ugr | Universidad de Granada

Trabajo 2 - Aprendizaje Automático

Realizado por:
Francisco Solano López Rodríguez
DNI: 20100444P
Email: fransol0728@correo.ugr.es

Índice

1. Ejercicio sobre la complejidad de H y el ruido **1**

2. Modelos lineales **5**

3. BONUS **9**

1. Ejercicio sobre la complejidad de H y el ruido

1. Dibujar una gráfica con la nube de puntos de salida correspondiente.

   a) Considere $N = 50$, $dim = 2$, $rango = [-50, +50]$ con $simula\_unif(N, dim, rango)$.

```
X = simula_unif(N=50, dims=2, size=(-50, 50))
plt.scatter(X[:, 0], X[:, 1])
plt.show()
```

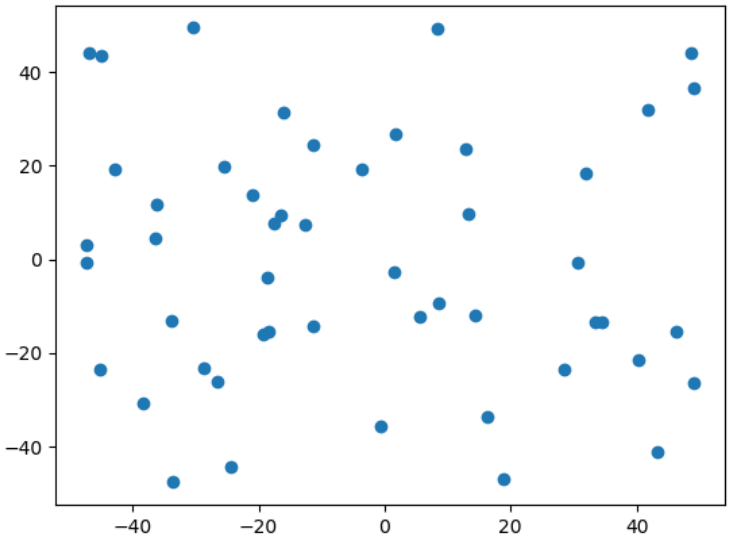


Figura 1: Distribución uniforme

   b) Considere $N = 50$, $dim = 2$ y $sigma = [5, 7]$ con $simula\_gaus(N, dim, sigma)$.

```
X = simula_gaus(size=(50,2),sigma=(5,7))
plt.scatter(X[:, 0], X[:, 1])
plt.show()
```

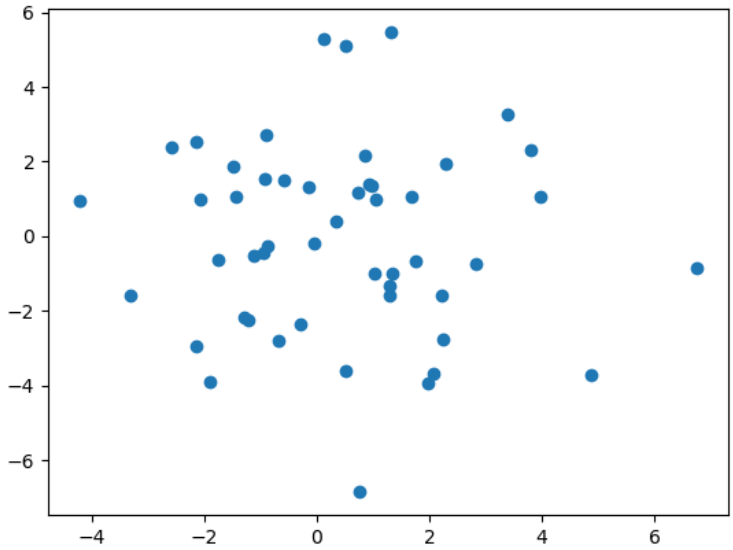


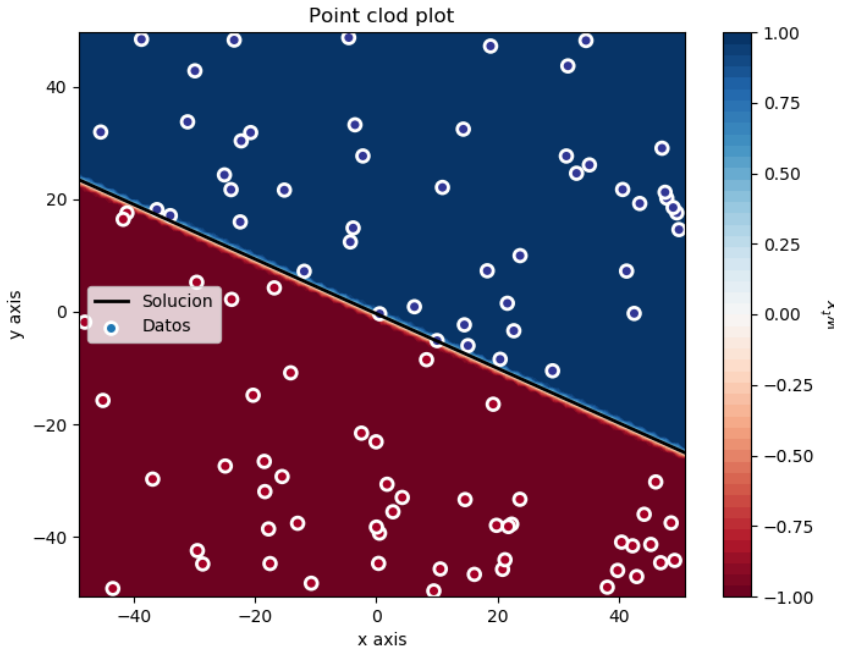
Figura 2: Distribución normal

2. 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*().

```
def f(X,a,b):
    return X[:, 1]-a*X[:, 0]-b

# simulamos una muestra de puntos 2D
X = simula_unif(N=100, dims=2, size=(-50, 50))

# simulamos una recta
a,b = simula_recta()

# agregamos las etiquetas usando el signo de la funcion f
y = np.sign(f(X,a,b))
```

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)

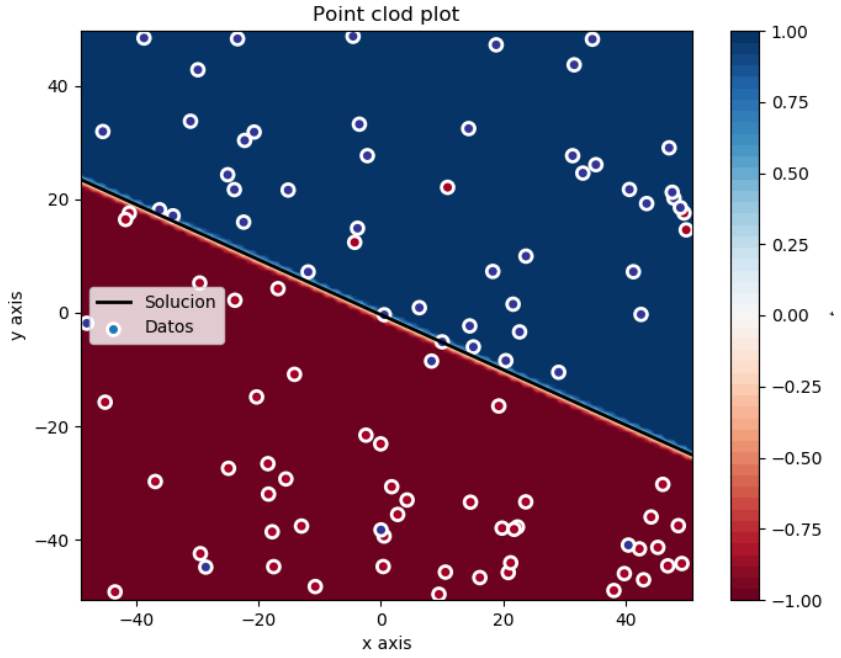
```
plot_datos_recta(X, y, a, b)
```



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)

```
# vector de indices correspondientes a los 'y' con etiqueta 1
ind_pos = []
# vector de indices correspondientes a los 'y' con etiqueta -1
ind_neg = []

# vector y de etiquetas que vamos a modificar
y_mod = np.array(y)

for i in range(0,y.size):
    if y[i] == 1:
        ind_pos.append(i)
```

```
    else:
        ind_neg.append(i)

ind_pos = np.array(ind_pos)
ind_neg = np.array(ind_neg)

# permutamos los indices
np.random.shuffle(ind_pos)
np.random.shuffle(ind_neg)

# Modificamos 10% de los positivos
for i in range(0,ind_pos.size//10):
    y_mod[ind_pos[i]] = -1

# Modificamos 10% de los negativos
for i in range(0,ind_neg.size//10):
    y_mod[ind_neg[i]] = 1

plot_datos_recta(X, y_mod, a, b)
```



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

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

```
def f1(X):
    return (X[:,0]-10)**2+(X[:,1]-20)**2-400

def f2(X):
    return 0.5*(X[:,0]+10)**2+(X[:,1]-20)**2-400

def f3(X):
    return 0.5*(X[:,0]-10)**2-(X[:,1]+20)**2-400
```

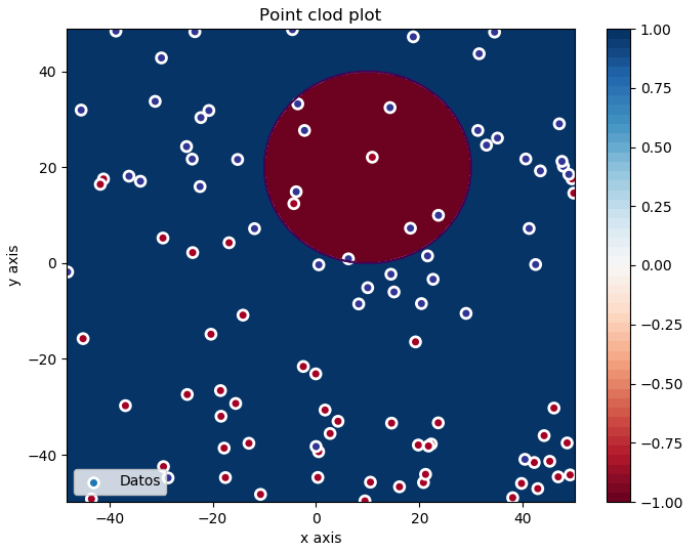
```
def f4(X):
    return X[:,1]-20*X[:,0]**2-5*X[:,0]+3
```

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.

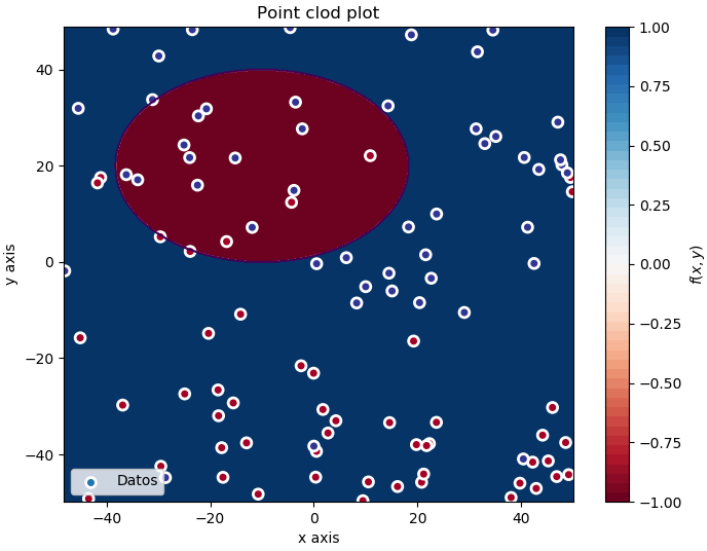
La función utilizada para visualizar las funciones junto con los datos es la siguiente:

```
def plot_datos_funcion(X, y, fz, title='Point clod plot',
                       xaxis='x axis', yaxis='y axis'):
    #Preparar datos
    min_xy = X.min(axis=0)
    max_xy = X.max(axis=0)
    border_xy = (max_xy-min_xy)*0.002

    #Generar grid de predicciones
    xx, yy = np.mgrid[min_xy[0]-border_xy[0]:max_xy[0]+
    border_xy[0]+0.001:border_xy[0],
    min_xy[1]-border_xy[1]:max_xy[1]+border_xy[1]+0.001:border_xy[1]]
    grid = np.c_[xx.ravel(), yy.ravel(), np.ones_like(xx).ravel()]
    pred_y = fz(grid)
    pred_y = np.clip(pred_y, -1, 1).reshape(xx.shape)

    #Plot
    f, ax = plt.subplots(figsize=(8, 6))
    contour = ax.contourf(xx, yy, pred_y, 50, cmap='RdBu',
    vmin=-1, vmax=1)
    ax_c = f.colorbar(contour)
    ax_c.set_label('$f(x, y)$')
    ax_c.set_ticks([-1, -0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75, 1])
    ax.scatter(X[:, 0], X[:, 1], c=y, s=50, linewidth=2,
    cmap="RdYlBu", edgecolor='white', label='Datos')

    ax.contour(xx, yy, pred_y, [0])
    ax.set(
    xlim=(min_xy[0]-border_xy[0], max_xy[0]+border_xy[0]),
    ylim=(min_xy[1]-border_xy[1], max_xy[1]+border_xy[1]),
    xlabel=xaxis, ylabel=yaxis)
    ax.legend()

    plt.title(title)
    plt.show()
```

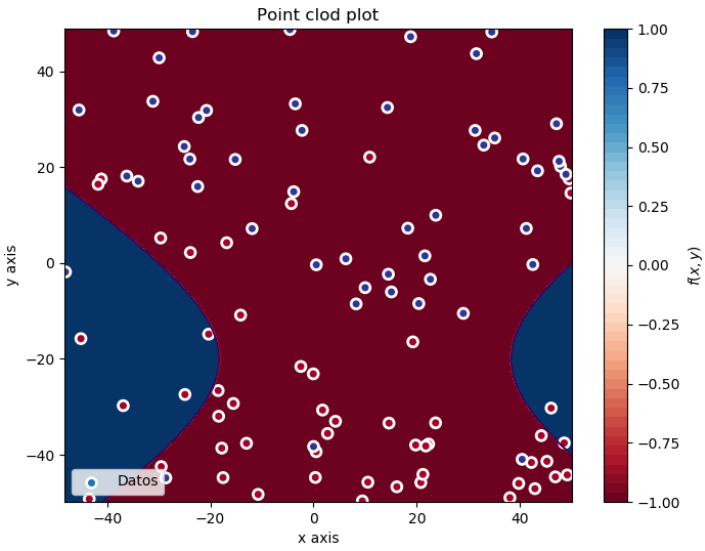
```
print('\nf(x,y) = (x-10)^2+(y-20)^2-400\n')
plot_datos_funcion(X, y_mod, f1)

print('\nf(x,y) = 0.5*(x+10)^2+(y-20)^2-400\n')
plot_datos_funcion(X, y_mod, f2)

print('\nf(x,y) = 0.5*(x-10)^2-(y+20)^2-400\n')
plot_datos_funcion(X, y_mod, f3)

print('\nf(x,y) = y-20x^2-5x+3\n')
plot_datos_funcion(X, y_mod, f4)
```



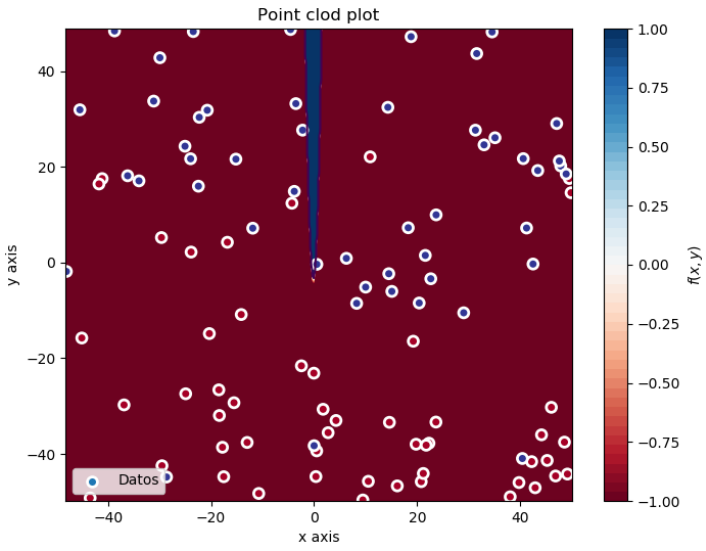
(a) $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$



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



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



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

Estas funciones a diferencia de la utilizada en el ejercicio anterior no son lineales. La primera se corresponde con la ecuación de una circunferencia centrada en el punto (10,20) y de radio 20. La segunda una elipse centrada en el punto (-10,20). La tercera se corresponde con el corte de un hiperboloide con el plano $z = 0$ y por último la cuarta se trata de una función cuadrática. Evidentemente estas funciones son más complejas que la lineal. Todas ellas tienen algo en común y es que tienen términos con exponente 2, a diferencia de la lineal que su máximo exponente era 1. Así podemos ver que está clase de funciones da lugar a funciones con una complejidad mayor que la lineal. La ventaja de estas funciones a pesar de ser más complejas es que pueden separar muchos casos en los cuales los datos no son linealmente separables, luego si extendemos la clase de la funciones lineales y añadimos estas funciones tendremos una clase más amplia y capaz de separar mas conjuntos de datos.

2. Modelos lineales

1. **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.

```
def ajusta_PLA(datos,label,max_iter,vini):
    w = vini
    x = np.concatenate((datos, np.ones((datos.shape[0], 1), np.float64)), axis=1)
    iters = 0
    while iters < max_iter:
        stop = True
        iters+=1
        for j in range(0,label.size):
            # si no tienen el mismo signo actualizamos w
            if np.sign(w.dot(x[j])) != label[j]:
                w = w + label[j]*x[j]
                stop = False
        # si no se ha actualizado w en ninguna de las iteraciones paramos
        if stop:
            break
    return w, iters
```

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.

```
vini = np.zeros(X.shape[1]+1, np.float64)
w, iters = ajusta_PLA(X, y, 500, vini)
print (' Iteraciones con vector cero:', iters)

suma = 0

for i in range(0,10):
    vini = simula_unif(N=1, dims=3, size=(0,1))
    w, iters = ajusta_PLA(X, y, 500, vini)
    suma = suma + iters

print (' Media iteraciones vector num aleatorios: ', suma/10)
```

Los resultados obtenidos han sido los siguientes:

```
Iteraciones con vector cero: 12
Media iteraciones vector num aleatorios: 14.1
```

El número de iteraciones con vector de números aleatorios ha sido en media mayor que el del vector cero, luego podemos intuir que vamos a obtener menos iteraciones con el vector cero. Aunque también es cierto que el número de veces que hemos realizado el experimento de iniciar el algoritmo con un vector de números aleatorios ha sido solo de 10 y no debería ser suficiente como para sacar conclusiones. Por ello repetí el experimento pero 100 veces en lugar de 10, obteniendo los siguientes resultados.

```
Iteraciones con vector cero: 12
Media iteraciones vector num aleatorios: 13.9
```

De nuevo vemos que el vector de cero ha realizado menos 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.

```
vini = np.zeros(X.shape[1]+1, np.float64)
w, iters = ajusta_PLA(X, y_mod, 500, vini)
print (' Iteraciones con vector cero:', iters)
```

```
suma = 0

for i in range(0,10):
    vini = np.random.uniform(0, 1, X.shape[1]+1)
    w, iters = ajusta_PLA(X, y_mod, 500, vini)
    suma = suma + iters

print (' Media iteraciones vector num aleatorios: ', suma/10)
```

En este caso los resultados obtenidos han sido:

```
Iteraciones con vector cero: 500.0
Media iteraciones vector num aleatorios: 500.0
```

El número de iteraciones obtenido en ambos casos se corresponde con el número máximo de iteraciones que fijé en el algoritmo. Esto ocurriría para cualquier número de iteraciones máximas que fijemos, ya que los datos no son linealmente separables por lo que el algoritmo no terminaría nunca.

2. **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 visualizable, 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 putos aleatorios del plano y calcular la línea que pasa por ambos. Seleccionar $N = 100$ puntos aleatorios $\{x_n\}$ de $\mathcal{X}$ y evaluar las respuestas $\{y_n\}$ de todos ellos respecto de la frontera elegida.

   A continuación simulamos la recta pedida, los 100 puntos aleatorios $\{x_n\}$ de $\mathcal{X}$ y evaluamos las respuestas $\{y_n\}$

```
a, b = simula_recta(intervalo=(0,2))
X = simula_unif(N=100, dims=2, size=(0,2))
y = np.sign(f(X,a,b))
```

   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, $1, 2, ..., N$, en el orden de los datos antes de usarlos en cada época del algoritmo.
   - Usar una tasa de aprendizaje de $\eta = 0{,}01$

```
def sigmoide(x):
    return 1/(np.exp(-x)+1)

# Regresion logistica Gradiente Descendente Estocastico
def rl_sgd(X, y, max_iters, tam_minibatch, lr = 0.01, epsilon = 0.01):
    x = np.concatenate((X, np.ones((X.shape[0], 1), np.float64)), axis=1)
    w = np.zeros(len(x[0]), np.float64)
    index = np.array(range(0,x.shape[0]))
    tam = tam_minibatch

    for k in range(0,max_iters):
        w_old = np.array(w)
        # permutamos los indices para el minibatch
```

```
        np.random.shuffle(index)
        for j in range(0,w.size):

            suma = np.sum(-y[index[0:tam:1]]*x[index[0:tam:1],j]*
            (sigmoide(-y[index[0:tam:1]]*(x[index[0:tam:1]].dot(w_old)))))

            w[j] -= lr*suma

        if np.linalg.norm(w-w_old) < epsilon:
            break

    return w
```

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).

```
def reetiquetar(y):
    y_ = np.array(y)

    for i in range(0, y_.size):
        if y_[i] == -1:
            y_[i] = 0

    return y_

def error_acierto(X,y,w):
    x = np.concatenate((X, np.ones((X.shape[0], 1), np.float64)), axis=1)
    tam = y.size
    suma = 0

    for i in range(0,tam):
        if np.abs(sigmoide(x[i].dot(w))-y[i]) > 0.5:
            suma += 1

    return suma/tam

w = rl_sgd(X ,y, 1000, 64)

y_ = reetiquetar(y)

print ('\nEin:',error_acierto(X,y_,w))

X_test = simula_unif(N=2000, dims=2, size=(0,2))
y_test = np.sign(f(X_test,a,b))

y_test_ = reetiquetar(y_test)

print ('\nEout:',error_acierto(X_test,y_test_,w))
```

Los resultados obtenidos han sido:

```
Ein: 0.02

Eout: 0.0375
```

Para el calculo del error se ha utilizado el error de acierto, para ello se han reetiquetado las etiquetas, pasando los -1 a 0. Se ha contado como fallo aquel en el que $\sigma(w^T x_i)$ distaba mas de 0.5 de y_i. Los errores obtenidos, tanto para E_{in} como para E_{out}, son bastante bajos, lo cual indica que el ajuste se ha hecho bastante bien, ya que hay un bajo porcentaje de fallo.

3. BONUS

Clasificación 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.
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 $\delta = 0{,}05$. ¿Qué cota es mejor?

A continuación se muestra la función utiliza para leer los datos en la que se seleccionan las muestras de los dígitos 4 y 8.

```
def readData(file_x, file_y):
   # Leemos los ficheros
   datax = np.load(file_x)
   datay = np.load(file_y)
   y = []
   x = []

   # Solo guardamos los datos cuya clase sea la 1 o la 5
   for i in range(0,datay.size):
      if datay[i] == 4 or datay[i] == 8:
         if datay[i] == 8:
            y.append(1)
         else:
            y.append(-1)
         x.append(np.array([datax[i][0], datax[i][1]]))

   x = np.array(x, np.float64)
   y = np.array(y, np.float64)

   return x, y
```

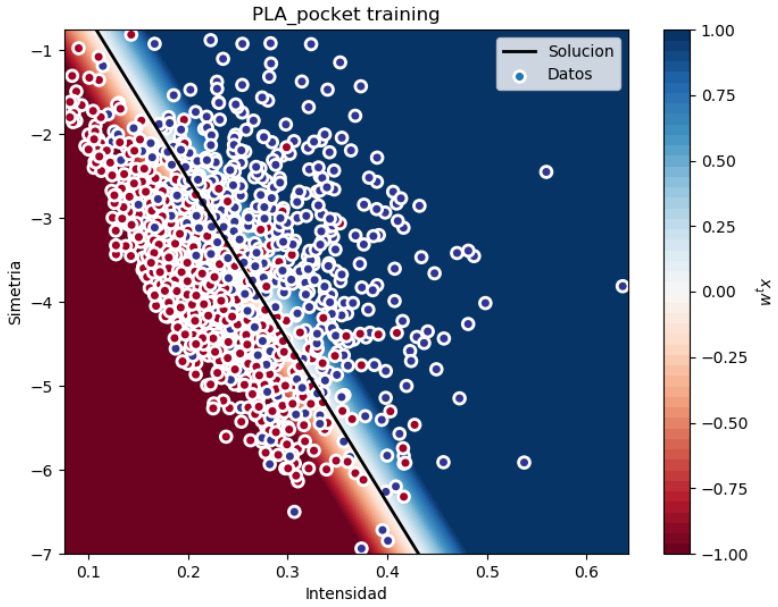
```
# Lectura de los datos de entrenamiento
x, y = readData('datos/X_train.npy', 'datos/y_train.npy')
# Lectura de los datos para el test
x_test, y_test = readData('datos/X_test.npy', 'datos/y_test.npy')
```

Como modelos de regresión lineal he utilizado el algoritmo de la pseudoinversa con el cual obtendremos un vector de pesos que utilizaremos como vector inicial para PLA pocket.

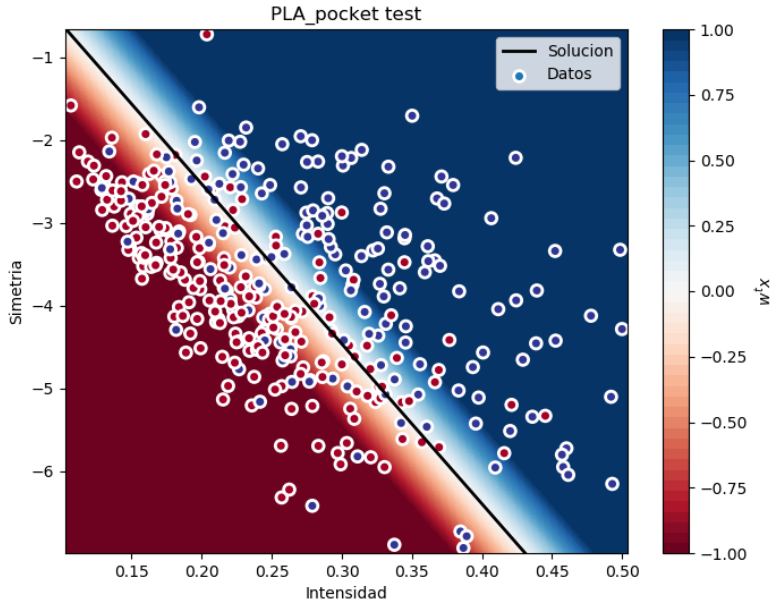
```
def pseudoinversa(X, y):
   x = np.concatenate((X, np.ones((X.shape[0], 1), np.float64)), axis=1)
   # Obtenemos la descomposicion en valores singulares
   u,d,vt = np.linalg.svd(x)
   # calculamos la inversa de d
   d_inv = np.zeros((d.size, d.size), np.float64)
   for i in range(0,d.size):
      d_inv[i,i] = 1/d[i]

   v = vt.transpose()
   # Calculo de la pseudoinversa de x
   x_inv = v.dot(d_inv).dot(d_inv).dot(v.transpose()).dot(x.transpose())
```

```
    w = x_inv.dot(y)
    return w

def error_acierto2(X,y,w):
    x = np.concatenate((X, np.ones((X.shape[0], 1), np.float64)), axis=1)
    tam = y.size
    suma = 0

    for i in range(0,tam):
        if np.sign(x[i].dot(w)) != y[i]:
            suma += 1

    return suma/tam

def PLA_pocket(datos,label,max_iter,vini):
    w = vini
    x = np.concatenate((datos, np.ones((datos.shape[0], 1), np.float64)), axis=1)
    # vamos a ir almacenando el mejor w encontrado junto al error que
    # se obtiene con dicho w
    mejor = w
    error_mejor = error_acierto2(datos,y,w)

    for i in range(0,max_iter):
        stop = True
        # Igual que en el PLA
        for j in range(0,label.size):
            if np.sign(w.dot(x[j])) != label[j]:
                w = w + label[j]*x[j]
                stop = False
        if stop:
            break
        else:
            error_actual = error_acierto2(datos,y,w)
            # si el error actual mejora al mejor actualizamos
            if error_actual < error_mejor:
                error_mejor = error_actual
                mejor = np.array(w)

    return mejor
```

Gráfica de los datos de entrenamiento junto con la función estimada.



Gráfica de los datos del test junto con la función estimada.




```
vini = pseudoinversa(x,y)
w = PLA_pocket(x, y, 500, vini)

a = -w[0]/w[1]
b = -w[2]/w[1]

# Dibujamos la recta obtenida con PLA_pocket junto con los datos de entrenamiento
plot_datos_recta(x, y, a, b, 'PLA_pocket training', 'Intensidad', 'Simetria')

# Dibujamos la recta obtenida con PLA_pocket junto con los datos para el test
plot_datos_recta(x_test, y_test, a, b, 'PLA_pocket test', 'Intensidad', 'Simetria')
```

Para calcular el error de nuevo se ha tenido en cuenta el error de acierto.

```
# Calculamos Ein y Etest
ein = error_acierto2(x,y,w)
etest = error_acierto2(x_test,y_test,w)

print ('\nBondad del resultado:\n')
print ('Ein: ',ein)
print ('\nEtest: ', etest)
```

Los resultados obtenidos han sido los siguientes:

```
Ein:  0.22529313232830822
Etest:  0.2540983606557377
```

Estos resultados indican que ha habido más de un 20 % de datos que no han sido clasificados correctamente, aún así viendo la gráfica y observando los datos junto a sus etiquetas se ve claramente que los datos no son separables linealmente, y que para haberse realizado el ajuste mediante una recta no está tan mal el error de acierto.

Para el cálculo de la cota sobre el valor de E_{out} he utilizado la fórmula dada en la siguiente desigualdad.

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

```
cotaein = ein + np.sqrt((8/y.size)*np.log(4*((2*y.size)**3 + 1)/0.05))
cotaetest = etest + np.sqrt((8/y_test.size)*np.log(4*((2*y_test.size)**3)/0.05))

print ('\nCotas sobre el valor de Eout\n')
print ('Cota basada en Ein:', cotaein)
print ('Cota basada en Etest:', cotaetest)
```

Los valores sobre la cota obtenidos han sido los siguientes:

```
Cotas sobre el valor de Eout:

Cota basada en Ein: 0.6562296377990118
Cota basada en Etest: 0.9809354817361753
```

La cota mejor ha resultado ser a de E_{in}, uno de los motivos, aparte de que el valor de E_{test} es mayor, es que el conjunto de test tenía menos datos que el de training.

Las cotas no nos dan valores muy optimistas, pero solo son cotas teóricas, en la práctica puede ser preferible obtener estimaciones sobre el valor de E_{out} mediante el valor de E_{test}, cuanto mayor sea el conjunto para el test mejor será la estimación realizada. Esto es debido a la siguiente desigualdad:

$$P(|E_{test}(g) - E_{out}(g)| > \epsilon) \leq 2e^{-2N\epsilon^2}$$

Cierta para todo ϵ y para todo g.

Conforme crece el valor de N la parte de la derecha de la desigualdad se hace menor y como es cierta para todo ϵ podemos tomar una valor de ϵ tan pequeño como queramos y al aumentar el valor de N hacer que la probabilidad de que $E_{out}(g)$ y $E_{test}(g)$ difieran más de un valor ϵ tienda a cero. Por lo que cuanto más datos para el test tengamos mejor será la estimación sobre el E_{out}.