ejercicio 1, por lo tanto omitiremos su explicación.

Experimento

Seleccione N=100 puntos aleatorios $\{x_n\}$ de $\mathcal{X}$ y evalúe las respuestas $\{y_n\}$ de todos ellos respecto de la frontera elegida. Ejecute Regresión Logística para encontrar la función solución g y evalúe el error E_{out} usando para ello una nueva muestra grande de datos (>999). Repita el experimento 100 veces y: Calcule el valor de E_{out} para el tamaño muestral N=100 y Calcule cuantas épocas tarda en converger en promedio RL para N=100 en las condiciones fijadas para su implementación

Para comprender mejor el experimento, primero haremos una explicación de qué se hará en cada una de las iteraciones.

En primer lugar, dado que la recta ya la tenemos elegida por el apartado anterior (función simula_recta()) lo que haremos es hacer uso de la función x = simula_unif(100, 2, [0,2]), que nos proporcionará el conjunto x de entrenamiento (una nube de 100 puntos en 2 dimensiones elegidos con una distribución uniforme en el intervalo [0,2]).

Posteriormente asignaremos una etiqueta a cada punto como se hizo en el ejercicio 1.

Después añadimos a x una columna al principio de unos (necesaria para obtener un hiperplano afín) y aplicamos el algoritmo implementado de regresión logística:

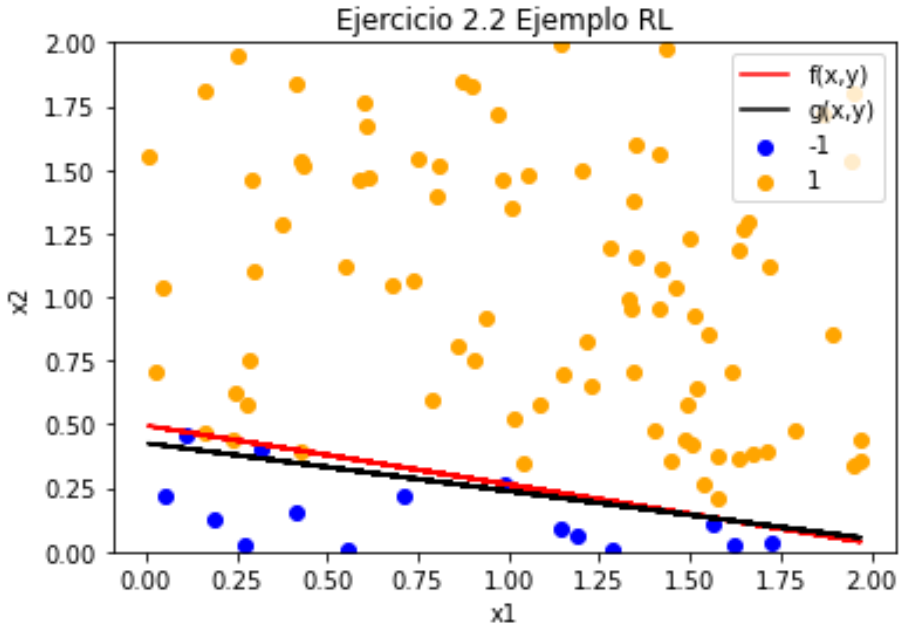
```
w,it=sgdRL(x,y,0.01,0.01)
```

Tal y como se pide en el ejercicio, se usa un learning rate y una tolerancia de 0.01, además el tamaño del minibatch es 1 pues no se especifica nada.

Tras aplicar el algoritmo obtenemos unos coeficientes para w, en mi caso son los siguientes:

```
Coeficientes obtenidos:  [[-3.27407346]
 [ 1.44582027]
 [ 7.68759712]]
```

Y si representamos la gráfica nos sale lo siguiente:



Como se puede observar la recta obtenida con RL se aproxima mucho a la real, de hecho si calculamos el E_{in} (Calculado con la función error de antes) y el número de épocas nos da lo siguiente:

```
Número de épocas en converger:  394
Calculamos el Error en la muestra (Ein):  0.10852776531767326
```

Lo cual es un error considerablemente bajo y un número de iteraciones "pequeño". Con lo que este método nos está garantizando que al menos dentro de la muestra, se comporta muy bien.

Después vamos a comprobar si ese buen comportamiento se mantiene en un conjunto de Test de 1000 puntos nuevos, para ello hacemos uso de nuevo de la función simula_unif() generando una nueva muestra de 1000 puntos y asignando etiquetas con la recta $f(x, y)$.

Tras esto representamos de nuevo en un gráfico los puntos con las rectas $g(x, y)$ y $f(x, y)$, obteniendo lo siguiente:

```
Calculamos el Error fuera de la muestra (Eout):  0.09644819420074052
```

Como podemos ver, su comportamiento en el Test Set es muy bueno también, y se produce un hecho interesante y es que el error en el Test set es menor incluso que en el Training set, esto quizá pueda deberse a que en el Training set hemos

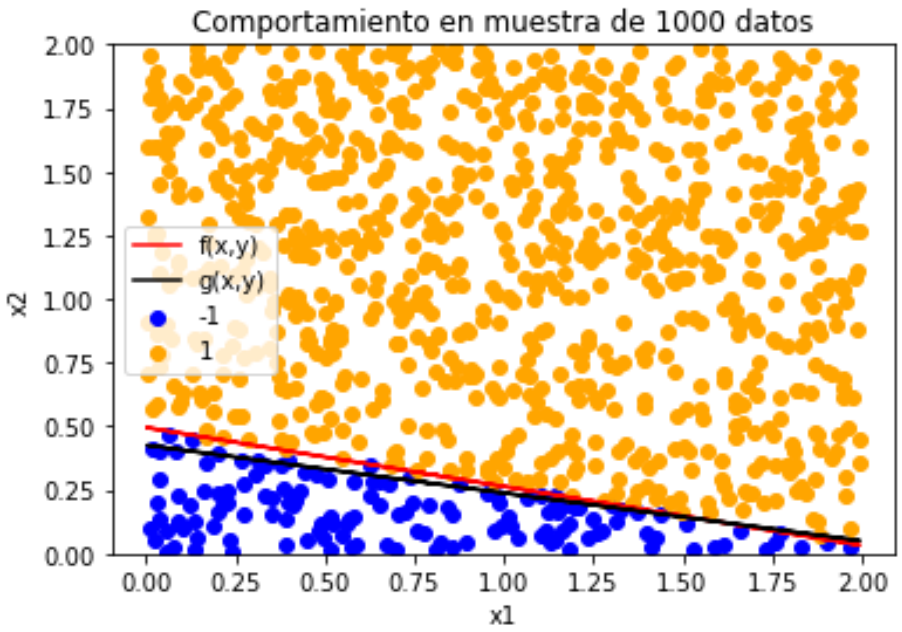


Figura 1: $f(x, y)$ y $g(x, y)$ en Test Set

usado un número muy bajo de elementos para entrenar en comparación con el Test set.

Finalmente, una vez explicado el experimento procedemos a realizar estos mismos pasos 100 veces y tomaremos la media de los errores cometidos en el Training set y Test Set así como el número medio de épocas empleado por el algoritmo para converger y sacaremos algunas conclusiones.

Resultados obtenidos:

```
 ---------------- TRAS 100 ITERACIONES ----------------
Ein medio:  0.09746667235805939
Eout medio:  0.10773833268204051
Número medio de épocas en converger:  367.79
```

Como podemos observar, el método es bastante bueno en general, pues consigue muy buenos resultados tanto en el conjunto de Entrenamiento como en el de Test, y además el número de épocas no es excesivamente alto. Por lo que este método nos proporciona una muy buena aproximación a la solución real del problema en un tiempo muy razonable, no obstante, por comentar algo en su contra, en este experimento la muestra de datos es linealmente separable pues no hay ruido, y el algoritmo PLA podría darnos una solución con error 0 a cambio de más tiempo de ejecución.

Conclusiones finales

Tras haber realizado los dos ejercicios podemos concluir que los dos algoritmos empleados, a pesar de que pueden usarse para resolver un mismo problema, tienen comportamientos diferentes y proporcionan soluciones distintas. En el caso del PLA si la muestra es linealmente separable es capaz de dar una solución óptima al problema, sin embargo si la muestra no es linealmente separable, en general, obtendremos un clasificador peor que empleando Regresión Logística, pues no se tiene en cuenta el error que se está cometiendo con cada actualización de los pesos, cosa que en RL pretendemos minimizar.

Por otro lado, si miramos el número de épocas empleadas en cada algoritmo:

- Si la muestra es linealmente separable, en general tendremos muchos datos, y aunque PLA de la solución óptima consumirá mucho tiempo de ejecución y realizará muchas iteraciones. En cambio RL consumirá menos tiempo de ejecución y realizará menos iteraciones en general, aunque no converja a una solución óptima.
- Si la muestra no es linealmente separable, entonces el número de iteraciones que use PLA no será tan relevante y además el algoritmo llegará hasta elmáximo de iteraciones permitidas, por otro lado, como ya se comentó, el error no es decreciente en cada iteración y la solución que obtendremos será peor en general que con RL (que tendrá un comportamiento similar al caso anterior).

Para ver estos hechos, he generado una muestra de 100 datos como los del Experimento del ejercicio de RL y les he introducido ruido. Además he aplicado el algoritmo de Regresión Logística y PLA obteniendo los siguientes resultados:

```
Número de épocas en convergerRL:  189
Número de épocas en convergerPLA:  1000
Calculamos el Error en la muestra de RL:  0.3780138220298664
Calculamos el Error en la muestra de PLA:  0.40349963594413013
```

Tal y como hemos dicho, en este caso RL obtiene mejores resultados tanto en tiempo de ejecución como en el error obtenido.

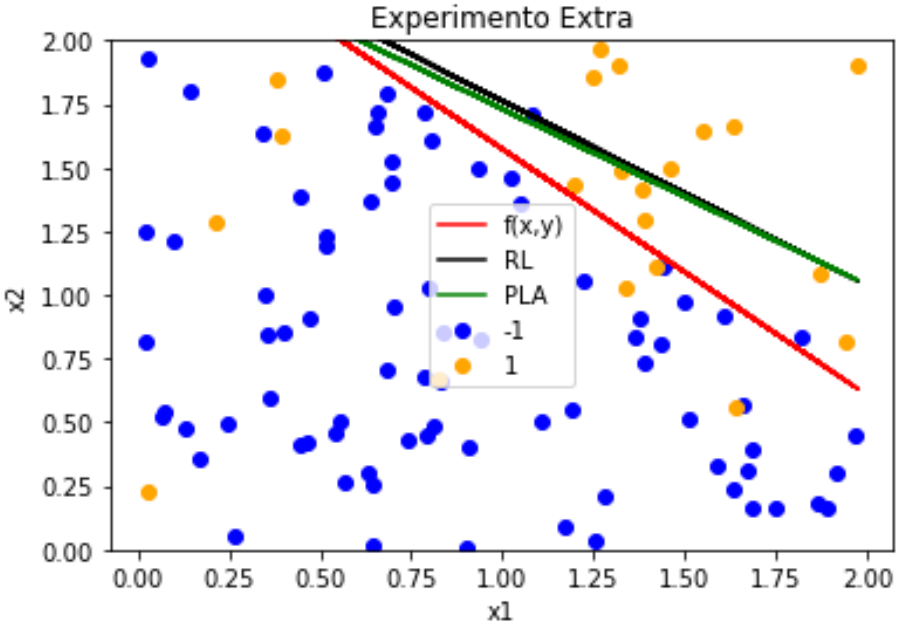


Figura 2: Comparativa

3. Bonus

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.

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

Al igual que se hizo en el en la práctica anterior, vamos a extraer de una muestra de dígitos 4 y 8 las características de Intensidad promedio y Simetría planteando el siguiente problema de clasificación binaria:

Llamamos $\mathcal{X}$ al espacio de características, en nuestro caso $1 \times \mathbb{R}^2\$$, $\mathcal{Y}$ al vector de etiqueta, en nuestro caso los posibles valores serían -1 y +1, de manera que -1 corresponde al dígito 4 y 1 al dígito 8. La función que queremos aproximar sería $f : \mathcal{X} \to \mathcal{Y}$, que es desconocida. La muestra que consideramos como conjunto de entrenamiento sería $\mathcal{D} = \{(x_n, y_n) \in \mathcal{X} \times \mathcal{Y} : n = 1..,1194\}$ (pues N=1194 es el tamaño del training set).

Tal y como hemos visto en teoría, suponemos una distribución de probabilidad $\mathcal{P}$ (desconocida también) de manera que los elementos de $\mathcal{D}$ están extraidos de forma independiente e idénticamente distribuidos.

Finalmente para aproximar f usaremos el siguiente conjunto de hipótesis $\mathcal{H} = \{g : \mathbb{R}^3 \to \mathbb{R} : g(x) = signo(w^T x), w \in \mathbb{R}^3\}$

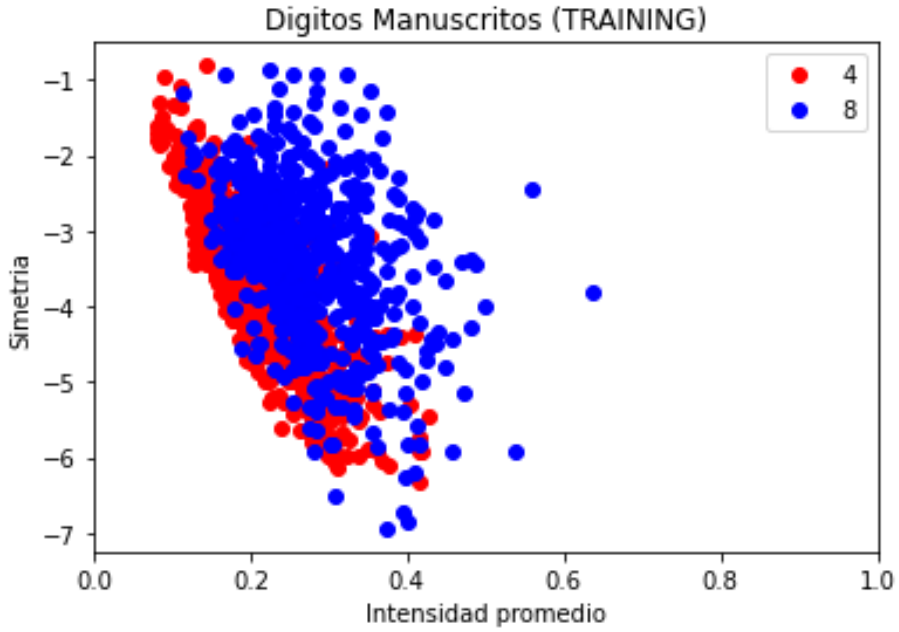
Finalmente para resolver el problema usaremos **ERM**(empirical risk minimization) utilizando como función de error:

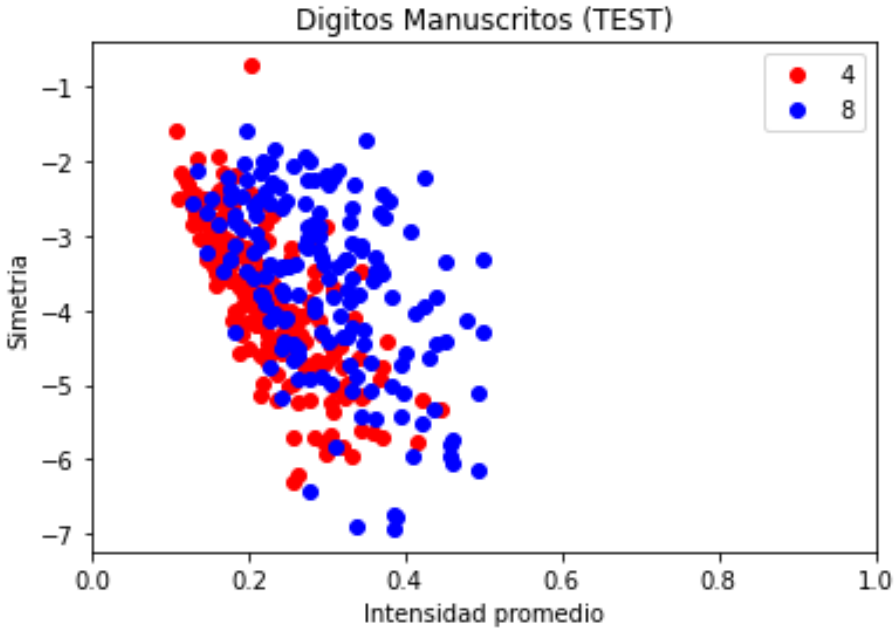
$$E_{in} = \frac{1}{N} \sum_{i=1}^{N} [g(x_i) \neq y_i]$$

Bajo estas condiciones, nuestro objetivo será encontrar el vector w de pesos que repreente el hiperplano que clasifica los puntos.

- **Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora. Responder a las siguientes cuestiones.**
  - *Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.*
  - *Calcular E_{in} y E_{test} (error sobre los datos de test).*
  - *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?*

En primer lugar, con ayuda del código que se nos proporciona en el template representamos los datos de Test Set y Training Set:





En primer lugar, haciendo uso del algoritmo de la Pseudoinversa de la práctica anterior calculamos los coeficientes w de la recta de regresión que separa ambas regiones y el error de clasificación que se comete. Los resultados obtenidos son los siguientes:

```
Usamos el algoritmo de la pseudoinversa para estimar la recta de regresión

Vector de pesos obtenido con Pseudoinversa:  [-0.50676351  8.25119739  0.44464113]
Error de clasificación cometido por la Pseudoinversa:  0.22780569514237856
```

Como podemos observar, ya de por sí el algoritmo de la Pseudoinversa nos proporciona un buen caslificador, cometiendo un error bastante bajo en el Training Set.

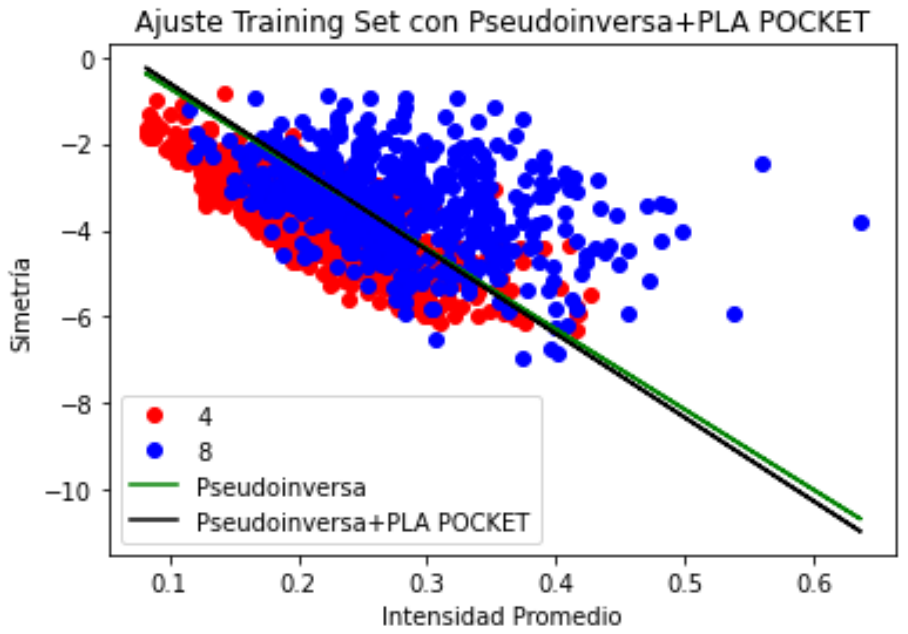
A continuación aplicamos el algoritmo Pocket, el cual es esencialmente el mismo que el PLA, a diferencia de que en este, tras cada época se evalúa si la actualización de los pesos que se ha producido es mejor o peor que la de la iteración pasada (entendemos que es mejor si comete un menor error de clasificación) y si es mejor los pesos se actualizan y si es peor se mantienen los de la iteración anterior. Así garantizamos que el error sea descendente en cada iteración del algoritmo y que tras x iteraciones se devuelve la mejor solución posible.

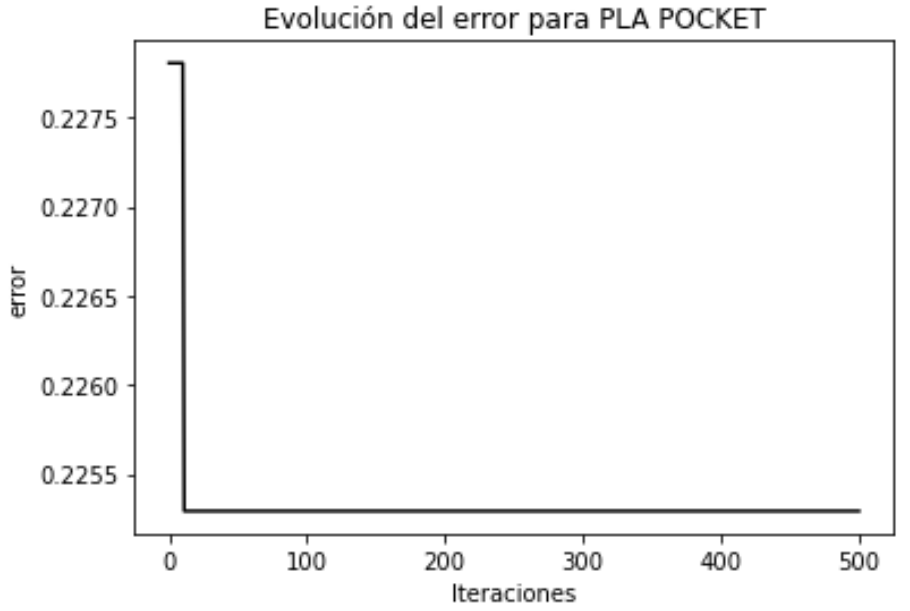
Los resultados obtenidos con el algoritmo POCKET con 500 iteraciones son los siguientes:

```
Vector de pesos obtenido con mejora PLA POCKET:  [[-6.50676351]
 [94.33278003]
 [ 4.88432863]]
Ein obtenido con Pseudoinversa+PLA POCKET:  0.22529313232830822
```

Como podemos observar, se logra mejorar el error en la muestra, pero muy poco.

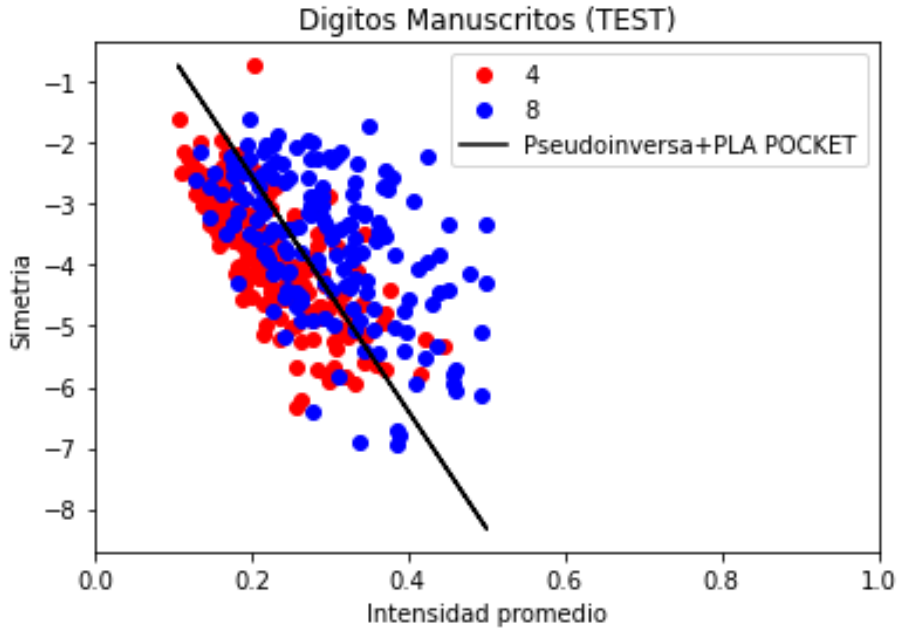
A continuación muestro en un gráfico todos los resultados hasta ahora y el hecho de que el error del PLA POCKET efectivamente es decreciente en cada iteración.





Finalmente valoramos el comportamiento del método Pseudoinversa+PLA POCKET en el Test set, obteniendo los siguientes resultados:

```
VEAMOS COMPORTAMIENTO SOBRE EL TEST SET
Etest obtenido con Pseudoinversa+PLA POCKET:  0.2540983606557377
```



Como podemos ver logra un error prácticamente idéntico al del Training Set, lo cual nos puede indicar (aunque esto no es seguro) que posiblemente sea un buen estimador de la función f buscada.

Como curiosidad y a modo de experimento vamos a aplicar como único método el PLA POCKET, obteniendo los siguientes resultados:

```
Ahora a usar el algoritmo POCKET sobre un vector de pesos inicializado a 0

Vector de pesos obtenido con mejora PLA POCKET:  [[ -8.        ]
 [138.57245022]
 [  8.095     ]]

Ein obtenido con PLA en 500 iteraciones:  0.228643216080402
VEAMOS COMPORTAMIENTO SOBRE EL TEST SET
Etest obtenido con PLA POCKET:  0.2459016393442623
```

Lo cual nos permite deducir que el algoritmo PLA POCKET se comporta bastante bien sobre los datos si ayuda de un algoritmo extra de regresión. De hecho obtiene un error menor en el Test Set, pero de nuevo esto no garantiza que el E_{out} sea menor.

Finalmente mostramos la gráfica de cómo desciende el error en cada iteración.

Cotas del error

Las dos cotas para E_{out} que hemos visto en clase son dos, la desigualdad de Hoeffding y la de Vapnik-Chervonenkis.

Hoeffding

La expresión de la cota es la siguiente:

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{1}{2N} \log \frac{2|\mathcal{H}|}{\delta}}$$

Es también llamada cota de generalización, la N sería el tamaño de la muestra, el δ sería la tolerancia y $|\mathcal{H}|$. El inconveniente es que se utiliza el cardinal de $\mathcal{H}$ y en nuestro caso dicho cardinal es ∞. No obstante, al estar programado el ejercicio en un ordenador, podemos "discretizar" el espacio. Dado que un flotante ocupa 64 bits, podemos suponer que $|\mathcal{H}| \approx 2^{64*3}$.

Finalmente comentar que esta desigualdad puede aplicarse con E_{test}, la única diferencia en este caso es que $|\mathcal{H}| = 1$ pues el conjunto $\mathcal{H}$ solo continene una función, la que hemos elegido en el Training Set.

Los resultados obtenidos son los siguientes:

```
COTA DE HOEFFDING
```

```
Pseudoinversa+PLA POCKET:
Ein:  0.4646154745114327
Etest:  0.3250874640883532

PLA POCKET:
Ein:  0.4679655582635265
Etest:  0.3168907427768778
```

Como podemos ver es una cota bastante fina, en concreto la del E_{test} pues al estar realizada sobre una muestra que no hemos entrenado con nuestro modelo previamente es más representativa de la realidad.

Por otro lado la cota que se nos garantiza es bastante aceptable, lo cual nos puede indicar que nuestro clasificador es muy adecuado (tanto para la Pseudoinversa+PLA POCKET como para PLA POCKET únicamente). De hecho es curiso que la cota para el PLA POCKET sin Pseudoinversa es más baja que para la Pseudoinversa, esto se debe a que como habíamos visto antes, con PLA POCKET sin pseudoinversa se lograba un mejor resultado para el error en el Test Set, y como el término de la raiz es el mismo tanto para PLA POCKET como para Pseudoinversa+POCKET este hecho explica que la cota sea menor.

Vapnik-Chervonenkis

La expresión de la cota es la siguiente:

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} \log \frac{4((2N)^{dvc} + 1)}{\delta}}$$

Dónde N es el tamaño de la muestra y *dvc* es la dimensión de Vapnik-Chervonenkis del clasificador utilizado. Como en este caso nos encontramos ante el problema del Perceptrón en 2D, tal y como se explicó en Teoría $dvc = 3$ en este caso. Luego para las condiciones del enunciado las cotas obtenidas serían:

```
COTA DE VC

Pseudoinversa+PLA POCKET:
Ein:  0.6562296377990118


PLA POCKET:
Ein:  0.6595797215511057
```

Como podemos ver, estas cotas son menos finas que las obtenidas anteriormente con Hoeffding. No obstante tienen una ventaja con respecto a las anteriores, y es que no necesitamos saber el cardinal de $\mathcal{H}$ para dicha cota, luego no tenemos el problema que tuvimos en el ejemplo anterior, en el cual tuvimos que "discretizar" $\mathcal{H}$.

Notas

La explicación de cómo se han realizado las gráficas y cómo se han representado las rectas la he omitido pues se ha hecho exactamente igual que en la práctica anterior, donde ya se explicó.