Herramientas Software para Tratamiento de Imágenes
Máster Oficial en Visión Artificial
Hoja de Problemas Evaluable

Instrucciones para la entrega

Una vez resueltos los problemas enunciados a continuación, el alumno utilizará el formulario habilitado en el moodle de la asignatura para subir los fuentes en un fichero zip. El nombre del fichero deberá formarse siguiendo el siguiente esquema: “Apellido1_Apellido2_Nombre.zip”.

Fecha límite de entrega: -

Enunciados

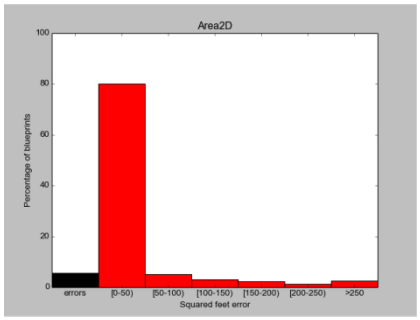
1. Queremos comparar los resultados obtenidos por un algoritmo de reconocimiento de objetos 3D. Por cada objeto reconocido se conoce su area2D, area3D y nivel de complejidad expresado como un número entero. Tanto los resultados generados por el algoritmo, como el *groundtruth*, se encuentran en ficheros csv diferentes (ver ficheros adjuntos con el enunciado).
   Se desea analizar cada característica por separado, generando una gráfica por cada una de ellas. Interesa que este paso se lleve a cabo de forma automática, el algoritmo evoluciona, y no queremos perder tiempo generando manualmente las gráficas cada vez que mejoramos (o empeoramos) el método. Por su simplicidad a la hora de manejar grandes colecciones de datos y calcular estadísticos, elegiremos Python como lenguaje para desarrollar la tarea. Pasos a seguir:
   - Leer los dos ficheros .csv y cargar los datos en dos NumPy arrays (`numpy.loadtxt`)
   - Generar una gráfica de barras para el área2D similar a la que se muestra a continuación. Cada barra muestra el porcentaje de objetos cuya área2D difiere del original en un rango establecido (`matplotlib.pyplot.bar`). De forma adicional, la gráfica debe mostrar el porcentaje de objetos para los que no se pudo calcular área porque el proceso de reconocimiento falló (nota: un fallo en el cálculo de la una característica se marca con un carácter ‘-‘ en el fichero csv).
   - Generar una gráfica equivalente, pero con el Área 3D.
   - Generar una tercera gráfica donde se muestren las diferencias en nivel de complejidad. La complejidad se mide como un valor entero, por lo que cada barra representará el porcentaje de objetos cuya complejidad difiere en una determinada cantidad. Mostrar también una barra en la

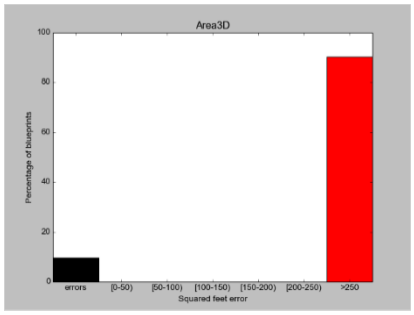
gráfica que indique el porcentaje de objetos para los que no se pudo calcular su complejidad.

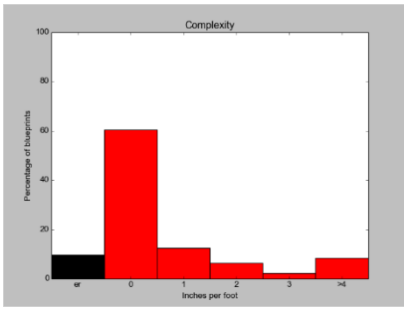
Para la resolución de la práctica se recomienda el uso de las siguientes funciones sobre NumPy arrays: `np.count_nonzero`, `np.isnan`, `np.sum`

El código deberá presentar como mínimo una función en Python llamada `computeStats` que reciba tres argumentos:

- Dos rutas donde se encuentran los ficheros a comparar
- Y una tercera donde guardar las imágenes correspondientes a cada gráfica.







2. Dado un vídeo correspondiente al tráiler de una película (por ejemplo), utilizar un clasificador cascada (previamente entrenado) para detectar las caras y los ojos de las personas que aparecen en escena. El módulo de detección de objetos de OpenCV incorpora la posibilidad de trabajar con este tipo de clasificadores. Su uso es muy sencillo:

```
faceCascade = cv2.CascadeClassifier(faceCascadePath)
rects = faceCascade.detectMultiScale(image,
                                     scaleFactor = 1.1,
                                       minNeighbors =5,
                                       minSize=(30,30),
                          flags = cv2.CASCADE_SCALE_IMAGE)
```

La primera llamada carga un clasificador previamente entrenado (utilizar los ficheros .xml proporcionados con la práctica para caras y ojos). La segunda detecta el objeto de interés, buscándolo en diferentes escalas de la imagen.

Nuestro programa en Python debe aceptar al menos un parámetro de entrada `-video` donde se indica la ruta del vídeo a cargar, y otro parámetro `-out` que indique la ruta donde se va a guardar el video resultante de la detección. La llamada a nuestro detector debe poder hacerse de la siguiente manera:

python eyefacedetector.py –video=./media/avengers.avi

–out=./avengers_result.avi

Para ello utilizaremos el paquete `argparse`:

```
import argparse
…
ap = argparse.ArgumentParser()
ap.add_argument("-v", "--video", required = False,
    help = "path to where the video file resides")
args = vars(ap.parse_args())
…
path = args['video']
```

3. Seguimos interesados en reconocer objetos en imágenes. Esta vez es el turno de peatones. Para ello utilizaremos la secuencia de imágenes proporcionada por el profesor. Para leer de una forma sencilla el directorio de imágenes se recomienda el uso del paquete `glob`.
   En esta ocasión utilizaremos un detector basado en histogramas de gradientes orientados (previamente entrenado para detectar personas).

```
hog = cv2.HOGDescriptor()
hog.setSVMDetector( cv2.HOGDescriptor_getDefaultPeopleDetector()
)
…

rects, weights = hog.detectMultiScale(img, winStride=(8,8),
                                padding=(32,32), scale=1.05)
```

   Nuestro programa en Python debe aceptar al menos un parámetro de entrada `–images` donde se indica la ruta donde se encuentra almacenada la secuencia, y otro parámetro `–out` que indique la ruta donde se va a guardar el video resultante de la detección. La llamada a nuestro detector debe poder hacerse de la siguiente manera:

   python pedestriandetector.py –images=./images

   –out=./pedestrian_result.avi

4. Estamos interesados en identificar imágenes. Para ello vamos a trabajar en un enfoque basado en la detección y *matching* de puntos característicos. Por suerte, OpenCV presenta un módulo que implementa diferentes descriptores (SURF, SIFT, ORB, etc…) que simplifican la identificación de puntos característicos en imagen. Para comenzar a familiarizarnos con estos puntos característicos lo primero que haremos será seguir el tutorial que se nos proporciona desde la documentación de OpenCV. En él se

detallan los pasos a seguir para identificar puntos homólogos en dos imágenes distintas.

Una vez tengamos el código del tutorial funcionando lo vamos a utilizar para identificar imágenes similares. Para ello nos crearemos un conjunto de fotos (podemos hacerlas con el móvil o descargarlas de internet) de carátulas de libros, cds, dvds…(lo que el alumno considere). A partir de aquí el alumno desarrollará una función en Python que reciba dos argumentos:

- Ruta donde se encuentra la imagen a emparejar
- Colección de imágenes candidatas

```
python mediamatcher.py -query=./cover_The_Hobbit
                                -covers=./my_media_database/
```

La función deberá mostrar en una ventana la imagen con la queda emparejada la imagen de consulta. De todas las imágenes se seleccionará aquella que obtenga mayor número de *matches* (o puntos emparejados). Fijaremos un umbral de un mínimo de 50 puntos emparejados. Si varias imágenes superan dicho umbral, se mostrará aquella que presente mayor número de *matches*.

5. Queremos desarrollar una aplicación de visión artificial que nos permita reconocer dígitos manuscritos. Nuestro sistema constará de dos partes. Una primera dedicada a entrenamiento donde el sistema aprenderá a reconocer dígitos, y una segunda de validación, donde demostraremos el correcto funcionamiento del sistema entrenado.

   Para el entrenamiento del sistema utilizaremos un dataset de dígitos llamado MNIST. En concreto descargaremos una versión en formato csv del mismo que simplifica la carga de información en memoria. Para entrenar nuestro sistema el profesor proporciona (entre otros archivos) un fichero Python `train.py`. La labor del alumno en esta parte será la de entender el código y entrenar el sistema en su ordenador. El sistema entrenado resultante se almacenará en disco para su posterior uso. NOTA: probablemente para poder entrenar el sistema sea necesaria la instalación del paquete mahotas (`pip install mahotas`)

   Tras el entrenamiento, el siguiente paso consiste en poner a prueba al sistema. Para ello el alumno debe desarrollar una función en Python llamada `classify.py`. Dicha función recibirá dos argumentos. El primero será el modelo previamente entrenado, y el segundo, una imagen que contenga dígitos manuscritos (coger un folio, escribir una secuencia de números y hacerle una foto con el móvil). Un ejemplo de la llamada se muestra a continuación:

python classify.py –model=./trained_svm

–image=./imagen.png

La función `classify.py` debe mostrar la imagen de entrada. Cada dígito deberá aparecer recuadrado y debajo se deberá mostrar el digito reconocido por el sistema (ver imagen al final del texto). Para ello se deben seguir los siguientes pasos:

1. Cargar el modelo utilizando la librería cPickle (`cPickle.load`)
2. Cargar la imagen y convertirla a escala de grises
3. Suavizar la imagen con un filtrado gaussiano de 5x5 y calcular sus bordes con `cv2.Canny`
4. Detectar los contornos en la imagen resultante. Si todo va bien tendremos uno por dígito. Si aparecen más, podemos utilizar la relación de aspecto de los dígitos para filtrar falsos contornos.
5. Una vez que tengamos un contorno por dígito, será nuestro sistema previamente entrenado el que determine de que dígito se trata. Para ello, dado un contorno calcularemos su bounding box y recortaremos dicha región de la imagen en escala de grises.
6. Por cada recorte, umbralizaremos utilizando otsu, e invertiremos el resultado (en MNIST el dígito se presenta en blanco y el fondo en negro).
7. Dado el recorte umbralizado, aplicaremos las operaciones de `deskew` y `center_extent` de la misma forma que lo hicimos en el entrenamiento.
8. Calcular el histograma de gradientes orientados del recorte y pasárselo al sistema para que identifique de que dígito se trata:

```
hist = hog.describe(crop_region)
digit = model.predict(hist)[0]
```

9. Dibujar el bounding box de cada dígito y debajo el dígito reconocido por el sistema (ver imagen)