Técnicas de Inteligencia Artificial

Segunda práctica

Aprendizaje

Cristian Aguilera Martínez

19 de mayo de 2010

Índice

1. Detalles de diseño e implementación.....3
1.1. ImageFilter, Cara y Main.....5
1.2. Hiperplano.....7
1.3. Clasificador débil.....8
1.4. Clasificador fuerte.....9
1.5. AdaBoost.....10

2. Experimentación.....11

3. Perceptrón simple.....12

1. Detalles de diseño e implementación

El objetivo de esta práctica es implementar el algoritmo AdaBoost que construya un sistema de aprendizaje para predecir si una cara está de frente o de perfil.

Para el desarrollo de esta práctica se ha partido desde un proyecto en Eclipse que ha proporcionado el profesorado de la asignatura. Este proyecto incluye algunas clases ya completadas y otras en las que se debe añadir el código.

En los siguiente apartados se estudia cada una de las clases que componen el código del proyecto. Ver la figura 1 para ver cómo se relacionan cada una de las clases y qué dependencias aparecen (sobre todo, existen dependencias con la clase Cara, que se utiliza en todas las clases).

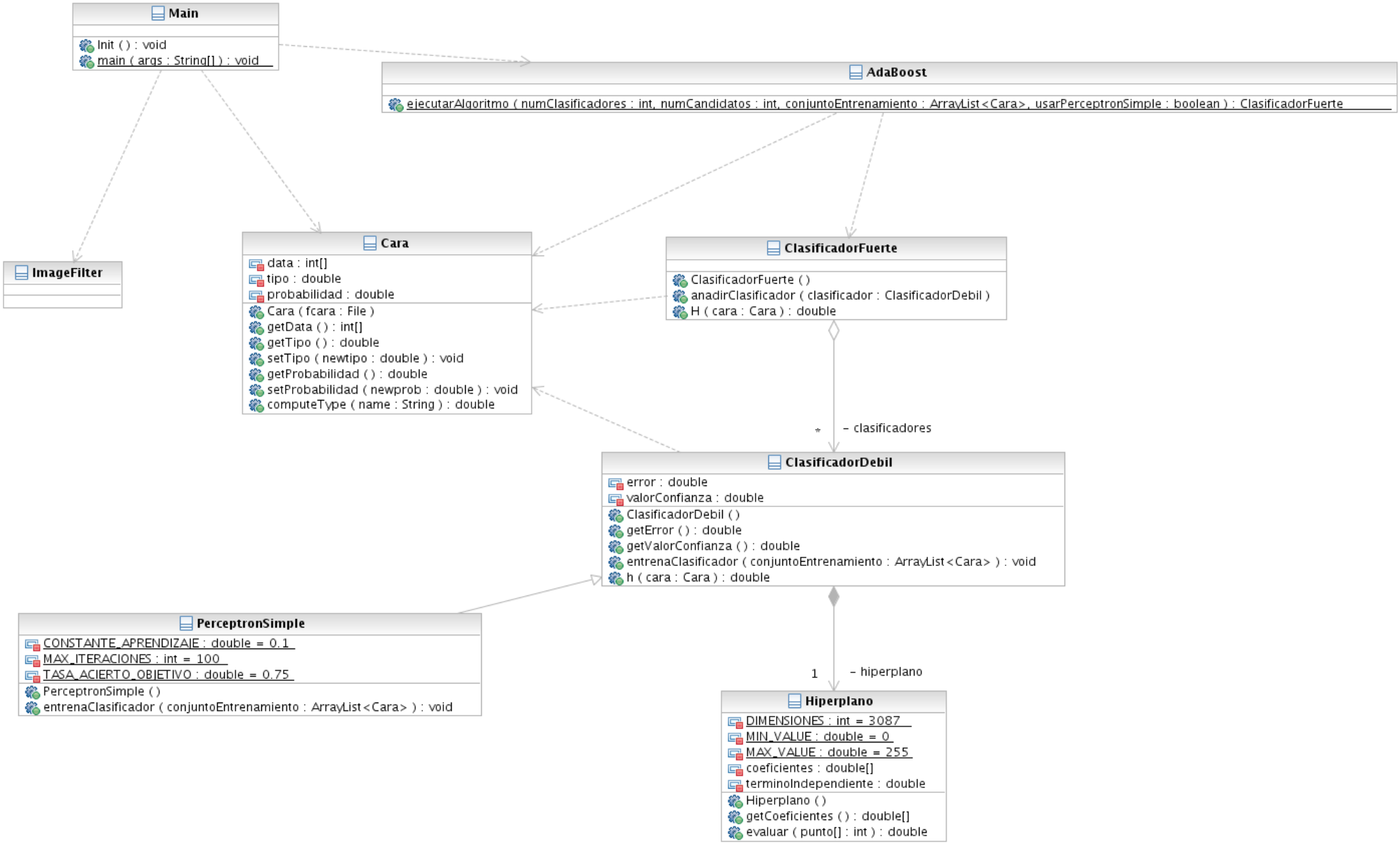

Figura 1: Diagrama de clases.

1.1. ImageFilter, Cara y Main

La clase Cara representa una imagen que contiene la foto de una cara de frente o de perfil. Además, la clase también presenta un atributo para saber si está de frente o no, así como un valor que indica el peso de la cara (que se tendrá en cuenta para entrenar los clasificadores). La clase ImageFilter se utiliza para identificar el formato de las imágenes de las caras que se van a cargar en los objetos Cara. Ninguna de estas clases se ha modificado y todos los métodos y atributos tienen el mismo valor que en el proyecto original de Eclipse.

En la clase Main, que es el punto de entrada del programa, se realiza la carga de las imágenes de las caras y se ejecuta el algoritmo AdaBoost. Además, se establecen los parámetros de ejecución tal como muestra la tabla 1. Como se verá en el apartado 3 de la parte optativa, también se han añadido otros parámetros para configurar el perceptrón simple (la regla delta).

Opción	Descripción
-t	*test rate*
-T	cantidad de candidatos (hiperplanos aleatorios) para cada clasificador
-c	cantidad de clasificadores para el clasificador fuerte

Tabla 1: Parámetros de entrada del programa.

Una vez que en el Main se cargan las imágenes de las caras, se introducen los objetos Cara en una lista. Esta lista se subdividirá en 2 partes atendiendo al valor *test rate* que se introduce en la entrada del programa. Una de las partes se utilizará para el entrenamiento del clasificador (aprendizaje basado en supervisión) y la otra parte se encargará de evaluar dicho clasificador con patrones que nunca ha visto en la fase de entrenamiento.

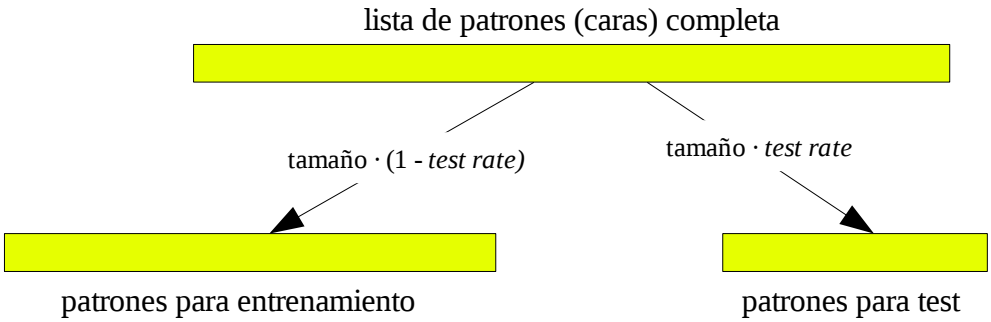

Figura 2: Subdivisión de la lista de patrones en 2 sublistas.

En total, se disponen de 311 caras diferentes. Al cargar las imágenes e introducirlas en una lis-

ta, ésta se reordenará aleatoriamente para evitar, en la medida de lo posible, que los resultados de diferentes ejecuciones sean similares. Para ellos se utiliza la función *shuffle()* de la clase Collections de Java, que permite reordenar una colección según una distribución aleatoria (supuestamente).

Cuando se disponga de una lista de entrenamiento y una lista de test, sólo falta invocar al algoritmo AdaBoost con la lista de entrenamiento para que calcule el clasificador fuerte y evaluar finalmente la lista de test. Al algoritmo AdaBoost hay que indicarle también los otros 2 parámetros: número de clasificadores y número de candidatos.

A continuación se muestra un pseudocódigo de cómo se ejecuta AdaBoost.

```
Lista<Cara> listaCompleta = cargarLista();
Collections.shuffle(listaCompleta);
Lista<Cara> listaEntrenamiento = subdividirLista(listaCompleta, 1 –
testRate);
Lista<Cara> listaTest = subdividirLista(listaCompleta, testRate);

ClasificadorFuerte cF = AdaBoost.ejecutarAlgoritmo(numClasificadores,
numCandidatos, listaEntrenamiento);

int aciertos = 0;
para cada Cara i de listaTest
{
     if (cF.H(i) == i.getTipo())
        aciertos++;
}
```

Pseudocódigo 1: Ejemplo de uso del algoritmo AdaBoost implementado.

Más adelante veremos que AdaBoost recibe un parámetro más que determina si se utiliza el perceptrón simple o no.

1.2. Hiperplano

Esta clase representa un hiperplano que utiliza una ecuación lineal para determinarse en el espacio multidimensional. Concretamente, estos planos tienen 3087 dimensiones debido a que los patrones se determinan con 3087 (cada imagen de una cara tiene un total de 63 x 49 = 3087 píxeles). A continuación se muestra la ecuación lineal del hiperplano de 3087 dimensiones, donde *a* son los coeficientes del vector normal del hiperplano y *D* es el término independiente.

$$a_1 x_1 + a_2 x_2 + a_3 x_3 + ... + a_{3087} x_{3087} + D = 0$$

Cuando se cree un hiperplano utilizando el constructor de la clase Hiperplano, éste se generará aleatoriamente:

1. Se traza un vector de dirección para la normal del hiperplano aleatoriamente y se normaliza para que su módulo sea 1. Cada componente de este vector normal se corresponde con cada componente de *a* en la ecuación lineal del hiperplano.
2. Se calcula aleatoriamente un punto en el hiperespacio por el que pase el hiperplano. Este punto estará limitado a los valores que pueden tomar los patrones en cada una de las dimensiones. En este caso, al tratarse de imágenes en blanco y negro, los valores que tome cada componente de dicho punto aleatorio estarán comprendidos entre 0 y 255 (256 tonos de gris diferentes).
3. Finalmente, se calcula el término independiente sustituyendo ese punto aleatorio en la ecuación lineal del hiperplano y resolviendo la incógnita *D*.

Con el vector de dirección normal y el término independiente, obtenemos la ecuación lineal del hiperplano. Si en esa ecuación sustituimos las incógnitas *x* por un punto cualquiera, podremos determinar si dicho punto está contenido en el hiperplano, o si está por encima o por debajo de él:

- Si al sustituir se obtiene 0, el punto está contenido en el plano.
- Si al sustituir se obtiene un valor positivo, el punto está por encima del plano.
- Si al sustituir se obtiene un valor negativo, el punto está por debajo del plano.

En la clase Hiperplano existe un atributo que contiene los coeficientes del vector normal y otra variable que contiene el término independiente. También posee un método (*evaluar()*) para comprobar un punto y determinar si está por encima o por debajo del hiperplano.

1.3. Clasificador débil

La clase ClasificadorDebil se compone de un Hiperplano. Cuando se construye la clase utilizando el constructor, se crea también un hiperplano aleatorio tal como se ha comentado en el apartado anterior.

El clasificador débil proporciona la función *h()* que recibe una parámetro de la clase Cara. Esta función evalúa la cara sobre el hiperplano y determina si está por encima del hiperplano o por debajo. Cuando esté por encima, se considerará que la Cara está de frente y si está por debajo se considerará que la cara está de perfil o medio perfil. Una Cara posee una secuencia de píxeles que es lo que se evalúa realmente sobre el hiperplano del clasificador, es decir, cada píxel de la imagen se corresponde con una componente del punto que se sustituye en la ecuación lineal del plano.

Para determinar cuánto de bueno es el clasificador, hay que entrenarlo según una lista de patrones de entrenamiento. Esta lista de entrenamiento contiene objetos de la clase Cara las cuales están de frente o de perfil. Además, se trata de un entrenamiento supervisado, por lo que cada Cara conoce cuál es su posición (con el método *getTipo()*). Se recorrerá la lista de Caras y se evaluará con la función *h()* comentada en el párrafo anterior. Si el resultado de *h()* no coincide con el tipo de la cara, se incrementa la tasa de error del clasificador (ϵ).

Como ya se comentó en los primeros apartados, cada Cara tiene un peso asociado que AdaBoost irá modificando en cada iteración. El error del clasificador se calcula sumando el peso de cada Cara (que se obtiene con *getProbabilidad()*) que falle a la hora de clasificarla durante el entrenamiento.

```
double error = 0;
for (Cara i : conjuntoEntrenamiento)
      if (h(i) != i.getTipo())
            error += i.getProbabilidad();
```

Pseudocódigo 2: Cálculo del error durante el entrenamiento de un clasificador débil.

A partir del error se obtiene un valor de confianza que se calcula con la expresión:

$$\alpha = \frac{1}{2}\log_{10}\left(\frac{1-\epsilon}{\epsilon}\right)$$

Es en la función *entrenaClasificador()* donde se recorre la lista de caras para entrenar el clasificador y fijar tanto el error como el valor de confianza.

1.4. Clasificador fuerte

El clasificador fuerte es un agregado de clasificadores débiles cuya función de predicción consiste en devolver el resultado en función de la mayoría obtenida entre todos los clasificadores. Supongamos que se evalúa una cara sobre un clasificador fuerte que dispone de 10 clasificadores débiles y que los resultados obtenidos son

- 8 clasificadores débiles dicen que la cara está de perfil
- 2 clasificadores débiles dicen que la cara está de frente

En ese caso, el clasificador fuerte dirá que la cara está de perfil puesto que hay mayoría.

Esta función está implementada en el método *H()* de ClasificadorFuerte. En esta implementación también se tiene en cuenta el valor de confianza del clasificador. Si un clasificador tiene un valor de confianza alto, será más relevante a la hora de tomar la decisión el clasificador fuerte; es decir, no depende sólo de lo que diga la mayoría sino, también, qué calidad tiene esa mayoría.

1.5. AdaBoost

Dado un conjunto de patrones de diferentes clases, este algoritmo se encarga de entrenar un clasificador que predice a qué clase pertenece un determinado patrón que no pertenece al conjunto con el que fue entrenado. El clasificador que devuelve el algoritmo está compuesto por múltiples clasificadores débiles que se entrenan uno a uno con el mismo conjunto de entrenamiento. Cada clasificador aporta cierta cantidad de decisión al clasificador final (clasificador fuerte) según la tasa de acierto que posee dicho clasificador débil. La cantidad de clasificadores débiles se define con un valor concreto y será uno de los parámetros de entrada del algoritmo.

A continuación se comentan brevemente los pasos del algoritmo para obtener el clasificador fuerte dado un conjunto de entrenamiento:

- Se inicializan los pesos de cada patrón del conjunto de entrenamiento. La suma de todos los pesos será 1 por lo que, si hay N patrones, cada uno pesará 1/N.
- El siguiente bucle repite tantas veces como clasificadores débiles se quieran obtener para generar el clasificador débil (*t* es el iterador del bucle):
    1. Se generan aleatoriamente múltiples clasificadores débiles tal como se ha explicado en el apartado del Hiperplano. De todos los que se generen, se seleccionará el que mejor resultados proporcione; es decir, el que produzca una tasa de aciertos sobre el conjunto de entrenamiento. Este clasificador (h_t) elegido al azar en esta iteración formará parte del clasificador fuerte y se entrenará con el conjunto de entrenamiento para calcular su error y el valor de confianza.
    2. Para dicho clasificador débil se calculará su valor de confianza en función de su error, tal como se ha explicado en el apartado de Clasificador débil.

$$\alpha_t = \frac{1}{2}\log_{10}\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

       Siendo α_t el valor de confianza del clasificador de esta iteración y ϵ_t el error del clasificador de esta iteración.

    3. Una vez se ha generado el clasificador débil de esta iteración, se actualizan los pesos de cada uno de los patrones en función de dicho clasificador. No interesa evaluar los patrones que son fáciles de clasificar, por lo tanto, aquéllos que se han clasificado correctamente en el clasificador débil decrementan su peso. En cambio, los que han fallado son más controvertidos y merecen la pena examinarlos mejor, por lo que se incrementa su peso. En la siguiente iteración el clasificador débil que se escoja al azar tendrá en cuenta estos nuevos pesos e intentará clasificar mejor aquéllos patrones cuya decisión es más costosa.

$$Z_t = \sum_{i=1}^{n} D_t(i)$$

Se calcula Z como un valor de normalización, siendo n el total de patrones y $D_t(i)$ el peso del patrón i en la iteración t.

Para cada patrón se actualiza su peso según la expresión:

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

Donde y_i es la clase del patrón real y $h_t(i)$ es la clase que devuelve el clasificador de esta iteración al evaluar el patrón i.

Cuando y_i no coincide con $h_t(i)$, el exponente de e toma un valor positivo y, si no, toma un valor negativo. Es por eso que, cuando no coincide, el peso del patrón se incrementa. Si coincide, decrementa por ser negativo.

4. Al final de esta iteración se comprueba qué tasa de error se obtiene al evaluar el conjunto de entrenamiento sobre el clasificador fuerte; es decir, sobre la totalidad de los clasificadores débiles. Si la tasa de error es 0, ya no es necesario repetir el bucle.

- El conjunto de clasificadores débiles es finalmente el resultado que devuelve el algoritmo AdaBoost.

La implementación del algoritmo se encuentra en un método estático de la clase AdaBoost. Este método recibe los parámetros siguientes:

- **numClasificadores**: cantidad de clasificadores que van a entrenarse para el clasificador fuerte.
- **numCandidatos**: cantidad de hiperplanos aleatorios que se generarán en cada iteración para obtener el clasificador débil de la misma.
- **listaEntrenamiento**: conjunto de patrones que se utilizarán en el entrenamiento.

Dicho método estático (*ejecutarAlgoritmo()*) devuelve finalmente un objeto de la clase ClasificadorFuerte.

2. Experimentación

3. Perceptrón simple

Descripción

Para la parte optativa de la práctica se pide la implementación de la regla delta para perceptrones de una única neurona para entrenar los hiperplanos del clasificador débil.

La regla delta permite ajustar iterativamente el hiperplano. Se asume que el incremento de los pesos es proporcional a la diferencia entre la salida obtenida por el clasificador débil y la salida que realmente se corresponde con la clase del patrón evaluado sobre el clasificador.

Al entrenar un clasificador con la regla delta, el hiperplano del clasificador se ajusta a la lista de entrenamiento. Es decir, se modifican las componentes del hiperplano para reducir el error que se genera con el conjunto de entrenamiento. Además, se realizan múltiples iteraciones con dos posibles condiciones de parada:

- Se supere un límite máximo de iteraciones.
- Se alcance una tasa de acierto objetivo (por ejemplo, un 90% de aciertos).

A continuación se muestra la regla delta con el significado de cada término:

$$w_i' = w_i + \eta(d-y)x_i$$

- w: coeficientes del vector normal del hiperplano del clasificador débil
- w_i: componente i del vector normal del hiperplano del clasificador débil
- η: constante de aprendizaje (entre 0 y 1 que se determina según experimentación)
- d: clase a la que pertenece el patrón (1 o -1 en nuestro caso)
- y: clase que el clasificador débil predice para el patrón (1 o -1 en nuestro caso)
- x: dimensiones del patrón
- x_i: componente i del punto en el que se sitúa el patrón

En el código de la práctica la clase PerceptronSimple hereda de la clase ClasificadorDebil, por lo que incluye todos los atributos y métodos de esta clase.

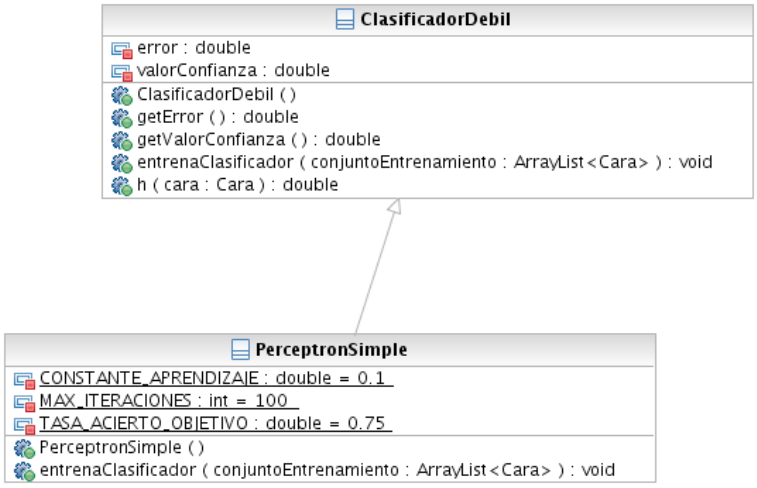

Figura 3: PerceptronSimple que hereda de ClasificadorDebil.

El método *entrenaClasificador()* se sobreescribe de la siguiente forma:

```
@Override
public void entrenaClasificador(ArrayList<Cara> conjuntoEntrenamiento) {
	int iteraciones = 0;
	double tasaAciertos = 0;
	while (iteraciones < MAX_ITERACIONES && tasaAciertos < TASA_ACIERTO_OBJETIVO) {
		// Ajustamos el hiperplano asociado a la neurona.
		for (Cara i : conjuntoEntrenamiento) {
			double d = i.getTipo();
			double y = h(i);
			int[] x = i.getData();
			for (int j = 0; j < Hiperplano.DIMENSIONES; j++) {
				hiperplano.getCoeficientes()[j] = hiperplano.getCoeficientes()[j] +
CONSTANTE_APRENDIZAJE * (d - y) * x[j];
			}
		}

		// Incrementamos el número de iteraciones y recalculamos la tasa de aciertos.
		iteraciones++;
		tasaAciertos = 0;
		for (Cara i : conjuntoEntrenamiento) {
			if (h(i) == i.getTipo())
				tasaAciertos++;
		}
		tasaAciertos /= conjuntoEntrenamiento.size();
	}

	// Finalmente se calcula el error del igual forma que en el clasificador débil base.
	error = 0;
	for (Cara i : conjuntoEntrenamiento) {
		if (h(i) != i.getTipo())
			error += i.getProbabilidad();
	}
	valorConfianza = 0.5 * Math.log10((1 - error) / error);
}
```

En el código aparece un bucle *while* que representa el total de iteraciones que se aplican en el entrenamiento. En cada iteración se ajustará el hiperplano según el conjunto de entrenamiento.

Cada patrón del conjunto de entrenamiento modificará las componentes del hiperplano a placer, de forma que en cada iteración el hiperplano encaje mejor; es decir, la tasa de aciertos se incre-

mentará en cada iteración (hasta cierto punto, claro). Por lo tanto, con el perceptrón simple no sólo lanzamos hiperplanos aleatoriamente, sino que dichos hiperplanos se desplazan poco a poco hasta obtener mejores resultados.

La parte final del método coincide con el entrenamiento del ClasificadorDebil, en el que sólo se calculaba el error y el valor de confianza según el conjunto de entrenamiento.

Para configurar el perceptrón simple se han añadido 3 nuevos parámetros al Main:

Opción	Descripción
-n	constante de aprendizaje (entre 0 y 1)
-i	cantidad de iteraciones máximas
-o	tasa de acierto objetivo (entre 0 y 1)

Tabla 2: Más parámetros del punto de entrada.

Estos 3 valores se reciben en el punto de entrada y se establecen en 3 atributos estáticos de la clase PerceptronSimple. Cualquier instancia de PerceptronSimple que se ejecute utilizará dichos valores. Además, ahora AdaBoost recibe un cuarto parámetro de tipo booleano que indica si ha de utilizarse PerceptronSimple o ClasificadorDebil. Cuando en el Main se especifique alguno de los 3 parámetros anteriores, AdaBoost se ejecutará con el PerceptronSimple.

Como PerceptronSimple hereda de ClasificadorDebil, en la implementación AdaBoost puede aplicarse la técnica de polimorfismo para ejecutar el método *entrenaClasificador()*. Independientemente del tipo de clasificador, éste se entrenará según el comportamiento que defina el método *entrenaClasificador()*. Es decir, en la implementación de AdaBoost sólo cambia el hecho de crear el clasificador débil; o bien se utiliza el constructor de ClasificadorDebil o bien se utiliza el constructor de PerceptronSimple, pero el resto de código no se modifica porque el polimorfismo hace el resto.