Práctica 2 Sistemas Inteligentes

Algoritmo Adaboost para un simple detector de imágenes

Introducción

Se nos pide implementar un detector de imágenes el cual utilizará el algoritmo **Adaboost** (Adaptative Boosting) usando como clasificadores una serie de hiperplanos generados aleatoriamente que separarán los píxeles de cada imagen a un lado o a otro de cada uno de éstos.

Algoritmo Adaboost

Su funcionamiento se basa en la **combinación de un conjunto de clasificadores simples ponderados**. A partir de un conjunto de entrenamiento, se le van asignando pesos (inicialmente equivalentes) a cada elemento, los cuales se van actualizando para cada próximo clasificador a generar de tal manera que se le vaya dando más peso a aquellos elementos que no hayan sido clasificados correctamente por el clasificador actual para que el próximo los tenga más en cuenta, realizando así lo que denominamos un **descenso por gradiente**.

Podemos entender de una manera visual su funcionamiento con este vídeo

Clasificador aleatorio: hiperplano

Necesitamos ser capaces de distribuir las imágenes en un espacio. Para ello, como **todas las imágenes son de tamaño 24*24** = 576, podemos definir cada imagen como un vector de 576 componentes, o lo que es lo mismo, una **n-tupla de tamaño 576**.

Tal y como podíamos observar en el vídeo anterior, para clasificar una serie de puntos en el espacio empleábamos una recta (con un clasificador complejo, una combinación de éstas) para separar un conjunto de puntos de otro. Sin embargo, no podemos representar una imagen como un punto y una recta que los separe, por lo que deberemos de generalizar este método a un **hiperplano de dimensión 576**, que son los píxeles que contiene la imagen.

```
private static final int DIMM = 24 * 24;
```

Si una recta tiene como ecuación Ax + By - C = 0; un plano, Ax + By + Cz - D = 0; la del hiperplano será $A_1x_1 + A_2x_2 + A_3x_3 + ... - C = 0$, siendo los **puntos** el **valor de cada píxel** y los **vectores** el **tipo de clasificación** posible. Almacenaremos cada punto y cada vector en arrays de tamaño 576 que son las dimensiones de las imágenes que tenemos en nuestra base de datos. La **constante** será otro dato que se calculará como la **suma de todos los puntos por su vector** correspondiente.

```
private final double[] vectors = new double[DIMM];
//Aunque usamos puntos, no los almacenamos. Más adelante se explica.
private double constant = 0d;
```

Como dijimos al principio, **la generación** de estos planos **será aleatoria**, por lo que generaremos valores para cada punto y su vector. **Los límites se establecerán entre 0 (negro en RGB) y 255 (blanco en RGB) para los puntos y -1 (NO-CARA) y 1 (CARA) para el vector**. De este modo, acotaremos la generación de hiperplanos para que todos los que generemos se queden cerca de las posiciones de las imágenes y su clase.

```
Hyperplane() {
  SplittableRandom random = new SplittableRandom(); //Generador de números bastante más eficiente que Random
  for (int i = 0; i < DIMM; i++) {
    vectors[i] = random.nextDouble(Cara.NO_CARA, Cara.CARA);
    //Sólo necesitamos los puntos para calcular la constante; no hace falta guardarlos
    constant += vectors[i] * random.nextInt(Cara.MIN_RGB, Cara.MAX_RGB);
  }
}
```

Una vez tengamos el hiperplano creado, podremos **evaluar una imagen** a través de él. Se basará en el cálculo de un valor numérico que será el **sumatorio del producto de cada píxel de la imagen por el vector correspondiente del hiperplano**. **El signo del resultado determinará el tipo** de la imagen que el hiperplano ha interpretado.

```
double evaluate(int[] pixels) {
  double evaluation = 0;
  for (int i = 0; i < DIMM; i++) evaluation += vectors[i] * pixels[i];
  return evaluation - constant;
}
```

Clasificador débil (simple): el mejor hiperplano de un conjunto

Una vez que tenemos el clasificador de imágenes creado, deberemos de poder **determinar qué tan bueno es para, a partir de un una serie de diferentes hiperplanos, quedarnos con el mejor de ellos**.

Lo primero que realizamos es añadir un nuevo atributo `error` al hiperplano que nos indique numéricamente qué tasa de error tiene en un conjunto de aprendizaje dado.

```
private double error; //tasa de error del hiperplano
double getError() { return error; }
void setError(double error) { this.error = error; }
```

Y una referencia al mejor hiperplano (el de menor error) que consigamos obtener:

```
private Hyperplane best;
Hyperplane best(){ return best; }
```

Así, dada la cantidad de hiperplanos a generar y un conjunto de aprendizaje, iremos generándolos secuencialmente, calcularemos cúantas imágenes falla y nos quedaremos con aquel que haya fallado menos:

```
WeakLearner(int n, ArrayList<Cara> faces) {
  for (int i = 0; i < n; i++) {
    Hyperplane h = new Hyperplane();
    double error = 0d;
    for (Cara f : faces) if (pointLocation(h, f) != f.getTipo()) error++;
    h.setError(error);
    if (i == 0) best = h; //Si es el primer hiperplano que genero, será el único que tengo (de momento)
    else if (h.getError() < best.getError()) best = h;
  }
}
```

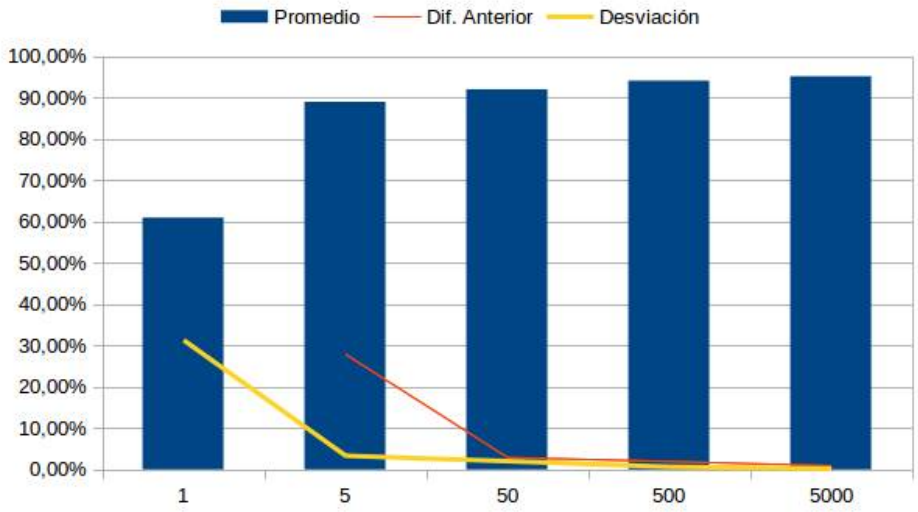
Para averiguar si un hiperplano ha fallado o no en la evaluación de la imagen, **llamamos al método de evaluación del hiperplano, nos quedamos con el signo** de su resultado (negativo -> no-cara; positivo -> cara) **y lo comparamos con el verdadero tipo** de la imagen (esto último lo hacemos en el método anterior).

```
int pointLocation(Hyperplane h, Cara c) { return h.evaluate(c.getData()) > 0d ? Cara.CARA : Cara.NO_CARA; }
```

Probando la cantidad de hiperplanos que debemos de generar para que el clasificador simple tenga algún sentido, hemos obtenido los siguientes resultados (partiendo de la premisa de escoger un 70% de las imágenes para aprendizaje y el resto para test, que es la elección más común):

Hiperplanos	Promedio	Dif. Anterior	Desviación
1	60,97%		31,46%
5	88,99%	28,02%	3,48%
50	92,05%	3,06%	2,17%
500	94,14%	2,09%	0,81%
5000	95,18%	1,04%	0,36%

Gráficamente se puede observar con mayor facilidad:



Podemos observar cómo **500 hiperplanos es la mejor elección**, pues el índice de aciertos para las imágenes de test es bastante alto y estable (una desviación típica menor a 1%). Con 5000 hiperplanos también obtenemos muy buenos resultados, el problema es que tarda demasiado tiempo en comparación con el anterior, y la ganancia que obtiene no merece tanto la pena.

En lo que a tiempo respecta, tarda **aproximadamente unos 7*10^-3 segundos en generar cada hiperplano** (y unos 0'016 para 5, 0'094 para 50, 0'774 para 500 y 6'949 para 5000); obviamente, cuantos más hiperplanos queramos, más tiempo necesitaremos. **Para la parte de aprendizaje**, esto debería de sernos totalmente **indeferente**, siempre y cuando **para la parte del test** el clasificador tenga un tiempo de respuesta **rápido**.

Clasificador fuerte (complejo): clasificadores débiles ponderados

Ahora que ya tenemos nuestro clasificador débil operativo, pasaremos a emplear el algoritmo Adaboost.

Primero deberemos de establecer **cómo vamos a asignarle el peso a una imagen**:

- Una opción es disponer de un **array** de pesos cuya posición se asocie a la de cada imagen. Una opción totalmente válida que he visto en muchas implementaciones.

```
//(Dentro del constructor)
double[] pesos = new double[faces.size()];
```

- Otra es **agregarle a la clase `Cara` un nuevo atributo `peso`** que sea el peso que Adaboost asocie a cada imagen. De esta forma (que es la que he escogido) hacemos uso de una de las características más importantes de la programación orientada a objetos: la Encapsulación.

```
private double peso = 1;
double getPeso() { return peso; }
void setPeso(double value) { peso = value; }
```

Ahora que las imágenes tienen peso, **la contabilización de errores de un hiperplano que realizaba el clasificador débil ahora se calculará en función del peso de cada una de éstas**.

```
//ANTES:
for (Cara f : faces) if (pointLocation(h, f) != f.getTipo()) error++
//AHORA:
for (Cara f : faces) if (pointLocation(h, f) != f.getTipo()) error += f.getPeso();
```

Así, fallar una imagen tendrá más o menos importancia que fallar otra. Sin embargo, cuando apliquemos únicamente el clasificador débil seguirá funcionando igual que antes debido a que el peso que recibe cada imagen por defecto es 1 (el mismo que se acumulaba antes por cada error).

Por último, como el **clasificador complejo** será una **combinación ponderada de clasificadores simples**, almacenaremos todos ellos en un ArrayList:

```
private final ArrayList<WeakLearner> weakLearners;
```

Ahora podemos aplicar **Adaboost**, cuyo algoritmo en **pseudo-código** es el siguiente:

```
· Inicializamos distribución de pesos D_1(i) = 1/N sobre el conjunto de entrenamiento
· para t=1 hasta T
  · Entrenar clasificador débil h_t a partir de D_t
  · Calcular el valor de confianza para ht: α_t = 1/2 Ln(1-ε_t/ε_t ) donde ε_t = Pr_{Dt}[h_t(x_i) != y_i]
  · Actualizar distribución D sobre el conjunto de entrenamiento:
      D_{t+1}(i) = (D_t(i)*e^(-α_t*y_i*h_t(x_i)))/(Z_t)
    Donde:
      D_{t+1}(i) es el peso de cada imagen para el próximo clasificador (los que se evaluarán en la próxima iter
      D_t(i) es el peso de cada imagen del clasificador actual
      α_t es la confianza del clasificador actual
      y_i es el tipo de imagen
      h_t(x_i) es el tipo de imagen que el clasificador actual cree que tiene
      Z_t es una constante de normalización de los nuevos pesos (el sumatorio de todos ellos)
  · Evaluar la cantidad de aciertos que tiene el clasificador fuerte actual. Si lo ha adivinado todo, salir del
· fin_para
· devolver clasificador fuerte
```

Y la **implementación en Java** que he realizado, ésta:

```
StrongLearner( ArrayList<Cara> faces, ArrayList<Cara> facesTest, int numWeakClassifiers, int numHyperplanes) {
  weakLearners = new ArrayList<>(numWeakClassifiers);
  for (Cara f : faces) f.setPeso(1d/faces.size());
  for (int t = 0; t < numWeakClassifiers; t ++) {
    WeakLearner learner = new WeakLearner(numHyperplanes, faces);
    weakLearners.add(learner);
    double Z_t = 0d;
    for (Cara f : faces) {
      f.setPeso(f.getPeso() * Math.pow(
          Math.E, -learner.getConfidence() * f.getTipo() * learner.pointLocation(learner.best(), f)
      ));
      Z_t += f.getPeso();
    }
    for (Cara f : faces) f.setPeso(f.getPeso() / Z_t);
    int[] stats = getClassifiersMatches(faces);
    if(stats[0] == faces.size()) break;
  }
}
```

El método `getClassifiersMatches` **comprueba la cantidad de aciertos o fallos** (tanto de caras como de no-caras) y devuelve los tres valores (que luego usamos para mostrarlos a modo debug):

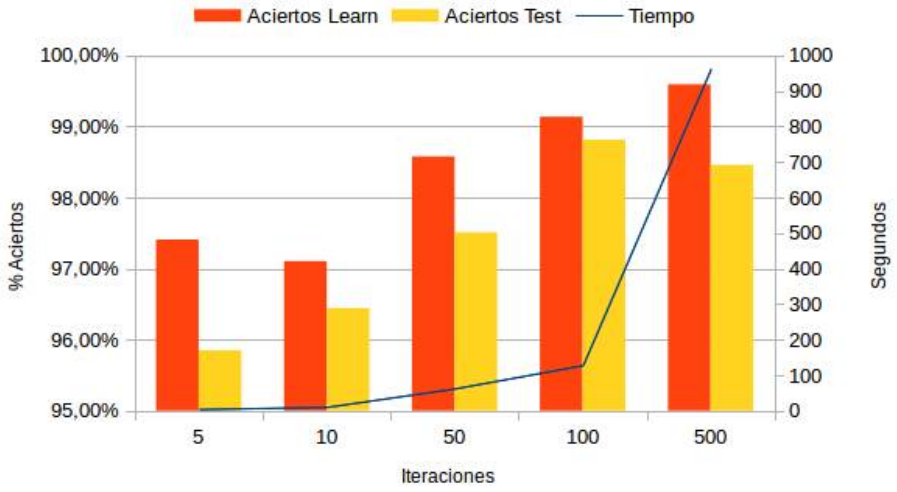
```
private int[] getClassifiersMatches( ArrayList<Cara> faces) {
  int matches, facesErrors, noFacesErrors;
  matches = facesErrors = noFacesErrors = 0;
  for (Cara f : faces) {
    if (pointLocation(f) == f.getTipo()) matches ++;
    else if (f.getTipo() == Cara.CARA) facesErrors ++;
    else noFacesErrors ++;
  }
  return new int[] {matches, facesErrors, noFacesErrors};
}
```

Y `pointLocation` es el **análogo al del clasificador débil**, excepto por que ahora el resultado es la **suma ponderada de todos los clasificadores por su confianza**:

```
int pointLocation( Cara f) {
  double sum = 0d;
  for (WeakLearner learner : weakLearners) sum += learner.getConfidence() * learner.best().evaluate(f.getData
  return sum > 0d ? Cara.CARA : Cara.NO_CARA;
}
```

Partiendo de la base de que usamos **500 hiperplanos por cada clasificador** (como bien decidimos tras las anteriores pruebas) **y una proporción 70%-30%** para el aprendizaje y el test, veamos qué **resultados** obtenemos en función de las iteraciones que el algoritmo realice:

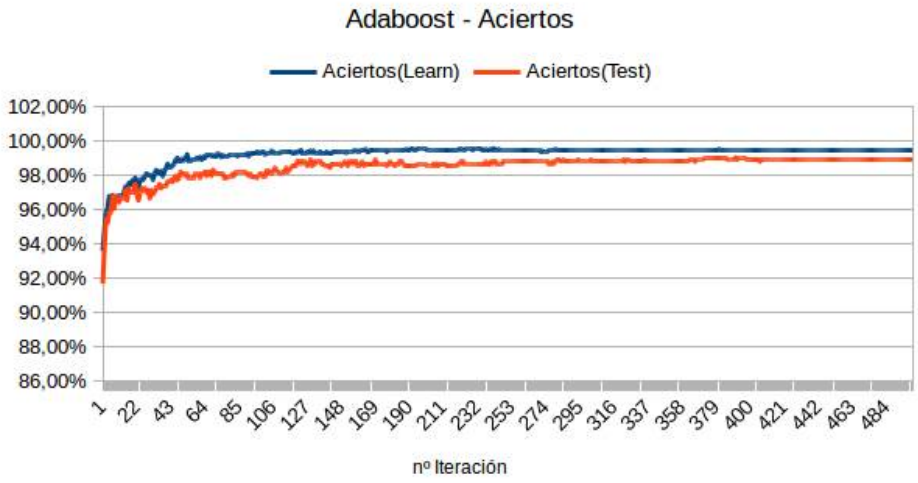
Iteraciones	Tiempo	Aciertos Learn	Aciertos Test
5	4,872	97,41%	95,85%
10	10,968	97,10%	96,44%
50	63,049	98,58%	97,51%
100	128,545	99,14%	98,81%
500	963,642	99,59%	98,46%



Como podemos observar, **con 100 iteraciones ya hemos obtenido un resultado más que aceptable**. Con 500 vemos cómo apenas mejora e incluso los aciertos de las imágenes para test disminuyen (aunque apenas lo haga). En lo que a tiempo respecta, no nos supone un factor muy importante por lo mencionado anteriormente, pero cabe destacar la **drástica subida de tiempo entre 100 y 500 iteraciones** debido a que, como la cantidad de clasificadores débiles va aumentando, el método `pointLocation` cada vez se va volviendo más lento y costoso.

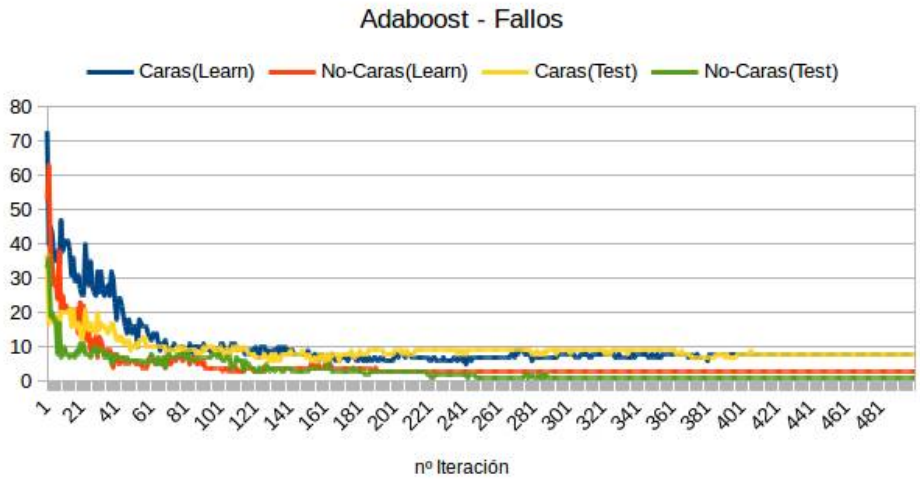
Profundizando en Adaboost

A modo de debug y con el fin de observar el funcionamiento y evolución del clasificador fuerte, mostramos por pantalla el **porcentaje de aciertos y la cantidad de imágenes falladas (caras y no-caras) con 500 iteraciones**. Esto es lo que hemos podido observar:



Aquí comprobamos cómo **cada vez la tasa de acierto es superior**, teniendo una subida muy drástica al inicio de la ejecución. Más tarde, **sobre la iteración 150, se estabiliza**. A partir de la iteración 200 la tasa apenas varía, y lo mismo le ocurre con el test. Podemos decir, entonces, que **este clasificador, utilizando una proporción 70-30 de imágenes para aprendizaje y test, no sufre de sobreentrenamiento**, pues ambas tasas se mantienen a lo largo de las iteraciones sin apenas cambios, y los pocos que

se observan son siempre al alza. **Esto posiblemente no sea así con otras proporciones y, sobre todo, utilizando un conjunto de imágenes más variado**.



Y aquí vemos cómo en todo momento **cuando el clasificador falla** (el porcentaje restante al de los aciertos) **lo hace siempre en base a caras**; detecta más fácilmente la ausencia de éstas. Si bien **la diferencia entre los fallos de las imágenes de aprendizaje y test al inicio son muy acusadas, con el paso de las iteraciones van reduciendo distancias** (el clasificador "aprende" correctamente). Por otra parte, a partir de la iteración 200 la tasa de fallos se mantiene bastante estable, quedando prácticamente intacta a partir de la 300.

Conclusiones

El algoritmo **parece funcionar bastante bien... Sobre el conjunto de imágenes que tenemos** disponible, el cual es minúsculo y demasiado repetitivo. En la vida real, para bien o para mal, las caras son muy distintas y suelen ser fotografiadas bajo multitud de ángulos y expresiones. Dudo mucho que esta propuesta fuese capaz de detectar esos casos más complicados (lo comprobaremos en la parte optativa).

Por otra parte, **el tiempo de aprendizaje es, con todo, bastante largo**. Sin embargo, **este algoritmo parece ser un buen candidato para ser paralelizado**, y así lo demuestra este ***paper*** en donde explican detalladamente **cómo implementar este algoritmo en un entorno paralelo y distribuido** (una joyita).

Referencias

1. Apuntes de clase y explicaciones del profesor
2. Discusiones y resoluciones de dudas entre los compañeros de clase
3. Un poco de teoría
4. Un ejemplo de implementación
5. Otro ejemplo

Pavel Razgovorov (pr18@alu.ua.es)