Práctica 2 Optativa Sistemas Inteligentes

Detección de caras usando el clasificador entrenado en la anterior práctica

Introducción

Una vez tenemos nuestro algoritmo **Adaboost implementado**, tenemos que implementar un pequeño **programa que lo ponga en funcionamiento**. Para ello, deberemos de poder **guardar el clasificador** que nos creamos **para más adelante poder cargarlo**. El nuevo programa leerá el clasificador y una lista de imágenes y, por cada cara que detecte, pintará un recuadro alrededor. Las imágenes procesadas también se almacenarán en el disco para que podamos ver el resultado.

Antes de comenzar

Para poder implementar la nueva práctica, tenemos que hacerle unos pequeños arreglos a la anterior. Veamos cuáles son:

- Debemos de **poder guardar y leer el clasificador generado**, esto es, hacer que sea persistente. Para ello, simplemente **hemos guardado toda la información** relevante de éste (los vectores y la constante de cada mejor hiperplano de cada clasificador débil que forma parte del fuerte) **en un archivo JSON** haciendo uso de la librería Gson gracias a la cual con tan sólo un par de líneas de código podemos guardar y leer todos lo datos de la clase `StrongLearner` sin tocar nada del código original.

```
public void saveLearner(String fileName, boolean pretty) throws IOException {
  Gson gson = pretty ? new GsonBuilder().setPrettyPrinting().create() : new Gson();
  JsonParser jp = new JsonParser();
  JsonElement je = jp.parse(gson.toJson(this));
  Files.write(Paths.get(fileName), gson.toJson(je).getBytes());
}

public static StrongLearner loadLearner(String fileName) throws FileNotFoundException {
  Gson gson = new Gson();
  JsonReader reader = new JsonReader(new FileReader(fileName));
  return gson.fromJson(reader, StrongLearner.class);
}
```

- Para que al ejecutar el programa nos guarde (y lea) el clasificador generado, hemos añadido este código justo debajo de donde llamábamos al constructor del clasificador:

```
try {
  System.out.println("Guardando en disco... ");
  learner.saveLearner("learner.json", true);
  System.out.println("Leyendo de disco... ");
  learner = StrongLearner.loadLearner("learner.json");
} catch (IOException ex) {
  Logger.getLogger(Practica2SI.class.getName()).log(Level.SEVERE, null, ex);
}
```

- **A la hora de generar el clasificador, íbamos probando iteración a iteración qué tan bueno era** (aciertos, fallos de cara y fallos de no-cara sobre los dos conjuntos de imágnes), **cosa que consumía algo de tiempo**. Para evitar esta sobrecarga, **hemos creado un nuevo atributo para el programa,** `VERY_VERBOSE` (que se activa con el argumento "-V"), **al cual accede el clasificador fuerte para evaluar si queremos mostrar el progreso de aprendizaje o no**. Además, como antes comprobábamos si había acertado todas mediante éstos datos calculados, ahora lo evaluamos a través del **error del clasificador débil actual** encontrado (**si el error es 0, ha acertado todas** las imágenes). Esto lo hemos realizado para poder guardar varios clasificadores en el menor tiempo posible.

```
//ANTES:
int[] stats = getClassifiersMatches(faces);
int[] testStats = getClassifiersMatches(testFaces);
System.out.println((t+1) + "\t" + format.format((double)stats[0]/faces.size()) + "\t" + stats[1] + "\t" + sta
        + "\t" + format.format((double)testStats[0]/testFaces.size()) + "\t" + testStats[1] + "\t" + testStat
```

```
if(stats[0] == faces.size()) break;

//AHORA:
if (Practica2SI.VERY_VERBOSE) {
  int[] stats = getClassifiersMatches(faces);
  int[] testStats = getClassifiersMatches(testFaces);
  System.out.println((t+1) + "\t" + format.format((double)stats[0]/faces.size()) + "\t" + stats[1] + "\t" + s
      + "\t" + format.format((double)testStats[0]/testFaces.size()) + "\t" + testStats[1] + "\t" + testStats[
}
if (learner.best().getError() == 0) break;
```

- Para detectar que el usuario quiere que se muestre la evolución del aprendizaje, hemos añadido un nuevo caso al `switch` del `main`:

```
case 'V':
    VERY_VERBOSE = true; //Por defecto false
    paso = 1;
    break;
```

- Por último, **a la hora de analizar la imagen en nuestro nuevo programa deberemos de poder trabajar con sub-imágenes cuyo tipo desconocemos**; por tanto, **hemos añadido un nuevo constructor de `Cara`** que tan sólo recibe el vector de píxeles y le aplica la máscara (al igual que se hacía anteriormente). De esta forma **podremos pasarle al clasificador que leamos cada sub-imagen que extraigamos y que éste nos diga de qué tipo es**.

```
public Cara(int[] data) {
  for (int i = 0; i < data.length; i++) data[i] &= 0x000000FF;
  this.data = data;
}
```

Implementación de la nueva práctica

Una vez terminado de editar **el anterior proyecto, lo exportamos como jar** para que nos sirva de librería. Con ella, **agregamos una referencia al clasificador**, que deberemos de leer obligatoriamente para que el programa se ejecute:

```
static StrongLearner learner;
public static void main(String[] args) {
  if (args.length >= 1) {
    try {
        learner = StrongLearner.loadLearner(args[0]); //El clasificador es siempre el primer argumento
        for...
    } catch (FileNotFoundException e) {
      System.err.println("JSON del clasificador no encontrado ( " + new File(args[0]).getAbsolutePath() + ")")
    }
    catch...
  } else {
    System.err.println("Parámetros inválidos. Uso: \n\t"
      + Practica2SIOptativa.class.getSimpleName().intern() //Nombre de la clase simplificado
      + " (<Archivo-JSON>) [<Imágenes>...] ");
  }
}
```

Cuando ya ha leído el clasificador, pasa a leer las imágenes (si es que las hay) que ha recibido por argumento:

```
...
  for (int i = 1; i < args.length; i++) {
    ImageWithFaces imageWithFaces = new ImageWithFaces(new File(args[i]));
    imageWithFaces.detectFacesAndPaint();
    //El nombre será sólo el nombre del fichero, quitando la ruta
    imageWithFaces.save(new File(args[i]).getName());
  }
...
} catch (IOException e) {
  System.err.println("Error de ficheros de imagen:  " + e.getLocalizedMessage());
}
```

Hemos creado la clase `ImageWithFaces` que es quien **se encarga de encapsular toda la lógica de negocio relacionada con leer la imagen, analizarla, extraer las subimágenes, comprobar si son caras** (en caso de que lo sean, pintar un recuadro alrededor de ellas) **y de guardar el resultado**. Pasemos a ver cada funcionalidad:

- Para leer una imagen, usamos la clase **BufferedImage**, al igual que en la clase `Cara`. Para ser exactos, **leemos la imagen dos veces**: una será para el análisis y otra en donde iremos pintando los recuadros (si lo hiciéramos en una, por cada recuadro pintado estaríamos manchando y, por tanto, dificultando la detección de las próximas caras).

```
private BufferedImage img, copy;
ImageWithFaces(File fimg) throws IOException {
  try {
    img = ImageIO.read(fimg);
    copy = ImageIO.read(fimg);
  } catch (IOException e) {
    throw new IOException("No se puede abrir " + fimg.getAbsolutePath());
  }
}
```

- **Para analizar la imagen** e ir substrayendo partes de ella símplemente **la recorremos como si de una matriz se tratara**. **Por cada píxel recorrido, extraemos una sub-imagen de tamaño 24x24** de los píxeles adyacentes, la cual encapsulamos en una instancia de `Cara`. A continuación, la pasamos por el clasificador y, si la detecta como una cara, pintamos un recuadro de color rojo.

```
void detectFacesAndPaint() {
  for (int x = 0; x < img.getWidth(); x++)
    for (int y = 0; y < img.getHeight(); y++)
      //Si tengo hueco para buscar la cara y ésta efectivamente lo es
      if (x <= img.getWidth() - 24 && y <= img.getHeight() - 24 &&
            learner.pointLocation(new Cara(getNext24x24MaskedPixels(x, y))) == Cara.CARA)
        paint24x24Edge(x, y, Color.RED);
}
```

- El método `getNext24x24MaskedPixels` es quien se encarga de extraer los 24x24 píxeles adyacentes para poder formar el array `data` de la instancia de `Cara`.

```
private int[] getNext24x24MaskedPixels(int x, int y) {
    int[] subdata = new int[24 * 24];
    int count = 0;
    //No sé por qué pero hay que acceder de forma traspuesta
    for (int i = y; i < y + 24; i++)
        for (int j = x; j < x + 24; j++)
            subdata[count++] = img.getRGB(j, i);
    return subdata;
}
```

- Y `paint24x24Edge` simplemente pinta un recuadro en los 24x24 píxeles adyacentes.

```
private void paint24x24Edge(int x, int y, Color color) {
  //No sé por qué pero hay que acceder de forma traspuesta
  for (int i = y; i < y + 24; i++)
    for (int j = x; j < x + 24; j++)
      if ((i == y || i == y + 23) || (j == x || j == x + 23)) //Sólo pintar bordes
        copy.setRGB(j, i, color.getRGB());
}
```

- Por último, para guardar la imagen pintada, simplemente escribimos los datos de `copy` en en el directorio `media/salida_imagenes/` (si no existe lo crea).

```
void save(String fileName) throws IOException {
try {
  File dir = new File("media/salida_imagenes/");
  if (dir.mkdirs()) System.out.println("Ruta " + dir + " inexistente creada");
  fileName = dir + "/" + fileName;
  //Como formato cojo el nombre de la extensión
  ImageIO.write(copy, fileName.substring(fileName.lastIndexOf('.') + 1), new File(fileName));
} catch (IOException e) {
```

```
    throw new IOException("No se puede guardar  " + new File(fileName).getAbsolutePath());
  }
  }
```

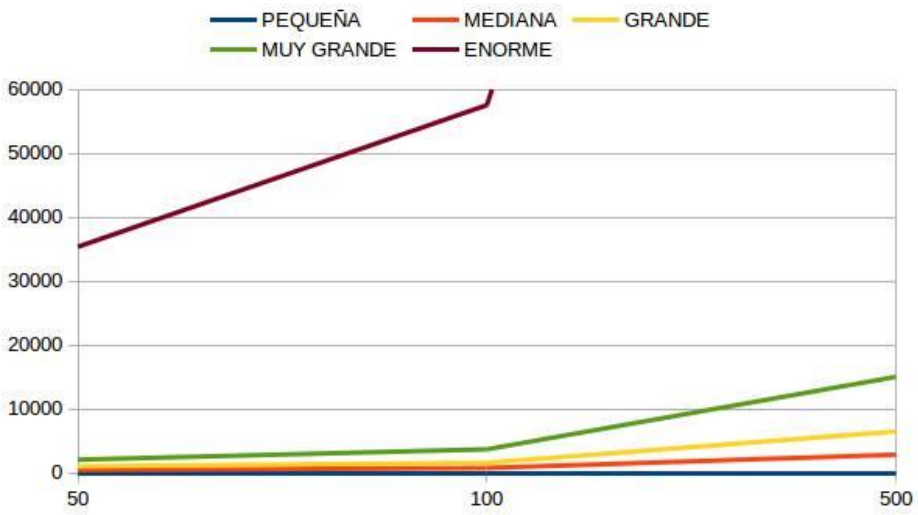
Pruebas de rendimiento

Para realizar las pruebas de rendimiento, hemos elegido **dos parámetros** que influyen en el coste computacional del programa: la **cantidad de clasificadores débiles** que tiene el clasificador y el **tamaño de la imagen** a procesar. Para realizar esta prueba, hemos elegido las siguientes muestras (son fotos de la base de datos combinadas; la última un equipo de fútbol):

El resultado ha sido el siguiente:

CLASIFICADORES	PEQUEÑA	MEDIANA	GRANDE	MUY GRANDE	ENORME
50	18	513	1074	2105	35394
100	27	805	1614	3722	57554
500	50	2888	6522	15054	251325

*el tiempo es en ms

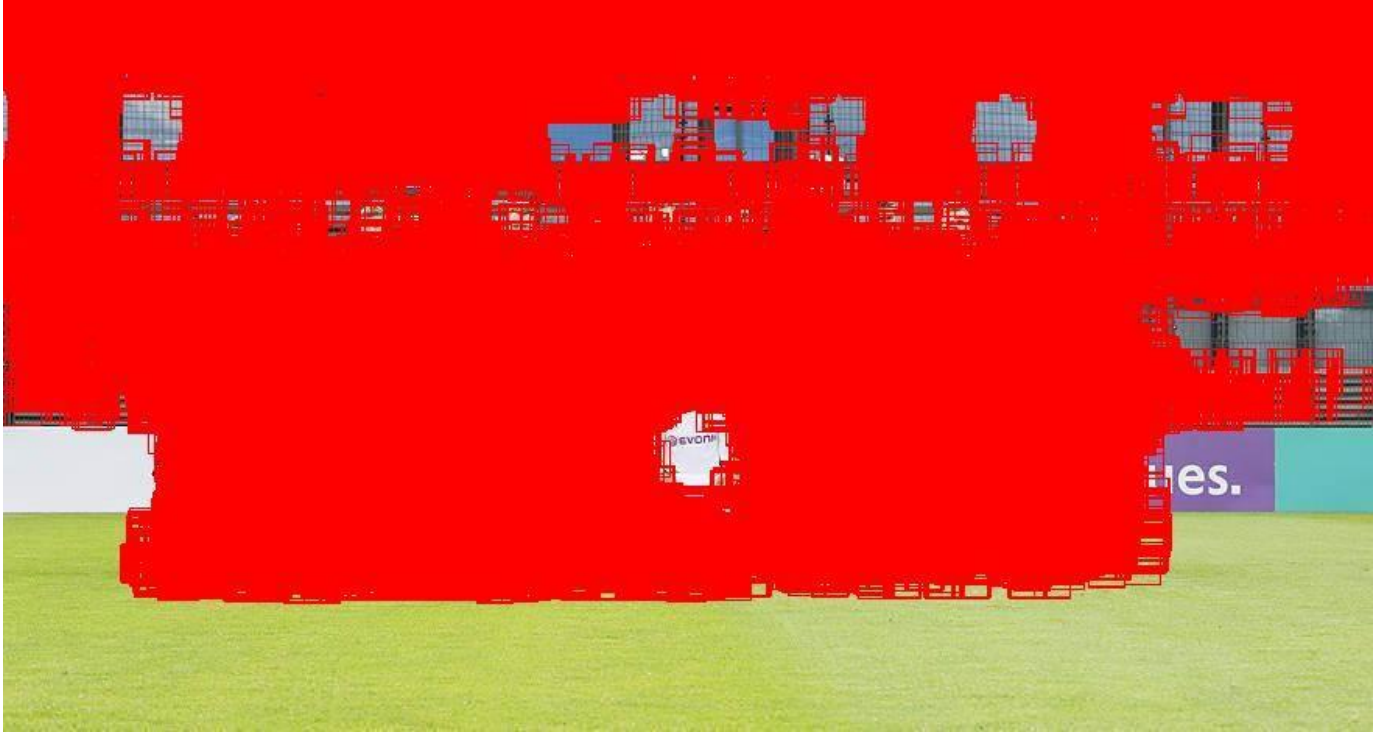


Como podemos observar, **el análisis de las imágenes es extremadamente lento** para todos los casos. Teniendo en cuenta que las muestras tienen un tamaño minúsculo (incluso las páginas web tienen imágenes de mayor resolución), **la aplicación real de este clasificador resulta impracticable** (el objetivo de respuesta rápida de este sistema no se cumple ni de lejos).

Análisis de los resultados

Probamos a analizar la imagen del equipo de fútbol (caras que no están en la base de datos) para ver qué tal funcionaba el clasificador. Sin comentarios...

Después de mi descontento por este resultado, pensé en que quizás **la base de datos de la que disponíamos era muy pequeña**, así que **busqué nuevas muestras para agrandarla** (y después las pasé a escala de grises y redimensioné a 24x24 píxeles), consiguiendo un total de 3.061 caras y 10.529 noCaras. **Volví a entrenar los clasificadores**, que ahora tardaban mucho más (por ejemplo, para 500 iteraciones me llevó casi 65 minutos) **y probé a analizar la imagen con esta nueva base de datos**. El resultado fue incluso mejor que el anterior:



Ahora, al parecer, **sí tenía una base de datos algo más consistente, pero la proporción entre caras y noCaras estaba muy descompensada** (además, el propio clasificador no lograba acertar más de un 80% de las muestras en ninguno de los casos). Para resolverlo, **eliminé las nuevas caras dejando solamente las que venían en la base de datos original**, dejando finalmente 247 caras y 10.529 noCaras. **Entrené de nuevo un último clasificador de 500 iteraciones y el resultado final fue el siguiente**:

Efectivamente, **seguía siendo muy impreciso, pero detectaba la mayoría de caras** y, lo más importante, no se equivocaba con aquellas sub-imágenes que no lo eran (no, al menos, con tanta frecuencia). Posiblemente con una muestra de imágenes de mayor volumen hubiera sido más efectivo, pero no lo puedo afirmar con seguridad.

Conclusiones

Una vez hemos observado qué tal se comporta nuestro clasificador en un entorno algo cercano a la realidad, podemos decir que **su funcionamiento deja mucho que desear**. Además de ser lento e impreciso, el fundamento de la clasificación es totalmente aleatoria, cuyo funcionamiento resulta bastante sospechoso. Aun con todo, **en muchas de las páginas sobre Adaboost se habla muy bien del algoritmo diciendo que es el más usado y estudiado, y que se aplica en numerosos campos**, cosa que hace que piense que mi implementación no sea correcta, o quizá el problema radique en el tamaño de la muestra a partir de la cual aprendemos.

En la Wikipedia también aparecen otras **variantes de este algoritmo**, las cuales parecen solucionar algunos de los posibles problemas que tiene este algoritmo (como pueden ser las muestras con "ruido", casos especiales de clasificar). Por otra parte, si

buscamos sobre **detección de caras**, nos hablan de **"eigenfaces" o algoritmos genéticos**. Si buscamos también cómo realiza la **detección de caras** la librería de visión computacional **OpenCV** nos habla sobre **"clasificadores en cascada basados en caracterísitcas de Haar", y dice que "en cascada"** está relacionado con el Boosting (lo que hace pensar que con esta librería se ha implementado Adaboost mediante clasificadores algo más inteligentes que los nuestros).

En definitiva, **es posible que con un clasificador con Adaboost bien implementado podamos detectar bien las caras**.

Referencias

1. Gson (Descarga)
2. Imagen de prueba BVB
3. Lectura de imágenes para Java: me basé en el código de la clase Cara (y muchas preguntas en StackOverflow)
4. Reconocimiento de imágenes (según Wikipedia)
5. Paper con mucha información sobre Adaboost
6. Paper sobre detección de caras en tiempo real
7. Detección de caras en OpenCV (C++)
8. Detección de caras en OpenCV (Python)

Pavel Razgovorov (pr18@alu.ua.es)