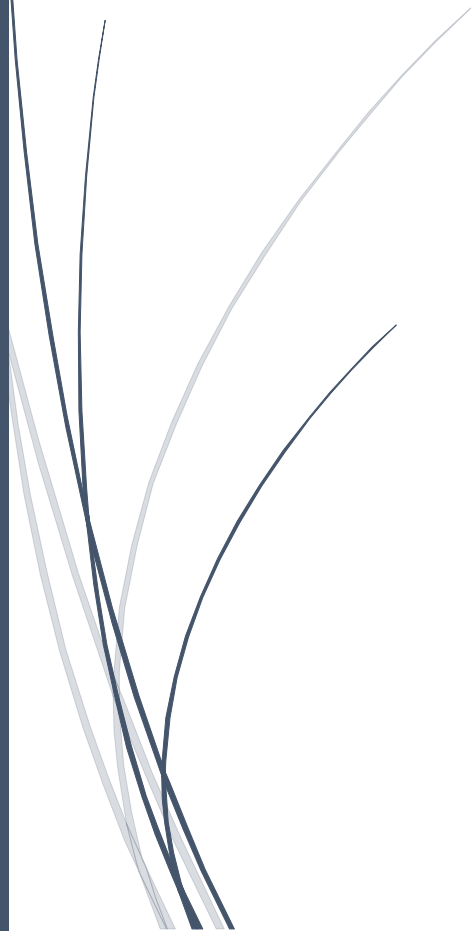
BRI PÉREZ, ANTONIO

Práctica 2: Ada Boost

HITO 3



UNIVERSIDAD DE ALICANTE
INGENIERIA INFORMÁTICA

Contenido

1. Introducción

Esta práctica es la segunda práctica de la asignatura sistemas inteligentes en la cual desarrollaremos un sistema que será capaz de distinguir números manuscritos. Para ello implementaremos un sistema de aprendizaje supervisado.

La entrada al sistema, las operaciones aplicadas sobre esta entrada y la salida obtenida serán explicadas a lo largo de la introducción

1.1. MNIST

MNIST es una base de datos de dígitos manuscritos en la que constan un total de 60.000 ejemplos y un test de 10.000 imágenes de números manuscritos. Y tan solo se trata de una versión reducida del conjunto NIST el cual posee más datos para desarrollar nuestro sistema. Las imágenes que nos encontraremos en la base de datos serán como el siguiente ejemplo:

Ilustración 1

Sin embargo, nosotros usaremos solo los 1000 primeros dígitos de este conjunto. Este conjunto se proporciona desde la asignatura.

1.2. Aprendizaje supervisado

Este conjunto mencionado anteriormente será el que utilicemos para entrenar a nuestro sistema. Utilizaremos, por tanto, el método de aprendizaje supervisado.

Este método consta de un conjunto de datos X del cual conocemos la clase a la que pertenece Y. Entonces, mediante un cálculo ponderado de pesos podemos asignarle más importante a aquellos elementos que están más alejados de la solución correcta para que nuestro sistema los tenga más en cuenta en siguientes iteraciones.

Ahora que ya hemos introducido los conceptos teóricos que utilizaremos vamos a ver como hemos organizado la estructura del proyecto.

2. Estructura del proyecto

Nuestro proyecto de NetBeans posee la siguiente estructura de clases:

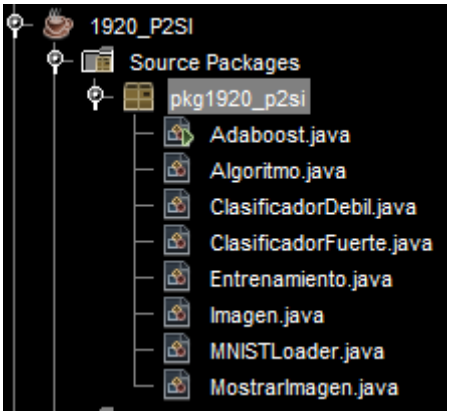


Ilustración 2

Se han implementado, desde 0 las clases: **Adaboost.java** (contiene el método main), la clase **Algoritmo.java**, **ClasificadorDebil.java**, **ClasificadorFuerte.java** y **Entrenamiento.java**.

Las clases Imagen.java, MNISLoader.java y MostrarImagen.java venían con la plantilla.

En los siguientes apartados se explicará que contiene cada clase y que función desempeña en el programa.

Solo se van a comentar los métodos principales de las clases para no adjuntar mucho código ya que este código acompaña a este documento y puede revisarse y probarse.

2.1. Adaboost

Esta es la clase principal del programa y la que se encargará de dirigir el código por la opción de entrenamiento o la fase de carga de los clasificadores fuertes. En el método main podemos recibir dos tipos de argumentos. En el método main podemos recibir dos tipos de argumentos: el entrenamiento o la carga.

```
24  public class Adaboost{
25      ArrayList aprender = new ArrayList();
26      public final MNISTLoader imageLoader = new MNISTLoader();
27
28      public static void main(String[] args) throws IOException{
29
30          if(args.length != 2){
31              System.err.println("Error en los argumentos");
32          }
33
34          else if("-t".equals(args[0])){
35
36              ArrayList<ClasificadorFuerte> fuertes = entrenarAlgoritmo();
37
38              guardaFuertesFichero(fuertes,args[1]);
39          }
40
41          // Adaboost <fichero_origen_CF> <ruta_imagen>
42
43          else{
44
45              faseDeCarga(args);
46          }
47      }
```

Ilustración 3

En esta clase también implementamos otros métodos muy importantes como el siguiente:

```java
private static ArrayList<ClasificadorFuerte> entrenarAlgoritmo() throws IOException {
    Entrenamiento entrenamiento = new Entrenamiento();
    ArrayList<ClasificadorFuerte> fuertes = new ArrayList();
    double umbral = 0.08;
    for(int i=0; i<=9; i++){
        boolean aux = true;
        while(aux){
            entrenamiento.entrenaAlgoritmo(i);
            double error = entrenamiento.valida(entrenamiento.fuertesEnt.get(i),i);
            System.out.println(i + " : " + error);
            if(error <= umbral){
                aux = false;
                fuertes.add(entrenamiento.fuertesEnt.get(i).getCF());
            }
            else{
                System.err.println(i + ": "+ error);
            }
        }
    }
    imprimirPorcentajes(fuertes,entrenamiento);
    return fuertes;
}
```

Ilustración 4

Este método se encarga entrenar el algoritmo y posteriormente de imprimir los porcentajes por pantalla.

Después para la fase de carga disponemos del siguiente método:

```java
private static void faseDeCarga(String[] args) throws NumberFormatException, IOException {
    String path = args[1]; //ruta de la imagen
    ArrayList<ClasificadorDebil> debiles = new ArrayList<>();
    ArrayList<ClasificadorFuerte> fuertes = new ArrayList<>();
    CargaFuertes(args, debiles,fuertes);
    ArrayList<Imagen> arrImg = new ArrayList<>();
    File fileIm = new File(path);
    if(fileIm.isFile()){
        Imagen im = new Imagen(fileIm.getAbsoluteFile());
        arrImg.add(im);
        int resultado = resolver(arrImg, fuertes);
        if(resultado==-1){
            System.err.println("No se ha podido reconocer el numero");
        }
        else{
            System.out.println("EL DIGITO CARGADO ES UN: " + resultado);
        }

        //Descomentar si se quiere ver la imagen por parametro

        /*
        Imagen img = new Imagen(fileIm);
        MostrarImagen imgShow = new MostrarImagen();
        imgShow.setImage(img);
        imgShow.mostrar();
        */
    }
    else{
        System.err.println("Error al cargar la imagen");
    }
}
```

Ilustración 5

Este método se encarga de cargar la imagen por argumento y de llamar al método que resuelve la imagen mediante el uso de factores de pertenencia.

Esta clase implementa muchos más métodos relacionados con la lectura y escritura de ficheros, cálculos auxiliares etc. Estos métodos no se comentan aquí

2.2. Clasificador Débil

Esta clase representa el objeto que se encargará de realizar una clasificación leve de la imagen que le pasemos. Los atributos de esta clase son:

```
15  public class ClasificadorDebil {
16      int pixel;
17      int direccion; //Si 0 --> "<=" si 1 --> ">"
18      int umbral;
19      double confianza=0; //valor de confianza del clasificador debil
20      double error;
```

Ilustración 6

El método más importante de esta clase es el siguiente:

```
54      public List<Integer> aplicarClasificadorDebil(List<Imagen> imgs){
55          List<Integer> resultados;
56          resultados = new ArrayList<>();
57
58          imgs.forEach((im) -> {
59              if(this.direccion == -1){
60                  if((im.getImageData()[this.pixel]) < this.umbral){
61                      resultados.add(1);
62                  }
63                  else{
64                      resultados.add(-1);
65                  }
66              }
67              else{
68                  if((im.getImageData()[this.pixel])>= this.umbral){
69
70                      resultados.add(1);
71                  }
72                  else{
73                      resultados.add(-1);
74                  }
75              }
76          });
77          return resultados;
78      }
```

Ilustración 7

Este método se encarga de aplicar un clasificador sobre un vector de imágenes. Primero miramos si la dirección es 1 o -1. Si es -1 comprobamos si el valor de la imagen en el píxel de nuestro clasificador se encuentra a ese lado del clasificador o no. Si no se encuentra, significa que ha fallado (añadimos un -1 al vector de resultados). Si está a ese lado del clasificador hemos acertado (añadimos un 1).

La otra condición es lo mismo, pero hacia el otro lado del umbral. Este método resulta clave para el correcto funcionamiento de nuestro sistema de aprendizaje.

2.3. Clasificador Fuerte

Un clasificador fuerte se define como la combinación de varios clasificadores débiles. Al final de nuestro proyecto, deberemos tener un clasificador fuerte para cada dígito. Cuando carguemos una imagen veremos cuál de todos los clasificadores fuerte arroja mayor factor de pertenencia y el sistema apostará por el dígito que corresponda al clasificador escogido.

Esta clase solo posee un atributo:

```
15  @SuppressWarnings("unchecked")
16  public class ClasificadorFuerte {
17
18      public ArrayList<ClasificadorDebil> debiles;
```

Ilustración 8

Este atributo se encargará de almacenar los clasificadores débiles que al combinarlos constituirán un clasificador fuerte.

Implementa el siguiente constructor, muy simple:

```
20  public ClasificadorFuerte(ArrayList<ClasificadorDebil> cd){
21      debiles = new ArrayList();
22
23      for(int i=0; i<cd.size();i++){
24          debiles.add(cd.get(i));
25      }
26  }
```

Ilustración 9

A pesar de ser una clase muy sencilla nos permite organizar el proyecto de forma cómoda para trabajar con objetos. Podríamos prescindir de esta clase tan sencilla, pero a la hora de generar un conjunto de clasificadores fuertes (vendría a ser el cerebro de nuestro sistema) será muy útil disponer de esta clase.

2.4. Entrenamiento

Esta clase se encarga de dirigir el entrenamiento de nuestro sistema de información (que no hacerlo). Para ello dispone de los siguientes atributos:

```
23  @SuppressWarnings("unchecked")
24  public class Entrenamiento {
25      public ArrayList<Imagen> test;
26      public ArrayList<Imagen> validSet;
27      public ArrayList<Algoritmo> fuertesEnt;
28
```

Ilustración 10

El vector test es el que utilizaremos para cargar las imágenes de la base de datos que utilizaremos para entrenar nuestro algoritmo (como veremos en el constructor)

El vector validSet lo utilizaremos para validar los resultados que nos arroja nuestro algoritmo AdaBoost y de esa manera pasarle al main un umbral máximo aceptable para que todos los números tengan un mínimo de garantía de ser identificados.

El vector fuertesEnt es un vector que contiene objetos Algoritmo que nos permitirá realizar las llamadas correspondientes al algoritmo AdaBoost que implementa la clase mencionada.

El método más importante de esta clase es su constructor:

```
32  public Entrenamiento(){
33      imageLoader = new MNISTLoader();
34      imageLoader.loadDBFromPath("./mnist_1000");
35      test = new ArrayList<>();
36      int porcentajeEnt=80;
37
38      //cargamos imagenes para el test
39
40      for(int i=0; i<10;i++){
41          //Cogemos 80 imagenes de cada número para el test
42          test.addAll(imageLoader.getImageDatabaseForDigit(i).subList(0,
43                  porcentajeEnt));
44
45      }
46
47
48      validSet = new ArrayList<>();
49      for(int i=0; i<10;i++){
50          //Cogemos 20 imagenes de cada número para la validación
51          validSet.addAll(imageLoader.getImageDatabaseForDigit(i).subList(porcentajeEnt,
52          imageLoader.getImageDatabaseForDigit(i).size()));
53
54      }
55
56      fuertesEnt = new ArrayList<>(10);
57      Algoritmo alg = new Algoritmo(1);
58
59      while(fuertesEnt.size() < 10){
60          fuertesEnt.add(alg);
61      }
62
63  }
```

Ilustración 11

Cargamos la base de datos y cargamos las imágenes desde la 0 a la 80 para todos los dígitos para nuestro test.

Cargamos nuestro set de validación para validar los datos de AdaBoost.

Creamos un objeto algoritmo y lo guardamos en nuestro vector de algoritmos tantas veces como dígitos halla (necesitamos un clasificador fuerte para cada dígito).

El resto de métodos son auxiliares. Ahora vamos a ver los métodos principales de la clase algoritmo y dejaremos de lado el código.

2.5. Algoritmo

Esta clase es la encargada de recibir los atributos del objeto entrenamiento y por lo tanto aplicar la algoritmia AdaBoost sobre estos datos. Primero tenemos el método generarClasifAzar

```
32      public ClasificadorDebil generarClasifAzar(int dimension) {
33          int pixel = (int)(Math.random()*dimension);
34          int umbral = (int) (Math.random()*255);
35          int dir = (int)(Math.random()*2);
36          if(dir == 0){
37              dir = -1;
38          }
39          return (new ClasificadorDebil(pixel,umbral,dir));
40
41      }
```

Ilustración 12

Que devuelve un clasificador débil aleatorio. A continuación, tenemos el método entrenarAda que es el algoritmo AdaBoost

```
66      void entrenarAda(List<Imagen> test, int digito) {
67          int i = 0;
68          //la m indica el número de clasificadores
69          //cuantos mas mejor
70          for(int m=0; m<400 /* valor experimental */;m++){
71              ClasificadorDebil debil;
72              ArrayList<ClasificadorDebil> debiles = new ArrayList<>();
73              List<Integer> resultados = null;
74
75              double err=0;
76              for(int j = 0; j<500; /*valor experimental*/ j++){
77                  debil =  generarClasifAzar(784);
78                  resultados = debil.aplicarClasificadorDebil(test);
79                  err = obtenerErrorClasificador(debil,resultados,test, d,digito);
80                  debiles.add(debil);
81              }
82
83              debil = getMejor(debiles);
84
85              double confianza = 0.5*Math.log(((1-debil.error) / debil.error));
86              debil.confianza = confianza;
87              resultados = debil.aplicarClasificadorDebil(test);
88              actualizarPesos(confianza,digito,test,resultados);
89
90
91              fuerteAlg.add(debil);
92
93              i++;
94          }
95
96      }
```

Ilustración 13

El resto de los métodos son auxiliares y los utilizamos para actualizar pesos, quedarnos con el mejor clasificador etcétera. Vamos a explicar cómo ejecutar el proyecto.

3. Ejecución del proyecto

3.1. Fase de entrenamiento

En este apartado se va a explicar cómo ejecutar el proyecto en modo entrenamiento. Lo primero de todo es cargar el proyecto en NetBeans:

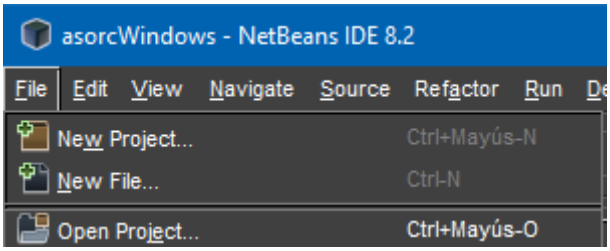


Ilustración 14

Y buscamos el proyecto que se llama “Plantilla” y lo cargamos:

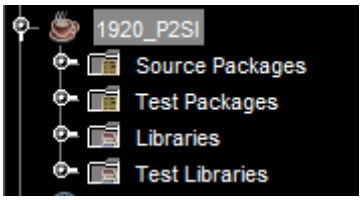


Ilustración 15

Lo siguiente que tendremos que hacer será abrir una consola de Windows y movernos hasta la ruta **./dist** de nuestro proyecto:

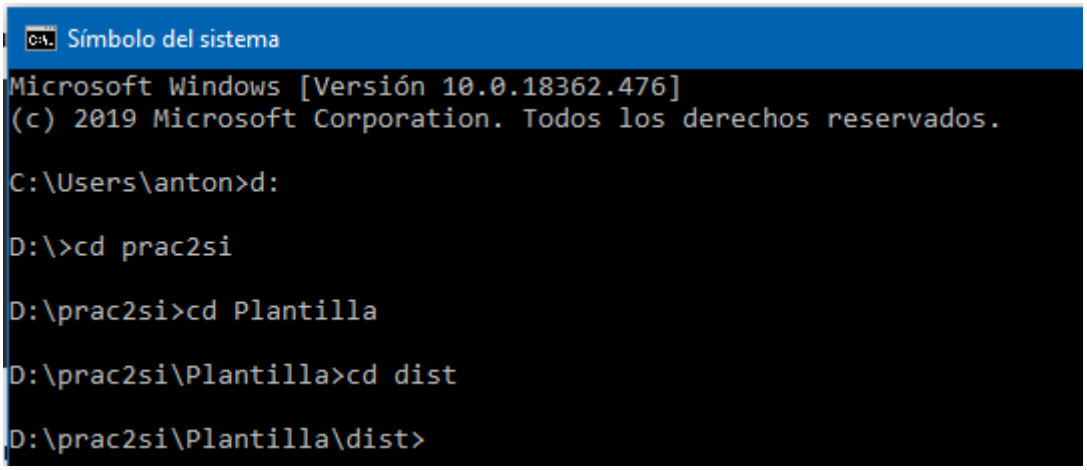


Ilustración 16

Una vez hecho esto tendremos que pulsar la tecla F11 en nuestro IDE y se nos construirá un fichero 1920_P2SI.jar

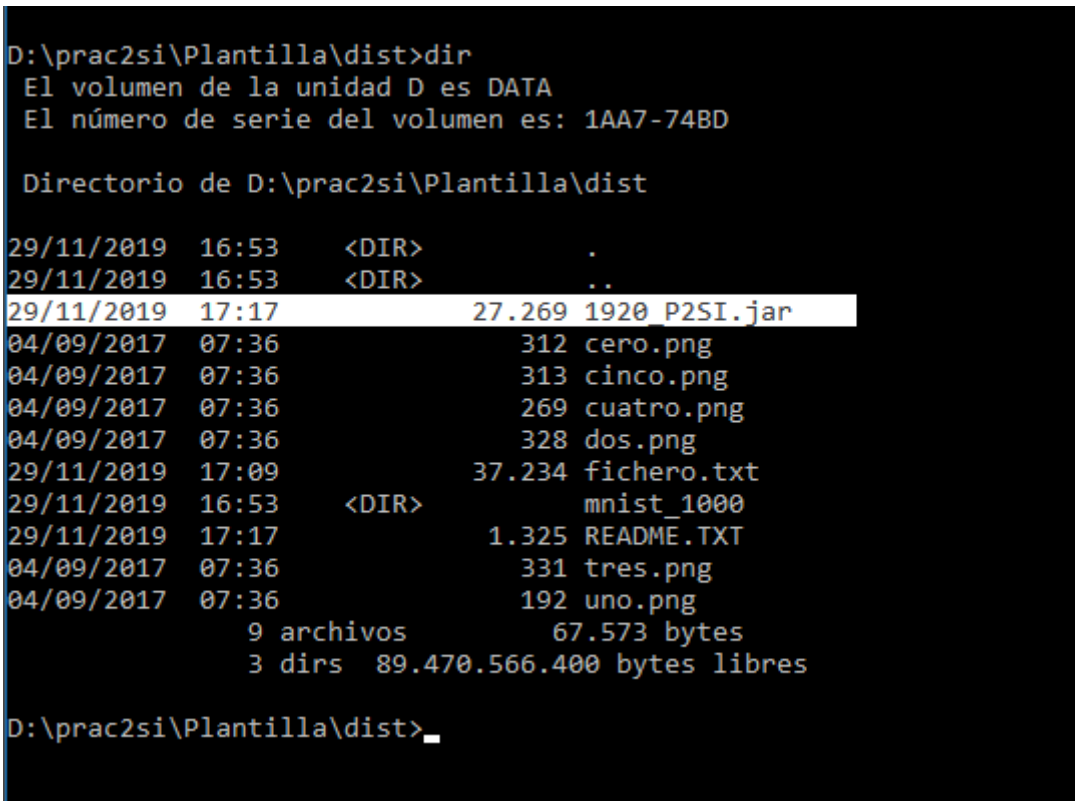


Ilustración 17

Para poder entrenar nuestro algoritmo tendremos que ejecutar de esta manera:

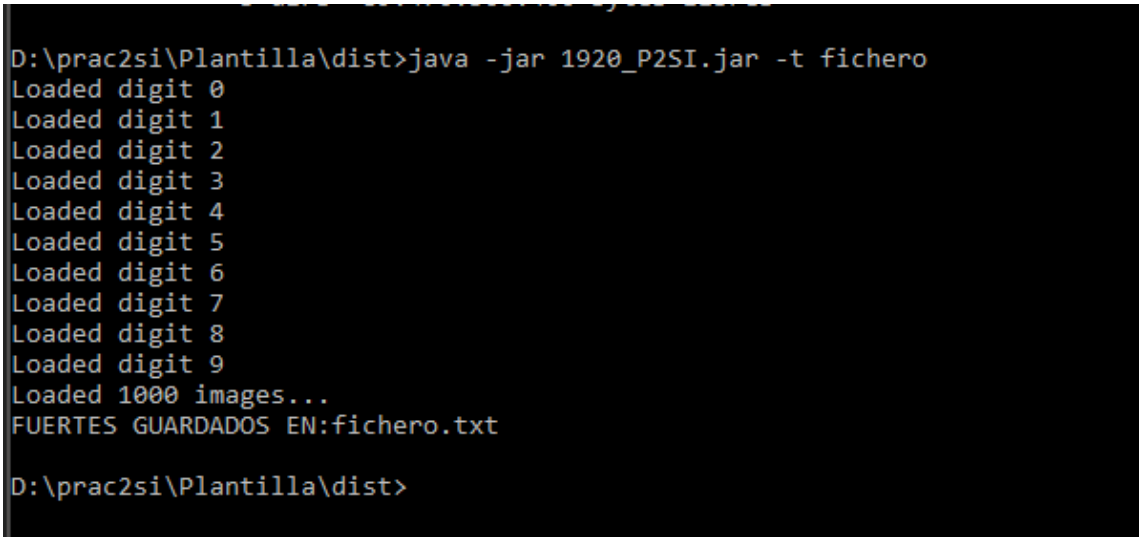


Ilustración 18

Esto entrenará nuestro algoritmo y creará en esa ruta un fichero de nombre “fichero” en este caso con todos los clasificadores:

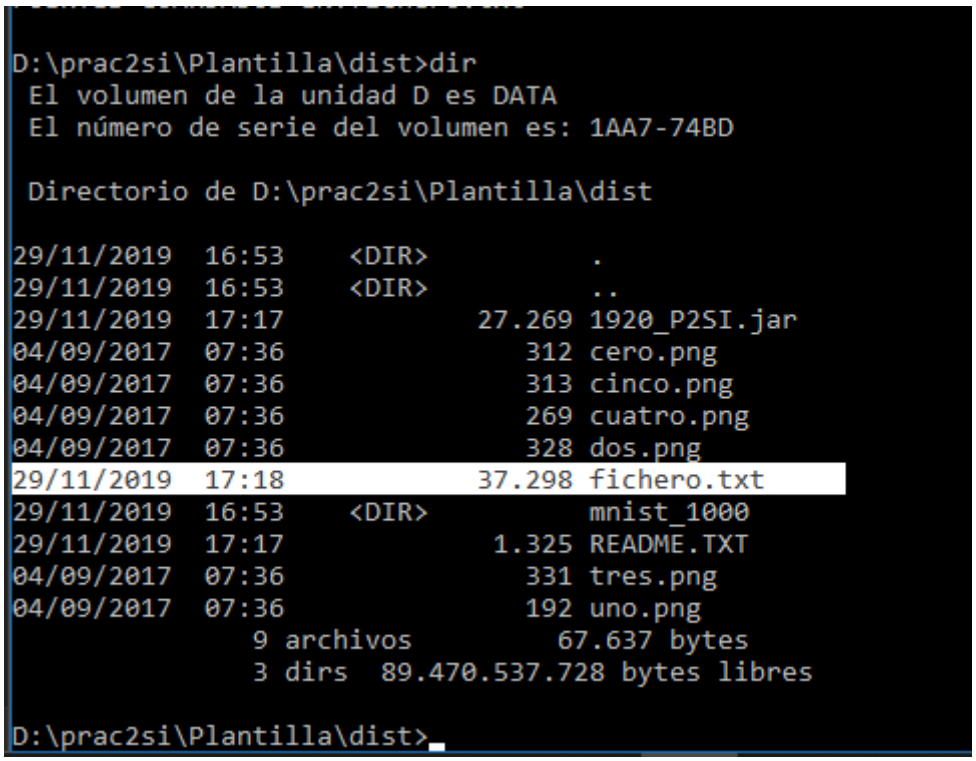


Ilustración 19

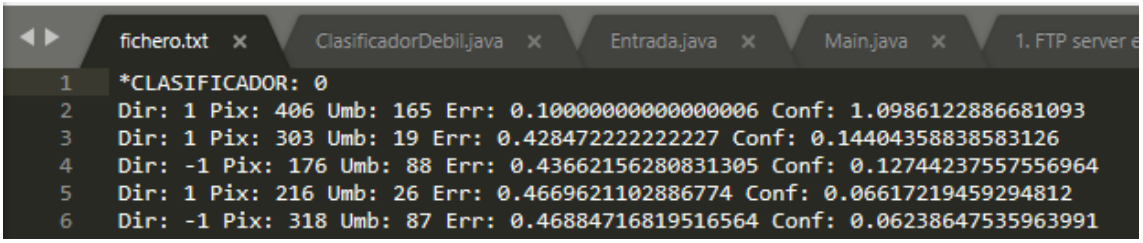


Ilustración 20

(Solo se muestran algunos)

Esos son los clasificadores que ha generado nuestro algoritmo y que ahora cargaremos en el programa.

3.2. Fase de carga

A continuación, explicaremos como cargar una imagen en el programa. Cabe destacar que si la imagen no se encuentra en la misma ruta que el programa .jar la imagen no cargará. No se muy bien porque ocurre esto pero aun poniendo la ruta completa de la imagen no cargaría en caso de encontrarse en otro directorio.

Igual que para la fase de entrenamiento abriremos una CMD de Windows y ejecutaremos el siguiente comando:

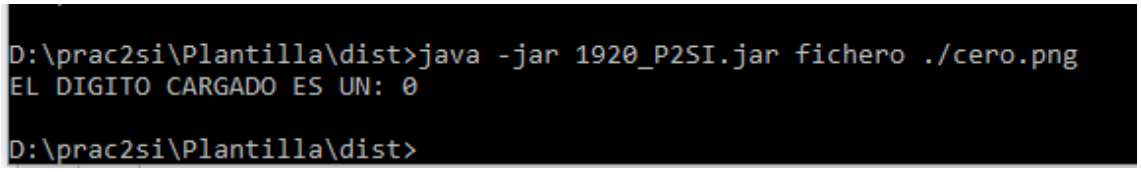


Ilustración 21

El primer argumento será el nombre del fichero (en este caso es fichero) y la imagen a cargar (./cero.png). El programa nos devolverá el resultado que considere.

Quiero destacar una parte del código y es el método que he utilizado para realizar la decisión del número:

```
331     public static int resolver (ArrayList<Imagen> img, ArrayList<ClasificadorFuerte> fuertes){
332        double pertenencia=0;
333        ArrayList<Double> pertenencias = new ArrayList();
334        for(Imagen imagen:img){
335            ArrayList<Imagen> im2 = new ArrayList<>();
336            im2.add(imagen);
337            double mejor = -9999;
338            for(int i=0; i<fuertes.size();i++){
339                ClasificadorFuerte aux = fuertes.get(i);
340                for(int j=0; j<aux.debiles.size();j++){
341
342                    double aux2 = aux.debiles.get(j).aplicarClasificadorDebil(im2).get(0);
343                    pertenencia +=aux2+aux.debiles.get(j).confianza;
344                }
345                pertenencias.add(pertenencia);
346                pertenencia=0;
347            }
348            //Controlamos el arrayList no tenga negativos
349            boolean allNegative=true;
350            for(int i=0; i<pertenencias.size();i++){
351                if(pertenencias.get(i)>0){
352                    allNegative = false;
353                }
354            }
355            if(allNegative){
356                System.err.println("No se ha reconocido el numero ");
357                return -1;
358            }
359            //Buscamos la maxima pertenencia y devolvemos su posicion
360            //que será el digito solución
361            double maximum=Collections.max(pertenencias);
362            for(int i=0; i<pertenencias.size();i++){
363                if(pertenencias.get(i)==maximum){
364                    return i;
365                }
366            }
367        }
368        return -1;
369     }
370
371 }
```

Ilustración 22

En esa captura vemos el método resolver que recibe una imagen y un vector de clasificadores fuertes. Lo que hago aquí es recorrer todos los clasificadores débiles de todos los clasificadores fuertes del fichero cargado y a cada clasificador fuerte le vinculo un factor de pertenencia (línea 343) basado en la confianza de cada uno de los clasificadores débiles por los que está compuesto. Esa confianza la guardo en un array de doubles. Por tanto, me queda para clasificador fuerte un valor de pertenencia. Cuando ya tengo mis diez pertenencias recorro ese vector buscando el valor mayor y me guardo la posición donde he encontrado ese valor mayor. La posición donde se encuentre ese valor mayor será el número que el programa ha reconocido (bucle de la línea 362). Esto es porque el clasificador fuerte para el 0 se encuentra en la primera posición del vector, el clasficador fuerte para el 1 se encuentra en la segunda y así sucesivamente. Entonces las pertenencias me quedan en el mismo orden.

Como es posible que el programa no reconozca el número podría darse el caso de que todo mi array fuera de negativos. Por ello, he decidido controlar esa excepción añadiendo la condición de la línea 355.

Ahora vamos a dar paso al análisis de los resultados.

4. Análisis de los resultados

4.1. Exposición de los resultados

4.1.1. Variando bucle 2

En primera instancia, hemos decidido realizar una distribución de 80% del conjunto para entrenar y el 20% para validar. Vamos a ver que pasa cuando modificamos los valores del bucle 2 de AdaBoost manteniendo esos porcentajes de entrenamiento y validación y el valor del bucle 1 a 10. En esta captura vemos a que bucles estoy haciendo referencia cuando hablo de bucle 1 y bucle 2:

```
72          //BUCLE 1:
73          for(int m=0; m<400 /* valor experimental */;m++){
74              ClasificadorDebil debil;
75              ArrayList<ClasificadorDebil> debiles = new ArrayList<>();
76              List<Integer> resultados = null;
77
78              double err=0;
79              //BUCLE 2
80              for(int j = 0; j<400; /*valor experimental*/ j++){
81                  debil =  generarClasifAzar(784);
82                  resultados = debil.aplicarClasificadorDebil(test);
83                  err = obtenerErrorClasificador(debil,resultados,test, d,digito);
84                  debiles.add(debil);
85              }
```

Ilustración 23

% de acierto	valor bucle 2
11,5	10
21	50
23	100
23,5	200
31,5	400
30	800

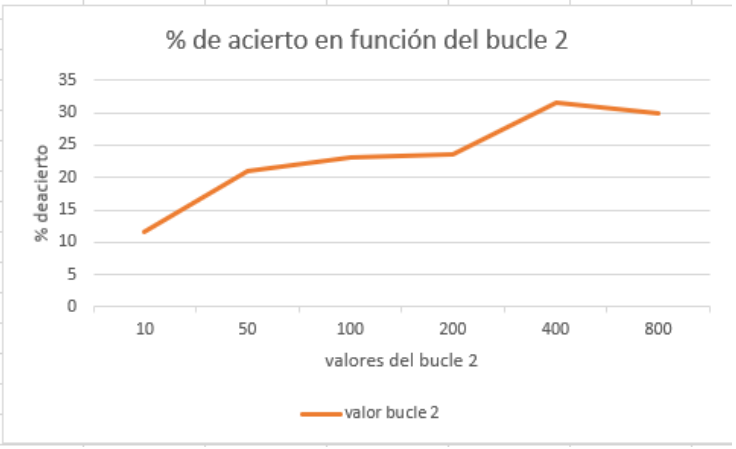


Ilustración 24

En el eje X nos encontramos con los valores del bucle 2 que hemos ido probando. En el eje Y tenemos el % de acierto en el test de validación. Como vemos, a pesar de incrementar en 20 veces las iteraciones (de 10 a 200) apenas conseguimos llegar de un 11,5 % de acierto a un 23,5 %. Es más, a pesar de incrementar el número de iteraciones de 400 a 800 no solo no ganamos en acierto, sino que estamos perdiendo.

4.1.2. Variando bucle 1

A continuación, vamos a ver que pasa cuando mantenemos el bucle 2 a 10 y cambiamos los valores del bucle 1:

% de acierto	valor bucle 1
11,5	10
27,5	50
34,5	100
40,5	200
51,5	400
64,5	800

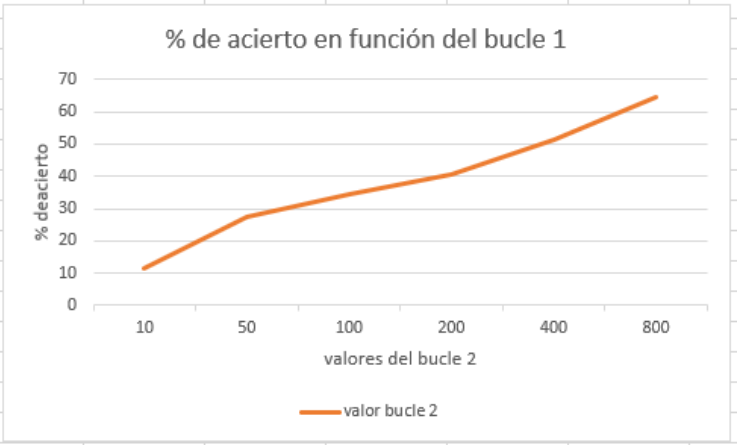


Ilustración 25

En este caso, obtenemos un incremento sostenido conforme aumentamos los valores del bucle 1. Podemos observar que se asemeja a una recta así que tenemos un incremento casi lineal. Vamos a ver que pasa si por ejemplo añadimos 1600 iteraciones al bucle 1. Veamos si se mantiene la linealidad.

% de acierto	valor bucle 1
11,5	10
27,5	50
34,5	100
40,5	200
51,5	400
64,5	800
43,5	1600

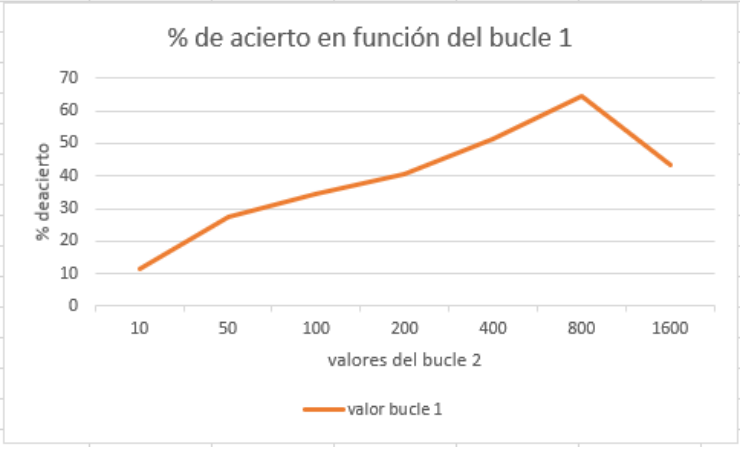


Ilustración 26

Como podemos observar ya no tenemos un incremento del % de acierto en el entrenamiento. Ahora caemos en picado, por tanto concluimos que no es buena estrategia incrementar uno de los bucles dejando estático el otro en bajas iteraciones.

Ahora veamos que ocurre cuando intentamos combinar ambos valores y los mantenemos siempre iguales.

% de acierto	valor bucle 1 y 2
12,5	10
40	50
49	100
67,5	200
77,5	400
79	800
80	1600

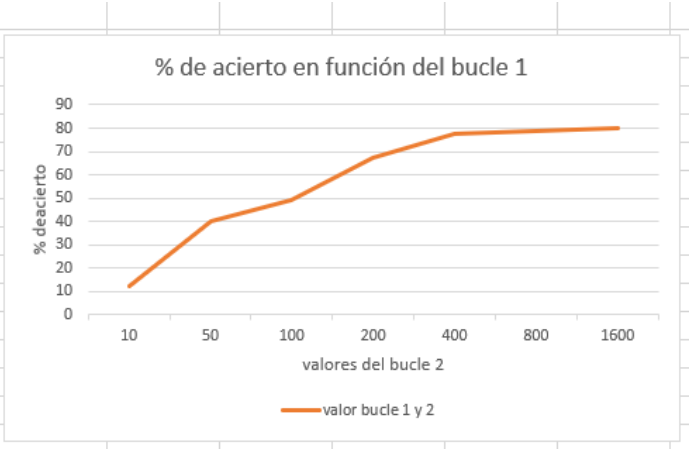


Ilustración 27

Como vemos obtenemos una grandísima mejora si combinamos los valores de ambos bucles. Sin embargo, cuando cargamos valores de 1600 en cada bucle obtenemos tiempos desproporcionados haciendo totalmente inviable esa configuración. Además, jugando con el umbral de validación podemos obtener porcentaje de acierto alrededor de 80 manteniendo 400 iteraciones (que parece el más aconsejable).

Por ejemplo, si reducimos el umbral de error admitido al 0,08 obtenemos estos resultados para 400 iteraciones en cada bucle:

```
76  private static ArrayList<ClasificadorFuerte> entrenarAlgoritmo(
77      Entrenamiento entrenamiento = new Entrenamiento();
78      ArrayList<ClasificadorFuerte> fuertes = new ArrayList();
        double umbral = 0.08;
80      for(int i=0; i<=9; i++){
81          boolean aux = true;
82          while(aux){
83              entrenamiento.entrenaAlgoritmo(i);
84              double error = entrenamiento.valida(entrenamiento.f
85
86              if(error <= umbral){
87                  aux = false;
88                  fuertes.add(entrenamiento.fuertesEnt.get(i).get
89              }
90              else{
91                  System.err.println(i + ": "+ error);
92              }
```

Ilustración 28

Ahí vemos umbral de 0,08. Obtenemos estos resultados:

Ilustración 29

El cual se acerca mucho a 800 y 1600 iteraciones en cada bucle manteniendo las 400 iteraciones. Esta parece la solución más responsable

4.1.3. Variando distribución entrenamiento y validación

Ahora que ya hemos encontrado nuestra distribución óptima para entrenar nuestro AdaBoost vamos a ver que pasa cuando variamos la configuración de distribución para el entrenamiento y la validación. Vamos a probar con un umbral de 0.2

% de acierto	% entrenamiento
43	10
58,25	20
62	40
82	85

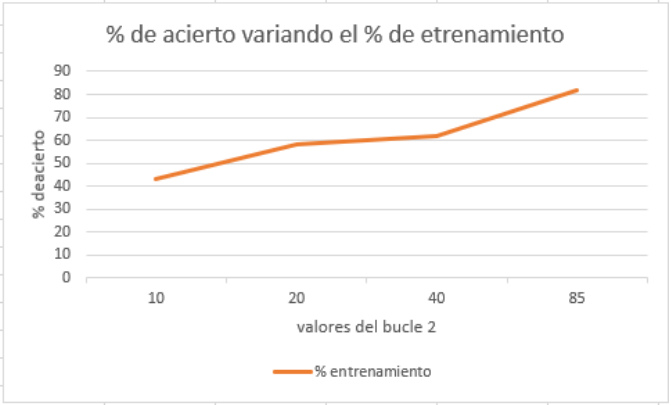


Ilustración 30

Como podemos ver obtenemos el % de acierto más grande con un % de entrenamiento de 85%. Si ahora probamos a reducir el umbral aceptable a 0.08 obtenemos el siguiente porcentaje de acierto:

```
78          ArrayList<ClasificadorFuerte> fuertes = new ArrayList();
79          double umbral = 0.08;
80          for(int i=0; i<=9; i++){
```

Ilustración 31

Con 0.08 obtenemos un 79,33% de acierto

```
D:\prac2si\Plantilla\dist>java -jar 1920_P2SI.jar -t fichero.txt
Loaded digit 0
Loaded digit 1
Loaded digit 2
Loaded digit 3
Loaded digit 4
Loaded digit 5
Loaded digit 6
Loaded digit 7
Loaded digit 8
Loaded digit 9
Loaded 1000 images...
Validacion: 79.33333333333333%
FUERTES GUARDADOS EN:fichero.txt.txt

D:\prac2si\Plantilla\dist>
```

Ilustración 32

Sin embargo, si aumentamos el umbral a 0.2 obtenemos un porcentaje de acierto de 80,67%. Esto es indicativo de que estaríamos sobreentrenando nuestro algoritmo

```
78        ArrayList<ClasificadorFuerte> fuertes = new ArrayList();
          double umbral = 0.2;
```

Ilustración 33

```
D:\prac2si\Plantilla\dist>java -jar 1920_P2SI.jar -t fichero.txt
Loaded digit 0
Loaded digit 1
Loaded digit 2
Loaded digit 3
Loaded digit 4
Loaded digit 5
Loaded digit 6
Loaded digit 7
Loaded digit 8
Loaded digit 9
Loaded 1000 images...
Validacion: 80.66666666666667%
FUERTES GUARDADOS EN:fichero.txt.txt

D:\prac2si\Plantilla\dist>
```

Ilustración 34

4.2. Conclusiones

Como conclusiones sacamos que el bucle 1 es más importante a la hora de obtener buenos resultados que el bucle 2 ya que iterar demasiado sobre el segundo bucle puede provocar el efecto contrario al deseado. También es clave ajustar de forma correcta el porcentaje de entrenamiento y validación ya que de otra manera estaríamos realizando un entrenamiento insuficiente (menor al 50%) de nuestro algoritmo. Seguidamente tenemos que controlar que no restringimos demasiado el umbral ya que en otro caso estaríamos sobreentrenando nuestro algoritmo de forma que obtendríamos el efecto contrario además de aumentar el tiempo de respuesta de nuestro programa.

En conclusión, el correcto ajuste de parámetros para que todos funcionen en armonía es la mejor opción para conseguir los resultados óptimos. Ir probando es esencial en esta disciplina

5. Cuestiones acerca del clasificador

5.1. Cuestión 1

¿Cuál es el número de clasificadores que se han de generar para que un clasificador débil funcione? Muestra una gráfica que permita verificar lo que comentas.

Para que un clasificador débil debe tener un error inferior al 50% para que no se considere puramente azaroso. Por tanto, vamos a realizar varias simulaciones para ver a partir de que número empezamos a obtener clasificadores con un error menor al 50%. Para realizar estas pruebas tenemos que modificar el bucle 2 de AdaBoost ya que es el que se encarga de generar los clasificadores. Este bucle:

```
for(int j = 0; j<400; /*valor experimental*/ j++){
    debil =  generarClasifAzar(784);
    resultados = debil.aplicarClasificadorDebil(test);
    err = obtenerErrorClasificador(debil,resultados,test, d,digito);
    debiles.add(debil);
}

debil = getMejor(debiles);
```

Ilustración 35

Para comprobarlo, vamos a ver cuantos clasificadores débiles con error menor a 50 hay para un solo clasificador fuerte. Para ello hemos creado este método auxiliar que nos devuelve cuantos clasificadores débiles hay con un error mayor o igual a 50:

```
76      private static void pruebaCDvalido(ArrayList<ClasificadorFuerte> fuertes){
77          int contador=0;
78          for(int i=0; i<fuertes.size();i++){
79              ClasificadorFuerte aux = fuertes.get(i);
80              for(int j=0; j<aux.debiles.size();j++){
81                  if(aux.debiles.get(j).error >= 0.5){
82                      contador++;
83                  }
84              }
85              break;
86          }
            System.out.println("Hay " + contador + " clasificadores malos");
88      }
```

Ilustración 36

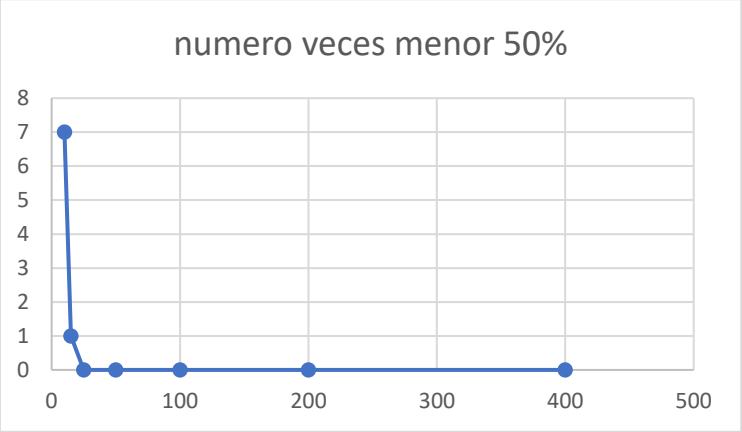


Ilustración 37

El eje X representa el número de clasificador débiles generados y el eje Y representa el número de clasificadores débiles con un error mayor o igual a 0.5. Como vemos, cuando generamos tan solo 15 clasificadores débiles obtenemos un clasificador con un error mayor o igual a 0.5 lo cual indica que ya estamos obteniendo clasificadores no válidos. Cuando generamos solo 10 clasificadores débiles 7 de ellos son mayores o igual a 50. Cabe decir que estos resultados han sido obtenidos con el bucle 1 de AdaBoost iterando hasta 400. Si modificamos este valor cabe esperar que estos valores sean incluso peores.

5.2. Cuestión 2

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

Para comprobar esto es tan sencillo como realizar una recogida de tiempos de la fase de aprendizaje modificando el valor del bucle 2 que es el que se encarga del número de clasificadores generados.

numero CD's generados	tiempo en segundos
400	21
200	12
100	5
50	3
25	2

Ilustración 38

Ilustración 39

En esta gráfica se aprecia claramente el tiempo empleado en la generación de los clasificadores débiles. Para estos cálculos hemos mantenido el valor del bucle 1 a 300 en todas las ejecuciones para sacar la relación correcta.

5.3. Cuestión 3

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

En mi caso he realizado una distribución 75% entrenamiento y 25% test ya que subir a 80% entrenamiento me arrojaba los mismos resultados que 75 y además de esta forma tengo un ¼ del conjunto disponible para validar mi sistema.

Esta división es fundamental ya que si elegimos una distribución muy alta de entrenamiento estaremos sobreentrenando nuestro algoritmo. Esto quiere decir qué aunque aumentemos el conjunto de entrenamiento obtenemos peores resultados a la hora de validarlo. Sin embargo, si escogemos una distribución pobre en entrenamiento podremos obtener resultados cercanos al 50% lo cual no tendría mucha diferencia con un sistema azaroso.

5.4. Cuestión 4

¿Has observado si se produce sobre entrenamiento? 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.

No entiendo porque, pero en mi caso siempre obtengo un 100% de acierto en el entrenamiento. A pesar de que obtengo ambos porcentajes de la misma forma en el método imprimirPorcentajes de la clase Adaboost.java. Sin embargo, sí que puedo detectar si se está produciendo sobre entrenamiento con la prueba de validación. Esto se refleja en el apartado donde modifico el bucle 2.

5.5. Cuestión 5

Comenta detalladamente el funcionamiento de AdaBoost teniendo en cuenta que tasa media de fallos obtienes para aprendizaje y test.

Como he dicho antes, obtengo siempre 100% de acierto en el entrenamiento, pero no así en el test. Sin embargo, vamos a explicar detalladamente el funcionamiento de este algoritmo:

Esta clase recibe dos parámetros, la lista de imagen que queremos utilizar para entrenar y el dígito sobre el que queremos trabajar. En primera instancia entramos en un bucle de parámetro variable. Una vez hemos entrado nos creamos una lista de clasificadores débiles y otra de enteros.

```
72          //BUCLE 1:
73          for(int m=0; m<400 /* valor experimental */;m++){
74              ClasificadorDebil debil;
75              ArrayList<ClasificadorDebil> debiles = new ArrayList<>();
            List<Integer> resultados = null;
```

Ilustración 40

A continuación, entramos a un bucle donde iteramos un número variable también y que será el número de clasificadores débiles que contenga nuestro clasificador fuerte. Dentro de este bucle generamos clasificadores débiles al azar y nos quedamos de todos ellos con el mejor. Ahora asignaremos a ese mejor clasificador un valor de confianza basado en la confianza y siguiendo esta ecuación

$$\textbf{Start}$$
$$\alpha_t = 1/2 \log_2 \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$$
$$\textbf{End}$$

Ilustración 41

Una vez hemos obtenido la confianza de nuestro clasificador tendremos que ver que resultados nos ofrece para nuestro conjunto de test. En función de los resultados obtenidos tendremos que estudiarlos para actualizar los pesos de nuestra muestra y ajustarlo para la siguiente generación de clasificadores.

```
79              //BUCLE 2
80              for(int j = 0; j<1600; /*valor experimental*/ j++){
81                  debil =  generarClasifAzar(784);
82
83                  resultados = debil.aplicarClasificadorDebil(test);
84                  err = obtenerErrorClasificador(debil,resultados,test, d,digito);
85                  debil.error=err;
86
87                  debiles.add(debil);
88              }
89
90              debil = getMejor(debiles);
91
92              double confianza = 0.5*Math.log(((1-debil.error) / debil.error));
93              debil.confianza = confianza;
94              resultados = debil.aplicarClasificadorDebil(test);
95              actualizarPesos(confianza,digito,test,resultados);
96
97
98              fuerteAlg.add(debil);
99
100             i++;
101         }
```

Ilustración 42

Para el ajuste de pesos seguimos la siguiente ecuación matemática:

13: **End**

14: Actualizar D_{t+1}

15: **Start**

16: $D_{t+1} = \frac{D_t(i) \cdot e^{-\alpha_t \cdot y_i h_t(x_i)}}{Z_t}$

17: $Z_t = \sum_i D_t(i)$

18: **End**

Ilustración 43

Cuyo equivalente en código sería:

```
118     public void actualizarPesos(double confianza, int digito, List<Imagen> test,
119             List<Integer> resultados) {
120
121         double nuevoD;
122         double z = 0;
123
124         for(int i=0; i<d.size();i++){
125             int res = resultados.get(i);
126             int y = 1;
127             if((res == 1 && test.get(i).numero != digito) ||
128                 res == -1 && test.get(i).numero == digito){
129                 y=-1;
130             }
131             nuevoD = d.get(i)*Math.exp(-confianza*y);
132             d.set(i, nuevoD);
133             z+=nuevoD;
134         }
135         for(int j=0; j<d.size();j++){
136             d.set(j, d.get(j)/z);
137         }
138
139     }
```

Ilustración 44

Básicamente lo que hacemos es cuanto más equivocado esté nuestro algoritmo a la hora de clasificar un dígito más peso tendremos que asignarle a ese error.

Una vez hallamos hecho todos los ajustes se nos generará un clasificador fuerte para el dígito que recibió AdaBoost por parámetro.

5.6. Cuestión 6

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

Esta cuestión la respondo al final del apartado de fase de carga.

6. Conclusión

En conclusión, esta práctica basada en el aprendizaje supervisado de un sistema de reconocimientos manuscritos deja patente que el correcto ajuste de parámetros experimentales, así como la elección correcta de la distribución entrenamiento-validación resulta crucial para la optimalidad del sistema. También es importante no descuidar los tiempos de ejecución ya que si fuera un sistema en producción el cliente querría que el sistema respondiese de forma eficaz por tanto también es un factor para tener en cuenta.