MIARFID Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Memoria Prácticas Biometría

Autor: Pascual Andrés Carrasco Gómez

Índice

Sistemas de verificación: Curva ROC

Este ejercicio está formado por un único programa python (versión 2.7):

- medidas_calidad.py: A través de una interfaz de usuario (terminal) nos permite elegir que opción queremos elegir para mostrarnos los resultados obtenidos.

Los datos utilizados en este ejercicio son los proporcionados en la asignatura, formados por los scores de clientes e impostores A y los scores de clientes e impostores B.

Todos los programas que se han realizado en esta memoria incorporan un control de parámetros para que el usuario conozca qué parámetros se necesitan para el correcto funcionamiento del programa, para obtener dicha información simplemente se ejecuta el programa sin parámetros:

```
$ python2.7 medidas_calidad.py
Uso: python2.7 medidas_calidad <scores_clientes> <scores_impostores>
```

El programa se ejecuta de la siguiente forma:

```
$ python2.7 medidas_calidad.py scores/scoresA_clientes scores/scoresA_impostores
```

La interfaz de usuario es un menú formado por siete opciones definidas en el rango de números del 1 al 7, donde el número 7 nos permite finalizar la ejecución del programa.

```
-----------------------------------
              MENU
-----------------------------------
1) Curva ROC
2) FP(FN = X) y umbral
3) FN(FP = X) y umbral
4) FP = FN y umbral
5) Area bajo la curva ROC
6) D-Prime
7) Salir
Escoge una opcion: [1-7]
```

A continuación vamos a comentar los resultados proporcionados por cada una de las opciones del programa para los scores de A y de B.

1) Curva ROC

Esta opción nos muestra la gráfica correspondiente a la curva ROC, para su correcto funcionamiento es necesario tener instalada la librería de python matplotlib, para ello han de estar instalados los siguiente paquetes:

```
apt-get install python-matplotlib
apt-get install python-tk
```

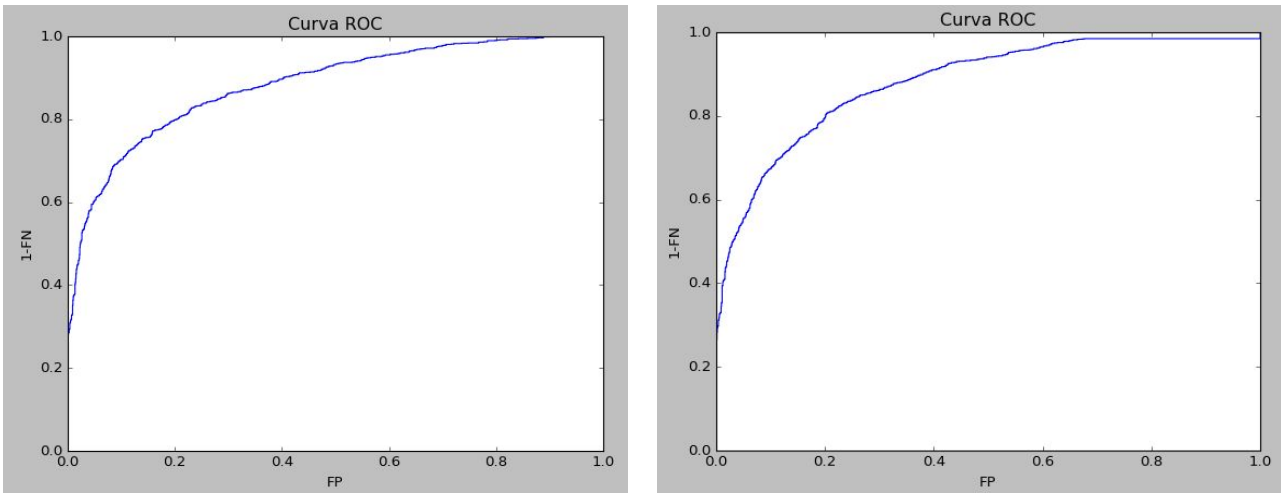


Imagen 1: Izquierda, Curva ROC obtenida mediante scoresA_clientes y scoresA_impostores.
Derecha, Curva ROC obtenida mediante scoresB_clientes y scoresB_impostores.

El código correspondiente para la obtención de la curva ROC se muestra a continuación:

```
# Valores de la grafica
x = [] # FP
y = [] # 1-FN
q_c = 0
q_i = 0
for i in range(0,len(scores)):
        acceso = scores[i][1]
        if acceso == 1: # cliente
                q_c += 1
        elif acceso == 2: # impostor
                q_i += 1
        x.append((len(scores_impostores)-q_i)/float(len(scores_impostores)))
        y.append(1-(q_c/float(len(scores_clientes))))
```

Los valores (puntos) de la gráfica se almacenan en dos listas x e y que corresponden a los puntos de la gráfica, x corresponde a FP e y corresponde a 1-FN. Los puntos de la gráfica únicamente se calculan una vez y son utilizados posteriormente por las opciones del menú del programa.

```
# Curva ROC
if op == 1:
        # Grafica (plot)
        plt.xlabel('FP')
        plt.ylabel('1-FN')
        plt.title('Curva ROC')
        plt.plot(x,y)
        plt.show()
```

La opción 1 como observamos en el código nos muestra la gráfica de la curva ROC.

2) FP(FN = X) y umbral

Esta opción nos devuelve el valor FP dado un valor FN indicado por el usuario y su correspondiente umbral.

```
# FP(FN = X) y umbral
elif op == 2:
        fn = input("Introduce un valor para FN: [0.0-1.0]\n")
        indice = -1
        d_min = float("Inf")
        for i in range(0,len(y)):
                if abs(y[i]-fn) < d_min:
                        indice = i
                        d_min = abs(y[i]-fn)
        print "FP(FN=" + str(fn) + "):",x[indice]
        print "FN: ",1-y[indice]
        print "umbral:",scores[indice][0]
```

Mostramos a continuación los resultados de la opción 2 para un valor FN = 0.51.

Scores A	Scores B
FP(FN=0.51): 0.0262820512821 FN: 0.49020979021 umbral: 0.1019	FP(FN=0.51): 0.0346153846154 FN: 0.49020979021 umbral: 0.163127

3) FN(FP = X) y umbral

Esta opción nos devuelve el valor FN dado un valor FP indicado por el usuario y su correspondiente umbral.

```
# FN(FP = X) y umbral
elif op == 3:
        fp = input("Introduce un valor para FP: [0.0-1.0]\n")
        indice = -1
        d_min = float("Inf")
        for i in range(0,len(x)):
                if abs(x[i]-fp) < d_min:
                        indice = i
                        d_min = abs(x[i]-fp)
        print "FN(FP=" + str(fp) + "):",(1.0-y[indice])
        print "FP:", x[indice]
        print "umbral:",scores[indice][0]
```

Mostramos a continuación los resultados de la opción 3 para un valor FP = 0.51.

Scores A	Scores B
FN(FP=0.51): 0.0643356643357 FP: 0.510256410256 umbral: 0.024081	FN(FP=0.51): 0.0587412587413 FP: 0.510256410256 umbral: 0.006098

4) FP = FN y umbral

Esta opción nos devuelve el valor en el que FP y FN son iguales (en el caso de que no se de la igualdad, devuelve los dos valores con menor diferencia entre ellos) y su correspondiente umbral.

```
# FP = FN y umbral
elif op == 4:
        indice = -1
        d_min = float("Inf")
        for i in range(0,len(x)):
                if abs(x[i]-(1-y[i])) < d_min:
                        indice = i
                        d_min = abs(x[i]-(1-y[i]))
        print "FP: ",x[indice]
        print "FN: ",1-y[indice]
        print "umbral: ",scores[indice][0]
```

Mostramos a continuación los resultados de la opción 4:

Scores A	Scores B
FP: 0.201282051282 FN: 0.201398601399 umbral: 0.050333	FP: 0.201282051282 FN: 0.201398601399 umbral: 0.044444

5) Área bajo la curva ROC

El área bajo la curva ROC se ha obtenido mediante el método de Mann-Whitney, que nos permite obtener el área de forma eficiente.

```
# Area bajo la curva ROC
elif op == 5:
        aux_suma = 0.0
        for s_c in scores_clientes:
                for s_i in scores_impostores:
                        aux_suma += h(s_c[0]-s_i[0])
        aroc = (1.0/(len(scores_clientes)*len(scores_impostores)))*aux_suma
        print "AROC: ",aroc
```

Mostramos a continuación los resultados de la opción 5:

Scores A	Scores B
AROC: 0.883163439125	AROC: 0.883082526448

6) D-Prime

Esta opción nos devuelve el factor d' (d-prime) que nos proporciona una medida de discriminabilidad sobre la técnica usada, para ello tiene en cuenta la separación de las distribuciones de scores de clientes e impostores, así como el grado de dispersión de ambas.

```
# d-prime
elif op == 6:
        m_clientes = 0 # media clientes
        for score in scores_clientes:
                m_clientes += score[0]
        m_clientes = m_clientes/float(len(scores_clientes))
        dt_clientes = 0 # Desviacion tipica clientes
        for score in scores_clientes:
                aux = (score[0]-m_clientes)*(score[0]-m_clientes)
                dt_clientes += aux
        dt_clientes = dt_clientes/len(scores_clientes)
        m_impostores = 0 # media impostores
        for score in scores_impostores:
                m_impostores += score[0]
        m_impostores = m_impostores/float(len(scores_impostores))
        dt_impostores = 0 # Desviacion tipica impostores
        for score in scores_impostores:
                aux = (score[0]-m_impostores)*(score[0]-m_impostores)
                dt_impostores += aux
        dt_impostores = dt_impostores/len(scores_impostores)
        dprime = (m_clientes-m_impostores)/math.sqrt(dt_clientes+dt_impostores)
        print "D-Prime: ",dprime
```

Mostramos a continuación los resultados de la opción 6:

Scores A	Scores B
D-Prime: 0.759953697523	D-Prime: 0.873481856448

Detección facial

DataSet

Para este ejercicio se nos ha proporcionado dos ficheros, un fichero que contiene 6099 imágenes que corresponden a caras y un segundo fichero que contiene 10000 imágenes que corresponden a no caras. Se ha aumentado el *dataset* original utilizando los programas que se han adjuntado en la memoria y que describimos a continuación:

- caras_desde_ojos.py: Utilizando el *dataset* “BioID-FaceDatabase-V1.2” se han obtenido el conjunto de caras a partir de las posiciones de los ojos indicadas por ficheros “.eye” en el *dataset*.

```
Uso: python2 caras_desde_ojos.py <corpus_BioID-FaceDatabase-V1.2> <f_salida>
$ python2 caras_desde_ojos.py BioID-FaceDatabase-V1.2/ mas_caras
```

- crop_imagen.py: A partir de un conjunto de imagenes que no contienen caras se han obtenido 100000 imágenes de dimensión 24x24 con la finalidad de ampliar el *dataset* original de no caras.

```
Uso: python2 crop_imagen.py <dir_imagenes> <size_crop> <n_crops> <nombre_f_salida>
$ python2 crop_imagen.py ../crop_no_caras/ 24 90000 mas_no_caras
```

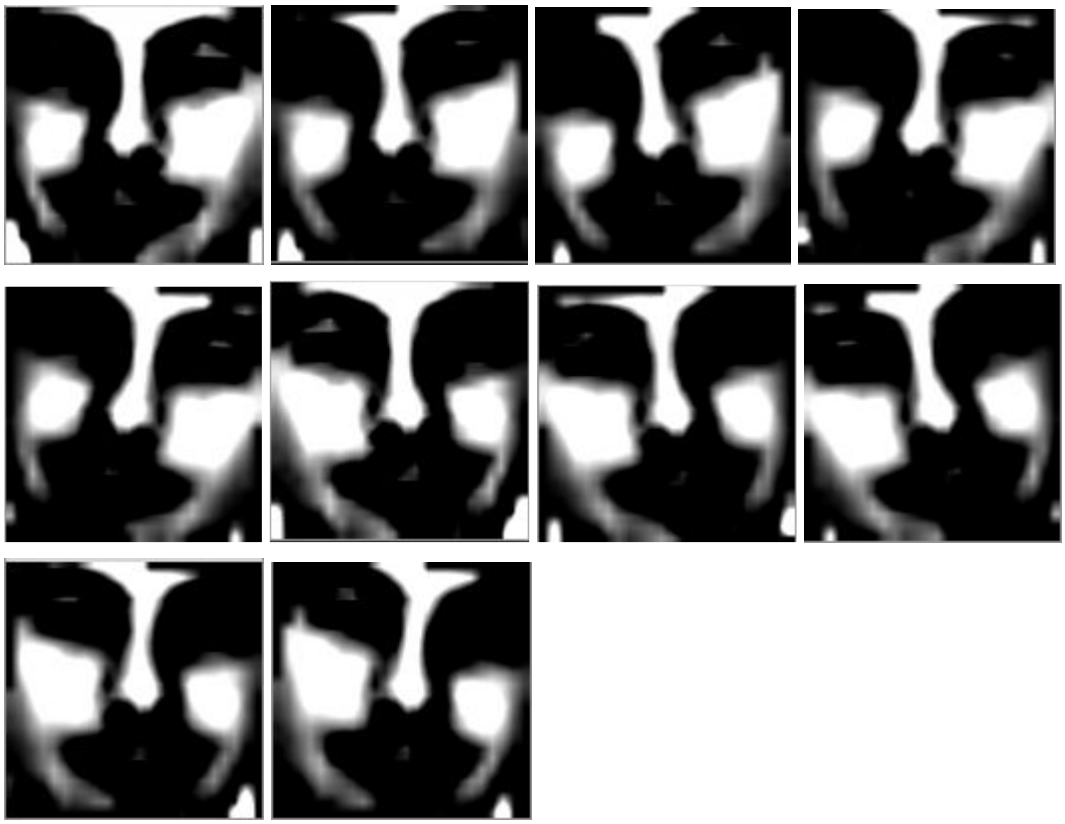
- unir_datos.py: Este programa se encarga de unir el *dataset* original con las respectivas ampliaciones obtenidas mediante los dos programas anteriores, almacenando los datos en disco en formato de matriz de numpy (ocupa menos en memoria).

```
Uso: python2 unir_datos.py <f_practicas> <f_incremental> <nombre_f_salida>
$ python2 unir_datos.py dfFaces_24x24_norm mas_caras aux_caras
$ python2 unir_datos.py NotFaces_24x24_norm mas_no_caras no_caras
```

- operaciones_caras.py: Realiza operaciones de rotacion, escalado y translación sobre el conjunto de caras obtenido.

```
Uso: python2 operaciones_caras.py <f_caras> <f_salida>
$ python2 operaciones_caras.py aux_caras caras
```

Para cada imagen obtenemos las siguientes imágenes al realizar las operaciones:



Por lo tanto para cada cara del dataset de caras obtenemos diez caras.
Nota: En los programas descritos en la definición del *dataset* se trabaja con imágenes normalizadas con media 0 y desviación típica 1.

El dataset con el que vamos a trabajar en los dos algoritmos implementados para detección facial se describe en la siguiente tabla:

DataSet	
caras	76160
no caras	100000

Neural Network-Based Face Detection

(Henry A. Rowley, Shumeet Baluja and Takeo Kanade)

Este ejercicio está compuesto por dos programas python:

- caras_neuronal.py: Este programa realiza el entrenamiento de la red neuronal y su posterior evaluación. Para realizar el entrenamiento y la evaluación se ha dividido el dataset en dos partes, 80% para entrenamiento y 20% para test.

```
Uso: python2 caras_neuronal.py <f_caras> <f_no_caras>
$ python2 caras_neuronal.py caras no_caras
```

- deteccion_caras: Este programa nos permite detectar caras sobre una imagen pasada como parámetro.

```
Uso: python2 deteccion_caras.py <image> <modelo> <size_window> <size_stride>
$ python2 deteccion_caras.py personas.jpg modelo.h5 24 10
```

A continuación vamos a mostrar el código implementado respecto a la estructura de la red neuronal utilizada como los resultados de la evaluación obtenidos, todo este código se encuentra en el programa “caras_neuronal.py”. Para implementar la red neuronal se ha utilizado la librería keras para python utilizando el backend Theano, por defecto el backend de keras es TensorFlow por lo tanto lo que se ha hecho es cambiar la configuración por defecto de la siguiente forma: https://keras.io/backend/

```
#----------------------------------------------------
# Estructura de la red
#----------------------------------------------------
model = Sequential()
model.add(Dense(64, init='uniform', input_shape=(len(x_train[0]),)))
model.add(noise.GaussianNoise(0.5))
model.add(Activation('relu'))
model.add(Dense(32, init='uniform'))
model.add(Activation('relu'))
model.add(Dense(2, init='uniform'))
model.add(Activation('softmax'))
#----------------------------------------------------
# Parametros
#----------------------------------------------------
model.compile(loss='categorical_crossentropy',
              optimizer='adadelta',
              metrics=['accuracy'])
```

En el entrenamiento se han utilizado 2 *epochs* y un tamaño de *batch* de 500.
Respecto a la evaluación se han obtenido dos evaluaciones del sistema obtenido, la primera evaluación corresponde a la que nos devuelve keras (especificado por el usuario), cuyos resultados se basan en la *loss* y la *accuracy* del modelo obtenido.

```
loss:  0.015805690367
accuracy:  0.994682252532
```

La segunda evaluación corresponde a los valores de verdaderos positivos (VP), falsos positivos (FP), verdaderos negativos (VN) y falsos negativos (FN) del modelo obtenido.

```
-----------------------------------
      RESULTADOS
-----------------------------------
Verdaderos positivos (VP):  7581
Falsos positivos (FP):  35
Verdaderos negativos (VN) 7570
Falsos negativos (FN) 46
Caras analizadas:  7616
No caras analizadas:  7616
-----------------------------------
```

Las imágenes de la evaluación son casos muy similares a las imágenes con las que se ha entrenado el modelo (son subconjuntos del *dataset*), para ello se ha implementado el programa “deteccion_caras.py”, en el cual le pasamos una imagen con caras de personas y nos devuelve a través de recuadros la detección facial detectada por el modelos, vemos los resultados en las siguientes imágenes.

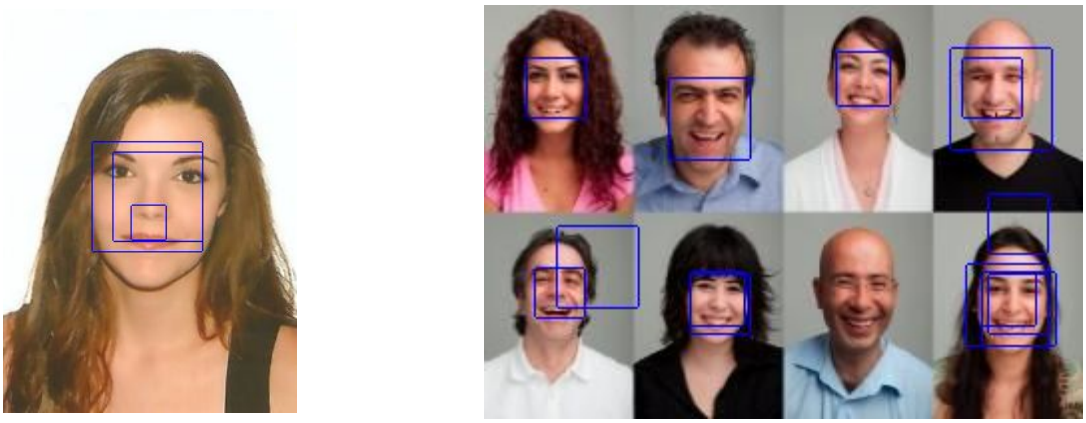


Imagen 2: Resultados de la detección facial utilizando el modelo entrenado con la red neuronal.

Schneiderman and Kanade

Este ejercicio está compuesto por tres programas python:

- entrenamiento.py: Este programa se encarga de dividir el dataset en un conjunto de entrenamiento (80%), un conjunto de desarrollo (20% sin extraerlo del conjunto) y un conjunto de evaluación (20%). Es el encargado de obtener el modelo, el cual está formado por tres estructuras de datos: q_caras, q_no_caras, m_pos_q_caras. También genera los modelos obtenidos para PCA y *kmeans.* Todos los modelos son almacenados en ficheros ".dat" que se utilizan en los programas que se describen a continuación.

```
Uso: python2 entrenamiento.py <f_caras> <f_no_caras>
$ python2 entrenamiento.py caras no_caras
```

- obtener_lambda.py: Este programa utiliza el conjunto de desarrollo "datos_dev.dat" para obtener el umbral que nos indica que es una cara y que no. El umbral se obtiene evaluando el sistema y dividiendo la probabilidad de no_caras entre la probabilidad de caras.

```
$ python2 obtener_lambda.py
```

- evaluacion.py: Este programa se encarga de mostrar por pantalla la evaluación del sistema obtenido mediante el entrenamiento. Para ello utiliza el fichero "datos_test.dat" (generado por "entrenamiento.py") y muesta los verdaderos positivos (VP), falsos positivos (FP), verdaderos negativos (VN) y falsos negativos (FN) obtenidos por el sistema para el conjunto de evaluación.

```
$ python2 evaluacion.py
```

- deteccion.py: Este programa nos permite detectar caras sobre una imagen pasada como parámetro.

```
Uso: python2 deteccion.py <image> <size_window> <scale> <stride>
```

A continuación vamos a comentar como se ha implementado el algoritmo basándonos en el programa python "entrenamiento.py". Para realizar la implementación de este algoritmo nos hemos encontrado con problemas de memoria RAM al generar las estructuras de datos necesarias, esto nos ha obligado a utilizar las librerías "IncrementalPCA" y "MiniBatchKMeans" de sklearn que permiten entrenar el modelo PCA y *kmeans* mediante bloques de datos particionados. Almacenar las imágenes del dataset como objetos numpy en vez de como listas también ha sido necesario para el correcto funcionamiento del programa. El programa primero obtiene el modelo PCA a partir de las 16 subimagenes obtenidas de cada imagen del conjunto de entrenamiento, tanto para imágenes de caras como imágenes de no caras. Una vez tenemos el modelo PCA proyectamos las subimagenes y obtenemos el modelo de *clustering* (kmeans) que nos proporciona el conjunto de etiquetas con las que vamos a obtener las tres estructuras de datos.

Este algoritmo requiere de dos estructuras de datos para modelar las caras (q_caras y m_pos_q_caras) y una estructura de datos para modelar las no caras (q_no_caras), las cuales se obtienen fácilmente mediante conteo. Para trabajar con probabilidades se realiza una normalización y para evitar productorios nulos se realiza un suavizado añadiendo un epsilon de 0.001 sobre la matriz “m_pos_q_caras” antes de normalizar.

EDA’s Caras

```
# Vector q de caras
q = [0]*clusters

print "- Obteniendo el vector q_i..."
# Actualizamos el vector q
for i in range(0,len(labels)):
	q[labels[i]] += 1

print "- Normalizando el vector q_i..."
# Normalizamos (para trabajar con probabilidades)
suma = sum(q)
for i in range(0,len(q)):
	q[i] = q[i]/float(suma)

# Almacenamos el objeto q en un fichero .dat para testing
fichero = file("q_caras.dat", "w")
pickle.dump(q, fichero, 2) # 2 = Almacenamiento binario
fichero.close()

# Matriz pos/q de caras
m_pos_q = []
for i in range(0,16):
	m_pos_q.append([0]*clusters)
print "- Obteniendo la matriz pos_i/q_i..."
# Actualizamos la matriz pos_i/qi
for i in range(0,len(labels)):
	pos = l_pos[i]
	q = labels[i]
	m_pos_q[pos][q] += 1

print "- Normalizando la matriz pos_i/q_i..."
# Suavizamos y normalizamos la matriz
for r in range(0,len(m_pos_q)):
	for c in range(0,len(m_pos_q[0])):
		m_pos_q[r][c] += 0.001
v_suma = []
for r in range(0,len(m_pos_q)):
	v_suma.append(sum(m_pos_q[r]))
for r in range(0,len(m_pos_q)):
	for c in range(0,len(m_pos_q[0])):
		m_pos_q[r][c] = m_pos_q[r][c]/float(v_suma[r])

# Almacenamos el objeto m_pos_q en un fichero .dat para testing
```

```
fichero = file("m_pos_q_caras.dat", "w")
pickle.dump(m_pos_q, fichero, 2) # 2 = Almacenamiento binario
fichero.close()
```

EDA's No caras

```
# Vector q de no caras
q = [0]*clusters

print "- Obteniendo el vector q_i..."
# Actualizamos el vector q
for i in range(0,len(labels)):
        q[labels[i]] += 1

print "- Normalizando el vector q_i..."
# Normalizamos (para trabajar con probabilidades)
suma = sum(q)
for i in range(0,clusters):
        q[i] = q[i]/float(suma)

# Almacenamos el objeto q en un fichero .dat para testing
fichero = file("q_no_caras.dat", "w")
pickle.dump(q, fichero, 2) # 2 = Almacenamiento binario
fichero.close()
```

Para obtener los modelos del sistema es necesario obtener dos parámetros que son el número de dimensiones a las que proyectamos PCA y el número de *clusters* que utilizamos en el modelo *kmeans*. Por otra parte también es necesario un tercer parámetro para realizar la evaluación del sistema y la detección de caras en una imagen, este parámetro es el umbral. Estos tres parámetros se han obtenido de forma empírica realizando un conjunto de pruebas.

Dimensiones (PCA)	***Clusters (kmeans)***	**Umbral (λ)**
8	60	2.38505529829

Los resultados del sistema obtenido se visualizan con el programa python "evaluacion.py" y son los que se muestran en la siguiente tabla:

```
-----------------------------------
      RESULTADOS
-----------------------------------
Verdaderos positivos (VP):  6387
Falsos positivos (FP):  1324
Verdaderos negativos (VN) 6292
Falsos negativos (FN) 1229
Caras analizadas:  7616
No caras analizadas:  7616
-----------------------------------
```

Nota: Se han realizado pruebas variando las dimensiones del dataset observando que a mayor número de muestras en el dataset mejores resultados se obtienen, por lo tanto es dependiente del número de caras y no caras que tengamos en el dataset y de la configuración de los tres parámetros descritos anteriormente.

A continuación se muestran las detecciones de este algoritmo sobre dos imágenes con caras, se observa que con el dataset que disponemos funciona mejor el modelo con redes neuronales implementado en el ejercicio anterior.

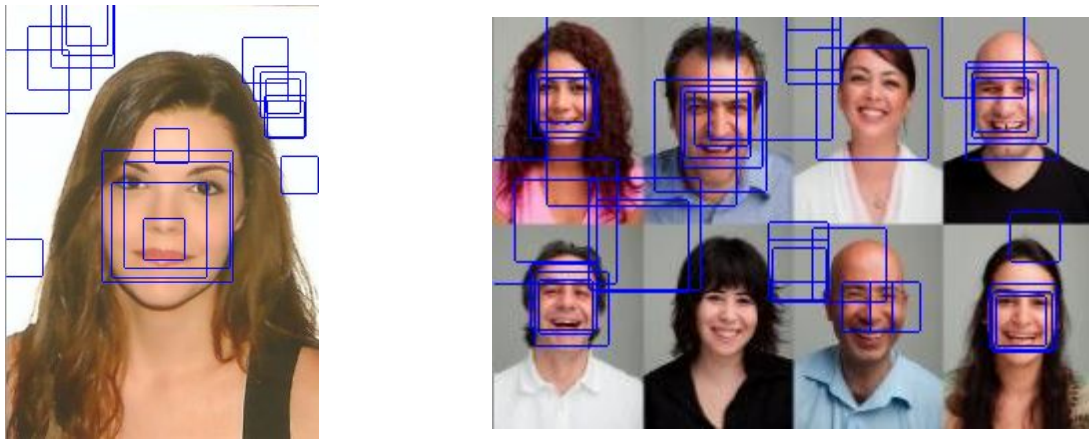


Imagen 3: Resultados de la detección facial utilizando el modelo entrenado con el algoritmo Schneiderman and Kanade.

Sistemas basados en características

DataSet

El dataset utilizado para estos ejercicios es “ORL face database”, es un *dataset* formado por imágenes tomadas a 40 personas, exactamente consiste en 10 imágenes obtenidas por persona, por lo tanto contiene un conjunto de 400 imágenes. El conjunto de imágenes se ha separado en un subconjunto para entrenamiento (imágenes 1-5 de cada individuo) y un subconjunto para evaluación (imágenes 6-10 de cada individuo), por lo tanto tenemos 200 imagenes para entrenamiento y 200 imagenes para test.

EigenFaces

Este ejercicio está compuesto por un único programa que describimos a continuación:

- eigenfaces.py: Se encarga de separar el subconjunto de entrenamiento y de test, realizar el entrenamiento y posteriormente la evaluación mostrando la gráfica obtenida.

```
Uso: python2 eigenfaces.py <dir_ORL>
$ python2 eigenfaces.py ORL/
```

En este ejercicio se han tenido en cuenta las consideraciones prácticas que consisten en obtener la diagonalización de C' en vez de C (trabajar con dimensión nxn en vez de dxd por el problema que aparece cuando d es grande), de la diagonalización de C' se ha obtenido B' (eigenvectores') y D' (eigenvalores') de los cuales posteriormente se han obtenido B (eigenvectores) y D (eigenvalores), la matriz de eigenvectores (B) obtenida es ortogonal pero no es ortonormal por lo tanto se ha dividido por su módulo.
La evaluación se ha realizado utilizando un clasificador k-vecino más cercano con k=1 con distancia euclídea como métrica y usando un algoritmo kd-tree para entrenar el modelo.
La gráfica resultante, obtenida mediante la evaluación, que muestra la curva de error variando d' se muestra a continuación.

Imagen 4: Curva de error variando d' en el problema de eigenvectores.

Nota: Para realizar cálculos algebraicos se ha utilizado la librería de python numpy, en concreto linalg. Para obtener el modelo de k-vecinos más cercanos se ha utilizado la librería de python sklearn, en concreto se ha utilizado el clasificador KNeighborsClassifier.

Se observa que para este problema realizando una proyección (PCA) con d'=32 ya se obtendría un error de clasificación significativamente bajo, ya que a partir de d'=32 el error es prácticamente constante.

d'	**error**
32	0.110156
200	0.096094

FisherFaces

Este ejercicio está compuesto por un único programa que describimos a continuación:

- fisherfaces.py: Se encarga de separar el subconjunto de entrenamiento y de test, realizar el entrenamiento y posteriormente la evaluación mostrando la gráfica obtenida.

```
Uso: python2 fisherfaces.py <dir_ORL>
$ python2 fisherfaces.py ORL/
```

Este programa python utiliza la parte del código de proyección PCA del programa "eigenfaces.py" para realizar una primera proyección (d') antes de aplicar LDA con el objetivo de intentar evitar singularidades en los datos. Una vez tenemos los datos proyectados con PCA se obtiene la matriz de distribución entre clases (SB) y la matriz de distribución intra clases (SW). Con las matrices SB y SW obtenemos la matriz C de la cual obtenemos los eigenvectores (B) y los eigenvalores (D) que nos permiten proyectar con LDA. La reducción de dimensionalidad (d'') de LDA se realiza en el rango de valores [1,C-1], ya que disponemos de 40 clases las gráficas muestran resultados de d'' = 1 a d'' = 39. La evaluación se ha realizado utilizando un clasificador k-vecino más cercano con k=1 con distancia euclídea como métrica y usando un algoritmo kd-tree para entrenar el modelo.
Las gráficas resultantes, obtenidas mediante la evaluación, que muestran la curva de error variando d'' se muestran a continuación.

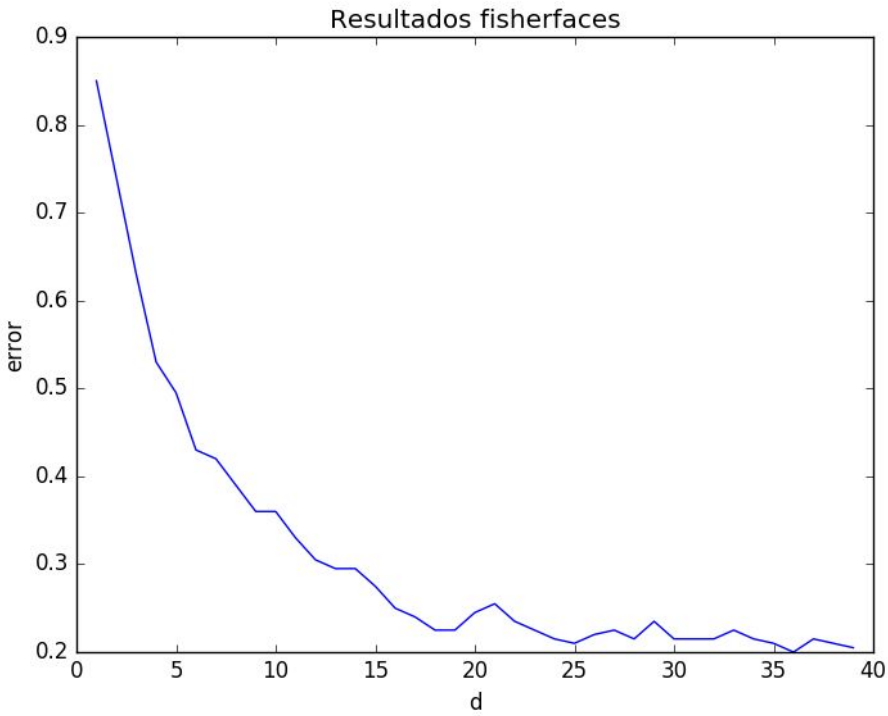


Imagen 5: Gráfica FisherFaces con proyección PCA a d' = 200.

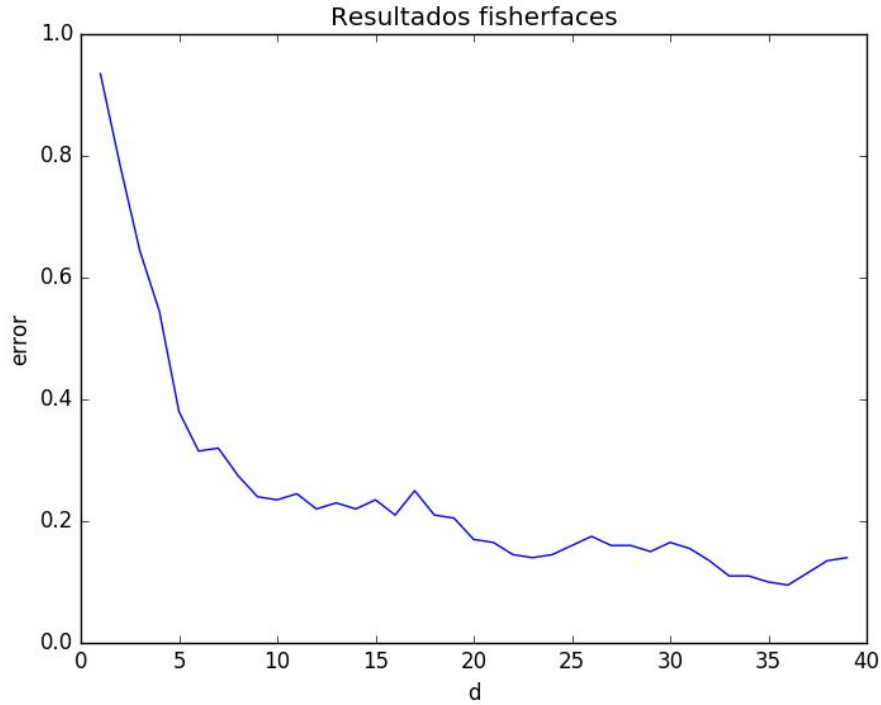


Imagen 6: Gráfica FisherFaces con proyección PCA a d' = 160.

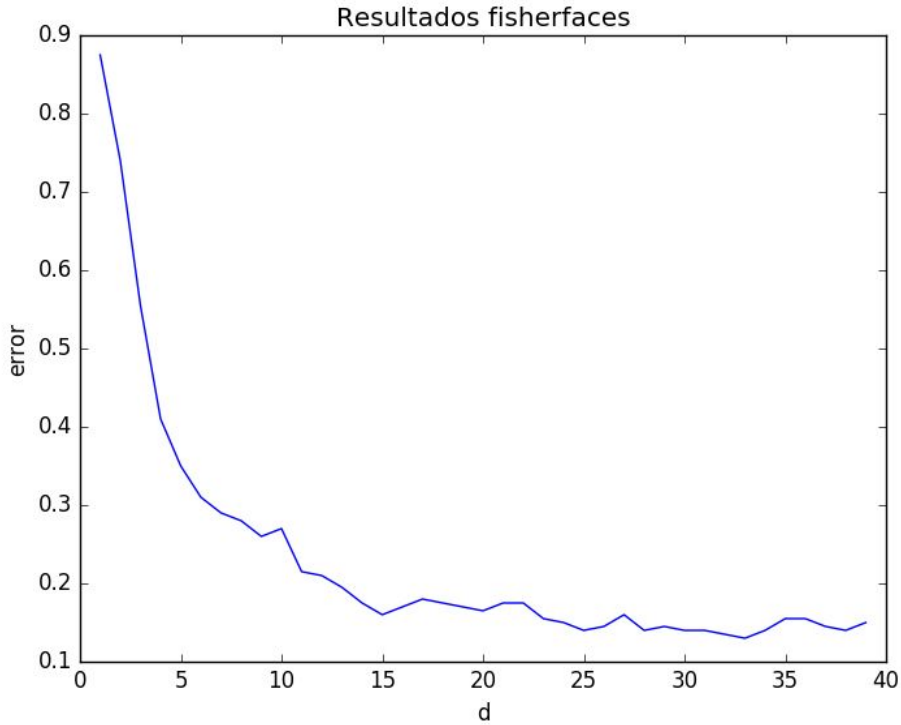


Imagen 7: Gráfica FisherFaces con proyección PCA a d' = 100.

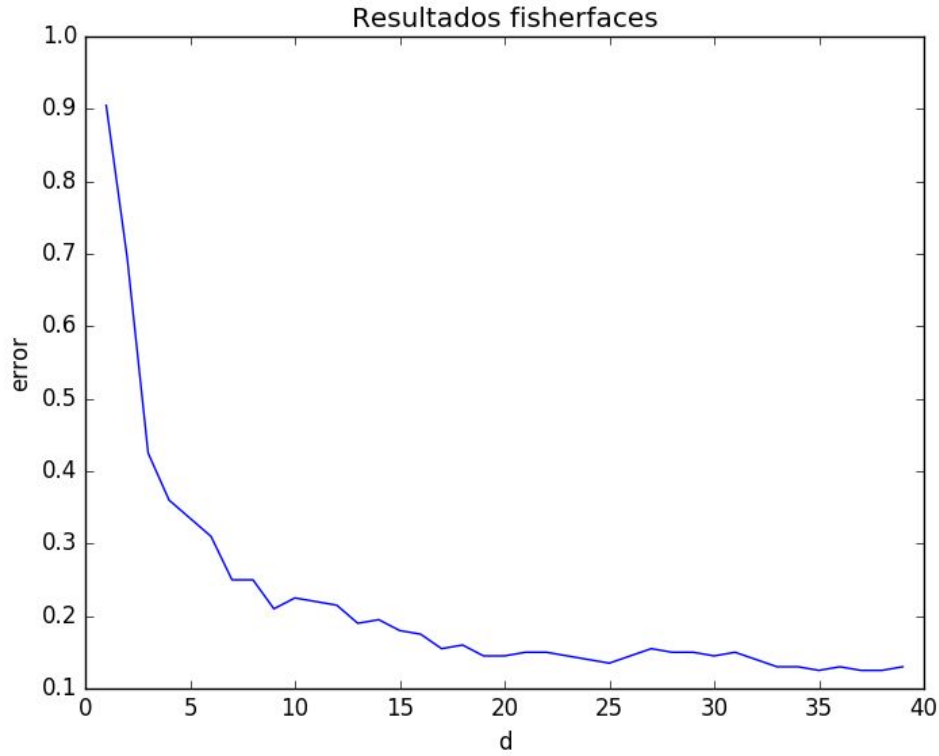


Imagen 8: Gráfica FisherFaces con proyección PCA a d' = 40.

Nota: Para realizar cálculos algebraicos se ha utilizado la librería de python numpy, en concreto linalg. Para obtener el modelo de k-vecinos más cercanos se ha utilizado la librería de python sklearn, en concreto se ha utilizado el clasificador KNeighborsClassifier.

d' (PCA)	**error (mínimo)**	**d'' (LDA)**
200	0.2	36
160	0.089	36
100	0.127	33
40	0.119	37

Se observa que el mejor resultado obtenido es con una reducción de dimensionalidad d' = 160 (PCA) y con una reducción de dimensionalidad d'' = 36 (LDA) con un error de 0.089.