Mayte Giménez

Prácticas de Biometría

17 de enero de 2014

Índice general

1 | Implementación de la curva ROC

En este primer ejercicio hemos implementado el método de verificación de sistemas informáticos consistente en la curva ROC. Este sistema de verificación nos permitirá de una manera visual concer las diferentes tasas de error en función del umbral escogido.

1.1. | Datos de entrada

Las distintas tasas (*scores*) del sistma biométrico las obtenemos mediante ficheros de texto plano, en un ficheros tendremos las tasas de los clientes y en otro las tasas de los impostores.

1.2. | Descripción del trabajo realizado

Hemos desarrollado un programa en python que lee los ficheros de entrada y que los almacena en una estructura de dattos adecuada para evaluar y consultar el sistema biomédico que estamos evaluando.

La estructura básica de esta clase es la que podemos ver en el segmento de código 3.1

Listing 1.1: EDA con la que gestionar la curva ROC

```
class RocData(object):
    def __init__(self, name="", c_score=None, i_score=None):
        self.name = name
        self.c_score = c_score
        self.i_score = i_score
        # Get all possible threasholds and inserts a 0.
        self.thrs = np.unique(
            np.insert(np.concatenate(
                [self.c_score, self.i_score]), 0, 0))
        self.fpr = None
        self.fnr = None
        self.tpr = None
        self.tnr = None

    def solve_ratios(self):
```

```
    """ Dado un conjunto de scores obtener los ratios """

    def plot(self, save_path):
    """ Dibujar la curva ROC """

    def aur(self, plot, save_path):
    """ Calcular el área bajo la curva ROC usando
    el método del trapecio """

    def aur_aprox(self, plot, save_path):
    """ Calcular el área bajo la curva ROC usando
    un método aproximado """

    def plot_aur(self, aur, save_path):
    """ Dibujar el área bajo la curva ROC """

    def dprime(self, plot):
    """ Obtener el valor D' """
```

Además de esto hemos desarrollado el script que lee los ficheros, recoge los argumentos y llama a las disntintas funciones para resolver la curva ROC.

1.3. | Resultados

1.3.1. Uso

Como comentábamos en el apartado anterior el programa que hemos desarrollado es un script de consola con las opciones que podemos ver en la sección de código 1.2

Listing 1.2: Uso del script para calcular la curva ROC

```
\$ python roc_curve.py --help
usage: roc_curve.py [-h] [-c C] [-i I] [-fp FP] [-fn FN]
                    [-p] [-a] [-aA] [-d]
                    [-s FILENAME]

Solve the ROC curve

optional arguments:
  -h, --help            show this help message and exit
  -c C, --clients C     Clients filename
  -i I, --impostors I   Impostors filename
  -fp FP                False positive
  -fn FN                False negative
  -p, --plot            Make plot
```

```
-a, --aur             Get area under the ROC curve
                      using trapezoidal rule
-aA, --aurAprox       Get area under the ROC curve
                      using an aproximation
-d, --dprime          Get dprime
-s FILE, --save FILE  Path where save ROC curve plot
```

Curva ROC

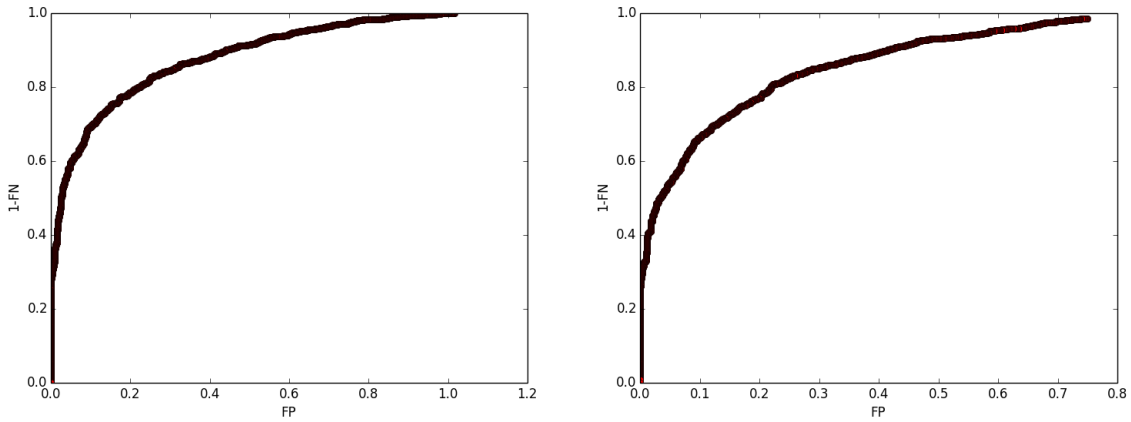
Lo primero que podemos realizar es calcular la curva ROC de un sistema.

Para esto, leeremos los datos, calcularemos los ratios y a partir de estos ratios dibujar la curva ROC. Damos al usuario la opción de guardar su gráfica.

En el segmento de código 1.3, vemos un ejemplo de cómo se lanzaría este script. Del mismo modo en la figura 1.1 vemos las curvas ROC de estos dos sistemas biométricos.

Listing 1.3: Calculo de la curva ROC para sistemas biométricos

```
$ python roc_curve.py -c ../data/scores/scoresB_clientes.txt
  -i ../data/scores/scoresB_impostores.txt -s scoresB
$ python roc_curve.py -c ../data/scores/scoresA_clientes.txt
  -i ../data/scores/scoresA_impostores.txt -s scoresA
```



(a) Curva ROC del sistema biométrico A (b) Curva ROC del sistema biométrico b

Figura 1.1: Curva ROC para dos sistemas biométricos

Obtener el valor FNR o el valor de FPR

Uno de los datos que nos puede interesar para comparar dos sitemas biométricos o incluso para establecer el valor a partir del cual aceptamos o rechazamos, es que dado el valor del ratio de los falsos positivos obtener el valor del ratio de los falsos negativos y el umbral para obtener este valor. Análogamente podemos querer el valor de los falsos postivos y el umbral dado un valor de falsos negativos.

Podemos ver un ejemplo de la ejecucición en el segmento de código 1.4.

Listing 1.4: Obtención de ratios y umbrales

```
$ python roc_curve.py -c ../data/scores/scoresB_clientes.txt
-i ../data/scores/scoresB_impostores.txt -fp 0.6
Dado el valor de fp: 0.6, el valor
de fnr es: 0.0458041958042 y el umbral: 0.00434

$ python roc_curve.py -c ../data/scores/scoresB_clientes.txt
-i ../data/scores/scoresB_impostores.txt -fn 0.3 -p
Dado el valor de fn: 0.3, el valor
de fpr es: 0.129020979021 y el umbral: 0.079034
```

Antes de continuar a la siguiente sección, aclaramos que los valores obtenidos en este apartado se obtienen si existen directamente de los datos y sino se interpolan a partir de la curva ROC.

Obtener el área bajo la curva ROC

El área bajo la curva ROC es una buena medida para evaluar numéricamente varios sistemas biométricos.

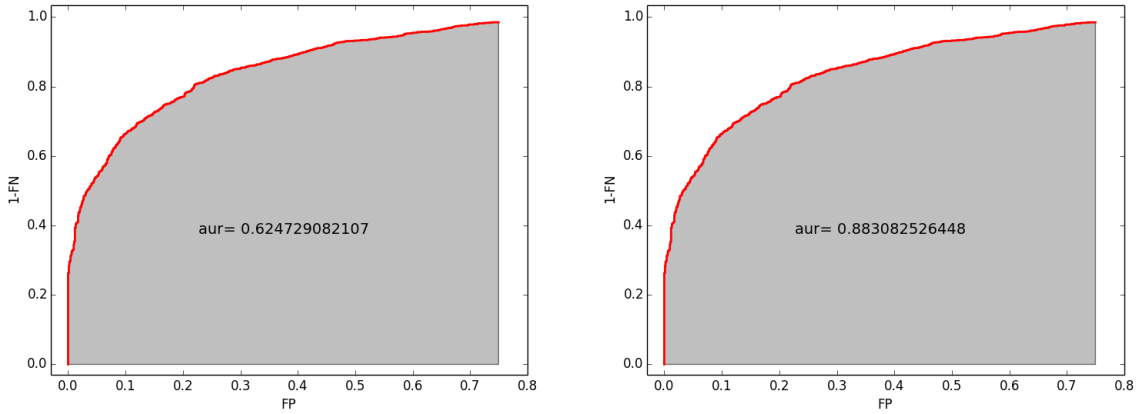
Hemos implementado esta medida mediante una aproximación a la integral de la curva y mediante la resolución de la integral mediante el método del trapecio. De los resultados que podemos ver en el segmento de código 1.5, vemos que tanto el área como los tiempos de ejecución varian en función del método empleado.

En la figura 1.2, vemos los resultados que hemos obenido.

Listing 1.5: Calculo del área bajo la curva ROC para sistemas biométricos

```
\$ python roc_curve.py -c ../data/scores/scoresB_clientes.txt
-i ../data/scores/scoresB_impostores.txt -a
El área bajo la curva roc es igual a 0.624729082107
(Coste temporal: 0.000113964080811)

\$ python roc_curve.py -c ../data/scores/scoresB_clientes.txt
-i ../data/scores/scoresB_impostores.txt -aA
El área bajo la curva roc es igual a 0.883082526448
(Coste temporal : 5.70780491829)
```



(a) AUR método aproximado (b) AUR método del trapecio

Figura 1.2: AUR

Obtener el factor d'

Por útlimo calculamos el valor del factor d', que mide la discriminalidad de la técnica empleada. En el segmento de código 1.6

Listing 1.6: Calculo del factor d-prime

```
\$ python roc_curve.py -c ../data/scores/scoresB_clientes.txt
-i ../data/scores/scoresB_impostores.txt -d -s dprime
El factor dprime es 0.873481856448
```

2 | Implementación de PCA

El siguiente ejercicio desarrollado es el análisis de las principales componentes de una imagen. Como entrada tenemos un conjunto de imágenes de caras de distintos sujetos y el número de dimeniones a las que queremos proyectar dichas imágenes.

La proyección en las nuevas dimensiones no es discriminativa, pero podemos utilizarla para crear clusters e identificar imágenes.

2.1. | Descripción del trabajo realizado

Una vez leídas las imágenes y cargadas en memoria como un vector de número reales, reservamos un conjunto de imágenes para el anális de las principales componentes y otro conjuto para probar cómo funciona este análisis. En este caso hemos reservado un 50 % de las imágnes para el entrenamiento y el restante 50 % para las pruebas.

Para cada una de las imágenes calculamos su proyección en el nuevo espacio de componentes de dimensionalidad d'. El caso general el algoritmo PCA, obtendría los eigenfaces (vectores propios) siguiendo el algoritmo que podemos ver en 2.1.

Listing 2.1: PCA

```
C = 1.0/n * np.dot(A,A.T)
D,B = la.eigh(C)
# Ordenamos los vectores propios, primero los que más varianza rec
order = np.argsort(D)[::-1] # sorting the eigenvalues
# Ordenamos los vectores propios & los valores propios
B = B[:,order]
D = D[order]
```

Sin embargo en el caso de que el número de imagenes sea mucho menor que el número de dimensiones de cada imagen, realizaremos el análsis de las principales componentes siguiento el algoritmo 2.2.

Listing 2.2: PCA cuado n es menor d

```
if d_prime > n:
    d_prime = n
# C: Matriz de covarianzas
C_prime = 1.0/d * np.dot(A.T,A)
#Delta=eigenvalues B=eigenvectors
```

```
D_prime,B_prime = la.eigh(C_prime)

for i in xrange(n):
    B_prime[:,i] = B_prime[:,i]/np.linalg.norm(B_prime[:,i])

B = np.dot(A, B_prime)
D = d/n * D_prime
# Ordenamos los vectores propios, primero los que más varianza rec
order = np.argsort(D, axis=0)[::-1]
# Ordenamos los vectores propios & los valores propios
B = B[:,order]
D = D[order]
```

Si ejecutamos PCA siguiendo este caso particular, el número máximo de dimensiones al que podremos reducir las imágenes corresponderá con el número de imágenes. Por las características de la máquina en la que hemos desarrollado esta práctica hemos empleado esta segunda implementación de PCA para tratar las imágenes.

2.1.1. Resultados

Con el 50 % de las imágenes que habíamos reservado de cada uno de los sujetos realizaremos la fase de pruebas. Proyectaremos las imágenes en el número de dimensiones y mediante vecino más cercano etiquetaremos la imagen.

Al proyectar en tan pocas dimensiones, desde 1 a 5, se pierde gran parte de la información y por eso los resultados que hemos obtenido no son los mejores.

En el segmento de código 2.3, vemos un ejemplo de cómo ejecutar el algoritmo PCA. Recorrería todas las cara que encuentre en la ruta que le hemos especificado, utilizando el 50 % de las imágenes para entrenar y el resto para predecir una etiqueta. Los resultados de estas predicciones se guardan en un fichero de texto plano para después dibujar la gráfica con los resultados.

Listing 2.3: Ejecución de PCA

```
python pca.py -p ../data/caras/ORL -d 100 > caras100
```

Hemos desarrollado un pequeño script en python para mostrar estos resultados en una gráfica como la que podemos ver en la imagen 2.1.

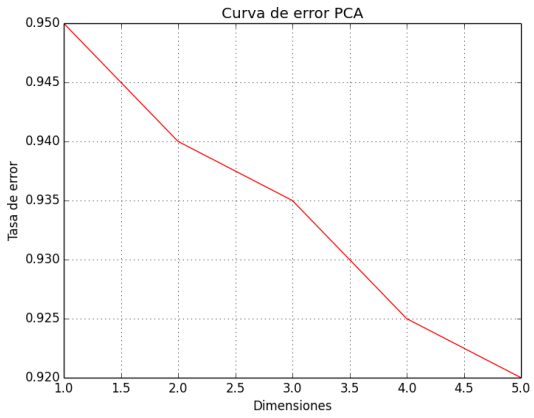


Figura 2.1: Resultado PCA

3 | Schneiderman y Kanade

3.1. | Datos de entrada

En primer lugar hemos preparado las imágenes para disponer de un conjunto de pruebas, otro para el desarrollo y un último para el test. El conjunto de entrenamiento lo empleamos para calcular las probabilidades que emplearemos en para la clasificación de una imagen en cara o no cara. El conjunto de desarrollo es un conjunto de imagenes distintas a las imágenes de entrenamiento empleadas para estimar el valor de λ. Finalmente tendremos un conjunto de test lo emplrearemos para calcular como funciona nuestro sistema.

Disponemos como parámetros de entrada de una serie de imágenes para el entreamiento expersadas mediante una serie de números de tipo flotante que corresponden con el valor del nivel de gris de la imagen normalizado. El primer paso que hemos desarrollado a consistido en leer estas imágenes y separar correctamente un porcentaje para entrenamiento y otro para test. El 80 % corresponderá al entrenamiento y el 20 % restante al test.

3.2. | Descripción del trabajo realizado

3.2.1. Entrenamiento

Para cada una de las imágenes del test, las dividimos en regiones y cuatificamos estas regiones utilizando el algorimto c-medias con 100 iteraciones, seleccionamos 256 niveles, por lo tanto las regiones que en este punto son matrices de 5x5 se colapsan a un único valor. A partir de estos valores calculamos las probabilidades de que este valor que hemos cuantificado pertenezca a cara o no caras, es decir $P(q|caras)$ y $P(q|nocara)$, para esto hemos empleado vectores inicializaods a 0 y después normalizados, pero como el conjunto de entrenamiento que estabamos empleando era muy disperso hemos realizado un suavizado de Laplace. También deberemos calcular la probabilidad de que en una posición de la imagen dado un valor cuantificado y si es cara o no cara, es decir $P(pos|q, caras)$ y $P(pos|q, nocara)$. Todas estas probabilidades nos las devolverá la función train que vemos en el segmento de código **??**, así como el tamaño de la ventana entrenado.

Listing 3.1: EDA con la que gestionar la curva ROC

```
def train(faces, not_faces, num_regions, q_levels):
        ...
    return p_q_faces, p_q_notFaces, p_pos_q_faces,
           p_pos_q_notFaces, width
```

3.2.2. Ajuste del parámetro λ

Una vez tenemos las probabilidades entrenadas utilizaremos el conjunto de test y calcularemos la probabilidad de que sea cara o no cara utilizando la regla de decisión descrita por Schneiderman y Kanade e imprimimos la media de las probabilidades de que la imagen sea cara y la media de las probabilidades de que la imagen no sea cara. Podemos ver un ejemplo de como ejecutar el algoritmo en modo desarrollo en el segmento de código 3.2.

Listing 3.2: Ejecución en modo desarrollo

```
$ python sk.py -f ../data/caras/dev_faces400
               -n ../data/caras/dev_notFaces400 -d
```

Estas medias las escribiremos en un fichero de texto plano, *test* y mediante otro pequeño programa en python, *lamda.py*, dónde leemos las probabilidades medias que hemos obtenido para caras y no caras y establecemos como lamda un valor que se encuentre en un punto medio entre la probabilidad de que la imagen sea cara y que sea no cara. Claramente este valor se verá muy afectado por datos anómalos ya que influyen mucho en la media.

3.2.3. Prueba del sistema

Una vez determinado el valor de lambda probaremos como funciona el sistema que hemos entrenado con datos de prueba. Para esto cogemos un nuevo conjunto de datos que no hayamos empleado para entrenar el sistema y calcularemos los veredaderos y falsos positivos así como los verdaderos y falsos negativos. Para ejecutar el sistema en modo de prueba deberíamos lanzar una orden como la que vemos el segmento de código 3.3.

Listing 3.3: Ejecución en modo pruebas

```
$ python sk.py -f ../data/caras/dev_faces2000
               -n ../data/caras/dev_notFaces2000
               -l 0.090909091113 -d
               -ft ../data/caras/faces_test600
               -nt ../data/caras/notfaces_test600 -t-f
```

Ejecutaríamos este análisis del sistama con distintos conjuntos de prueba y obtendríamos una medida de la robustez del sistema. Sin embargo, como el conjunto de test y de prueba que hemos empleado pertenecen a una misma base de datos de imagenes

el sistema se comporta demasiado favorablemente, esta sobreentrenado. Esto lo comprobamos al probar nuestro sistema de reconocimiento con una imagen distinta. Los restultados pueden encontrarse en el fichero de texto plano *test_error*

Uso del reconocedor

La útlima fase consiste en con el reconocedor entrenado buscar las caras dentro de una imagen.

Leemos y cargamos la imagen en memoria, cogemos secciones del tamaño de la ventana que hemos entrenado ya al igual que en la fase de pruebas calculamos la probabilidad de que en una region dada haya una cara si es mayor al lambda que hemos establecido en la fase de test.

Nos ha quedado pendiente desarrollar el dibujado de las regiones dónde ha encontrado cara, así como reducir el tamaño de la imagen para que las caras que encontremos sean invariantes al tamaño como explican Schneiderman y Kanade en su algoritmo.

La ejecución se realiza como vemos en el segmento de código 3.4

Listing 3.4: Ejecución en modo reconocedor

```
\$ python sk.py -f ../data/caras/dev_faces2000
                -n ../data/caras/dev_notFaces2000
                -l 0.090909091113 -t -i ../data/caras/jr.png
```

4 | Fusión de scores

Cuanto disponemos de varios resultados obtenidos de distintas fuentes biométricas para poder evaluar como afectan cada uno de los resultados en la decisión desarrollaremos un sistema de fusión de caracterísicas biométricas.

4.1. | Datos de entrada

Para este ejercicio disponemos de ficheros con texto plano cuya última columna nos indica si el dato es un cliente o un impostor y en las columnas anteriores tenemos los resultados de los distintos sistemas biométricos. En este caso únicamente 2 scores.

4.2. | Desarrollo realizado

4.2.1. Tratamiento de los datos de entrada

En primer lugar desarrollamos un pequeño script bash, que separe los scores de los clientes y de los impostores y se quede con una subsección de los mismos, para acelerar los cálculos. En caso de llevar este sistema a producción el número de elementos en el subconjunto deberá se el total de los elementos disponibles.

4.2.2. Fusión de scores

Con los datos ya preparados realizamos un script en python que lea estos datos y calcule el peso para cada score de modo que se minimice el área bajo la curva ROC.

En el listado de código 4.2 podemos ver las opciones para lanzar el script.

Listing 4.1: PCA cuado n es menor d

```
$ python fusion.py --help
usage: fusion.py [-h] [-tr TR] [-te TE] [-p]

Fuse some scores
optional arguments:
  -h, --help            show this help message and exit
  -tr TR, --train TR    Scores for train data
```

```
  -te TE, --test TE    Scores for test data
  -p, --plot           Make plot
```

4.3. | Resultados

A continuación, en el segmento de código 4.2, vemos los resultados de la fusión de los dos scores de modo que minimicen el área bajo la curva ROC. Y en la figura 4.1 vemos como se distribuyen los dos scores tanto en entrenamiento como en pruebas.

Listing 4.2: PCA cuado n es menor d

```
$ python fusion.py -tr ../data/fusion/train_small -te ../data/fusion/test_
El valor máximo del área bajo la curva ROC= 1.423700
Con los pesos:
  [ 0.60, 0.40 ]
```

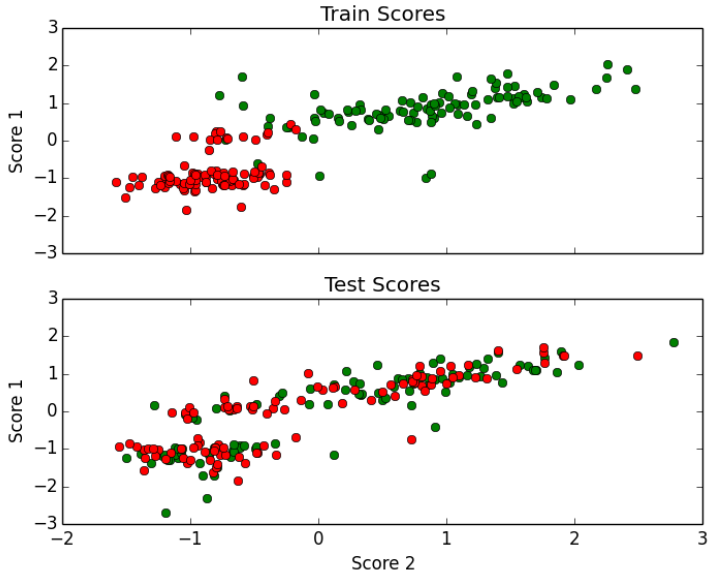


Figura 4.1: Scores en train y en test