Extracción de características en imágenes

Descriptores

ugr | Universidad de Granada

Máster en Ciencia de Datos e Ingeniería de los Computadores

Besay Montesdeoca Santana Graduado en Ingeniería Informática

1. Implementación del descriptor Local Binary Pattern(LBP)

En esta práctica se eligió implementar el descriptor LBP para realizar la extracción de los rangos de las distintas imágenes facilitadas en la actividad. En este apartado se pide implementar dos funciones matlab, la primera, llamada LBPu, será la encarga de calcular los códigos LBP uniformes correspondientes a cada uno de los píxeles:

```
function lbp = LBPu(im, T)
   % Implementamos la estrategia zero-padding para resolver el problema
   % de los bordes
   imSize = size(im);
   imP = zeros(imSize(1) + 2, imSize(2) + 2);
   imNew = imP;
   imP(2:imSize(1) + 1, 2:imSize(2) + 1) = im;

   % Creamos una tabla para realizar el etiquetado de cada pixel. En la
   % primera columna tenemos el patrón (en decimal) y en la segunda su
   % correspondiente etiqueta. En este proceso se calculan el número de
   % transiciones de 1 a 0 para realizar el etiquetado correctamente.
   tab = zeros(256, 2);
   label = 0;
   for i = 1:256
       tab(i) = i;
       bin = dec2bin(i-1, 8);
       bin = [bin, bin(1)];

       % Cálculo del número de transiciones
       state = bin(1);
       cont = 0;
       for t = 1:9
           if(bin(t) ~= state)
               cont = cont + 1;
               state = bin(t);
           end
       end

       % Etiquetado
       if(cont <= 2)
           tab(i, 2) = label;
           label = label + 1;
       else
           tab(i, 2) = 58;
       end
   end

   % Cálculo LBP de cada pixel. Para cada uno de ellos se consulta en la
   % tabla anteriormente calculada para obtener la etiqueta y escribirla
   % en la nueva imagen.
   for i = 2:imSize(1)+1
       for j = 2:imSize(2)+1

           % Calculamos el vector binario
           binaryPattern = zeros(1,8);
           if(abs(imP(i-1, j-1) - imP(i, j)) >= T) binaryPattern(1) = 1; else binaryPattern(1) = 0;
end
           if(abs(imP(i-1, j) - imP(i, j)) >= T) binaryPattern(2) = 1; else binaryPattern(2) = 0; end
           if(abs(imP(i-1, j+1) - imP(i, j)) >= T) binaryPattern(3) = 1; else binaryPattern(3) = 0;
end
           if(abs(imP(i, j+1) - imP(i, j)) >= T) binaryPattern(4) = 1; else binaryPattern(4) = 0; end
```

```
            if(abs(imP(i+1, j+1) - imP(i, j)) >= T) binaryPattern(5) = 1; else binaryPattern(5) = 0;
end
            if(abs(imP(i+1, j) - imP(i, j)) >= T) binaryPattern(6) = 1; else binaryPattern(6) = 0; end
            if(abs(imP(i+1, j-1) - imP(i, j)) >= T) binaryPattern(7) = 1; else binaryPattern(7) = 0;
end
            if(abs(imP(i, j-1) - imP(i, j)) >= T) binaryPattern(8) = 1; else binaryPattern(8) = 0; end

            imNew(i, j) = tab(bin2dec(int2str(binaryPattern))+ 1, 2);
        end
    end

    % Devolvemos la imagen sin el padding
    lbp = imNew(2:imSize(1)+1, 2:imSize(2)+1);

end
```

La segunda función que se pedía implementar era la encargada de recibir la matriz con los LBP de una ventana de tamaño 128x64 y se encarga de calcular el descriptor respetando tamaño de los bloques y el solapamiento indicado:

```
function x = lbp_features(patch, windowSize, desp)
    patchSize = size(patch);

    % Número de ventanas posibles que recorrerán la imagen y por tanto número de histogramas
    nWindows = (patchSize(1)/windowSize(1) + ((patchSize(1)/windowSize(1))-1) *
((windowSize(1)/desp)-1)) *
...
    (patchSize(2)/windowSize(2) + ((patchSize(2)/windowSize(2))-1) * ((windowSize(2)/desp)-1));

    x = zeros(1, nWindows*59);
    index = 1;
    for i = 1:desp:patchSize(1)- desp
        for j = 1:desp:patchSize(2)-desp
            h = hist(patch(i:i+windowSize(1)-1, j:j+windowSize(2)-1), 59);
            x(index:index+58) = sum(h');
            index = index + 59;
        end
    end
    x = x/norm(x);
end
```

2. Cálculo de características

Seguidamente también era necesario implementar un Script que, utilizando las funciones anteriores, lea las imágenes y calcule sus características, creando las matrices con los datos y etiquetas que se utilizarán para el entrenamiento y evaluación del clasificador. Para ello se ha implementado una función llamada *extractFeatures* que recibe como parámetros de entrada la ruta donde se ubican las imágenes, el nombre del fichero csv donde se guardaran las características y por último el umbral T de corte para el algoritmo LBP.

```
function [] = extractFeatures(pathFiles, nameFile, T)
    d = dir(strcat(pathFiles, '*.png'));
    r = zeros(length(d), 6195);
    for i = 1:length(d)
        path = strcat(pathFiles, d(i).name);
        im = rgb2gray(imread(path));
```

```
        patch = LBPu(im, T);
        no = lbp_features(patch, [16, 16], 8);
        disp([i, length(d)]);
        r(i,:) = no;
    end
    csvwrite(nameFile, r);
end
```

3. Entrenamiento y clasificación (R)

Una vez extraídas las características de las imágenes se procedió a construir los modelos que tomarán como entrada estas características y realizarán la clasificación para detectar peatones en las imágenes.

Para obtener la base de datos completa primero fue necesario realizar un pequeño preprocesamiento donde fue necesario añadir manualmente las etiquetas y pegar los distintas datos:

```
rm(list = ls())

# Cargamos los datos
trainP = read.csv("data/trainPedestrians2.csv", header=FALSE)
trainP = cbind(trainP, sapply(1:dim(trainP)[1], function(n){return("P")})) # Le añadimos la clase correspondiente (Pedestrians -> P)
colnames(trainP)[length(trainP)] = "Class"

trainB = read.csv("data/trainBackground2.csv", header=FALSE)
trainB = cbind(trainB, sapply(1:dim(trainB)[1], function(n){return("B")})) # Le añadimos la clase correspondiente (Background -> B)
colnames(trainB)[length(trainB)] = "Class"

dataTrain = rbind(trainP, trainB)

testP = read.csv("data/testPedestrians2.csv", header=FALSE)
testP = cbind(testP, sapply(1:dim(testP)[1], function(n){return("P")}))
colnames(testP)[length(testP)] = "Class"

testB = read.csv("data/testBackground2.csv", header=FALSE)
testB = cbind(testB, sapply(1:dim(testB)[1], function(n){return("B")}))
colnames(testB)[length(testB)] = "Class"

dataTest = rbind(testP, testB)
```

Una vez obtenido el dataset, el siguiente paso fue implementar los modelos de clasificación y realizar el entrenamiento. Para esta práctica se han utilizado tres modelos distintos: el árbol de decisión, el Random Forest y el SVM.

Como medidas de rendimiento se ha implementado una función que realiza la predicción y calcula la Precisión, la Sensibilidad, la Especificidad y el G-mean para dicha predicción del modelo a evaluar:

```
library(randomForest)
library(caret)
library(e1071)
```

```
library(party)

# Función que realiza la predicción y el cálculo de las medidas
prediction = function(model, data){
 testPred = predict(model, newdata = data)

 # Cálculo de las medidas de precisión
 results = confusionMatrix(table(testPred, data[,ncol(data)]))
 accuracy = results$overall[1]
 sensitivity = results$byClass[1]
 specificity = results$byClass[2]
 gmean = sqrt(sensitivity * specificity)
 m = c(accuracy, sensitivity, specificity, gmean)
 names(m) = c("Accuracy", "Sencivility", "Specificity", "Gmean")
 print(m)
}

# Realizamos un shuffle de los datos
dataTrain = dataTrain[sample(nrow(dataTrain)),]
dataTest = dataTest[sample(nrow(dataTest)),]

# Modelos
mtree = ctree(Class ~ ., data = dataTrain)
rfM = randomForest(Class ~ ., dataTrain)
svmM = svm(Class ~ ., data = dataTrain)

prediction(mtree, dataTest)
prediction(rfM, dataTest)
prediction(svmM, dataTest)
```

Podemos observar en la siguiente tabla que el modelo que mejor resultados ofrece es el basado en las Máquinas de Soporte Vectorial(SVM). Como podemos apreciar, con dicho modelo obtenemos tasas de acierto superiores con respecto a los otros modelos aunque realmente por muy poco, sobretodo con respecto al modelo Random Forest.

Modelos	Precisión	Sensibilidad	Especificidad	G-mean
Árbol de Decisión	0.8509091	0.8200000	0.8766667	0.8478601
SVM	0.9481818	0.9240000	0.9683333	0.9459070
Random Forest	0.9145455	0.8980000	0.9283333	0.9130407