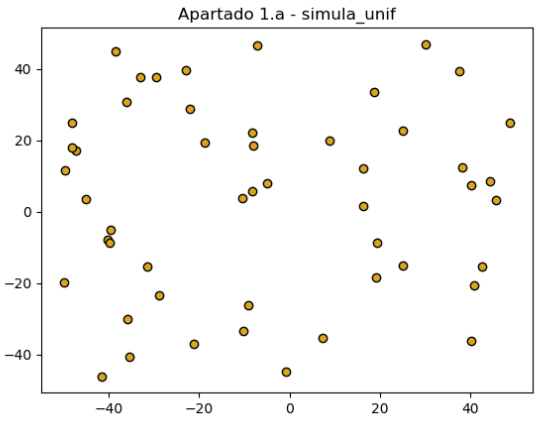
Trabajo 2: Programación

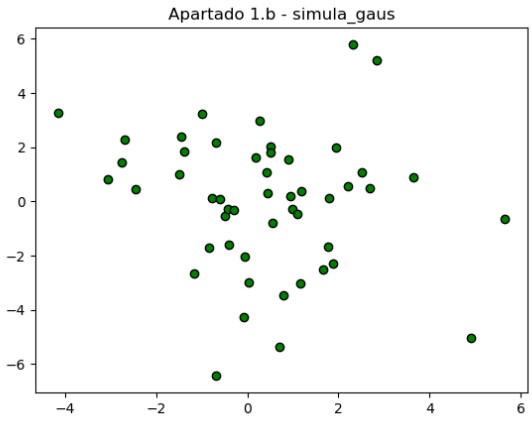
Ignacio Vellido Expósito

1. Ejercicio sobre la complejidad de H y el ruido

Apartado 1




```
x = simula_unif(50, 2, [-50,50])
```

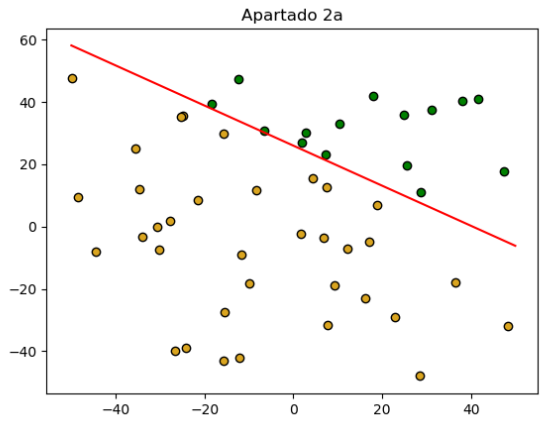



```
x = simula_gaus(50, 2, np.array([5,7]))
```

Apartado 2

Primeramente, para una muestra sin ruido:

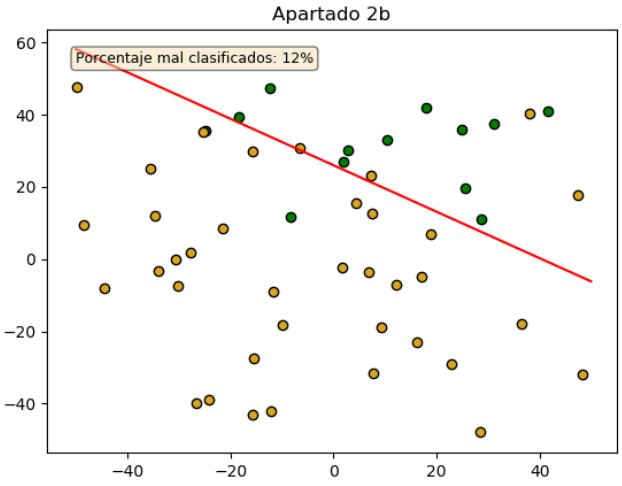
```
def f(x, y, a, b):
    return signo(y - a*x - b)


x = simula_unif(50, 2, [-50,50])
recta = simula_recta([-50,50])

# Asignamos las etiquetas
labels = np.array([[0,0]])
labels = np.delete(labels, (0), axis=0)

for (a,b) in zip(x[:,0], x[:,1]):
  labels = np.append(labels, f(a, b, recta[0], recta[1]))
```



Como dice el enunciado, la recta clasifica perfectamente la muestra.

Tras introducir error con un 10% de probabilidad:

```
"""
No suele ser exactamente un 10%, pero considero que esta es una forma más
relista de simular el ruido
"""
# Cambiamos el signo con un 10% de probabilidad
for i, l in enumerate(labels):
  n = np.random.choice(2, 1, p=[0.9, 0.1])
  if n == 1:
    labels[i] = -l

# Comprobamos el nº de puntos mal clasificados
misclassified = 0

truelabels = np.array([[0,0]])
truelabels = np.delete(truelabels, (0), axis=0)

for (a,b) in zip(x[:,0], x[:,1]):
  truelabels = np.append(truelabels, f(a, b, recta[0], recta[1]))

for (a,b) in zip(labels, truelabels):
  if a != b:
    misclassified += 1

misclassified /= len(truelabels)
misclassified = int(100*misclassified)

text = "Porcentaje mal clasificados: " + str(misclassified) +"%"
```



Y aquí vemos como la recta no clasifica perfectamente, pues el conjunto no es separable linealmente.

Apartado 3

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

def f2(X):
  return .5*((X[:,0] + 10)**2) + (X[:,1] - 20)**2 - 400

def f3(X):
  return .5*((X[:,0] - 10)**2) + (X[:,1] + 20)**2 - 400

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

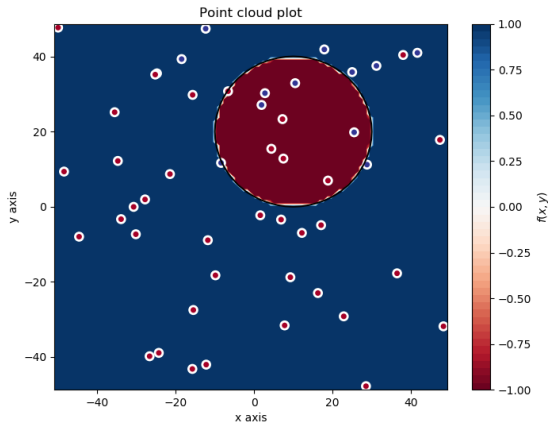
```python
plot_datos_cuad(x, labels2b, f1)

# Mostrar porcentaje de fallos al clasificar
valoresF1 = f1(x)

labels = np.array([[0,0]])
labels = np.delete(truelabels, (0), axis=0)

for i in valoresF1:
  labels = np.append(labels, signo(i))


truelabels = np.array([[0,0]])
truelabels = np.delete(truelabels, (0), axis=0)

for (a,b) in zip(x[:,0], x[:,1]):
  truelabels = np.append(truelabels, f(a, b, recta[0], recta[1]))


# Comprobamos el nº de malas clasificiaciones
misclassified = 0

for (a,b) in zip(labels, truelabels):
  if a != b:
    misclassified += 1

misclassified /= len(truelabels)
misclassified = int(100*misclassified)

print("F1 - Porcentaje mal clasificados: " + str(misclassified) +"%")
```

```python
plot_datos_cuad(x, labels2b, f2)

# Mostrar porcentaje de fallos al clasificar
valoresF2 = f2(x)

labels = np.array([[0,0]])
labels = np.delete(truelabels, (0), axis=0)

for i in valoresF2:
  labels = np.append(labels, signo(i))


truelabels = np.array([[0,0]])
truelabels = np.delete(truelabels, (0), axis=0)

for (a,b) in zip(x[:,0], x[:,1]):
  truelabels = np.append(truelabels, f(a, b, recta[0], recta[1]))


# Comprobamos el nº de malas clasificiaciones
misclassified = 0

for (a,b) in zip(labels, truelabels):
  if a != b:
    misclassified += 1

misclassified /= len(truelabels)
misclassified = int(100*misclassified)

print("F2 - Porcentaje mal clasificados: " + str(misclassified) +"%")
```

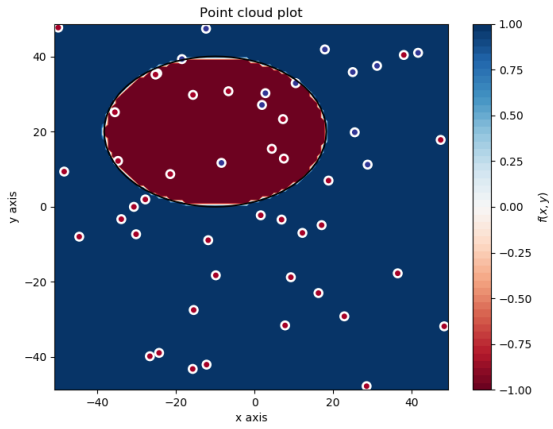
```python
plot_datos_cuad(x, labels2b, f3)

# Mostrar porcentaje de fallos al clasificar
valoresF3 = f3(x)

labels = np.array([[0,0]])
labels = np.delete(truelabels, (0), axis=0)

for i in valoresF3:
  labels = np.append(labels, signo(i))


truelabels = np.array([[0,0]])
truelabels = np.delete(truelabels, (0), axis=0)

for (a,b) in zip(x[:,0], x[:,1]):
  truelabels = np.append(truelabels, f(a, b, recta[0], recta[1]))


# Comprobamos el nº de malas clasificiaciones
misclassified = 0

for (a,b) in zip(labels, truelabels):
  if a != b:
    misclassified += 1

misclassified /= len(truelabels)
misclassified = int(100*misclassified)

print("F3 - Porcentaje mal clasificados: " + str(misclassified) +"%")
```

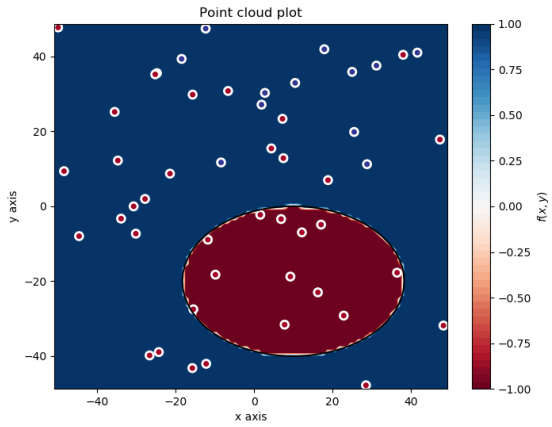
```python
plot_datos_cuad(x, labels2b, f4)

# Mostrar porcentaje de fallos al clasificar
valoresF4 = f4(x)

labels = np.array([[0,0]])
labels = np.delete(truelabels, (0), axis=0)

for i in valoresF4:
  labels = np.append(labels, signo(i))


truelabels = np.array([[0,0]])
truelabels = np.delete(truelabels, (0), axis=0)

for (a,b) in zip(x[:,0], x[:,1]):
  truelabels = np.append(truelabels, f(a, b, recta[0], recta[1]))


# Comprobamos el nº de malas clasificiaciones
misclassified = 0

for (a,b) in zip(labels, truelabels):
  if a != b:
    misclassified += 1

misclassified /= len(truelabels)
misclassified = int(100*misclassified)

print("F4 - Porcentaje mal clasificados: " + str(misclassified) +"%")
```

f_1

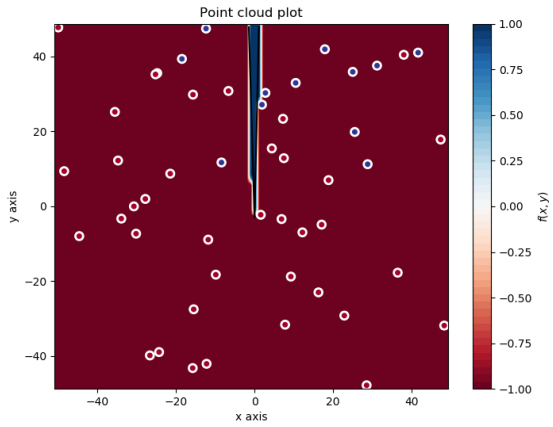


f_2





f_3



f_4

```
F1 - Porcentaje mal clasificados: 42%
F2 - Porcentaje mal clasificados: 42%
F3 - Porcentaje mal clasificados: 42%
F4 - Porcentaje mal clasificados: 40%
```

Se deduce que un separador más complicado no tiene por qué representar mejor el problema que uno más simple. En este caso ocurre lo contrario, mientras que con el lineal se consigue un porcentaje de error en torno al 10% con los no lineales no bajamos del 40%.

Pese a tener mayor flexibilidad, las fronteras de estas funciones son totalmente arbitrarias, y con un porcentaje de error de ese nivel se puede afirmar que no se aprende nada (aunque en este caso no hemos utilizado algoritmos de aprendizaje).

2. Modelos lineales

Apartado 1 – Algoritmo Perceptron

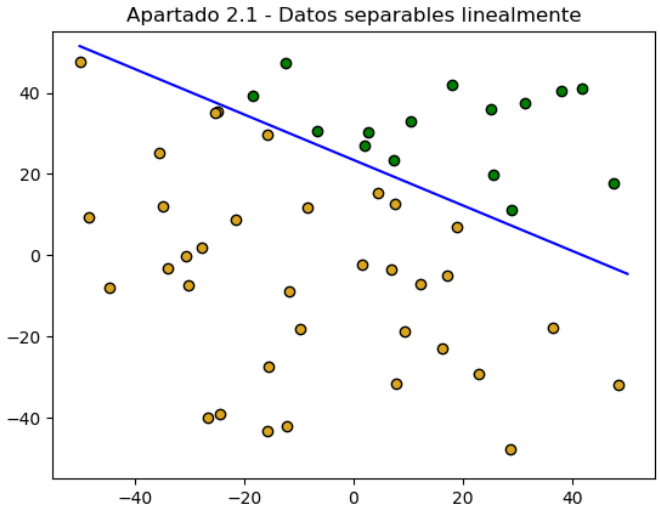
```python
def ajusta_PLA(datos, label, max_iter, vini):
    # Variables auxiliares
    w = vini.copy()
    w_old = w.copy()
    noChange = False
    iterations = 0

    while (not noChange) and iterations < max_iter:
      for (x,y) in zip(datos, label):
        iterations += 1
        h = w.transpose().dot(x)

        if signo(h) != y:
          w = w + np.dot(y, x)

      # Comprobamos si se han modificado los pesos
      if np.array_equal(w_old,w):
        noChange = True

      # Guardamos el valor de w en esta iteración
      w_old = w.copy()

    return (w, iterations)
```

```python
# Apartado a
print("Ejercicio 2.1.a\n")

# Añadir columna de unos
c0 = np.full(len(data2a),1)
puntos = np.copy(data2a)
data2a = np.column_stack((c0, data2a))

# Con vector 0 ------------------------------------------------------
max_iter = 50_000
# Creamos array con tantas columnas como las de la matriz de datos
vini = np.zeros(np.size(data2a,1))
w_cero, iteration = ajusta_PLA(data2a, labels2a, max_iter, vini)

print("Comenzando con vector de ceros")
print('Valor de iteraciones necesarias para converger: ' + str(iteration))
print("\n-----------\n")
```

```python
# Random initializations ---------------------------------------------
iterations = []
for i in range(0,10):
    vini = np.random.uniform(low=0, high=1, size=np.size(data2a,1))

    w_rand, iteration = ajusta_PLA(data2a, labels2a, max_iter, vini)
    print("Iteración: " + str(iteration))

    iterations.append(iteration)


print("\nComenzando con vector de números aleatorios")
print('Valor medio de iteraciones necesario para converger: {}'\
      .format(np.mean(np.asarray(iterations))))
```

```
Ejercicio 2.1.a

Comenzando con vector de ceros
Valor de iteraciones necesarias para converger: 2200

-----------

Comenzando con vector de números aleatorios

        Iteración: 3200
        Iteración: 3050
        Iteración: 1750
        Iteración: 4300
        Iteración: 1750
        Iteración: 1900
        Iteración: 3450
        Iteración: 2350
        Iteración: 3050
        Iteración: 2350

Valor medio de iteraciones necesario para converger: 2715.0
```

Empezando con números aleatorios la cantidad de iteraciones media sobrepasa la de un vector de ceros, pero la diferencia no es tan grande. Si miramos las llamadas a la función una por una, vemos que con un vector inicial mal elegido llegamos al doble que con ceros. Aun así, para este número de datos el algoritmo converge de forma rápida.



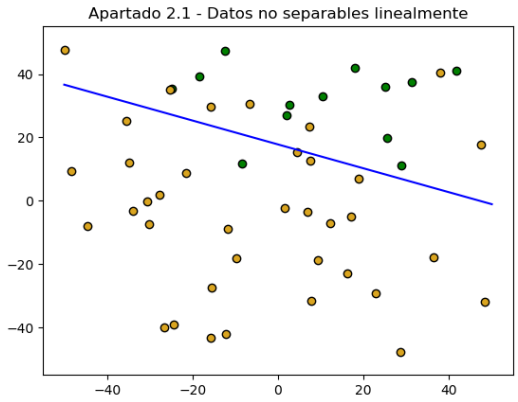
Comenzando con un vector de ceros

```python
# Ahora con los datos del ejercicio 1.2.b ------------
print("Ejercicio 2.1.b\n")

# Añadir columna de unos
c0 = np.full(len(data2b),1)
puntos = np.copy(data2b)
data2b = np.column_stack((c0, data2b))

# Con vector 0 -------------------
max_iter = 30000
# Creamos array con tantas columnas como las de la matriz de datos
vini = np.zeros(np.size(data2b,1))
w_cero, iteration = ajusta_PLA(data2b, labels2b, max_iter, vini)

print("Comenzando con vector de ceros")
print('Valor de iteraciones necesarias para converger: ' + str(iteration))
print("\n-----------\n")

# Random initializations----------
iterations = []
for i in range(0,10):
    vini = np.random.uniform(low=0, high=1, size=np.size(data2b,1))

    w_rand, iteration = ajusta_PLA(data2b, labels2b, max_iter, vini)

    iterations.append(iteration)


print("\nComenzando con vector de números aleatorios")
print('Valor medio de iteraciones necesario para converger: {}'\
      .format(np.mean(np.asarray(iterations))))
```

```
Ejercicio 2.1.b

Comenzando con vector de ceros
Valor de iteraciones necesarias para converger: 30000

-----------

Comenzando con vector de números aleatorios

Valor medio de iteraciones necesario para converger: 30000.0
```

Aquí vemos que no converge, esto tiene sentido puesto que el error que hemos introducido en las etiquetas ha hecho el conjunto linealmente no separable, y PLA no es un algoritmo capaz de clasificar perfectamente con este tipo de datos.

Un algoritmo derivado que nos podría servir sería el Pocket Algorithm, donde lanzaríamos el Perceptron con un número máximo de iteraciones y nos quedaríamos con la mejor iteración (aquella que tenga los valores de w que hagan mínimo el error).



Comenzando con un vector de ceros

Apartado 2 – Regresión Logística

```python
# Cálculo del gradiente, derivada de Ein
def dEin(x, y, w):
    N = len(x)
    yx = np.dot(y, x)           # yn * xn
    den = 1 + (np.e)**(np.dot(yx, w.transpose()))
    sumatoria = np.dot(yx, (1/den))

    return np.dot(-(1/N), sumatoria)
```

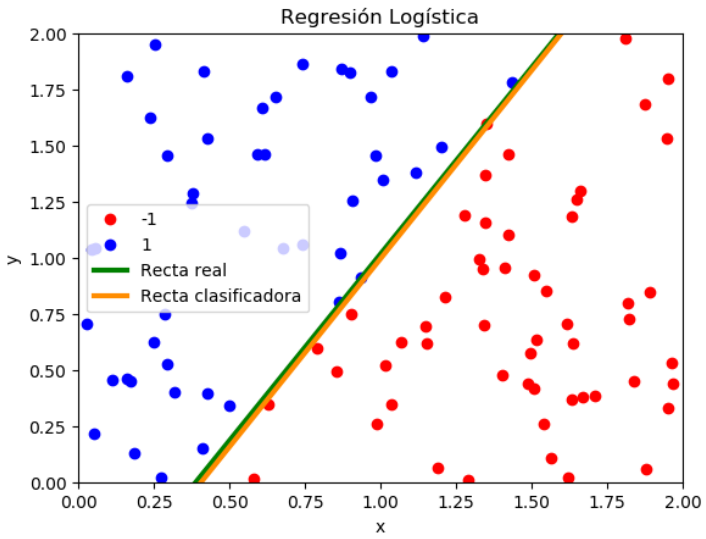
```python
def Ein(x, y, w):
    N = len(x)
    ein = 0
    for (a,b) in zip(x,y):
      yx = np.dot(a,b)
      ein += np.log(1 + np.e**(-np.dot(yx, w.transpose())))

    return ein / N
```

```python
def sgdRL(data, labels, vini, eta, epochs):
    # Variables auxiliares
    w = vini.copy()
    w_old = w.copy()
    minibatch_size = 1

    for i in range(epochs):
        # Ordenamos aleatoriamente las muestras
        fullbatch_x, fullbatch_y = shuffle(data, labels, random_state=0)

        # Separamos en minibatches
        batches_x = np.array_split(fullbatch_x, np.ceil(len(data) / minibatch_size))
        batches_y = np.array_split(fullbatch_y, np.ceil(len(data) / minibatch_size))

        for (minibatch_x, minibatch_y) in zip(batches_x, batches_y):
            error = dEin(minibatch_x, minibatch_y, w)
            gradient = eta * error

            w = w - gradient


        resta = w - w_old
        surpassedMinError = True if np.all(np.abs(resta)) < 0.01 else False

        # Devolvemos si hemos alcanzado el error
        if surpassedMinError:
            return w

        w_old = w.copy()

    return w
```

```python
# Generamos los puntos
data = simula_unif(100, 2, [0,2])

# Seleccionamos dos aleatoriamente
p = np.random.randint(0, len(data), 2)

x1 = data[p[0]][0]
x2 = data[p[1]][0]
y1 = data[p[0]][1]
y2 = data[p[1]][1]

# Calculamos la recta
a = (y2-y1)/(x2-x1) # Calculo de la pendiente.
b = y1 - a*x1       # Calculo del termino independiente.

# Etiquetamos
labels = np.array([[0,0]])
labels = np.delete(labels, (0), axis=0)

for x in data:
  labels = np.append(labels, f(x[0], x[1], a, b))

truelabels = labels.copy()

# Añadir columna de unos
c0 = np.full(len(data),1)
data = np.column_stack((c0, data))

# Lanzamos el algoritmo
vini = np.ones(np.size(data,1))
eta = 0.01
epochs = 1000

w = sgdRL(data, labels, vini, eta, epochs)

# Usar la muestra de datos etiquetada para encontrar
#nuestra solución g y estimar Eout usando para ello
# un número suficientemente grande de nuevas muestras.

# Generamos conjunto de test
test = simula_unif(100, 2, [0,2])

# Añadir columna de unos a test
c0 = np.full(len(test),1)
test = np.column_stack((c0, test))

# Calculamos el error fuera de la muestra
misclassified = 0

for (l,d) in zip(truelabels, data):
  v = signo(np.dot(d, w.transpose()))
  if l != v:
     misclassified += 1

misclassified /= len(data)
print("Pesos de g: " + str(w))
print("Porcentaje Eout: " + str(misclassified))
```

Ejecutando lo anterior obtenemos:

```
Pesos de g: [  4.51636405 -11.15989558   6.65631566]
Porcentaje Eout: 0.01
```



El algoritmo no llega a encontrar el clasificador perfecto, puesto que la condición de salida hace que acabe antes. Aun así, solo falla en un punto.

3. Bonus

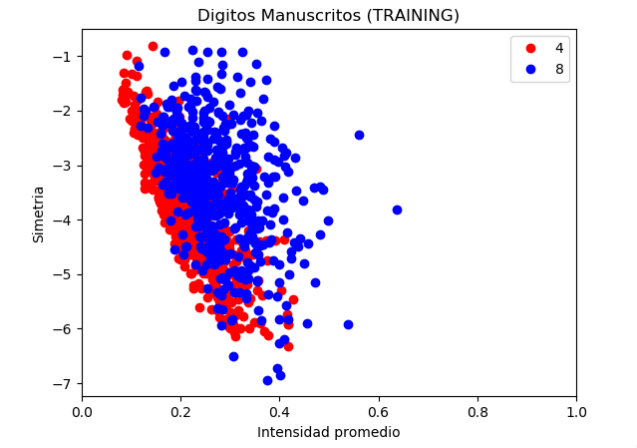
Apartado 1

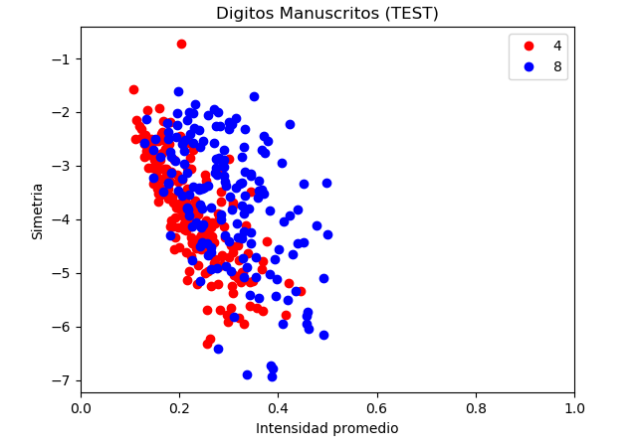
Problema: Identificar un dígito a partir de los datos de entrada
$\boldsymbol{X}$ = Características de los dígitos {intensidad promedio, simetría}
$\boldsymbol{Y}$ = Dígitos (4 y 8), se pueden representar con etiquetas {-1,1}.

Aplicando uno de los algoritmos vistos se obtendrá una hipótesis g (en forma de vector de pesos) que nos permitirá clasificar nuevos datos.

Apartado 2

Tenemos los datos:





Utilizando **Regresión** mediante SGD:

```
def Ein(x, y, w):
    N = len(x)
    ein = 0
    for (a,b) in zip(x,y):
      yx = np.dot(a,b)
      ein += np.log(1 + np.e**(-np.dot(yx, w.transpose())))

    return ein / N
```

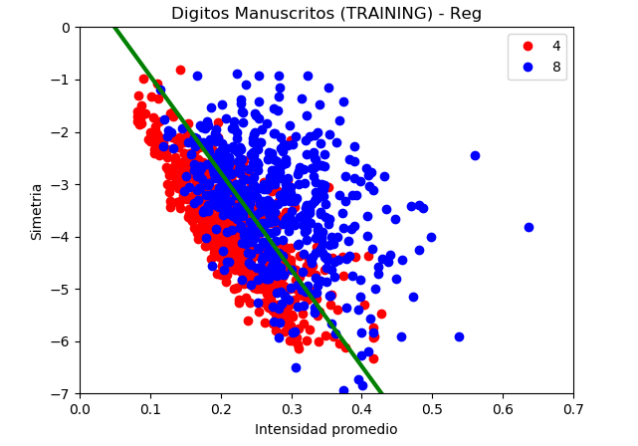
```
# Cálculo del gradiente, derivada de Ein
def dEin(x, y, w):
    N = len(x)
    yx = np.dot(y, x)            # yn * xn
    den = 1 + (np.e)**(np.dot(yx, w.transpose()))
    sumatoria = np.dot(yx, (1/den))

    return np.dot(-(1/N), sumatoria)
```

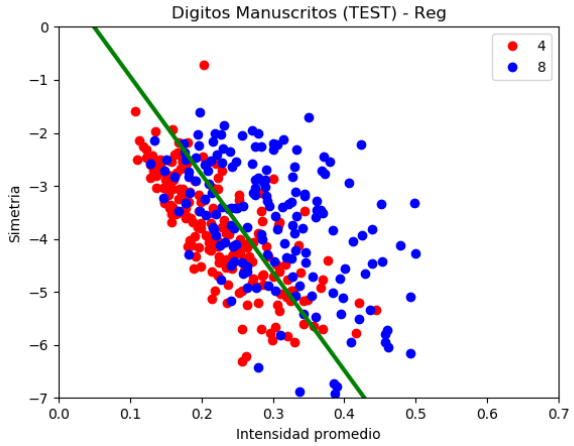
```
def sgdRL(data, labels, vini, eta, epochs):
    # Variables auxiliares
    w = vini.copy()
    w_old = w.copy()
    minibatch_size = 1

    for i in range(epochs):
        # Ordenamos aleatoriamente las muestras
        fullbatch_x, fullbatch_y = shuffle(data, labels, random_state=0)

        # Separamos en minibatches
        batches_x = np.array_split(fullbatch_x, np.ceil(len(data) / minibatch_size))
        batches_y = np.array_split(fullbatch_y, np.ceil(len(data) / minibatch_size))

        for (minibatch_x, minibatch_y) in zip(batches_x, batches_y):
            error = dEin(minibatch_x, minibatch_y, w)
            gradient = eta * error

            w = w - gradient


        resta = w - w_old
        surpassedMinError = True if np.all(np.abs(resta)) < 0.01 else False

        # Devolvemos si hemos alcanzado el error
        if surpassedMinError:
            return w

        w_old = w.copy()

    return w
```

```python
# Con Regresión (SGD) --------------------
vini = np.zeros(np.size(x,1))
eta = 0.01
epochs = 500

w = sgdRL(x, y, vini, eta, epochs)
w_reg = w.copy()


#CALCULO DE LOS ERRORES ---
ein = Err(x, y, w)
print("Regresión - Porcentaje Ein:\t" + str(ein))
etest = Err(x_test, y_test, w)
print("Regresión - Porcentaje Etest:\t" + str(etest))

ein = Ein(x, y, w)
print("Regresión - Valor Ein:\t" + str(ein))

etest = Ein(x_test, y_test, w)
print("Regresión - Valor Etest:\t" + str(etest))

#COTA SOBRE EL ERROR - VC Generalization Bound
tolerance = 0.05

# Sobre Ein
N = len(x)

# Break Point = 4, por tanto:
dvc = 2**(4-1)

a = (8/N) * np.log( (4*( ((2*N)**dvc) + 1)) / tolerance )
sqr = np.sqrt(a)
eout = ein + sqr

print("Regresión - Cota Eout sobre Ein:\t" + str(eout))

# Sobre Etest
eout = etest + sqr
print("Regresión - Cota Eout sobre Etest:\t" + str(eout))
```

Obteniendo:

```
Regresión - Porcentaje Ein:     0.22194304857621341
Regresión - Porcentaje Etest:   0.2595628415300549
Regresión - Valor Ein:  0.46429792180051077
Regresión - Valor Etest:        0.5263410501195098
```





Y para las cotas, aplicamos:

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

```
Regresión - Cota Eout sobre Ein:        1.1323419851361178
Regresión - Cota Eout sobre Etest:      1.1943851134551169
```

Utilizando **Pocket-Algorithm** inicializado con la salida de **SGD**:

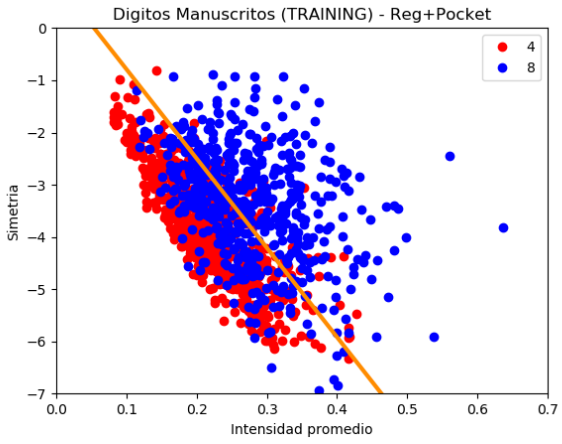
```python
# Media de símbolos mal clasificados
def Err(data, label, w):
  N = len(data)
  error = 0

  for (x,y) in zip(data, label):
    sign = signo(np.dot(w, x))
    if sign != signo(y):
      error += (1/N)

  return error
```

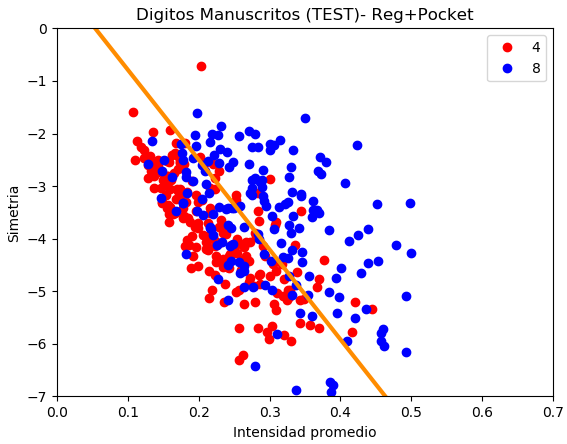
```python
#POCKET ALGORITHM
def pocket(datos, label, max_iter, vini):
    # Variables auxiliares
    w = vini.copy()
    pocket = w.copy() # La mejor de todas
    pocket_error = Err(datos, label, w)

    for i in range(max_iter):
      w, it = ajusta_PLA(datos, label, 1, w)

      error = Err(datos, label, w)

      if error < pocket_error:
        pocket = w.copy()
        pocket_error = error

    return pocket, pocket_error
```

```python
# Con Pocket a partir de Regresión --------------------
w = w_reg.copy()
max_iter = 1000
vini = np.ones(np.size(x,1))

w, ein = pocket(x, y, max_iter, vini)

#CALCULO DE LOS ERRORES
print("Reg+Pocket - Porcentaje Ein:\t" + str(ein))
etest = Err(x_test, y_test, w)
print("Reg+Pocket- Porcentaje Etest:\t" + str(etest))

ein = Ein(x, y, w)
print("Reg+Pocket - Valor Ein:\t" + str(ein))

etest = Ein(x_test, y_test, w)
print("Reg+Pocket - Valor Etest:\t" + str(etest))


#COTA SOBRE EL ERROR - VC Generalization Bound
tolerance = 0.05

# Sobre Ein
N = len(x)

# El Perceptron tiene Break Point = 4, por tanto:
dvc = 2**(4-1)

a = (8/N) * np.log( (4*( ((2*N)**dvc) + 1)) / tolerance )
sqr = np.sqrt(a)
eout = ein + sqr

print("Reg+Pocket - Cota Eout sobre Ein:\t" + str(eout))

# Sobre Etest
eout = etest + sqr
print("Reg+Pocket - Cota Eout sobre Etest:\t" + str(eout))
```

Tenemos:

```
Reg+Pocket - Porcentaje Ein:    0.23031825795644786
Reg+Pocket- Porcentaje Etest:   0.25136612021857946
Reg+Pocket - Valor Ein: 1.4031942058551794
Reg+Pocket - Valor Etest:       1.708555094410384
```





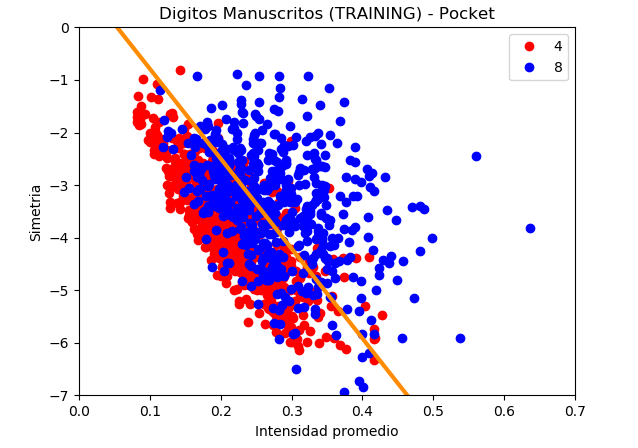
Y de cotas:

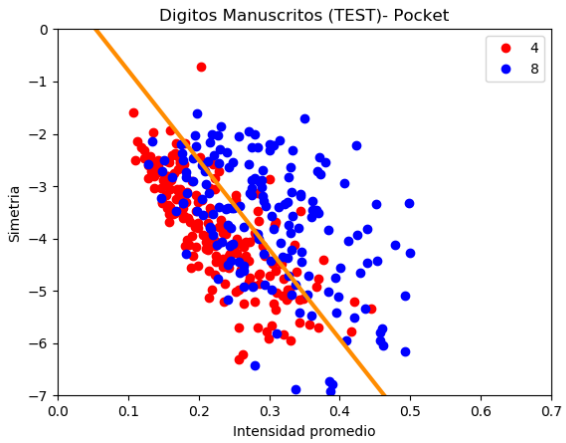
```
Reg+Pocket - Cota Eout sobre Ein:       2.0712382691907862
Reg+Pocket - Cota Eout sobre Etest:     2.376599157745991
```

Utilizando **Pocket-Algorithm** inicializado con un vector de unos:

```
# Con Pocket ---------------------------------------------------
# Llamando a Pocket
max_iter = 1000
vini = np.ones(np.size(x,1))

w, ein = pocket(x, y, max_iter, vini)

#CALCULO DE LOS ERRORES
print("Pocket - Ein:\t" + str(ein))

etest = Err(x_test, y_test, w)
print("Pocket - Etest:\t" + str(etest))


#COTA SOBRE EL ERROR - VC Generalization Bound
tolerance = 0.05

# Sobre Ein
N = len(x)

# El Perceptron tiene Break Point = 4, por tanto:
dvc = 2**(4-1)

a = (8/N) * np.log( (4*( ((2*N)**dvc) + 1)) / tolerance )
sqr = np.sqrt(a)
eout = ein + sqr

print("Pocket - Cota Eout sobre Ein:\t" + str(eout))

# Sobre Etest
eout = etest + sqr
print("Pocket - Cota Eout sobre Etest:\t" + str(eout))
```

```
Pocket - Porcentaje Ein:        0.23031825795644786
Pocket- Porcentaje Etest:       0.25136612021857946
Pocket - Valor Ein:     1.4031942058551794
Pocket - Valor Etest:   1.708555094410384
```





Y cotas:

```
Pocket - Cota Eout sobre Ein:   0.898362321292055
Pocket - Cota Eout sobre Etest: 0.9194101835541866
```

Análisis de resultados

Lo primero que vemos es que independientemente de cómo inicialicemos Pocket-Algorithm obtenemos los mismos resultados.

Con todos los algoritmos se ha conseguido un porcentaje de fallos al clasificar relativamente bajo, siendo un poco mejor los de Regresión dentro de la muestra, y Pocket con el conjunto de test. Con las gráficas hemos visto que estos clasificadores no permiten un gran margen de mejora si utilizamos separadores lineales, ya que trabajamos con un conjunto de datos no lineal.

Respecto a las cotas, con los tres métodos se han conseguido unas que son bastante altas al compararse con el error real, además de tener una diferencia bastante pequeña entre E_{out} y E_{test}.

Este cálculo de la cota considera todas las posibles hipótesis g en H, por tanto, no es de extrañar que se sobreestime de gran manera el valor de E_{out}.

```
Regresión - Porcentaje Ein:     0.22194304857621341
Regresión - Porcentaje Etest:   0.2595628415300549
Regresión - Valor Ein:  0.46429792180051077
Regresión - Valor Etest:        0.5263410501195098
```

```
Reg+Pocket - Porcentaje Ein:    0.23031825795644786
Reg+Pocket- Porcentaje Etest:   0.25136612021857946
Reg+Pocket - Valor Ein: 1.4031942058551794
Reg+Pocket - Valor Etest:       1.708555094410384
```

```
Pocket - Porcentaje Ein:        0.23031825795644786
Pocket- Porcentaje Etest:       0.25136612021857946
Pocket - Valor Ein:     1.4031942058551794
Pocket - Valor Etest:   1.708555094410384
```

```
Regresión - Cota Eout sobre Ein:        1.1323419851361178
Regresión - Cota Eout sobre Etest:      1.1943851134551169
```

```
Reg+Pocket - Cota Eout sobre Ein:       2.0712382691907862
Reg+Pocket - Cota Eout sobre Etest:     2.376599157745991
```

```
Pocket - Cota Eout sobre Ein:   2.0712382691907862
Pocket - Cota Eout sobre Etest: 2.376599157745991
```