

Práctica 2: clasificación de dígitos MNIST mediante AdaBoost

Introducción

En esta práctica se ha desarrollado un sistema para distinguir dígitos manuscritos del 0 al 9 en imágenes. El sistema implementado es de aprendizaje supervisado con AdaBoost. El objetivo de la práctica es entrenar al sistema, mediante un conjunto de imágenes de prueba, para clasificar imágenes en estas clases (dígitos del 0 al 9) aunque no hayan sido previamente analizadas por el sistema.



El banco de imágenes utilizado para las pruebas es la base de datos MNIST

Lo primero será importar las librerías necesarias

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
import random, math
```

```
2022-06-26 11:30:32.998175: W tensorflow/stream_executor/platform/default/dso_loader.c
c:64] Could not load dynamic library 'libcudart.so.11.0'; dLError: libcudart.so.11.0: c
annot open shared object file: No such file or directory
2022-06-26 11:30:32.998195: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore
above cudart dLError if you do not have a GPU set up on your machine.
```

Desde la librería de Keras podemos descargar la base de datos MNIST

```
In [ ]: (x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train.shape, y_train.shape
```

```
Out[ ]: ((60000, 28, 28), (60000,))
```

Podemos guardar la base de datos en un fichero con el siguiente comando

```
In [ ]: np.savez("mnist", x=x_train, y=y_train)
```

La cargamos para utilizarla

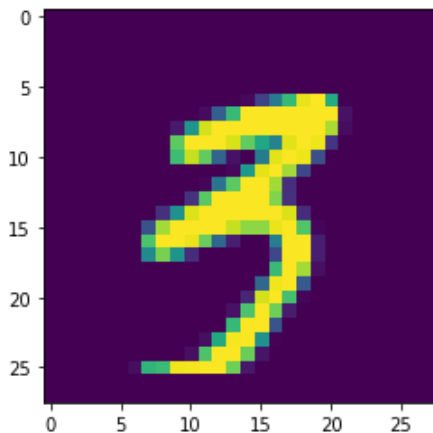
```
In [ ]: npzfile = np.load("mnist.npz")
mnist_X = npzfile['x']
```

```
mnist_Y = npzfile['y']
mnist_X.shape
```

Out[]: (60000, 28, 28)

Veamos alguna de las imagenes del banco de datos

```
In [ ]: plt.figure()
plt.imshow(mnist_X[1007])
plt.show()
print("Etiqueta: " + str(mnist_Y[1007]))
```



Etiqueta: 3

Implementación AdaBoost

Adaptación de los arrays de imágenes y etiquetas para AdaBoost en función de la clase para la que se va a entrenar

```
In [ ]: DIV = 10000
TEST = 20000

def adaptar_conjuntos(mnist_X, mnist_Y, clase):
    X = mnist_X[:DIV].reshape(DIV, 784)

    Y = []
    for i in range(len(mnist_Y[:DIV])):
        if mnist_Y[i] == clase:
            Y.append(1)
        else:
            Y.append(-1)

    return (X, Y)

def adaptar_conjuntos_test(mnist_X, mnist_Y):
    X = mnist_X.copy()
    X = np.reshape(X[DIV:TEST], (TEST - DIV, 784))

    return (X, mnist_Y)
```

Funciones necesarias para los clasificadores débiles

```
In [ ]: def generar_clasificador_debil(dimension_datos):
    return (random.randint(0, dimension_datos-1), random.randint(0, 255), random.choice
```

```
In [ ]: def aplicar_clasificador_debil(clasificador, imagen):
    v = imagen[:,clasificador[0]]
    return ((v >= clasificador[1]) * 1 + (v < clasificador[1]) * -1) * clasificador[2]
```

```
In [ ]: def obtener_error(clasificador, X, Y, D):
        return D * (aplicar_clasificador_debil(clasificador, X) != Y)
```

Función de entrenamiento adaboost que nos permite obtener un conjunto de clasificadores fuertes y sus respectivos valores α .

```
In [ ]: def adaboost(X, Y, T, A):
        clasificadores_debiles = []
        alphas = []
        D = np.full(len(X), 1.0/len(X))

        for t in range(T):
            conjuntos = []
            for k in range(A):
                c_d = generar_clasificador_debil(28*28)
                clas = aplicar_clasificador_debil(c_d, X)
                eps = np.sum(obtener_error(c_d, X, Y, D))
                conjuntos.append((c_d, eps, clas))

            pri = True
            for elem in conjuntos:
                if pri:
                    fc = elem
                    pri = False
                    continue
                if elem[1] < fc[1]:
                    fc = elem
                    # Guardamos el conjunto con menor error

            alpha = 0.5*math.log2((1-fc[1])/fc[1])
            alphas.append(alpha)
            clasificadores_debiles.append(fc[0])

        Z = np.sum(D)
        for i in range(len(D)):
            D[i] = (D[i]*(math.e**(-alpha*Y[i] * fc[2][i])))/Z

        return (clasificadores_debiles, alphas)
```

Función para aplicar nuestro clasificador fuerte una vez se ha entrenado. Esta función nos permite obtener, para una imagen, el valor que tiene en una clase, según los clasificadores debiles de esta clase ponderados por su α .

```
In [ ]: def aplicar_clasificador_fuerte(clasificadores, alphas, imagen):
        clasificacion = 0
        for i in range(len(clasificadores)):
            clasificacion += alphas[i] * aplicar_clasificador_debil(clasificadores[i], np.a

        return clasificacion
```

Además se ha implementado una función que entrena el sistema para las 10 clases haciendo uso de la función adaboost

```
In [ ]: def entrenar_sistema(mnist_X, mnist_Y, T, A):
        clasificadores = []
        alphas = []
        for clase in range(10):
            (trainX, trainY) = adaptar_conjuntos(mnist_X, mnist_Y, clase)
            (cl, al) = adaboost(trainX, trainY, T, A)
            clasificadores.append(cl)
            alphas.append(al)
        return (clasificadores, alphas)
```

Se ha implementado la siguiente función que permite probar para un set de imagenes, el índice de acierto que tiene el sistema para una clase concreta.

```
In [ ]: def testn(X, Y, clasificadores, alphas, n):
        correct = total = 0
```

```

for i in range(len(X)):
    if (Y[i] != n):
        continue
    total += 1
    clasif = (-1, 0)
    for j in range(10):
        punt = (aplicar_clasificador_fuerte(clasificadores[j], alphas[j], X[i]))
        if punt > clasif[1]:
            clasif = (j, punt)
    if n == clasif[0]:
        correct += 1

return correct/total

```

Veamos una prueba para la clase '6'.

```

In [ ]: T = 10
        A = 20
        print("Proceso de entrenamiento")
        (clasificadores, alphas) = entrenar_sistema(mnist_X, mnist_Y, T, A)
        (testX, testY) = adaptar_conjuntos_test(mnist_X, mnist_Y)
        print("Proceso de pruebas")
        print(f"Porcentaje de aciertos del sistema para la clase 6: {np.average(testn(testX, te

```

Proceso de entrenamiento
 Proceso de pruebas
 Porcentaje de aciertos del sistema para la clase 6: 9.368836291913215%

La función testn utilizada previamente, se puede generalizar para el conjunto de clases de la siguiente forma, de modo que permite calcular el índice de acierto que tiene el sistema en su conjunto.

```

In [ ]: def test(X, Y, clasificadores, alphas):
        correct = np.full(10, 0)
        total = np.full(10, 0)

        for i in range(len(X)):
            total[Y[i]] += 1
            clasif = (-1, 0)
            for j in range(10):
                punt = (aplicar_clasificador_fuerte(clasificadores[j], alphas[j], X[i]))
                if punt > clasif[1]:
                    clasif = (j, punt)
            if Y[i] == clasif[0]:
                correct[clasif[0]] += 1

        return correct/total

```

Vamos a realizar una prueba con valores de A y T pequeños (los definidos para el test anterior), que permita ver el funcionamiento

```

In [ ]: print("Proceso de entrenamiento")
        (uue, cu) = adaptar_conjuntos(mnist_X, mnist_Y, 3)
        (clasificadores, alphas) = entrenar_sistema(mnist_X, mnist_Y, T, A)
        (testX, testY) = adaptar_conjuntos_test(mnist_X, mnist_Y)
        print("Proceso de pruebas")
        print(f"Porcentaje de aciertos del sistema: {np.average(test(testX, testY, clasificador

```

Proceso de entrenamiento
 Proceso de pruebas
 Porcentaje de aciertos del sistema: 7.470834226498072%

Para realizar un análisis que ayude a determinar los valores adecuados para T y A se ha utilizado la siguiente función

```

In [ ]: def valorarAT(X, Y, A, T, At, Ta):
        resultsA = []

```

```

for i in At:
    (clasificadores, alphas) = entrenar_sistema(X, Y, T, i)
    (testX, testY) = adaptar_conjuntos_entr_test(X, Y)
    resultsA.append(test(testX, testY, clasificadores, alphas))

resultsT = []
for i in Ta:
    (clasificadores, alphas) = entrenar_sistema(X, Y, i, A)
    (testX, testY) = adaptar_conjuntos_entr_test(X, Y)
    resultsT.append(test(testX, testY, clasificadores, alphas))
return (resultsA, resultsT)

```

En esta función se entrena el sistema en función de los valores de A_t con T constante, y se obtienen los resultados para cada uno de estos entrenamientos. Después se hace lo propio pero para cada valor de T_a con A constante.

Cuestiones sobre la práctica

Explica brevemente cómo has adaptado la base datos MNIST al algoritmo AdaBoost

Para adaptar los elementos de la base de datos de MNIST al algoritmo de entrenamiento, se ha utilizado la función `adaptar_conjuntos` formulada previamente. Esta función consiste en darle una nueva forma a cada imagen. Por defecto las imágenes de MNIST son de 28x28 píxeles, pero en esta función se reestructura de forma que sean un único vector con los 784 píxeles seguidos, de forma que sea más sencillo de acceder a cada píxel individual. También se han editado las etiquetas, poniendo los valores 1, si es la clase a entrenar, y -1 en caso contrario. Por este motivo, para cada clase a entrenar, se debe adaptar de nuevo los datos. Además, del conjunto total de imágenes se han seleccionado para entrenar, las primeras 20000.

Por otro lado, se han tenido que adaptar también estos datos al algoritmo de prueba, con la función `adaptar_conjuntos_test`. En este caso, para adaptar las imágenes, se hace del mismo modo. La diferencia recae en las etiquetas, las cuales se quedan tal cual está en la base de datos. Para el conjunto de datos de testeo se han utilizado las siguientes 10000 imágenes, esto es, de la 20001 a la 30000.

Comenta detalladamente el funcionamiento de AdaBoost teniendo en cuenta que tasa media de fallos obtienes para aprendizaje y test. Correlaciona los porcentajes de acierto y de fallo con los valores de A y de T .

AdaBoost es un algoritmo mediante el cual obtener clasificadores fuertes a través de un entrenamiento. Cuenta con dos parámetros importantes, A y T . A define el número de clasificadores aleatorios que vamos a obtener para probar y obtener un clasificador débil. El parámetro T por otro lado, indica el número de clasificadores débiles que compondrán nuestro clasificador fuerte.

Con la función `valorarAT` definida previamente, se tratará de ver que papel cumplen A y T en la precisión del sistema. Para ello lanzaremos la siguiente prueba:

In []:

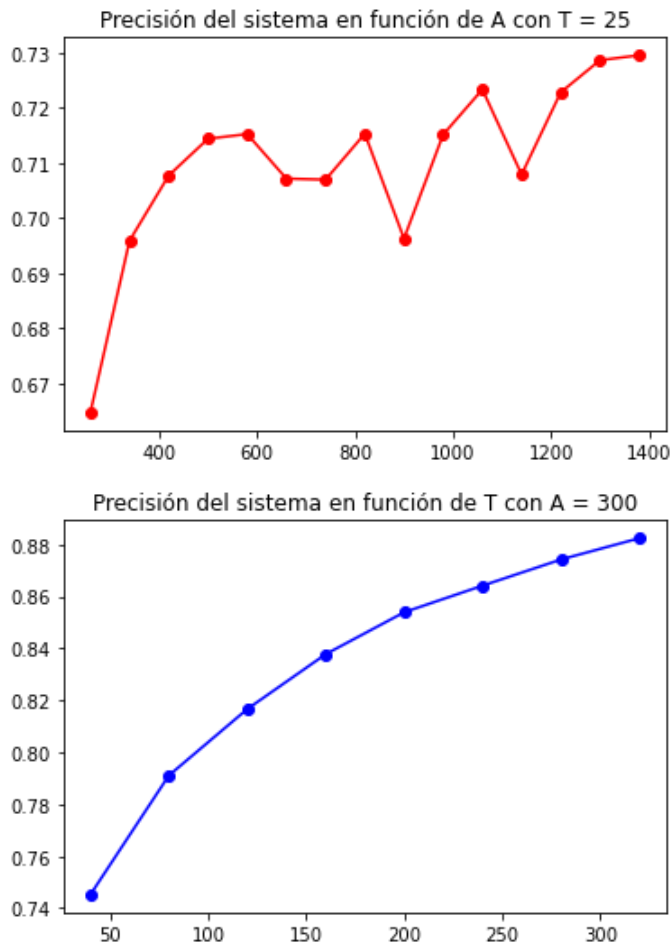
```

T = 25
A = 300
At = range(260, 1381, 80)
Ta = range(40, 321, 40)

(rA, rT) = valorarAT(mnist_X, mnist_Y, A, T, At, Ta)
rAcorr = [np.average(i) for i in rA]
rTcorr = [np.average(i) for i in rT]

```

```
plt.figure()
plt.plot(At, rAcorr, 'r-o')
plt.title(f"Precisión del sistema en función de A con T = {T}")
plt.show()
plt.figure()
plt.plot(Ta, rTcorr, 'b-o')
plt.title(f"Precisión del sistema en función de T con A = {A}")
plt.show()
```



Estas gráficas muestran detalles importantes de Adaboost.

El primero de ellos es que existe una relación lineal entre el valor de A y T con la precisión del sistema, esto es, aumentar el valor de A o de T, aumentará el índice de aciertos del sistema. Aún así, vemos que esta relación no es estricta, es decir, vemos que en algunas pruebas con valores de A mayores, obtenemos un acierto menor. Esto se debe a que el modo de obtener clasificadores es aleatorio, y aún siendo cierto que a mayor número de clasificadores probados, aumentará la fiabilidad, no es seguro que el clasificador final sea mejor que uno obtenido con menor valor de A. Respecto a T vemos que la precisión parece estancarse en cierto punto.

El segundo detalle es la influencia de A y T en la gráfica de precisión. El valor de A es el que da estabilidad a la gráfica para diferentes valores de T, pues al aumentar A se reduce la aleatoriedad del clasificador y viceversa. Por otro lado, el valor de T desplaza la curva para diferentes valores de A verticalmente, esto es, a mayor valor de T, veríamos unos mayores porcentajes de acierto y a la inversa.

¿Cómo afecta el número de clasificadores generados al tiempo empleado para el proceso de aprendizaje? ¿Qué importancia le darías? Justifica tu respuesta.

El tiempo de aprendizaje del algoritmo depende linealmente de básicamente tres variables, que son, el número de clasificadores débiles a obtener (T), los clasificadores débiles aleatorios por cada iteración (A) y

la cantidad de imágenes utilizadas (X). No modificando A ni X , el tiempo de aprendizaje dependerá linealmente de T , esto es, aumentar el valor de T en una proporción hará aumentar el tiempo destinado al aprendizaje en esta misma proporción.

A mayor número de clasificadores débiles entrenemos en nuestro algoritmo, obtendremos unos resultados mejores, esto es un porcentaje de acierto mayor. A pesar de ser cierto esto, el incremento de precisión del sistema no es lineal con el incremento de T , sino que más bien describe una función $\log(T)$. Es por esto que no se puede escoger un número muy elevado para T , pues mientras el tiempo de entrenamiento crece linealmente con T , la precisión acaba estancandose.

Resumiendo, debe escogerse un número de T apropiado al problema y a la precisión necesaria en este caso, teniendo en cuenta el tiempo disponible para el entrenamiento, y para obtener este valor de T , lo apropiado es realizar numerosas pruebas en las que se prueben diversos valores para escoger de entre estos el que consideremos óptimo.

¿Cómo has dividido los datos en conjunto de entrenamiento y test? ¿Para qué es útil hacer esta división?

Los diferentes conjuntos se han dividido a través de las dos funciones diferentes de adaptar conjuntos. En estas funciones se han definido unos valores `DIV` y `TEST` que permiten dividirlos. Para el conjunto de entrenamiento se han utilizado las primeras `DIV` imágenes de `mnist`, y para el test se han utilizado desde la imagen `DIV` hasta la imagen `TEST`.

No solo es útil hacer esta división, es necesario para hacer un test riguroso del sistema. Si utilizáramos las imágenes de entrenamiento en el test, obtendríamos resultados muy elevados de precisión, pues es con estas mismas imágenes con las que el sistema ha aprendido a diferenciar. Al utilizar otras imágenes, la fiabilidad del sistema bajará mucho, y es por esto que en ese se deberá utilizar mayores valores de A y T .

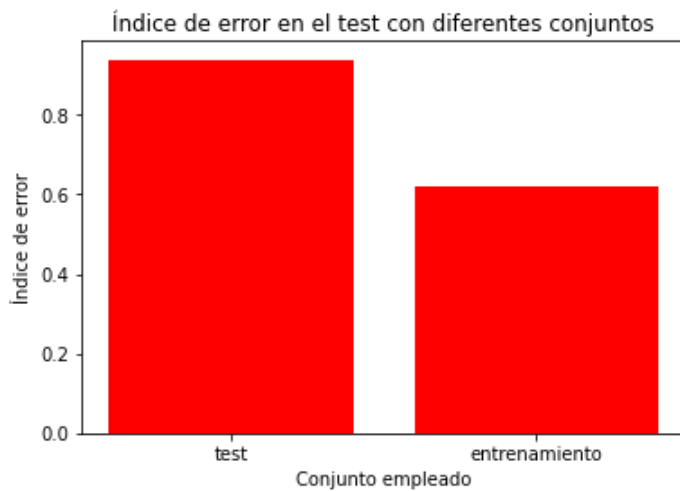
¿Has observado si se produce sobreentrenamiento? Justifica tu respuesta con una gráfica en la que se compare el error de entrenamiento y el de test a lo largo de las ejecuciones.

Ejecutar el testeo con el conjunto de entrenamiento conlleva un error reducido en comparación a usar el conjunto de test. Esto se debe a que a la hora de obtener los clasificadores se están utilizando las imágenes del conjunto de entrenamiento, y por tanto los clasificadores estarán optimizados para estas imágenes.

Veamos a continuación la diferencia a la hora de testear entre conjunto de test y de entrenamiento:

```
In [ ]: def adaptar_conjuntos_entr_test(mnist_X, mnist_Y):
        X = mnist_X.copy()
        X = np.reshape(X[:DIV], (DIV, 784))
        return (X, mnist_Y)

(x_ent, y_ent) = adaptar_conjuntos_entr_test(mnist_X, mnist_Y)
(x_tes, y_tes) = adaptar_conjuntos_test(mnist_X, mnist_Y)
plt.bar([0, 1], [1 - np.average(test(x_tes, y_tes, clasificadores, alphas)), 1 - np.ave
plt.title('Índice de error en el test con diferentes conjuntos')
plt.ylabel('Índice de error')
plt.xlabel('Conjunto empleado')
plt.xticks([0, 1], ('test', 'entrenamiento'))
plt.show()
```



Como se puede apreciar en el anterior gráfico, al utilizar el conjunto de entrenamiento se ha reducido en buena medida el error del sistema. Los índices de error son elevados en ambos casos pues se ha utilizado los clasificadores con $T = 10$ y $A = 20$.

¿Cómo has conseguido que Adaboost clasifique entre los 10 dígitos cuando solo tiene una salida binaria?

Al utilizar adaboost se obtiene un conjunto de clasificadores débiles para una clase con salida binaria. Estos clasificadores débiles componen un clasificador fuerte, que nos permite obtener un valor al ponderar cada uno de estos clasificadores débiles con su valor α .

A partir de esta idea, si se lanza el algoritmo Adaboost para cada una de las clases, obtendremos un clasificador fuerte para cada clase. A partir de este 'sistema' de clasificadores, es posible, para una imagen, al aplicar cada uno de estos clasificadores fuertes, que devolverán valores numéricos, determinar a qué clase pertenece la imagen con el valor del clasificador más alto. Por ejemplo, si para una imagen cualquiera se aplican los clasificadores del sistema, y obtenemos los valores para la clase 0 = 2.45 y para la clase 7 = 3.12, el sistema dirá que esta imagen contiene un 7.

Ampliación de Julio

Se ha implementado el algoritmo AdaBoost pero en esta ocasión, los clasificadores débiles serán de tipo hiperplano y no umbral. Lo que se pretende con el hiperplano es dividir el espacio en dos partes y especificar que los objetos que quedan a un lado se van a clasificar según una clase y los objetos que quedan al otro lado según la otra clase. Decimos que en un espacio de dimensión D , el hiperplano es el objeto geométrico plano que divide ese espacio en dos. Así, el hiperplano en $D=2$ es una recta, en $D=3$ es un plano, etc. De manera general, definimos un hiperplano en dimensión D como los puntos ($P = \{p_1, p_2, \dots, p_d\}$) del espacio D -dimensional que cumplen:

$$h^D : \sum_{i=1}^D (x_i p_i) - C = 0$$

De este modo, todos los puntos del espacio D que al ser sustituidos en la ecuación anterior nos devuelven un valor negativo estarán al lado negativo del espacio, lo cual se considera no pertenece a la clase, según el hiperplano y, al contrario, los que nos devuelven un valor positivo están al otro lado del espacio y por tanto pertenecerán a la clase. El hiperplano contará con 784 dimensiones, una por píxel de la imagen.

Lo primero es definir las funciones básicas de los clasificadores débiles:

```
In [ ]: def generar_clasificador_debil_hiperplano(dimension_datos):
        ran = [random.randint(0, 255) for _ in range(dimension_datos)]
```



```

vec = [random.uniform(-1,1) for _ in range(dimension_datos)]
denom = 0
for i in range(len(vec)):
    denom += vec[i]**2
normalv = vec / np.sqrt(denom)
C = 0
for i in range(dimension_datos):
    C += ran[i] * normalv[i]
return (normalv, C)

```

```

In [ ]: def aplicar_clasificador_debil_hiperplano(clasificador, imagen):
        ret = []
        for i in range(len(imagen)):
            res = 0
            for j in range(len(clasificador[0])):
                res += imagen[i][j] * clasificador[0][j]
            res -= clasificador[1]
            if res >= 0:
                ret.append(1)
            else:
                ret.append(-1)
        return ret

```

```

In [ ]: def obtener_error_hiperplano(clasificador, X, Y, D):
        return D * (aplicar_clasificador_debil_hiperplano(clasificador, X) != Y)

```

A continuación se define la función adaboost. Esta es igual que la implementada para los clasificadores umbral, pero utiliza las funciones de clasificadores hiperplano.

```

In [ ]: def adaboost_hiperplano(X, Y, T, A):
        clasificadores_debiles = []
        alphas = []
        D = np.full(len(X), 1.0/len(X))

        for t in range(T):
            conjuntos = []
            for k in range(A):
                c_d = generar_clasificador_debil_hiperplano(28*28)
                clas = aplicar_clasificador_debil_hiperplano(c_d, X)
                eps = np.sum(obtener_error_hiperplano(c_d, X, Y, D))
                conjuntos.append((c_d, eps, clas))

            pri = True
            for elem in conjuntos:
                if pri:
                    fc = elem
                    pri = False
                    continue
                if elem[1] < fc[1]:
                    # Guardamos el conjunto con menor error
                    fc = elem

            alph = 0.5*math.log2(abs((1-fc[1])/fc[1]))
            alphas.append(alph)
            clasificadores_debiles.append(fc[0])

            Z = np.sum(D)
            for i in range(len(D)):
                D[i] = (D[i]*(math.e**(-alph*Y[i] * fc[2][i])))/Z

        return (clasificadores_debiles, alphas)

```

Será necesario poder aplicar el clasificador fuerte obtenido.

```

In [ ]: def aplicar_clasificador_fuerte_hiperplano(clasificadores, alphas, imagen):
        clasificacion = 0
        for i in range(len(clasificadores)):
            clasificacion += alphas[i] * np.sum(aplicar_clasificador_debil_hiperplano(clasi

```

```
return clasificacion
```

Igual que antes, se ha definido una función que permite entrenar al sistema en conjunto.

```
In [ ]: def entrenar_sistema_hiperplano(mnist_X, mnist_Y, T, A):
        clasificadores = []
        alphas = []
        for clase in range(10):
            (trainX, trainY) = adaptar_conjuntos(mnist_X, mnist_Y, clase)
            (cl, al) = adaboost_hiperplano(trainX, trainY, T, A)
            clasificadores.append(cl)
            alphas.append(al)
        return (clasificadores, alphas)
```

Se probará ahora el funcionamiento mediante la función `testn`, igual que la anterior pero utilizando funciones de hiperplano.

```
In [ ]: def testn_hiperplano(X, Y, clasificadores, alphas, n):
        correct = total = 0

        for i in range(len(X)):
            if (Y[i] != n):
                continue
            total += 1
            clasif = (-1, 0)
            for j in range(10):
                punt = (aplicar_clasificador_fuerte_hiperplano(clasificadores[j], alphas[j]
                    if punt > clasif[1]:
                        clasif = (j, punt)
            if n == clasif[0]:
                correct += 1

        return correct/total
```

```
In [ ]: T = 3
        A = 10
        print("Proceso de entrenamiento")
        (clasificadores, alphas) = entrenar_sistema_hiperplano(mnist_X, mnist_Y, T, A)
        (testX, testY) = adaptar_conjuntos_test(mnist_X, mnist_Y)
        print("Proceso de pruebas")
        print(f"Porcentaje de aciertos del sistema para la clase 4: {np.average(testn_hiperplan
```

Proceso de entrenamiento

Proceso de pruebas

Porcentaje de aciertos del sistema para la clase 4: 3.571428571428571%

Igual que con los clasificadores umbral, se ha implementado la función `test` para probar el sistema en conjunto.

```
In [ ]: def test_hiperplano(X, Y, clasificadores, alphas):
        correct = np.full(10, 0)
        total = np.full(10, 0)

        for i in range(len(X)):
            total[Y[i]] += 1
            clasif = (-1, 0)
            for j in range(10):
                punt = (aplicar_clasificador_fuerte_hiperplano(clasificadores[j], alphas[j]
                    if punt > clasif[1]:
                        clasif = (j, punt)
            if Y[i] == clasif[0]:
                correct[clasif[0]] += 1

        return correct/total
```

```
In [ ]: print("Proceso de entrenamiento")
(clasificadores, alphas) = entrenar_sistema_hiperplano(mnist_X, mnist_Y, T, A)
(testX, testY) = adaptar_conjuntos_test(mnist_X, mnist_Y)
print("Proceso de pruebas")
print(f"Porcentaje de aciertos del sistema: {np.average(test_hiperplano(testX, testY, c
```

Proceso de entrenamiento
 Proceso de pruebas
 Porcentaje de aciertos del sistema: 9.664462072925959%

Cuestiones sobre la práctica: ampliación de Julio

¿Varía T y A entre ambos clasificadores débiles al usarlos en Adaboost? ¿Cuál requiere un T más elevado? ¿Porqué?

Al implementar hiperplanos el valor de T puede reducirse bastante. Esto se debe a que el clasificador umbral únicamente "mira" un pixel de la imagen y el hiperplano todos los píxeles, por tanto este segundo con un menor número de clasificadores es capaz de acercarse al objetivo mejor. Además, el proceso de entrenamiento con hiperplanos es considerablemente más lento.

A continuación se realizará un test en el cual valorar A y T como se hizo con los clasificadores umbral.

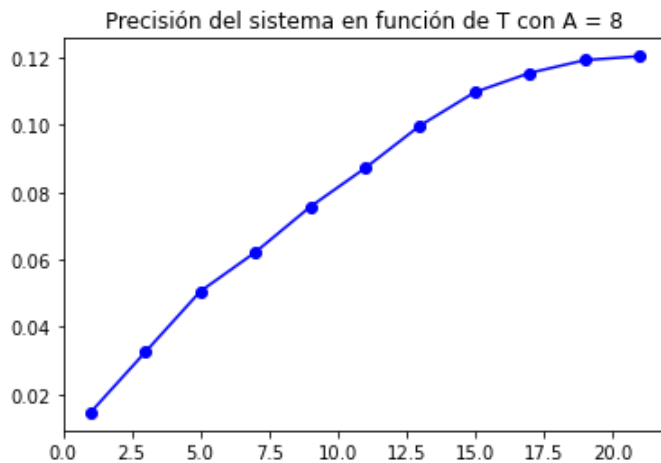
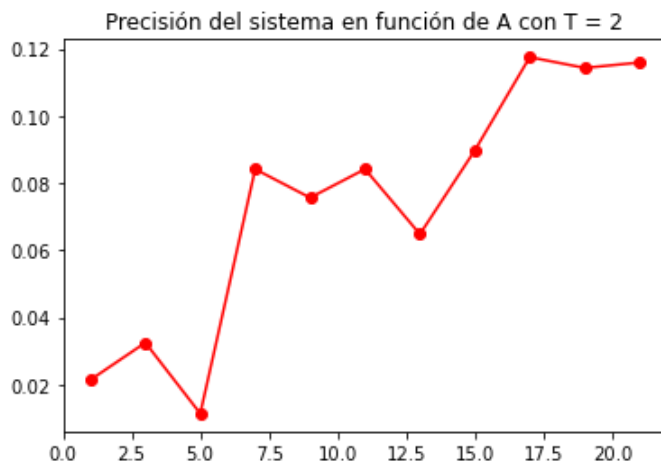
```
In [ ]: def valorarAT_hiperplano(X, Y, A, T, At, Ta):
    resultsA = []
    for i in At:
        (clasificadores, alphas) = entrenar_sistema_hiperplano(X, Y, T, i)
        (testX, testY) = adaptar_conjuntos_entr_test(X, Y)
        resultsA.append(test_hiperplano(testX, testY, clasificadores, alphas))

    resultsT = []
    for i in Ta:
        (clasificadores, alphas) = entrenar_sistema_hiperplano(X, Y, i, A)
        (testX, testY) = adaptar_conjuntos_entr_test(X, Y)
        resultsT.append(test_hiperplano(testX, testY, clasificadores, alphas))
    return (resultsA, resultsT)
```

```
In [ ]: T = 2
A = 8
At = range(1, 22, 2)
Ta = range(1, 22, 2)

(rA, rT) = valorarAT(mnist_X, mnist_Y, A, T, At, Ta)
rAcorr = [np.average(i) for i in rA]
rTcorr = [np.average(i) for i in rT]

plt.figure()
plt.plot(At, rAcorr, 'r-o')
plt.title(f"Precisión del sistema en función de A con T = {T}")
plt.show()
plt.figure()
plt.plot(Ta, rTcorr, 'b-o')
plt.title(f"Precisión del sistema en función de T con A = {A}")
plt.show()
```



Se puede apreciar una gráfica similar en forma a la que se usó con los clasificadores umbral. En este test los valores de A y T escogidos son muy inferiores a los utilizados en el clasificador umbral, y aportan precisiones similares. Por otro lado el tiempo de ejecución de este test ha sido muy alto.

¿Qué clasificador débil crees que funciona mejor para clasificar un dígito? ¿Y el problema completo con 10 dígitos?

Si únicamente se tuviera en cuenta el resultado final, se podría desarrollar sistemas en los que los valores de T y A fueran muy elevados y cualquiera de los dos clasificadores arrojaría resultados precisos.

Siendo esto un problema a nivel temporal y computacional, para un sistema más sencillo, el problema de clasificar un único dígito se resolvería mejor utilizando el clasificador umbral, pues es más sencillo de entrenar y es bastante especializable. El clasificador hiperplano también resuelve este problema de forma adecuada, pero el mayor coste computacional y temporal a la hora de entrenar al sistema es un lastre importante.

Por otro lado, el clasificador hiperplano sería una solución más adecuada en el caso de resolver el sistema completo. Al tener que aprender una variedad de formas, el clasificador umbral necesita mucho entrenamiento, mientras que el clasificador hiperplano con un menor entrenamiento, es capaz de solventar el problema correctamente.