Doble Grado en Ingeniería Informática y Matemáticas

APRENDIZAJE AUTOMÁTICO
(E. Computación y Sistemas Inteligentes)

TRABAJO-2: Programación

UNIVERSIDAD
DE GRANADA

Carlos Santiago Sánchez Muñoz

Grupo de prácticas 3 - Lunes

Email: carlossamu7@correo.ugr.es

30 de marzo de 2020

Índice

1. Ejercicio sobre la complejidad de $\mathcal{H}$ y el ruido

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 extraído de una distribución Gaussiana de media 0 y varianza dada, para cada dimensión, 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 $[-50, 50] \times [-50, 50]$.**

Apartado 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)`.

En este ejercicio sólo tenemos que llamar a la función `simula_unif` con los parámetros del enunciado. Después pintaremos los puntos obtenidos usando `pyplot.scatter`.

```
print("a) simula_unif(N, dim, rango) con N=50, dim=2 y rango=[-50,50].")
x_unif = simula_unif(50, 2, [-50, 50])
plt.scatter(x_unif[:, 0], x_unif[:, 1])
plt.title("Nube de puntos con simula_unif")
plt.gcf().canvas.set_window_title('Ejercicio 1 - Apartado 1a)')
plt.show()
```

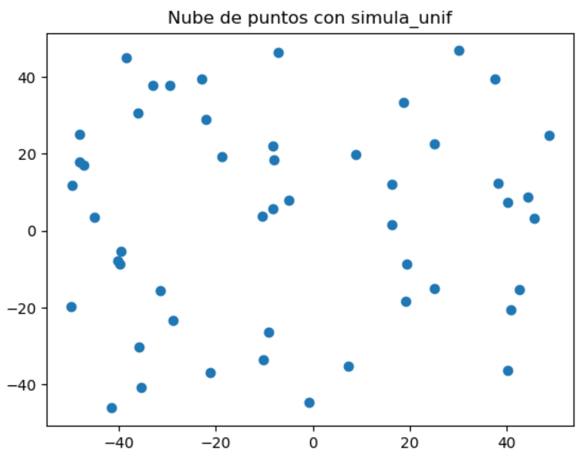

Imagen 1: Nube de puntos con `simula_unif`

La *Imagen 1* es el resultado de la implementación en donde encontramos `50` puntos distribuidos uniformemente en `[-50, 50]`.

b) Considere `N = 50`, `dim = 2` y `sigma = [5, 7]` con `simula_gaus(N, dim, sigma)`.

Este apartado es idéntico al anterior, llamaremos a la función `simula_gauss` con los parámetros indicados. Posteriormente pintamos los puntos.

```
print("\nb) simula_gaus(N, dim, sigma) con N=50, dim=2 y sigma=[5,7].")
x_gaus = simula_gaus(50, 2, np.array([5, 7]))
plt.scatter(x_gaus[:, 0], x_gaus[:, 1])
plt.title("Nube de puntos con simula_gaus")
plt.gcf().canvas.set_window_title('Ejercicio 1 - Apartado 1b)')
plt.show()
```

El resultado de la implementación debe ser una gráfica donde están representados `50` puntos $2D$ en la que cada punto ha sido extraído de una distribución gaussiana de media μ=`(0,0)` y varianza `(`σ_x^2`,`σ_y^2`)=(5,7)`. Resultado:

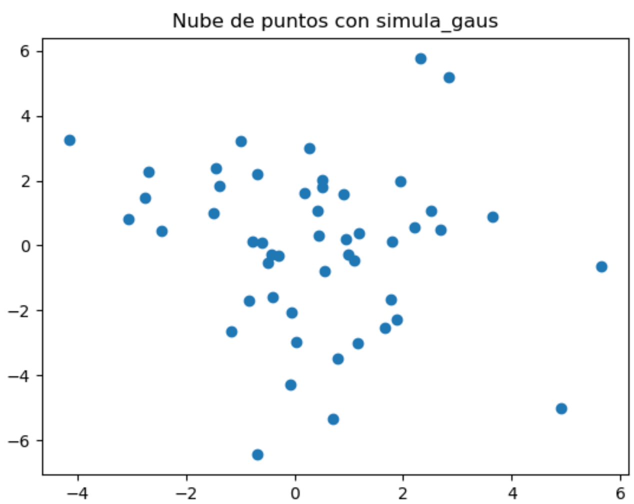

Imagen 2: Nube de puntos con `simula_gaus`

Efectivamente como resultado de la distribución Gaussiana los puntos están agrupados entorno a la media.

Apartado 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()`.

Voy a utilizar los parámetros `N=50` puntos $2D$ e intervalo `[-50, 50]` para llamar a `simula_unif` y generar la muestra. Para las etiquetas utilizo la función `simula_recta` con el mismo intervalo como parámetro y me devuelve la pendiente y la ordenada en el origen.

A partir de aquí utilizo una función `signo` implementada por mí con la peculiaridad de que para el valor 0 me devuelve 1 (signo positivo). Del mismo modo implementamos la función $f(x, y) = y - ax - b$. Por último, las etiquetas son el signo de evaluar f con los parámetros adecuados (las primeras y segundas componentes de cada punto de la muestra junto con `a` y `b`). Sólo voy a mostrar en la memoria el código de la llamada:

```
print ("\n### Apartado 2 ###\n")
N = 50
x = simula_unif(N, 2, [-50, 50])
a, b = simula_recta([-50, 50])
y = np.empty((N, ))
for i in range(N):
    y[i] = signo(f(x[i,0], x[i,1], 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)

Pintamos adecuadamente los puntos con su etiqueta correspondiente $\{1, -1\}$. Usaremos colores distintos para cada etiqueta y del mismo modo añadimos la gráfica de la recta usada para el cálculo de las etiquetas.

```
print("a) Grafica con las etiquetas de los puntos y la recta simulada.")
plt.scatter(x[y == -1][:, 0], x[y == -1][:, 1], label="Etiqueta -1")
plt.scatter(x[y == 1][:, 0], x[y == 1][:, 1], c="orange", label="Etiqueta 1")
points = np.array([np.min(x[:, 0]), np.max(x[:, 0])])
plt.plot(points, a*points+b, c="red", label="Recta simulada")
plt.legend()
plt.title("Clasificando puntos con la recta simulada")
plt.gcf().canvas.set_window_title('Ejercicio 1 - Apartado 2a)')
plt.show()
```

A continuación tenemos la gráfica resultante. En ella todos los puntos están bien clasificados respecto de la recta en rojo que es la que ha servido para etiquetar todos los puntos.

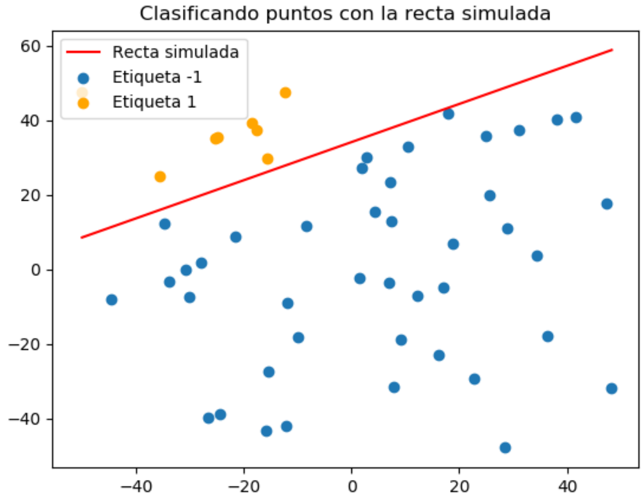

Imagen 3: Clasificando puntos con la recta simulada

El resultado es lo esperado y comentado anteriormente.

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)

Lo primero que vamos a hacer va a ser añadir ruido a las etiquetas. Más concretamente cambiaremos el signo a un 10 % de las etiquetas elegidas aleatoriamente. A continuación pintamos los resultados (con ruido) de nuevo.

```
print("b) Grafica con las etiquetas de los puntos con ruido y la recta.")
y_noise = np.copy(y)      # Introducimos ruido en el 10%
ind = np.random.choice(N, size=int(N/10), replace=False)
for i in ind:
    y_noise[i] = -y[i]

plt.scatter(x[y_noise == -1][:, 0], x[y_noise == -1][:, 1],
            label="Etiqueta -1")
plt.scatter(x[y_noise == 1][:, 0], x[y_noise == 1][:, 1],
            c="orange", label="Etiqueta 1")
plt.plot(points, a*points+b, c="red", label="Recta simulada")
plt.legend()
plt.title("Clasificando puntos (con ruido) con la recta simulada")
plt.gcf().canvas.set_window_title('Ejercicio 1 - Apartado 2b)')
plt.show()
input("--- Pulsar tecla para continuar ---")
```

La gráfica que obtenemos ya no tiene todos los puntos bien clasificados respecto de la recta en rojo. En concreto, como eran `N=50` puntos y hemos cambiado un 10 %, existen `5` puntos mal clasificados. Veamos:

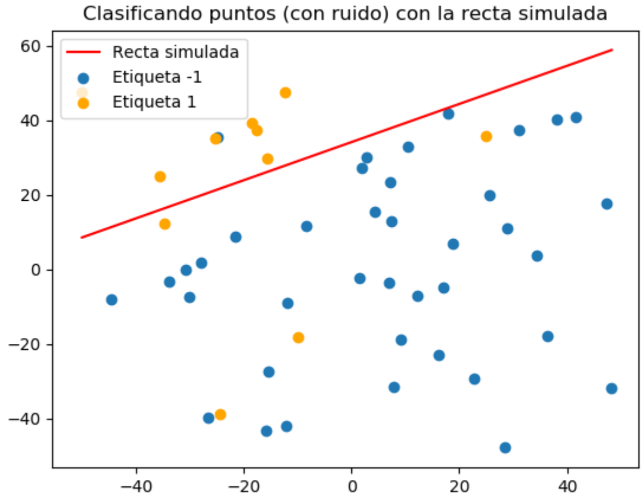

Imagen 4: Clasificando puntos (con ruido) con la recta simulada

Como predecíamos hay `4` puntos amarillos con los azules y `1` azul en la zona de los amarillos.

Apartado 3

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

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

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.

Como para cada función deseamos visualizar la muestra de los puntos, con su etiquetado y la gráfica de las funciones voy a implementar una función `print_graf` que realice este trabajo.

```
""" Para cada funcion pasada por argumento visualiza los puntos con sus
etiquetas y la grafica de la funcion como frontera de clasificacion.
- x: vector de puntos 2D que son las caracteristicas.
- y: vector de etiquetas.
- fun: funcion a representar.
```

```
- title: titulo de la funcion."""
def print_graf(x, y, fun, title=""):
    plt.scatter(x[y == -1][:, 0], x[y == -1][:, 1], label="Etiqueta -1")
    plt.scatter(x[y == 1][:, 0], x[y == 1][:, 1],
                c="orange", label="Etiqueta 1")
    #Generamos el contorno de fun
    x1, y1 = np.meshgrid(np.linspace(-50, 50, 100), np.linspace(-50, 50, 100))
    contorno = fun(x1,y1)
    plt.contour(x1, y1, contorno, [0], colors='red')
    plt.title("Clasificando puntos (con ruido) con la " + title)
    plt.gcf().canvas.set_window_title('Ejercicio 1 - Apartado 3')
    plt.legend()
    plt.show()
```

En segundo lugar he implementado una función `get_porc` que devuelve el porcentaje de puntos bien clasificados para los puntos `datos`, las etiquetas `labels` y la función `fun`. Para cada función llamaremos a `get_porc` y así manejaremos este dato también.

```
""" Calcula el porcentaje de puntos bien clasificados
- datos: datos.
- labels: etiquetas.
- fun: funcion clasificadora."""
def get_porc(datos, labels, fun):
    aciertos = labels*fun(datos[:, 0], datos[:, 1])
    return 100*len(aciertos[aciertos >= 0])/len(labels)
```

Provistos de todo lo anterior la ejecución de lo pedido en este Apartado 3 es sencillo. Una vez tengamos definidas las funciones clasificadoras, las cuales he denominado `f1`, `f2`, `f3` y `f4`, llamo para cada función a `print_graf` y muestro el procentaje de acierto con `get_porc`.

```
""" óFuncin en dos variables que representa una elipse
- x: primera variable de la ófuncin.
- y: segunda variable de la ófuncin."""
def f1(x, y):
    return (x-10)**2 + (y-20)**2 - 400

# El resto de funciones son análogas.

""" Funcion que ejecuta todo el apartado 3 """
def apartado3(x, y):
    print ("\n### Apartado 3 ###\n")
    print_graf(x, y, f1, "Elipse1")
    print("Acierto para '{}': {}%".format("Elipse1", get_porc(x, y, f1)))
    print_graf(x, y, f2, "Elipse2")
    print("Acierto para '{}': {}%".format("Elipse2", get_porc(x, y, f2)))
    print_graf(x, y, f3, "Elipse3")
    print("Acierto para '{}': {}%".format("Elipse3", get_porc(x, y, f3)))
    print_graf(x, y, f4, "áParbola")
    print("Acierto para '{}': {}%".format("áParbola", get_porc(x, y, f4)))
    input("--- Pulsar tecla para continuar ---")
```

Podemos observar las cuatro gráficas obtenidas.

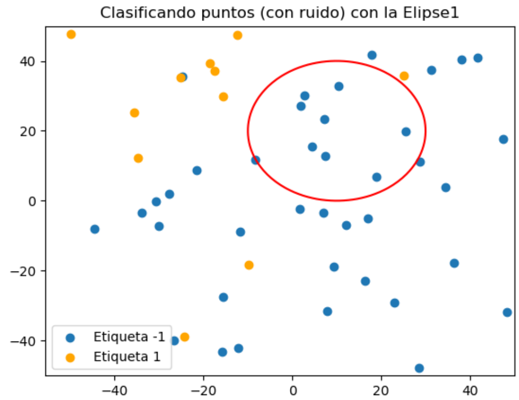

Imagen 5: Clasificando puntos (con ruido) con la Elipse1

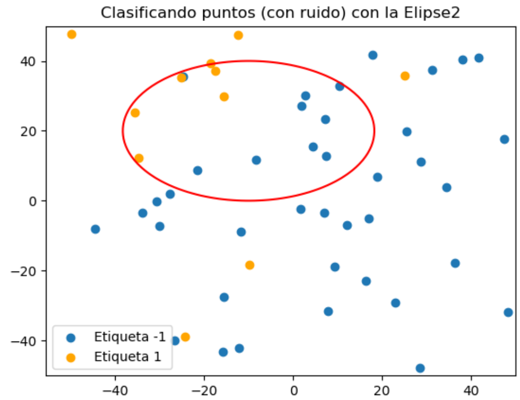

Imagen 6: Clasificando puntos (con ruido) con la Elipse2

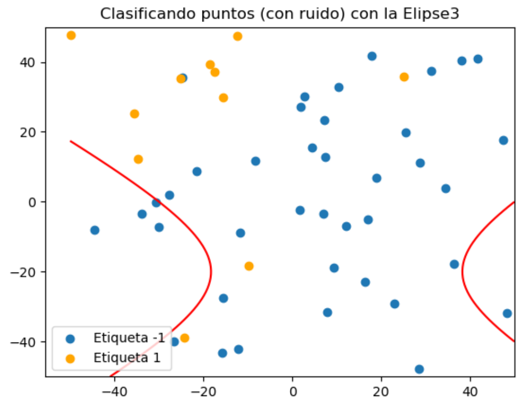

Imagen 7: Clasificando puntos (con ruido) con la Elipse3

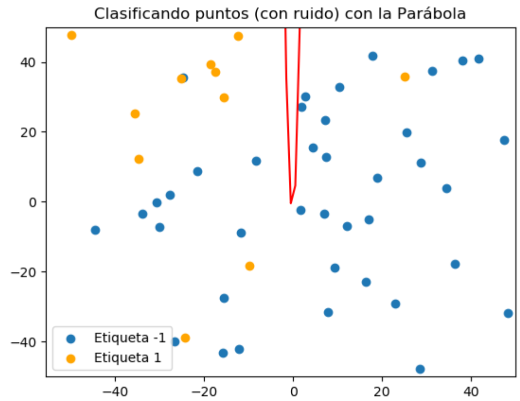

Imagen 8: Clasificando puntos (con ruido) con la Parábola

Escribir.

2. Modelos Lineales

Apartado 1

<u>Algoritmo Perceptron</u>: 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.

Escribir.

```
""" Calcula el hiperplano solucion a un problema de clasificacion binaria.
Devuelve el vector de pesos y el numero de iteraciones.
- datos: matriz de datos.
- labels: etiquetas.
- max_iters: numero maximo de iteraciones.
- vini: valor inicial."""
def ajusta_PLA(datos, labels, max_iters, vini):
    w = vini.copy()

    for it in range(1, max_iters + 1):
        w_old = w.copy()

        for dato, label in zip(datos, labels):
            if signo(w.dot(dato)) != label:
                w += label*dato

        if np.all(w == w_old):  # No hay cambios
            return w, it

    return w, it
```

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.**

Escribir. Hablar de `get_porc()`

```
""" Ejecuta ajusta_PLA() con vini un vector de ceros y luego 10 aleatorios
- datos: matriz de datos.
- labels: etiquetas.
- max_iters: numero maximo de iteraciones.
"""
def ejecuta_PLA(datos, labels, max_iters):
    print("    Vector inicial cero")
    w, it = ajusta_PLA(datos, labels, max_iters, np.zeros(3))
    print("    Num. iteraciones: {}".format(it))
    print("    Acierto: {}%".format(get_porc(datos, labels, w)))
```

```
    print("\n    Diez vectores iniciales aleatorios")
    iters = np.empty((10, ))
    percs = np.empty((10, ))
    for i in range(10):
        w, it = ajusta_PLA(datos, labels, max_iters, np.random.rand(3))
        iters[i] = it
        percs[i] = get_porc(datos, labels, w)
    print("    N. iteraciones: {}".format(np.mean(iters)))
    print("    Aciertos: {}%".format(np.mean(percs)))
```

Escribir.

```
print("a) Ejecutar PLA con los datos del ejercicio 1.2a).\n")
N = 50
a, b = simula_recta([-50, 50])
x = np.hstack((np.ones((N, 1)), simula_unif(N, 2, [-50, 50])))
y = np.empty((N, ))
for i in range(N):
    y[i] = signo(f(x[i,1], x[i,2], a, b))
ejecuta_PLA(x, y, 1000)
```

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.**

Escribir.

```
print("\nb) Ejecutar PLA con los datos del ejercicio 1.2b).\n")
y_noise = np.copy(y)    # Introducimos ruido en el 10%
ind = np.random.choice(N, size=int(N/10), replace=False)
for i in ind:
    y_noise[i] = -y[i]
ejecuta_PLA(x, y_noise, 1000)
```

Apartado 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 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 $x \in X$. Elegir una línea en el plano que pase por X como la frontera entre $f(x) = 1$ (donde y toma valores $+1$) y $f(x) = 0$ (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 $\{x_n\}$ 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 $\|w(t-1) - w(t)\| < 0,01$, donde $w(t)$ 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 $\eta = 0,01$.**

Escribir.

```
""" Calcula el gradiente de la Regresion Logistica.
- dato: un solo vector de caracteristicas.
- label: etiqueta del vector.
- w: vector de pesos.
"""
def grad_RL(dato, label, w):
    return -label*dato/(1 + np.exp(label*w.dot(dato)))

""" Algoritmo de regresion logistica con SGD.
- datos: matriz de datos.
- labels: etiquetas.
- eta: tasa de aprendizaje.
"""
def sgd_RL(datos, labels, eta):
    w = np.zeros(len(datos[0]))
    ind_set = np.arange(len(datos))
    changed = True          # indica si ha habido cambios en una época

    while changed:
        w_old = w.copy()
        ind_set = np.random.permutation(ind_set)
        for ind in ind_set:
            w = w - eta*grad_RL(datos[ind], labels[ind], w)
        changed = np.linalg.norm(w - w_old) >= 0.01

    return w
```

Escribir.

```
print("a) Implementar RL con SGD. Mostramos grafica con el resultado.")
# Calculamos datos y labels
N = 100; intervalo = [0, 2]
a, b = simula_recta([0, 2])
datos = np.hstack((np.ones((N, 1)), simula_unif(N, 2, intervalo)))
labels = np.empty((N, ))
for i in range(N):
    labels[i] = signo(f(datos[i, 1], datos[i, 2], a, b))

# Calculamos el vector de pesos usando RL+SGD
w = sgd_RL(datos, labels, 0.01)

# Representamos la recta obtenida
```

```
plt.scatter(datos[labels == -1][:, 1], datos[labels == -1][:, 2],
            label="Etiqueta -1")
plt.scatter(datos[labels == 1][:, 1], datos[labels == 1][:, 2],
            c="orange", label="Etiqueta 1")
points = np.array([np.min(datos[:, 1]), np.max(datos[:, 1])])
plt.plot(points, (-w[1]*points - w[0])/w[2], c="red", label="Recta RL")
plt.legend()
plt.title("Regresion Logistica con SGD")
plt.gcf().canvas.set_window_title('Ejercicio 2 - Apartado 2a)')
plt.show()
```

Escribir.

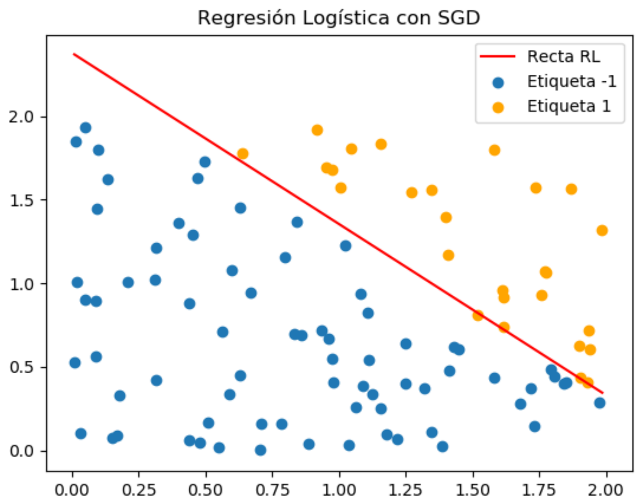

Imagen 9: Regresión Logística con SGD

Escribir.

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).

Escribir.

```
"""Calcula el error logistico.
- dato: un solo vector de caracteristicas.
- label: etiqueta del vector.
- w: vector de pesos."""
def Err_RL(datos, labels, w):
    return np.mean(np.log(1 + np.exp(-labels*datos.dot(w))))
```

Escribir.

```
print("\nb) Encontrar solucion g y estimar Eout con nuevas muestras.\n")
```

```python
# Calculamos datos y labels de test (uso el mismo N>999)
datos_test = np.hstack((np.ones((N, 1)), simula_unif(N, 2, intervalo)))
labels_test = np.empty((N, ))
for i in range(N):
    labels_test[i] = signo(f(datos_test[i, 1], datos_test[i, 2], a, b))

# Representamos la recta y el conjunto de test
plt.scatter(datos_test[labels_test == -1][:, 1],
        datos_test[labels_test == -1][:, 2], label="Etiqueta -1")
plt.scatter(datos_test[labels_test == 1][:, 1],
        datos_test[labels_test == 1][:, 2], c="orange", label="Etiqueta 1")
points = np.array([np.min(datos_test[:, 1]), np.max(datos_test[:, 1])])
plt.plot(points, (-w[1]*points - w[0])/w[2], c="red", label="Recta RL")
plt.legend()
plt.title("Regresion Logistica con SGD (test)")
plt.gcf().canvas.set_window_title('Ejercicio 2 - Apartado 2b)')
plt.show()

# Mostramos cálculos de porcentaje de aciertos y Eout
print("    Aciertos (test): {}%".format(get_porc(datos_test, labels_test, w)))
print("    Eout: {}".format(Err_RL(datos_test, labels_test, w)))
```

Escribir.

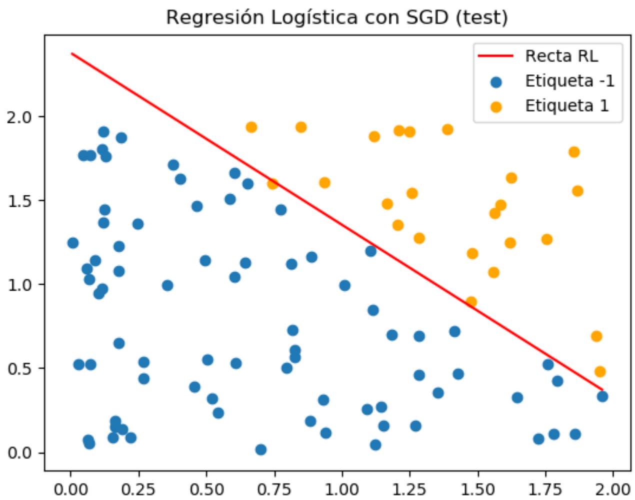

Imagen 10: Regresión Logística con SGD (test)

3. Bonus

<u>Clasificación de Dígitos</u>. 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.

Apartado 1

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

Escribir.

```
print("Leyendo los ficheros de datos de train y test.")
# 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')
```

Apartado 2

Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora. Responder a las siguientes cuestiones.

Escribir.

```
"""Calcula el hiperplano que hace de clasificador binario.
Devuelve el vector de pesos y el numero de iteraciones.
- datos: matriz de datos.
- labels: etiquetas.
- max_iters: numero maximo de iteraciones.
- vini: valor inicial."""
def PLA_Pocket(datos, labels, max_iters, vini):
    w = vini.copy()
    w_best = w.copy()
    err_best = Err(datos, labels, w_best)

    for it in range(1, max_iters + 1):
        w_old = w.copy()

        for dato, label in zip(datos, labels):
            if signo(w.dot(dato)) != label:
                w += label*dato

        err = Err(datos, labels, w)
        if err < err_best:
            w_best = w.copy()
            err_best = err

        if np.all(w == w_old):  # No hay cambios
            return w_best, it

    return w_best, it
```

a) **Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.**

Escribir.

```
# Calculamos los vectores de pesos
print("Calculando los vectores de pesos con un modelo de regresion lineal.")
w_pin = pseudoinverse(x, y)
print("Aplicando mejora de PLA-Pocket.\n")
w_pla, _ = PLA_Pocket(x, y, 1000, w_pin)

# Representamos las rectas obtenidas para train
print("a) Grafico de los datos de entrenamiento con la funcion estimada.")
plt.scatter(x[y == -1][:, 1], x[y == -1][:, 2], label="Etiqueta -1")
plt.scatter(x[y == 1][:, 1], x[y == 1][:, 2], c="orange",
            label="Etiqueta 1")
points = np.array([np.min(x[:, 1]), np.max(x[:, 1])])
plt.plot(points, (-w_pin[1]*points - w_pin[0])/w_pin[2], c="red",
         label="Pseudoinversa")
plt.plot(points, (-w_pla[1]*points - w_pla[0])/w_pla[2], c="green",
         label="PLA-Pocket")
plt.legend()
plt.title("Regresion sobre digitos manuscritos (train)")
plt.gcf().canvas.set_window_title('Bonus')
plt.show()

# Representamos las rectas obtenidas para test
print("   Grafico de los datos de test con la funcion estimada.")
plt.scatter(x_test[y_test == -1][:, 1], x_test[y_test == -1][:, 2],
            label="Etiqueta -1")
plt.scatter(x_test[y_test == 1][:, 1], x_test[y_test == 1][:, 2],
            c="orange", label="Etiqueta 1")
points = np.array([np.min(x_test[:, 1]), np.max(x_test[:, 1])])
plt.plot(points, (-w_pin[1]*points - w_pin[0])/w_pin[2], c="red",
         label="Pseudoinversa")
plt.plot(points, (-w_pla[1]*points - w_pla[0])/w_pla[2], c="green",
         label="PLA-Pocket")
plt.legend()
plt.title("Regresion sobre digitos manuscritos (train)")
plt.gcf().canvas.set_window_title('Bonus')
plt.show()
```

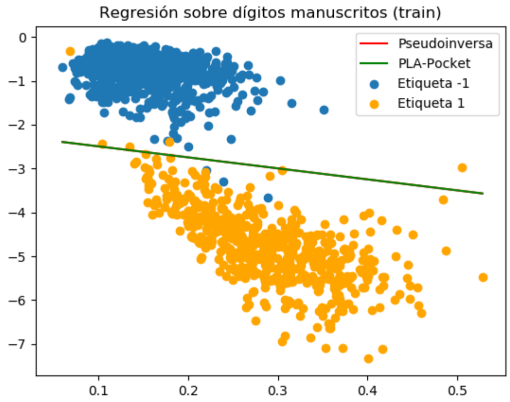

Imagen 11: Imágenes sobre dígitos manuscritos (train)

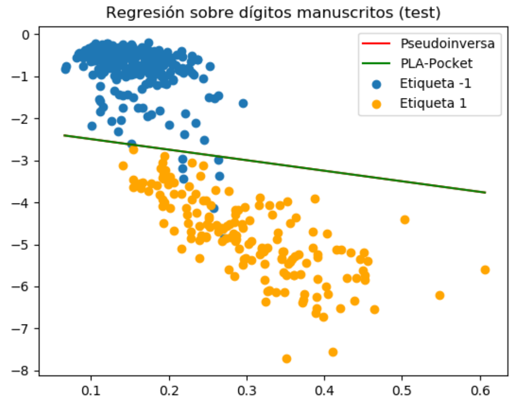

Imagen 12: Imágenes sobre dígitos manuscritos (test)

b) Calcular E_{in} y E_{test} (error sobre los datos de test).

Escribir.

```
# Imprimimos el cálculo de los errores
print("\nb) Calcular Ein y Etest.\n")
print("    Ein    para Pseudoinversa: ", Err(x, y, w_pin))
print("    Etest  para Pseudoinversa: ", Err(x_test, y_test, w_pin))
print("\n    Ein    para PLA-Pocket    : ", Err(x, y, w_pla))
print("    Etest  para PLA-Pocket    : ", Err(x_test, y_test, w_pla))
```

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?