Capítulo 8

Visão computacional em *python* utilizando as bibliotecas *scikit-image* e *scikit-learn*

Romuere R. V. Silva, José G. F. Lopes, Flávio H. D. Araújo, Fátima N. S. Medeiros e Daniela M. Ushizima

Abstract

Computer vision algorithms use matrices as input and produce a more compact information as output. These algorithms play a major role in applications of object classification and scene identification captured digitally. Nowadays, there are several libraries that implement computer vision algorithms. Our focus is to present python-based libraries, such as scikit-image and scikit-learn, that provide functions for digital image processing and machine learning, respectively. These open-source libraries allow writing sophisticate algorithms to solve problems in classification, clusterization, and content-based image retrieval. We illustrate the application of these libraries by describing a computer vision workflow that detects real situations of traffic imprudence from surveillance camera images.

Resumo

Algoritmos de visão computacional utilizam matrizes como entrada e produzem uma informação mais compacta como saída. Esses algoritmos são utilizados em aplicações para classificação de objetos e cenas capturadas digitalmente. Atualmente existem diversas bibliotecas que implementam algoritmos de visão computacional. Nosso foco é apresentar as bibliotecas scikit-image e scikit-learn, que possuem funções de processamento digital de imagens e aprendizado de máquina, respectivamente. Essas bibliotecas de código aberto permitem a escrita de sofisticados algoritmos para resolver problemas em classificação, clusterização e recuperação de imagens baseada em conteúdo. Nós ilustramos a aplicação dessas bibliotecas descrevendo um sistema de visão computacional para um problema real de imprudência no trânsito utilizando imagens de câmeras.

8.1. Introdução

A área de Visão Computacional utiliza modelos de descrição de objetos e cenas capturadas digitalmente para tomada de decisões, automatização de tarefas, seja na indústria de manufatura, farmacêutica, automobilística, dentre outras.

Atualmente existem diversas bibliotecas que implementam algoritmos de visão computacional tais como: *OpenCV* e *Matlab*. No entanto, as opções anteriores apresentam maiores desafios dado que a biblioteca *OpenCV* é geralmente usada por desenvolvedores mais avançados, e a licença do *software Matlab* apresenta custo elevado que inviabiliza seu uso em circunstâncias mais amplas, como em sala de aula, projetos de estudantes e pequenas empresas.

As bibliotecas *scikit-image* e *scikit-learn* são de código aberto e possuem uma extensa documentação com inúmeras funções para viabilizar o processamento digital de imagens, classificação, clusterização e recuperação de imagens baseada em conteúdo. Essas bibliotecas são utilizadas em diversas aplicações que dependem da linguagem *python* para desenvolvimento.

Esse capítulo tem o objetivo de introduzir os principais comandos da linguagem *python* que habilitam o desenvolvedor a realizar processamento de imagens, includindo a utilização das bibliotecas *scikit-image* e *scikit-learn* aplicadas a um problema real de Visão Computacional.

8.2. Visão Computacional

Com a popularização dos dispositivos eletrônicos para captura de imagens e vídeos, a análise automática desses dados através de programas computacionais tornou-se essencial nas mais variadas áreas de domínio da ciência. Para isso, algoritmos e representações são desenvolvidos para permitir que uma máquina reconheça objetos, pessoas, cenas e atividades. Nestes termos, o computador passa a desempenhar tarefas complexas e ser capaz de reconhecer objetos em figuras e filmes.

De forma geral, algoritmos de visão computacional utilizam matrizes bidimensionais ou hiperdimensionais como entrada e produzem uma informação mais compacta como saída, como por exemplo a classificação de um objeto. No caso de decisões em sistemas de controle de qualidade de materiais, a tomada de decisão geralmente é um módulo no sistema de monitoramento, responsável pela detecção de rachaduras a partir de imagens produzidas durante o processo de deformação para teste de resistência de materiais cerâmicos. A Figura 8.1 ilustra a relação entre as áreas de visão computacional, computação gráfica e processamento digital de imagens. Usualmente estas áreas se confundem, contudo elas se diferenciam principalmente pelo tipo de dados na entrada e saída do sistema.

8.3. Problema Abordado

É extensa a lista de aplicações possíveis dos algoritmos de visão computacional, que são particularmente relevantes na realização de tarefas insalubres e repetitivas que de outra forma seriam delegadas a seres humanos. Outras aplicações são auxiliar na percepção robótica, em agentes autônomos e aprimorar habilidades humanas tais como interface

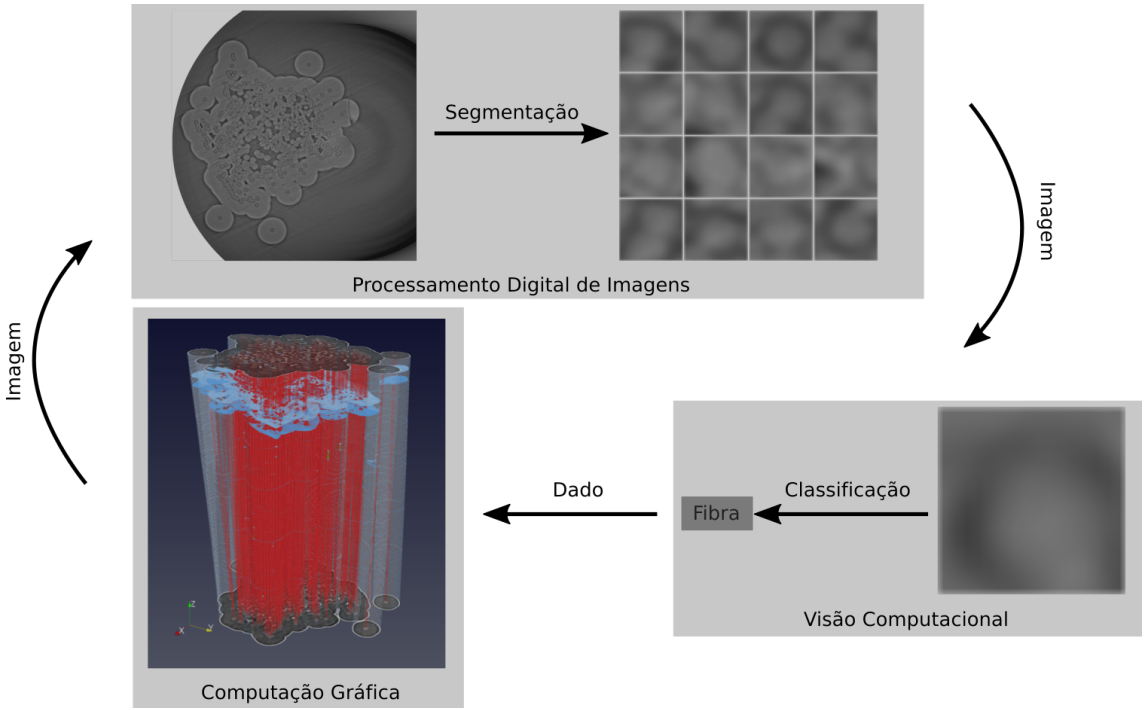

Figura 8.1: Visão Computacional × Computação Gráfica × Processamento Digital de Imagens. A entrada e a saída é o que define cada área. Adaptado de [Ushizima et al. 2014].

humano-computador e visualização.

Dentre as possíveis aplicações em visão computacional utilizaremos imagens de trânsito com foco na detecção de capacete de motociclistas. Alguns fatores determinam o interesse nesse tipo aplicação: 1) o número de acidentes de trânsitos envolvendo motociclistas tem aumentado nos últimos anos em vários países; 2) motocicletas são os veículos que se tornaram mais populares devido ao seu baixo custo; 3) o principal equipamento de segurança é o capacete e, apesar disso, muitos motociclistas não fazem uso do equipamento ou mesmo o fazem incorretamente [Silva et al. 2017]. Utilizaremos uma base de imagens que possui 255 imagens com motociclistas que estão ou não fazendo uso do capacete[1] adequadamente. A partir dessa base será possível realizar experimentos de segmentação para determinação de uma região de interesse, extração e seleção de atributos, e classificação. A Figura 8.2 mostra as principais etapas para a resolução do problema abordado.

8.4. Segmentação de Imagens

O processo de agrupamento dos pixels em uma componente conexa relevante é denominado de segmentação. Esse processo consiste em subdividir uma imagem em regiões ou objetos segundo um critério significativo. Agrupar pixels ou conjuntos de pixels está diretamente relacionado ao problema em estudo e geralmente requer algoritmos iterativos.

A separação dos pixels relativos a cada objeto, ou região, é uma etapa fundamental

[1]Imagens disponíveis em `https://github.com/romuere/databases`

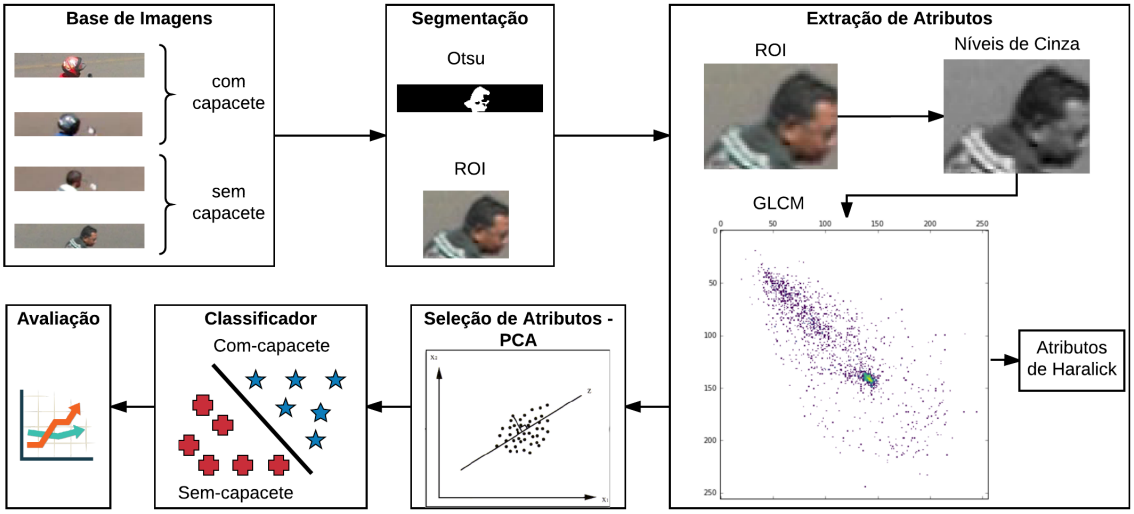

Figura 8.2: Metodologia fundamental para exploração de imagens usando visão computacional conforme proposto nesse tutorial.

para o sucesso da análise da imagem. Embora o ser humano possa facilmente identificar regiões de características semelhantes ou objetos presentes em uma imagem, a realização da mesma tarefa por um computador requer a implementação de algoritmos especialistas em analisar características dos pixels ou da distribuição dos mesmos na imagem.

Em outras palavras, a segmentação é um processo que divide uma região espacial (imagem) R em n sub-regiões. O valor de n varia de acordo com o problema a ser trabalhado. Por exemplo, no processamento e análise de imagens de células, a segmentação frequentemente consiste em identificar três regiões ($n = 3$), a saber, fundo (*background*), citoplasma e núcleo.

Durante o processamento digital de imagens, uma das formas de particionar a imagens em subconjuntos de pixels é realizar limiarização através do método clássico de Otsu [Otsu 1979]. Nesse método, assume-se que a imagem possui duas classes de pixels: de primeiro plano e de fundo. Então, calcula-se o valor ótimo de limiar que separa as classes de modo que a variação intra-classe seja mínima. O cálculo deste limiar é dado por:

$$\sigma_w^2(t) = w_0(t)\sigma_0^2(t) + w_1(t)\sigma_1^2(t), \tag{1}$$

em que os pesos $w_0(t)$ e $w_1(t)$ são as probabilidades das duas classes serem separadas pelo limiar t, sendo as variâncias dessas classes σ_0 e σ_1.

Para calcular o probabilidade da classe, utiliza-se o histograma com $L = 256$ níveis de cinza da imagem:

$$\begin{aligned} w_0(t) &= \sum_{i=0}^{t-1} p(i) \\ w_1(t) &= \sum_{i=t}^{L} p(i) \end{aligned} \tag{2}$$

O algoritmo é definido da seguinte forma:

1. Calcule o histograma e as probabilidades para cada nível de intensidade;
2. Selecione os valores inicias para $w_i(0)$;
3. Passe por todos os valores possíveis de limiar de t = 0, ..., 255;
   (a) Atualize os valores de w_i;
   (b) Calcule $\sigma_j^2(t)$;
4. o valor ótimo será o máximo; $\sigma_j^2(t)$.

Com o valor de limiar calculado, podemos obter uma segmentação como mostrado na Figura 8.3.

Figura 8.3: Exemplar da base de imagens após a segmentação utilizando Otsu.

8.4.1. Código em *python*

Assim como a maioria das bibliotecas de processamento digital de imagens, o *scikit-image* possui a implementação do limiar de Otsu. O objetivo de utilizar essa rotina foi obter uma Região de Interesse (*Region of Interest* - ROI) com o recorte do segmento contendo a cabeça do motociclista.

O Código Fonte 8.1 mostra os passos para realizar a segmentação de uma imagem baseado no limiar de Otsu. Após transformar a imagem em níveis de cinza em uma imagem binária (Linha 9), é feito um pós-processamento para remover pequenas regiões que foram segmentadas e que não fazem parte da região de interesse (Linhas 10 a 20), como mostra a Figura 8.4 com exemplos dessa etapa. Isso é feito para evitar que pequenas regiões, como cantos ou objetos, que não pertencem à região da cabeça do motociclista, sejam incluídos no recorte. O passo seguinte é fazer o recorte baseado na região segmentada e obter a imagem de onde serão extraídos os atributos para a posterior classificação do segmento (Figura 8.5).

```
1 from skimage.io import imread #funcao do pacote skimage para leitura de
      imagens
  from skimage.color import rgb2grey #funcao que transforma uma imagem
     RGB (colorida) em niveis de cinza
3 from skimage.filters import threshold_otsu #funcao que implementa o
     limiar de Otsu
  import numpy as np #pacote python que oferece suporte a arrays e
     matrizes multidimensionais
5 from skimage.measure import label,regionprops

7 path_image = "/home/users/image.png" #endereco da imagem
```

```
imagem = imread(path_image) #leitura da imagem
grey_image = rgb2grey(imagem)#Converte as imagens em niveis de cinzas
otsu = threshold_otsu(grey_image)/255 #Calcula o limiar utilizando o
    metodo de Otsu
img_otsu = grey_image < otsu #pixels com valores menores que 'otsu'
    serao brancos, caso contrario serao pretos

#-----------------------remove pequenas regioes-----------------------#
label_img = label(seg, connectivity = grey_image.ndim)#detecta regioes
    nao conectadas
props = regionprops(label_img)#calcula propriedade importantes de cada
    regiao encontrada (ex. area)

#Convert todas as regioes que possuem um valor de area menor que a
    maior area em background da imagem
area = np.asarray([props[i].area for i in range(len(props))])#area de
    cada regiao encontrada
max_index = np.argmax(area)#index da maior regiao
for i in range(len(props)):
    if(props[i].area < props[area].area):
        label_img[np.where(label_img == i+1)] = 0#regiao menor que a
    maior eh marcada como background

label_img = label_img*1#transforma imagem de valores booleanos para
    inteiros

#-------------------recorte da regiao de interesse-------------------#
# Obtendo os limites verticais das imagens segmentadas
ymin = np.min(np.where(label_img == True)[1])
ymax = np.max(np.where(label_img == True)[1])
imagem_cortada = imagem[:,ymin:ymax,:]
```

Código Fonte 8.1: Método de Otsu aplicado a duas imagens.

Figura 8.4: Imagem processada após a retirada doe artefatos.

8.5. Extração de Atributos

Atributos de forma, cor e textura são propriedades presentes em objetos registrados na maioria das imagens. Estas características podem ser quantificadas e usadas como descritores da imagem ou dos objetos numa cena. Esses descritores compõem um conjunto de propriedades representadas em um vetor de escalares, denominado descritor da imagem.

Figura 8.5: Exemplares da base de imagens após a determinação da região de interesse.

Assim sendo, cada objeto pode ser representado por um ponto em um espaço R^n, definido em termos das n características extraídas. Em alguns casos, é desejável que os descritores apresentem a propriedade de invariância a transformações afins, ou seja, invariante a rotação, escala, e translação.

Podemos representar uma região de dois modos: através de características externas, pertencentes a fronteira, ou em termos de características internas, pertencentes aos pixels que constituem a região. Escolher o modo de representação consiste em um passo essencial para permitir com que as imagens sejam processadas em um computador. O segundo passo é descrever a região com base na representação escolhida.

Após a extração de atributos, todas as informações obtidas são agrupadas em um vetor de escalares denominado vetor de atributos. A Figura 8.7 mostra um exemplo de extração de atributos.

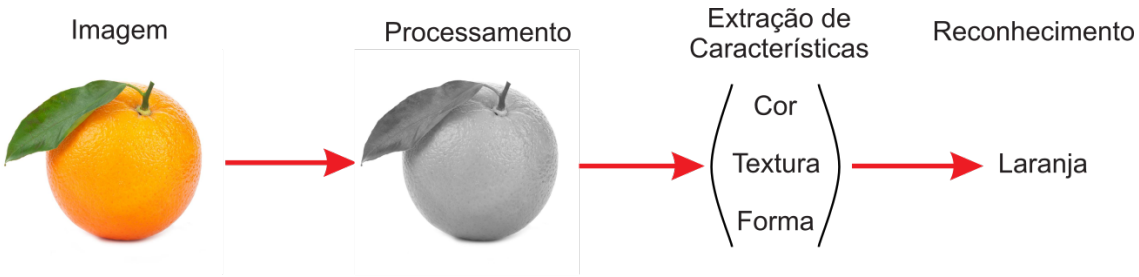

Figura 8.6: Extração de atributos para reconhecimento de padrões.

8.5.1. *Matriz de Coocorrência de Níveis de Cinza*

A matriz de coocorrência de níveis de cinza, do inglês *Grey-Level Co-occurrence Matrix* (GLCM) é uma técnica que tem como base a análise de textura em imagens. Na GLCM são analisadas as coocorrências existentes entre pares de pixels através de algum padrão. A matriz GLCM é sempre quadrada e armazena as informações das intensidades relativas dos pixels. Por este motivo, as imagens utilizadas são sempre em tons de cinza [Haralick 1973].

As probabilidades de coocorrências são calculadas entre dois níveis de cinza i e j, utilizando uma orientação Θ e uma distância conhecida como espaçamento entre pares de pixels. Essa orientação pode assumir os valores 0^o, 45^o, 90^o e 135^o graus. Para cada relacionamento espacial possível (distância e orientação) existe uma matriz de coocorrência. Desse modo, todas as informações sobre a textura de uma imagem estarão contidas nessa matriz [Baraldi and Parmiggiani 1995].

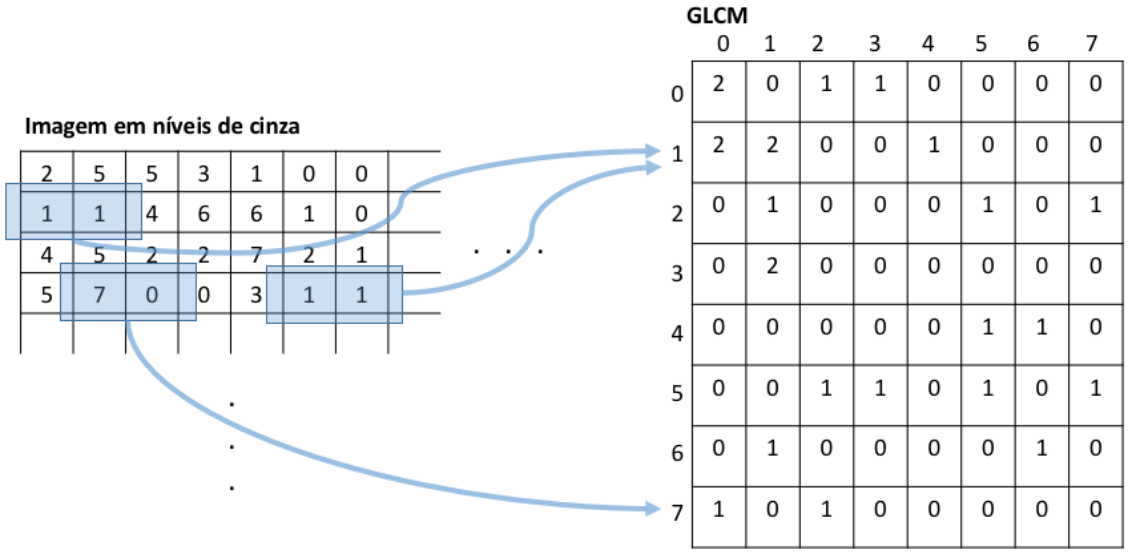

Figura 8.7: Exemplo de cálculo da matriz GLCM com espaçamento entre pares de pixels igual a 1 e $\Theta = 0^o$. Intensidade de pixels da imagem (esquerda), GLCM (direita).

A Figura 8.8 mostra a matriz de coocorrência calculada para duas amostras da base de imagens de motociclistas. Pode-se observar a diferença do padrão de espalhamento entre as matrizes (Figuras 8.8(b) e (d)), onde a primeira concentra informação ao longo da diagonal, enquanto a outra apresenta transições entre níveis de cinza mais heterogêneas.

Haralick [Haralick 1973] definiu 14 características significativas para a GLCM há mais de quarenta anos atrás, e muitas outras medidas derivadas dessas foram acrescentadas [Sabino et al. 2004]. Contudo, a utilização de algumas dessas características pode gerar melhor desempenho do que a utilização de todas. Assim, nesse trabalho são feitos cálculos dos seguintes atributos de textura: contraste (Equação 3), dissimilaridade (Equação 4), homogeneidade (Equação 5), energia (Equação 6), correlação (Equação 7) e segundo momento angular (ASM) (Equação 8).

$$Contraste = \sum_{i,j=0}^{L-1} P_{ij}(i-j)^2, \tag{3}$$

onde L é a quantidade de níveis de cinza e P é a GLCM.

$$Dissimilaridade = \sum_{i,j=0}^{L-1} P_{ij}|i-j|. \tag{4}$$

$$Homogeneidade = \sum_{i,j=0}^{L-1} \frac{P_{ij}}{1+(i-j)^2}. \tag{5}$$

$$Energia = \sqrt{\sum_{i,j=0}^{L-1} (P_{ij})^2}. \tag{6}$$

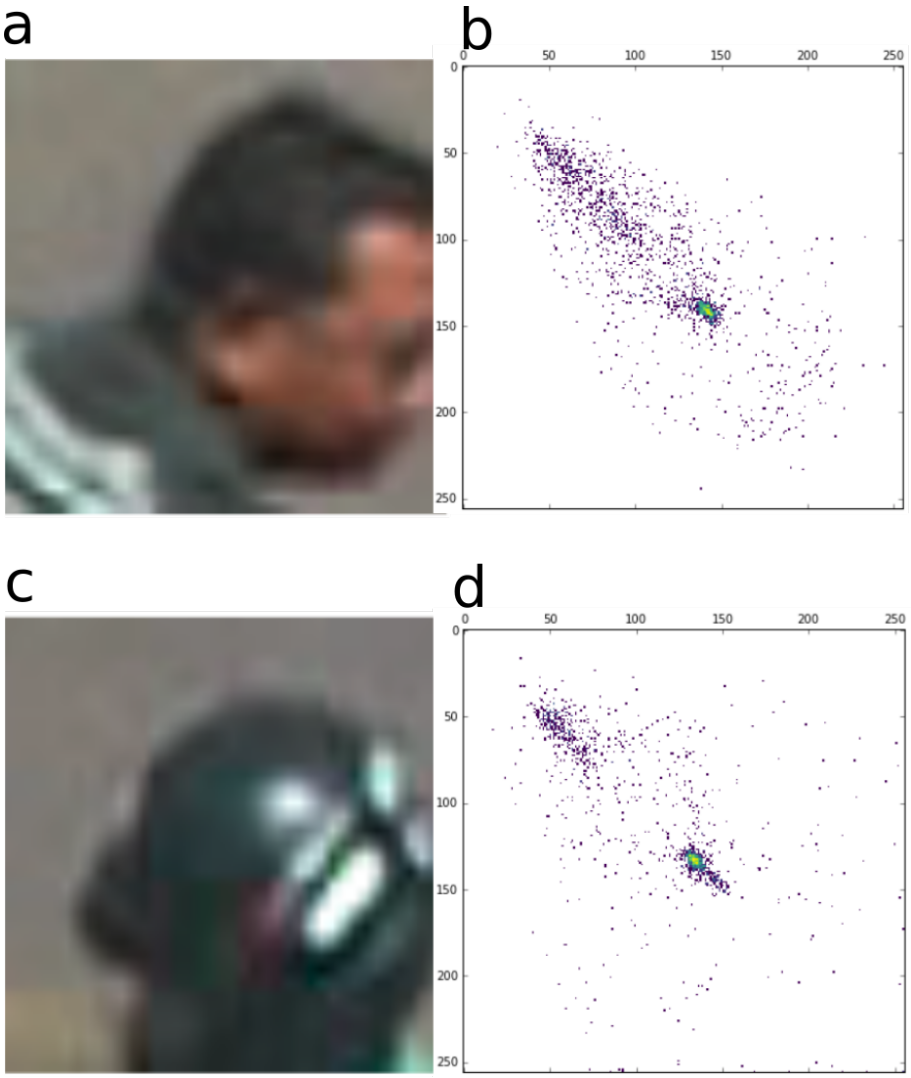

Figura 8.8: Amostras de imagens (a) e (c) e suas respectivas matrizes GLCM (b) e (d).

$$Correlacao = \sum_{i,j=0}^{L-1} \frac{(i-\mu_i)(j-\mu_j)P_{ij}}{\sqrt{(\sigma_i)^2(\sigma_j)^2}}, \tag{7}$$

onde μ é a média e σ é o desvio padrão.

$$ASM = \sum_{i,j=0}^{L-1} (P_{ij})^2. \tag{8}$$

8.5.1.1. Código em *python*

A obtenção da GLCM é realizada em duas etapas sendo a primeira o cálculo da matriz de coocorrências e a segunda o cálculo dos atributos de Haralick a partir dessa matriz. A biblioteca *sckit-image* possui duas funções com esse objetivo: *greycomatrix* é responsável

por calcular a matriz e tem como parâmetros a imagem, o ângulo, e a distância. O parâmetro *normed* é responsável por retornar a matriz normalizada caso seu valor seja atribuído como *True*; e *greycoprops* que recebe como parâmetro a matriz calculada anteriormente e o atributo que se deseja calcular. O Código Fonte 8.5 mostra as etapas de obtenção da GLCM.

```
from skimage.feature import greycomatrix,greycoprops #funcao do pacote
    skimage para calcular a matriz GLCM e os atributos da matriz

image_path = "/home/user/image.png" #endereco da imagem
image = imread(image_path) #leitura da imagem
image_gray = rgb2grey(image) #transformar de RGB para nivel de cinza
d = 1 #distancia entre pixels para GLCM
matrix = greycomatrix(image, [d], [0],normed=True) #calculo da matriz
    em 0 graus
props = np.zeros((6)) #vetor para armazenar atributos
props[0] = greycoprops(matrix,'contrast') #calcula contrast
props[1] = greycoprops(matrix,'dissimilarity') #calcula dissimilaridade
props[2] = greycoprops(matrix,'homogeneity') #calcula homogeneidade
props[3] = greycoprops(matrix,'energy') #calcula energia
props[4] = greycoprops(matrix,'correlation') #calcula correlacao
props[5] = greycoprops(matrix,'ASM') #calcula segundo momento angular
```

Código Fonte 8.2: Cálculo da matriz GLCM e dos atributos de Haralick.

8.6. Aprendizagem de Máquina e Classificação de Imagens

A tarefa de reconhecer padrões equivale a classificar determinado objeto ou situação como pertencente ou não a um certo número de categorias previamente estabelecidas. Em algoritmos de visão computacional o aprendizado de máquina é geralmente utilizado para classificar imagens. Para isso, são utilizados os vetores de atributos da etapa de extração de atributos.

Usualmente são utilizados algoritmos de aprendizado de máquina supervisionados, ou seja, é preciso apresentar dados pré-classificados e "ensinar" o algoritmo a identificar diferentes objetos. Nesse caso, é necessário uma base de dados que contenha ao menos parte dos dados previamente avaliados e rotulados por um especialista. Dessa forma o algoritmo poderá aprender os padrões dos objetos de cada classe. Os algoritmos de aprendizado de máquina supervisionados mais conhecidos são: Máquina de Vetor de Suporte (SVM) [Cortes and Vapnik 1995], Perceptron de Múltiplas Camadas (MLP) [Haykin 2001] e *Random Forest* (RF) [Breiman 2001].

8.6.1. Máquina de Vetor de Suporte - SVM

Os estudos sobre SVM foram introduzidos por Cortes e Vapnik [Cortes and Vapnik 1995]. De um modo geral, a SVM faz um mapeamento do espaço de entrada para um espaço de dimensionalidade maior. Em seguida, é calculado um hiperplano de separação ótimo. O hiperplano é escolhido de modo a maximizar a distância de separação entre as classes [Haykin 2001]. Diz-se que duas classes são linearmente separáveis se existe um hiperplano divisório entre as amostras de classes diferentes. A Figura 8.9 ilustra duas classes linearmente separáveis e seu hiperplano de separação ótimo.

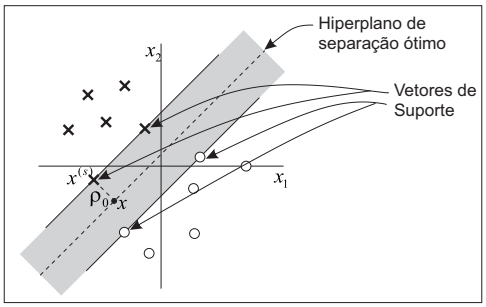

Figura 8.9: Hiperplano de separação ótimo para classes linearmente separáveis. Imagem adaptada de [Haykin 2001].

A SVM é um algoritmo de aprendizado supervisionado, ou seja, possui as fases de treinamento e teste. Na fase de treinamento, os atributos e as classes dos objetos são as entradas do classificador e o objetivo é projetar os vetores de suporte. Os vetores de suporte são utilizados para se obter o hiperplano de separação ótimo (Figura 8.9). Na fase de teste, o hiperplano é utilizado para encontrar uma saída a partir de um vetor de entrada.

Para utilizar SVM para reconhecimento de padrões, o algoritmo requer a transformação de funções não-linearmente separáveis em funções linearmente separáveis. Para isso, é necessário que seja aumentada a dimensionalidade do problema [Cover 1965]. As funções que aumentam a dimensionalidade do espaço de entrada são chamadas funções de *kernel* [Haykin 2001].

As funções de *kernel* são aplicadas nos vetores de entrada, os quais possuem dimensão N, e se obtém um novo vetor de dimensão X, onde $X > N$. Após esse cálculo, são obtidos os vetores de suporte, responsáveis por definir o hiperplano de separação ótimo. O hiperplano estará a uma igual distância (ρ_0) dos vetores de suporte de cada classe, como mostrado na Figura 8.9. Um novo objeto é classificado utilizando o hiperplano ótimo, com cada lado do hiperplano representa uma classe diferente.

8.6.2. *Random Forest*

O algoritmo *Random Forest* [Breiman 2001] é uma combinação de predições de diversas árvores em que cada árvore depende dos valores de um vetor independente, amostrados aleatoriamente e com a mesma distribuição para todas as árvores da floresta. Aqui, floresta é a nomenclatura dada a uma coleção de árvores de decisão. Após a geração de um grande número de árvores, as classes com maior número de votos são eleitas.

Em um *Random Forest*, cada nó é dividido usando o melhor dentre um subconjunto de indicadores escolhidos aleatoriamente naquele nó. Esta estratégia, apesar de um tanto contraditória, funciona adequadamente em comparação com muitos outros classificadores, além de ser robusto a superajuste nos parâmetros [Breiman 2001]. Além disso, é de fácil utilização pois possui apenas dois parâmetros: o número de variáveis no subconjunto aleatório em cada nó e o número de árvores da floresta.

A partir de um vetor de atributos, são gerados outros vetores de atributos, que são embaralhados em relação ao vetor original. É gerado um vetor para cada árvore do *Random Forest*. Em seguida, os vetores de atributos são passados como parâmetro

para as árvores de decisão. Cada árvore irá gerar um resultado para a classificação e, os resultados são combinados obtendo uma saída unificada. A Figura 8.10 mostra os passos do classificador *Random Forest*.

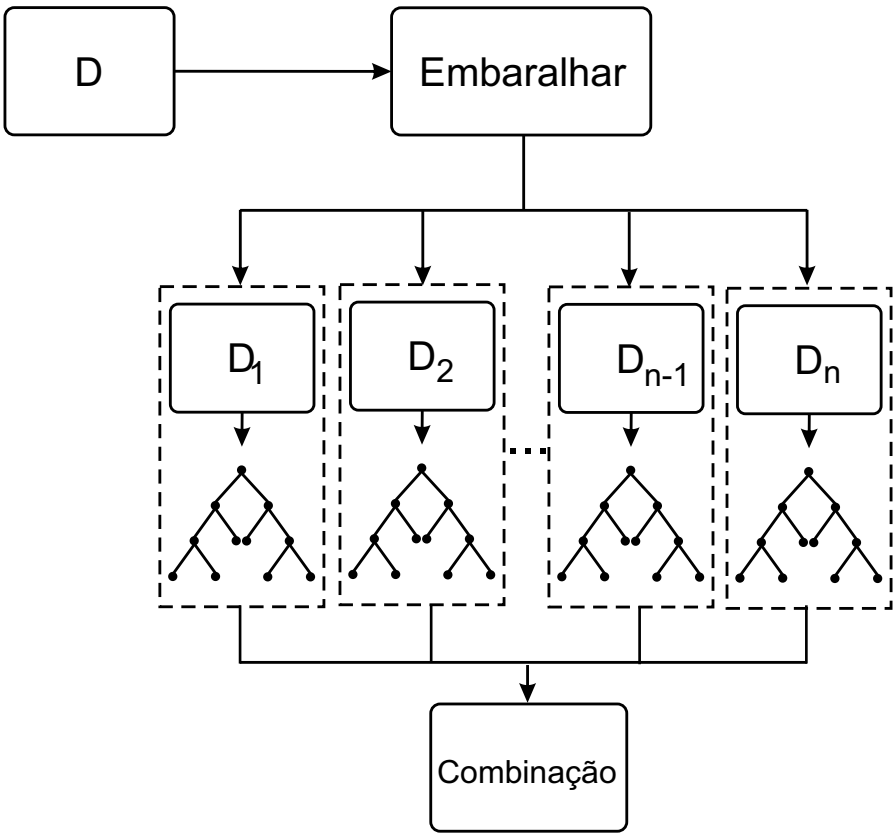

Figura 8.10: Passos do classificador *Random Forest*, onde D é o vetor de atributos original, D_1, $D_2 \ldots$, D_{n-1}, D_n são os vetores de atributos gerados a partir de D.

8.6.2.1. Código em *python*

A classificação de objetos em *python* pode ser realizada através da biblioteca *scikit-learn*. Esta biblioteca permite a utilização de algoritmos de aprendizagem de máquina, clusterização, seleção de atributos, dentre outros. As etapas básicas para classificação são: divisão do conjunto de dados em treino e teste; criação de uma instância do classificador utilizado; treino do classificador; predição utilizando os dados de teste; e cálculo da taxa de acerto seguindo alguma métrica de avaliação de desempenho. O Código Fonte 8.3 mostra as etapas de classificação de um conjunto de dados para os classificadores SVM e *Random Forest*.

```
from sklearn.svm import SVC #SVM para classificacao
from sklearn.ensemble import RandomForestClassifier #RF para
    classificacao
from sklearn.model_selection import train_test_split #pacote para
    dividir os dados em treino e teste
```

```
from sklearn.metrics import accuracy_score #metrica de avaliacao

train = 0.3 #30% dos dados para treinar o classificador
test = 1-train #restante dos dados (70%) para testar o classificador
'''features = matriz com todos os atributos calculado, cada linha
    possui os atributos de uma imagem, labels = vetor com a classe de
    cada imagem na mesma ordem da variavel matriz'''
X_train, X_test, y_train, y_test = train_test_split(features, labels,
    test_size=test) #divide os dados em conjuntos de treino e teste

clf_svm = SVC()#cria uma instancia da SVM
clf_svm.fit(X_train,y_train) #treinamento da SVM
pred_svm = clf.predict(X_test) #teste da SVM
acc_svm = accuracy_score(y_test, pred)

clf_rf = RandomForestClassifier()#cria uma instancia da Random Forest
clf_rf.fit(X_train,y_train) #treinamento da Random Forest
pred_rf = clf.predict(X_test) #teste da Random Forest
acc_rf = accuracy_score(y_test, pred) #compara os vetores de teste e
    predito para o calculo da taxa de acerto
```

Código Fonte 8.3: Classificação de imagens utilizando os classificadores Random Forest e Máquina de Vetor de Suporte.

Observando o Código Fonte 8.3 é possível ajustar alguns valores ou parâmetros, dentre os quais a taxa de treino-teste. O aumento dessa taxa pode melhorar a aprendizagem do classificador, visto que quanto mais dados disponíveis para esta etapa, mais robusto será o modelo a ser utilizado no teste do classificador. O *Random Forest* possui alguns parâmetros que podem melhorar a classificação dos dados, o principal deles é o número de árvores utilizadas que é geralmente padronizada e igual a 10. Em relação à SVM podemos modificar o tipo de kernel e seu coeficiente. A medida de avaliação de desempenho também pode ser modificada pois dependendo da aplicação a medida de acurácia pode não ser a melhor opção.

8.6.3. Análise de Componentes Principais

A análise por componentes principais, do inglês *Principal components analysis* (PCA) foi baseada na transformada de Karhunen–Loève. Essa transformada visa decompor uma matriz de dados em um conjunto de coeficientes, posteriormente esses coeficientes são quantizados e codificados para que se tenha uma determinada taxa de compressão [Pedrini 2007].

Considere um espaço multiespectral onde cada ponto tem seu respectivo vetor:

$$X = \begin{bmatrix} \vec{x_1} \\ \vec{x_2} \\ \vdots \\ \vec{x_k} \end{bmatrix} \tag{9}$$

O valor médio da posição dos pontos é dado por:

$$\vec{m_x} = E(X) = \frac{1}{K} \sum_{k=1}^{K} \vec{x_k}, \tag{10}$$

em que $E(.)$ é o operador esperança, $\vec{m_x}$ é o vetor médio de X e $\vec{x_k}$ são os vetores que representam cada ponto num total de K pontos.

A matriz de covariância de X é definida por:

$$C_x = \frac{1}{K-1} \sum_{k=1}^{K} (x_k - m_x)(x_k - m_x)^t. \tag{11}$$

A matriz de covariância tem como função medir a dispersão dos dados, onde os elementos da diagonal representam a variância, e os restantes a covariância das amostras [Richards 2006]. Uma vez que C_x é real e simétrica, é possível encontrar n autovetores. É importante para a análise de componentes principais, verificar se existe um novo sistema de coordenadas no qual as amostras seriam representadas com mínima correlação [Richards 2006]. Se $\vec{y}$ representa os dados em um novo sistema de coordenadas temos que:

$$\vec{y} = D^t(\vec{x} - \vec{m_x}), \tag{12}$$

em que t representa o operador de transposição e D é a matriz de transformação que mapeia as amostras que estavam no espaço $\vec{x}$ para o $\vec{y}$ essa expressão é chamada de *transformada de Hotelling*. As linhas da matriz D são formadas pelos autovetores de C_x ordenados de tal forma que a primeira linha corresponde ao autovetor relacionado ao maior autovalor e a última linha corresponde ao autovetor correspondente ao menor autovalor [Gonzalez 2007].

A matriz de covariância no novo espaço pode ser definida como:

$$C_y = DC_xD^t \tag{13}$$

em que C_y é diagonal e D é a matriz de rotação. Portanto, C_y pode ser representado da seguinte forma:

$$C_y = \begin{bmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \lambda_n \end{bmatrix}$$

em que λ_i representa os n autovalores de C_x. Os autovalores são organizados de tal maneira que: $\lambda_1 > \lambda_2 \cdots > \lambda_n$ [Richards 2006]. Por definição os elementos de C_y serão as variâncias da amostra de dados mapeadas no novo espaço.

A reconstrução de $\vec{x}$ por meio de $\vec{y}$ pode ser feita da seguinte forma:

$$\vec{x} = D^t \vec{y} + \vec{m_x}. \tag{14}$$

Utilizamos o PCA com o objetivo de diminuir a quantidade de atributos mantendo a acurácia na etapa de classificação.

8.6.3.1. Código em *python*

O Código Fonte 8.4 mostra as etapas para o cálculo do PCA e posterior classificação dos atributos obtidos com a GLCM.

```
from sklearn.decomposition import PCA#biblioteca que implementa a
    analise de componentes principais

#Quantidade de componenetes desejadas,  esses valores devem ser menores
     que a quantidade de atributos. Aqui extraimos 6 atributos de cada
    componente no sistema de cores RGB, totalizando 18 atributos
components = [2,4,8,10,12]

#Funcao que calcula a PCA
def pca(X_train, X_test,y_train, n_comp):
    """
    PCA transformation for using a 'training' set and a 'testing' set
    """
    pca = PCA(n_components=n_comp)
    pca.fit(X_train,y_train)
    transform = pca.transform(X_test)
    return transform

results = np.zeros(5)#variavel que armazena os resultados para cada
    componente

#para cada quantidade de componentes o PCA eh executado nos conjuntos
    de treino e teste. Apos isso, os dados sao classificados
for id_comp,comp in enumerate(components):
    X_train_pca = pca(X_train,X_train,y_train,comp)
    X_test_pca = pca(X_train,X_test,y_train,comp)
    c_rf = RandomForestClassifier()
    c_rf.fit(X_train_pca,y_train)
    pred = c_rf.predict(X_test_pca)
    acc = accuracy_score(y_test, pred)
    results[id_comp] = acc
```

Código Fonte 8.4: Cálculo das componentes principais

A Figura 8.11 mostra os resultados obtidos utilizando PCA para redução de atributos. É possível observar que com 10 componentes obtivemos o maior valor de acurácia para o classificador *Random Forest*, isso demonstra a eficácia da utilização de técnicas de seleção de atributos para classificação. Entretanto, em relação ao classificador SVM não houve mudança nos resultados.

8.7. Considerações Finais

Os resultados mostrados na Figura 8.11 concluem que sem uma quantidade de componentes superior a 4 os resultados de acurácia são menos precisos que resultados contendo mais componentes. Além disso, o método proposto aqui possui falhas em determinadas amostras de imagens como na Figura 8.12. Logo, algumas modificações podem ser feitas para melhorar a fase segmentação como, por exemplo, o uso da Transformada Circular de Hough [Silva et al. 2017].

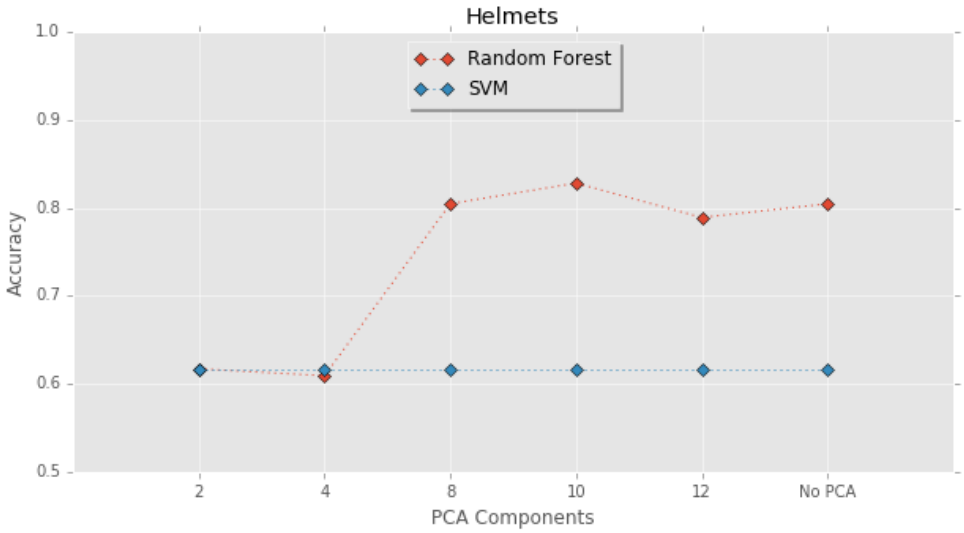

Figura 8.11: Acurácia para a classificação da base de dados de motociclistas utilizando análise de componentes principais como seletor de atributos.

Figura 8.12: Exemplo de uma imagem com falha na segmentação.

Referências

[Baraldi and Parmiggiani 1995] Baraldi, A. and Parmiggiani, F. (1995). An investigation of the textural characteristics associated with gray level cooccurrence matrix statistical parameters. *Geoscience and Remote Sensing, IEEE Transactions on*, 33(2):293–304.

[Breiman 2001] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

[Cortes and Vapnik 1995] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20:273–297.

[Cover 1965] Cover, T. M. (1965). Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, EC-14(3):326 –334.

[Gonzalez 2007] Gonzalez, R C, W. R. E. (2007). *Digital Image Processing*. Prentice Hall, Inc., Upper Saddle River, New Jersey, NJ, USA, 3 edition.

[Haralick 1973] Haralick, R.M., S. K. D. I. (1973). Textural features for image classification. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-3(6):610–621.

[Haykin 2001] Haykin, S. (2001). *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2 edition.

[Otsu 1979] Otsu, N. (1979). A threshold selection method from gray-level histograms. *IEEE transactions on systems, man, and cybernetics*, 9(1):62–66.

[Pedrini 2007] Pedrini, H, S. W. R. (2007). *Análise de Imagens Digitais: Princípios, Algoritmos e Aplicações*. Editora Thomson Learning Edições Ltda, São Paulo, São Paulo, SP, BRA, 1 edition.

[Richards 2006] Richards, J., J. X. (2006). *Remote Sensing Digital Image Analysis*. Springer-Verlag Berlin Heidelberg., 4 edition.

[Sabino et al. 2004] Sabino, D. M. U., da Fontoura Costa, L., Rizzatti, E. G., and Zago, M. A. (2004). A texture approach to leukocyte recognition. *Real-Time Imaging*, 10(4):205 – 216. Imaging in Bioinformatics: Part {III}.

[Silva et al. 2017] Silva, R. R. V. e., Aires, K. R. T., and Veras, R. d. M. S. (2017). Detection of helmets on motorcyclists. *Multimedia Tools and Applications*, pages 1–25.

[Ushizima et al. 2014] Ushizima, D. et al. (2014). Structure recognition from high resolution images of ceramic composites. *IEEE International Conference on Big Data*.

A. Código-fonte

O código abaixo reúne a etapas de segmentação, extração de atributos e classificação para a base de dados de motociclistas com e sem capacete para uma base de dados. Além disso, adicionamos etapas para otimização de parâmetros dos classificados.

```
#Computer Vision: scikit-image + scikit-learn

'''Problema: Classificar Imagens de Transito de Motociclistas com e sem capacete'''

#bibliotecas

import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread_collection,imsave
from sklearn.model_selection import train_test_split
from glob import glob
from scipy.stats import randint as sp_randint
import time
from skimage.color import rgb2grey
from skimage.filters import threshold_otsu
from skimage.measure import label,regionprops
from sklearn.decomposition import PCA
from skimage.feature import greycomatrix,greycoprops
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import make_scorer, accuracy_score
from sklearn.svm import SVC

#Read images from database

path = "/Users/romuere/Dropbox/MTAP/databases/database2_raw_data/"
```

```
comcapacete = glob(path+'comcapacete/*.png')
semcapacete = glob(path+'semcapacete/*.png')

images_cc = imread_collection(comcapacete)
images_sc = imread_collection(semcapacete)

def segmentation(im):
    '''Recebe uma imagem calcula limiar de otsu e fazer o
       recorte obdecendo a regiao resultante desse limiar'''

    grey_image = rgb2grey(im)
    otsu = threshold_otsu(grey_image)
    im_ = grey_image < otsu
    n_branco = np.sum(im_ == 1)
    n_preto = np.sum(im_ == 0)
    if n_branco > n_preto:
        im_ = 1-im_

    label_img = label(im_, connectivity = grey_image.ndim)#detecta
    regioes nao conectadas
    props = regionprops(label_img)#calcula propriedade importantes de
    cada regiao encontrada (ex. area)

    #Convert todas as regioes que possuem um valor de area menor que a
    maior area em background da imagem
    area = np.asarray([props[i].area for i in range(len(props))])#area
    de cada regiao encontrada
    max_index = np.argmax(area)#index da maior regiao
    for i in range(len(props)):
        if(props[i].area < props[max_index].area):
            label_img[np.where(label_img == i+1)] = 0#regiao menor que
    a maior eh marcada como background

    #------------------recorte da regiao de interesse-------------------#
    # Obtendo os limites verticais das imagens segmentadas
    ymin = np.min(np.where(label_img != 0)[1])
    ymax = np.max(np.where(label_img != 0)[1])
    imagem_cortada = imagem[:,ymin:ymax,:]
    return imagem_cortada


start = time.time()
for id_im,imagem in enumerate(images_cc):
    im_name = images_cc.files[id_im].split('/')[-1]
    imagem_segmentada = segmentation(imagem)
    imsave(path+'segmentacao/comcapacete/'+im_name,imagem_segmentada)

    #print(im_name)
for id_im,imagem in enumerate(images_sc):
    im_name = images_sc.files[id_im].split('/')[-1]
    imagem_segmentada = segmentation(imagem)
    imsave(path+'segmentacao/semcapacete/'+im_name,imagem_segmentada)
    #print(im_name)

end = time.time()
```

```
77 print('Tempo para a segmentacao das imagens:', end - start)

79 labels = np.concatenate((np.zeros(len(comcapacete)),np.ones(len(
       semcapacete))))


81
   #Extracting Features using GLCM
83 path_segmentada = "/Users/romuere/Dropbox/MTAP/databases/
       database2_raw_data/segmentacao/"
   comcapacete = glob(path_segmentada+'comcapacete/*.png')
85 semcapacete = glob(path_segmentada+'semcapacete/*.png')
   images = imread_collection(comcapacete+semcapacete)
87
   d = 15
89
   features = np.zeros((len(labels),18)) #6 features x 3 color channels
91 start = time.time()

93 for id_im,imagem in enumerate(images):
       for id_ch in range(3):
95         matrix0 = greycomatrix(imagem[:,:,id_ch], [d], [0],normed=True)
           matrix1 = greycomatrix(imagem[:,:,id_ch], [d], [np.pi/4],normed
       =True)
97         matrix2 = greycomatrix(imagem[:,:,id_ch], [d], [np.pi/2],normed
       =True)
           matrix3 = greycomatrix(imagem[:,:,id_ch], [d], [3*np.pi/4],
       normed=True)
99         matrix = (matrix0+matrix1+matrix2+matrix3)/4
           props = np.zeros((6))
101        props[0] = greycoprops(matrix,'contrast')
           props[1] = greycoprops(matrix,'dissimilarity')
103        props[2] = greycoprops(matrix,'homogeneity')
           props[3] = greycoprops(matrix,'energy')
105        props[4] = greycoprops(matrix,'correlation')
           props[5] = greycoprops(matrix,'ASM')
107        features[id_im,id_ch*6:(id_ch+1)*6] = props

109 end = time.time()
   print('Tempo para extrair atributos usando GLCM: ',end - start)
111


113 #Split Data

115 train = 0.5
   test = 1-train
117 X_train, X_test, y_train, y_test = train_test_split(features, labels,
       test_size=test)

119 #Random Forest Parameter Estimation
   def rf_parameter_estimation(xEst, yEst):
121
       clf = RandomForestClassifier(n_estimators=20)
123    # specify parameters and distributions to sample from
       hyperparameters = {"n_estimators": range(10,1000,50),
125                       "max_depth": range(1,100),
```

```
                  "max_features": sp_randint(1, xEst.shape[1]),
                  "min_samples_split": sp_randint(1, xEst.shape[1]),
                  "min_samples_leaf": sp_randint(1, xEst.shape[1]),
                  "bootstrap": [True, False],
                  "criterion": ["gini", "entropy"]}

    # run randomized search
    n_iter_search = 20
    random_search = RandomizedSearchCV(clf, param_distributions=
    hyperparameters, n_iter=n_iter_search, scoring=make_scorer(
    accuracy_score))
    random_search.fit(xEst, yEst)
    report(random_search.cv_results_)
    return random_search.best_params_

#SVM Parameter Estimation
def svm_parameter_estimation(xEst, yEst):

    hyperparameters = {'gamma': [1e-1, 1e-2,1e-3, 1e-4],'C': [1, 10,
    100, 1000]}

    clf = SVC(kernel='rbf')
    n_iter_search = 8
    random_search = RandomizedSearchCV(clf, param_distributions=
    hyperparameters,n_iter=n_iter_search,scoring=make_scorer(
    accuracy_score))
    random_search.fit(xEst, yEst)
    report(random_search.cv_results_)
    return random_search.best_params_

def report(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] == i)
        for candidate in candidates:
            print("Model with rank: {0}".format(i))
            print("Mean validation score: {0:.3f} (std: {1:.3f})".
    format(
                  results['mean_test_score'][candidate],
                  results['std_test_score'][candidate]))
            print("Parameters: {0}".format(results['params'][candidate
    ]))


#Classification unsing all features

start = time.time()
parameters = rf_parameter_estimation(X_train, y_train)
c_rf = RandomForestClassifier(**parameters)
c_rf.fit(X_train,y_train)
pred = c_rf.predict(X_test)
acc_rf = accuracy_score(y_test, pred)

end = time.time()
print('Tempo para classificacao usando Random Forest: ',end - start)
print('Random Forest Accuracy: ',acc_rf)
```

```
start = time.time()
parameters = svm_parameter_estimation(X_train, y_train)
c_svm = SVC(**parameters)
c_svm.fit(X_train,y_train)
pred = c_svm.predict(X_test)
acc_svm = accuracy_score(y_test, pred)

end = time.time()
print('Tempo para classificacao usando Random Forest: ', end - start)
print('Support Vector Machine Accuracy: ',acc_svm)


#Classification using PCA

def pca(X_train, X_test,y_train, n_comp):
    """
    PCA transformation for using a 'training' set and a 'testing' set
    """
    pca = PCA(n_components=n_comp)
    pca.fit(X_train,y_train)
    transform = pca.transform(X_test)
    return transform

components = [2,4,8,10,12]
results_rf = np.zeros(5)
results_svm = np.zeros(5)

start = time.time()
for id_comp,comp in enumerate(components):

    print('---------------','n comp. = ',comp,'---------------')

    X_train_pca = pca(X_train,X_train,y_train,comp)
    X_test_pca = pca(X_train,X_test,y_train,comp)
    #RF
    parameters = rf_parameter_estimation(X_train_pca, y_train)
    c_rf = RandomForestClassifier(**parameters)
    c_rf.fit(X_train_pca,y_train)
    pred = c_rf.predict(X_test_pca)
    acc = accuracy_score(y_test, pred)
    results_rf[id_comp] = acc

    print('------------------------------------------------------------')

    parameters = svm_parameter_estimation(X_train_pca, y_train)
    c_svm = SVC(**parameters)
    c_svm.fit(X_train_pca,y_train)
    pred = c_svm.predict(X_test_pca)
    acc = accuracy_score(y_test, pred)
    results_svm[id_comp] = acc

end = time.time()
print('Tempo para classificacao usando PCA:', end - start)
```

```
'''Plot do grafico com o resultado para os classificadores SVM e Random
    Forest'''

plt.style.use('ggplot')
fig = plt.figure(figsize = (10,5),dpi=400)
ax = plt.subplot(111)
ax.plot(range(1,7),np.concatenate((results_rf,[acc_rf]),axis=0),marker
    = 'D',linestyle = ':',label = 'Random Forest')
ax.plot(range(1,7),np.concatenate((results_svm,[acc_svm]),axis=0),
    marker = 'D',linestyle = ':',label = 'SVM')
ax.set_xlim([0,7])
ax.set_xlabel('PCA Components')
ax.set_ylabel('Accuracy')
ax.set_xticks(range(1,7))
ax.set_xticklabels(['2','4','8','10','12','No PCA'])
ax.set_ylim([0.5,1])
ax.set_title('Helmets')
legend = ax.legend(loc='upper center', shadow=True)
```

Código Fonte 8.5: Classificação de uma base de imagens utilizando o descritor GLCM e o classificador SVM.