Relatório - Atividade Prática 2
PDI - Limiarização e Kmeans

João Belfort[1], Miguel Ribeiro[1]
Instituto de Ciências Exatas e Tecnológicas
Universidade Federal de Viçosa- Campus Florestal (UFV-Florestal)
Rodovia LMG 818- km 6- Florestal- MG - Brasil
CEP: 35690-000

{joao.andrade1, miguel.a.silva}@ufv.br

Resumo. *Este artigo aborda a segunda atividade prática de relatório, centrada na limiarização e no algoritmo k-means. Destaca-se o emprego de clusters no k-means para colorir halteres conforme cores designadas, utilizando o método de Elbow. Além disso, foram exploradas diferentes técnicas de segmentação, incluindo Niblack, Wellner, adaptativa e gaussiana, com avaliação individual. Por fim, foram empregados métodos de limiarização e clusterização para a extração de placas de veículos. Os resultados apresentados oferecem compreensões para o entendimento da disciplina de Processamento Digital de Imagens (PDI), ressaltando a relevância do uso de bibliotecas disponíveis na linguagem de programação utilizada.*

1. Introdução

Este trabalho aborda uma série de atividades práticas no contexto do processamento digital de imagens, explorando técnicas e algoritmos fundamentais para análise e manipulação de imagens. Na primeira atividade, concentramo-nos no algoritmo K-means, uma ferramenta de aprendizado de máquina que se mostra promissora também em processamento de imagens. O K-means, baseado em clustering, é capaz de segmentar imagens em regiões distintas com base nas características de intensidade de pixel, o que o torna essencial na identificação e separação de objetos de interesse.

Na segunda atividade, exploramos o uso da segmentação, em particular na remoção de fundo em um vídeo. A segmentação é um processo fundamental que divide uma imagem em partes significativas, facilitando a análise e extração de informações específicas. Neste caso específico, aplicamos técnicas de segmentação para remover o fundo de um vídeo, destacando os objetos de interesse e simplificando a análise visual.

Por fim, na última atividade, nos dedicamos à limiarização, com o objetivo de extrair a placa de veículos de imagens. Utilizando o arquivo de placas, empregamos técnicas de limiarização e clusterização para isolar a placa do restante da imagem. Em seguida,

comparamos os resultados obtidos com a leitura da placa sem a aplicação dessas técnicas. As bibliotecas fundamentais para estas atividades foram Numpy[1], OpenCV[2], MatPlotLib[3] e Scikit-learn[4].

Ao longo deste trabalho, buscamos não apenas aplicar essas técnicas, mas também compreender seu funcionamento e avaliar sua eficácia em cenários práticos

Todas as atividades realizadas durante este trabalho são derivadas das práticas anteriores estabelecidas pelo coordenador da disciplina de código CCF 392, Processamento Digital de Imagens, Dr. Barros, A. C. F. professor de Graduação da Universidade Federal de Viçosa.

2. K-means

O algoritmo K-means é um método de aprendizado não supervisionado que se baseia em técnicas de clustering, onde clustering pode ser definido como a tarefa de identificar subgrupos nos dados de forma que os pontos de dados dentro do mesmo subgrupo (cluster) sejam muito semelhantes entre si, enquanto os pontos de dados em clusters diferentes sejam muito diferentes. Em outras palavras, o K-means é um algoritmo iterativo que tenta particionar o conjunto de dados em K subgrupos (clusters) distintos e não sobrepostos predefinidos, onde cada ponto de dados pertence a apenas um grupo (Figura 01). **[1]**

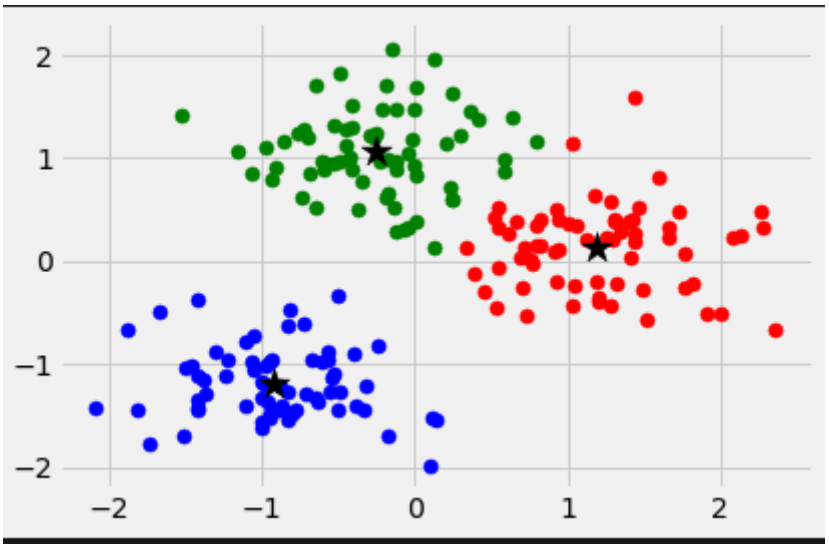

Figura 01 - K-Means clustering. Fonte: [2]

Na área de processamento digital de imagens, o K-means pode ser útil para a segmentação de imagens, que é o processo de dividir uma imagem em vários segmentos. O

[1] NumPy -
[2] opencv-python · PyPI
[3] Matplotlib
[4] Scikit-learn

objetivo da segmentação de imagens é transformar a representação de uma imagem em algo mais significativo e mais fácil de analisar. Por meio do K-means, é possível agrupar pixels de uma imagem com base em sua similaridade de cor, intensidade ou outras características relevantes. **[3]**

Apesar de sua utilidade na segmentação de imagens, o K-means pode apresentar algumas limitações, especialmente quando aplicado a conjuntos de dados grandes ou complexos **[3]**. Ele tende a funcionar melhor em conjuntos de dados pequenos, nos quais os clusters são bem definidos e separados. Em grandes conjuntos de dados ou em conjuntos nos quais os clusters têm formas complexas ou sobreposições, o desempenho do K-means pode diminuir e outras técnicas de clustering mais avançadas podem ser mais apropriadas (Figura 01).

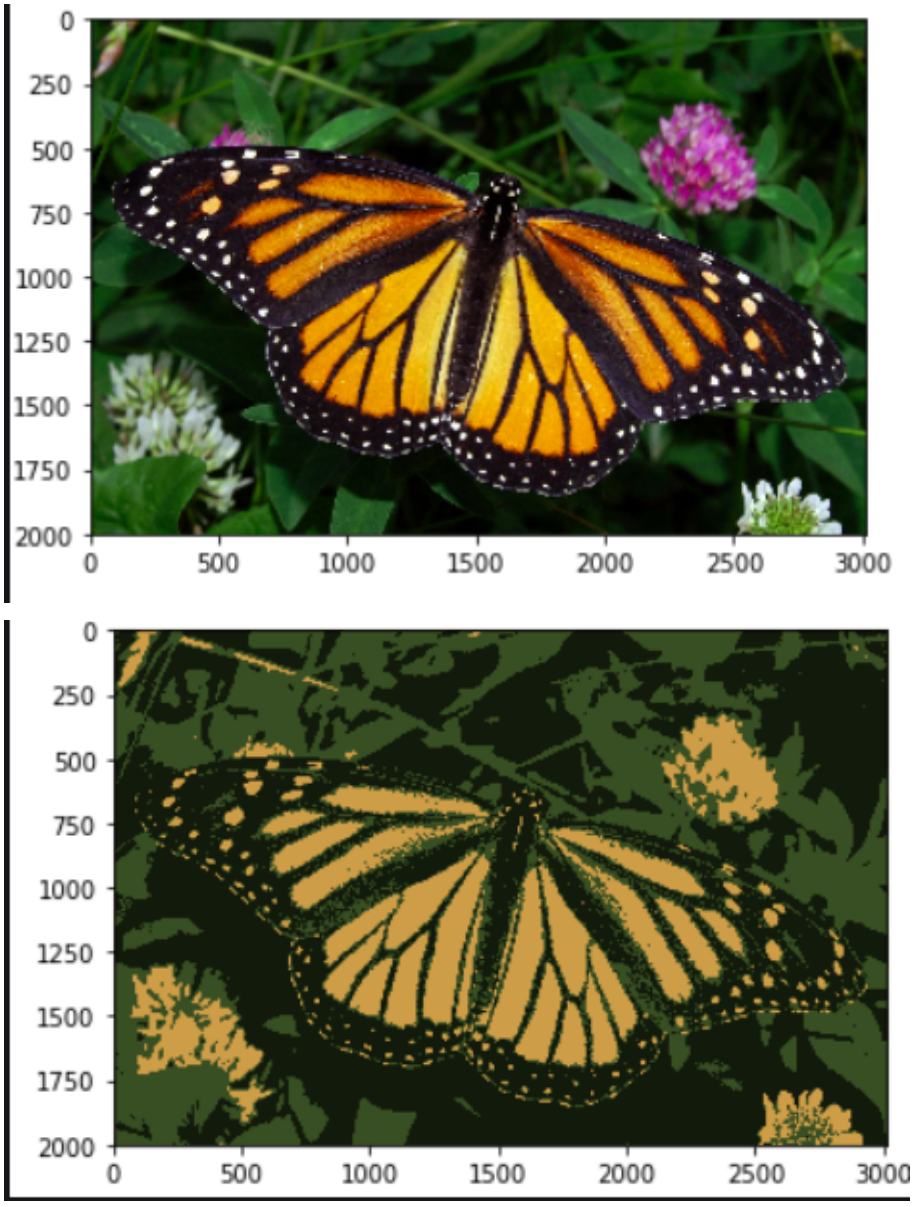

Figura 02 - Exemplo de aplicação do algoritmo K-means. Fonte: [3]

Partindo para técnicas de otimização do algoritmo K-means, o método *Elbow* é amplamente reconhecido como uma ferramenta valiosa para determinar o número ideal de clusters, representados por K, a serem utilizados no agrupamento de dados. Este método recebe o nome "Elbow" devido à forma característica do gráfico resultante, que se assemelha à curva de um cotovelo quando representamos a variância explicada pelos clusters em relação ao número de clusters.

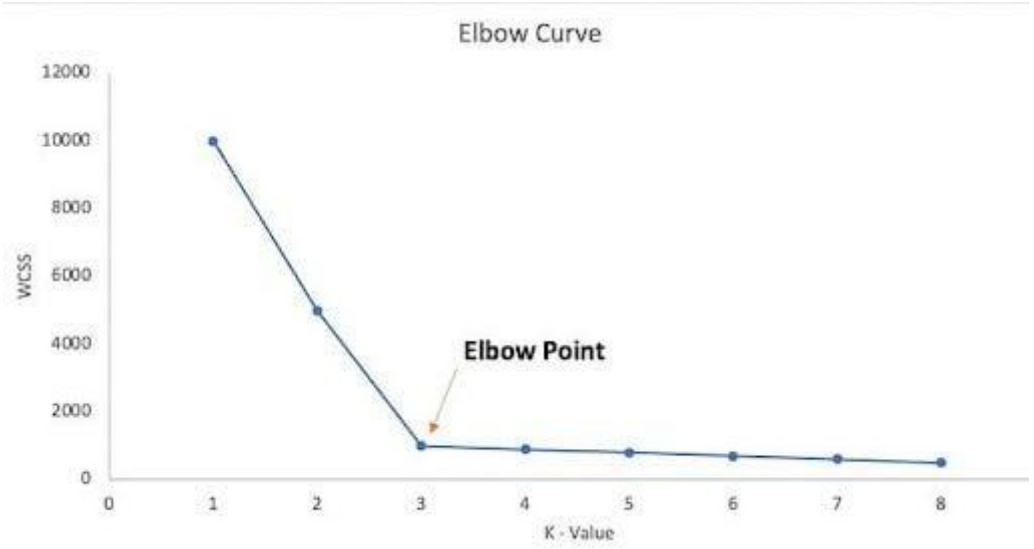

Figura 03 - Método do cotovelo. Fonte: [4]

Na atividade número 1, aplicamos o algoritmo K-means juntamente com técnicas como o método *Elbow* para a foto dos halteres (Figura 04). Esta foto é usada para diversas atividades ao longo deste curso, destacando seus resultados.

Figura 04 - halteres.jpg

2.1. Atividade 1 - *kmeans.ipynb*

O script "halteres.ipynb" tem como objetivo principal utilizar o algoritmo K-means

para agrupar as cores presentes na imagem dos halteres. Inicialmente, será realizado um agrupamento utilizando um número pré-definido de clusters, especificamente K=7. Em seguida, será empregado o método de *Elbow* para determinar o número ideal de clusters. O ponto de inflexão identificado pelo método de *Elbow* será utilizado para escolher o número ótimo de clusters (k) para segmentar a imagem. Posteriormente, serão aplicados novamente o algoritmo K-means para k-1, k, k+1 e k+2, a fim de explorar a variação na segmentação da imagem em torno do valor ideal de clusters. Por fim, será impressa a quantidade de cores que a imagem realmente utilizava antes do processo de segmentação, identificando o número de pares (b,g,r) distintos presentes na imagem original.

2.1.1. Carregamento da Imagem:

A imagem é carregada utilizando a função cv2.imread() do OpenCV. Isso resulta em uma matriz NumPy que representa a imagem, onde cada elemento da matriz corresponde a um pixel na imagem.

2.1.2. Transformação da Imagem em Array 2D de Pixels:

A imagem é então transformada em um array 2D de pixels usando o método *reshape*() do NumPy. Isso é feito para preparar a imagem para o algoritmo K-means, que requer um formato específico de entrada.

2.1.3. Definição dos Critérios de Parada e K=7:

Definimos os critérios de parada para o algoritmo K-means, especificando o número máximo de iterações e a precisão desejada. Além disso, fixamos o número de clusters (K) em 7 para a primeira execução do K-means.

- **cv2.TERM_CRITERIA_EPS**: Este critério indica que a iteração do algoritmo será interrompida quando a precisão desejada for atingida. Neste caso, a precisão é definida como 1.0, o que significa que o algoritmo K-means será interrompido quando a precisão da solução estiver dentro de uma margem de 1.0. Em outras palavras, quando a variação entre iterações sucessivas for menor que 1.0, o algoritmo para.

- **cv2.TERM_CRITERIA_MAX_ITER:** Este critério especifica o número máximo de iterações permitidas para o algoritmo K-means. Aqui, o valor escolhido foi 10, o que significa que o algoritmo será interrompido após 10 iterações, independentemente de ter alcançado a precisão desejada.

2.1.3. Aplicação do Algoritmo K-means:

O algoritmo K-means é então aplicado utilizando a função cv2.kmeans(). Ele agrupa os pixels da imagem em 7 clusters distintos com base na similaridade das cores.

```
# Aplica o algoritmo kmeans
ret, label, center = cv2.kmeans(Z, K, None, criteria, 10,
cv2.KMEANS_RANDOM_CENTERS)

center = np.uint8(center)  # Converte os centros de float32 para uint8
res = center[label.flatten()]  # Acessa o valor do centro para cada píxel
res2 = res.reshape(img.shape)  # Remodela a imagem para o formato original

final = np.concatenate((img, res2), axis=1)  # Concatena as imagens originais
e segmentadas
```

Figura 05 - Aplicação do K-means

2.1.4. Visualização dos Resultados:

Os resultados da segmentação são visualizados concatenando a imagem original com a imagem segmentada, permitindo uma comparação direta entre as duas (Figura 06).

Figura 06 - Imagens concatenadas

2.1.5. Método Elbow para Determinação do Número Ideal de Clusters:

Implementamos o método Elbow para determinar o número ideal de clusters. Calculamos a distorção para diferentes valores de K e plotamos um gráfico para visualizar a relação entre a distorção e o número de clusters.

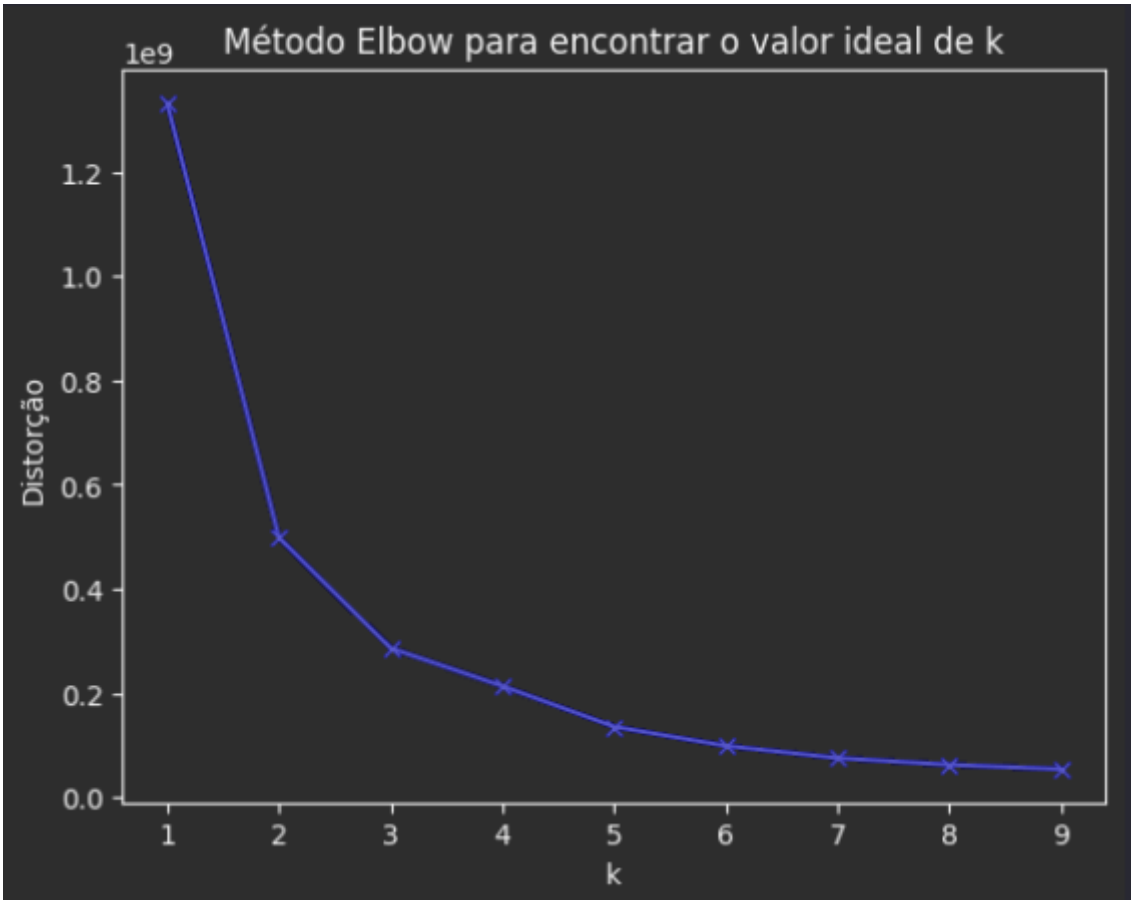

Figura 07 - Método elbow

Verifica-se, através do gráfico, que o número ideal de clusters é entre 3, pois:

Antes do ponto de cotovelo (região de underfitting): Nesta região, o número de clusters é muito baixo para capturar adequadamente a estrutura dos dados. Isso geralmente resulta em uma alta distorção.

Depois do ponto de cotovelo (região de overfitting): Aqui, o número de clusters é muito alto em relação à estrutura real dos dados. Adicionar mais clusters além desse ponto não oferece muitos benefícios na redução da distorção

2.1.6. Refinamento da Segmentação com o Número Ideal de Clusters:

Com base no ponto de inflexão identificado pelo método Elbow, refinamos a segmentação da imagem aplicando novamente o algoritmo K-means para k-1, k, k+1 e k+2. Isso nos permite explorar variações na segmentação em torno do valor ideal de clusters.

Figura 08 - k-1

Figura 09 - k

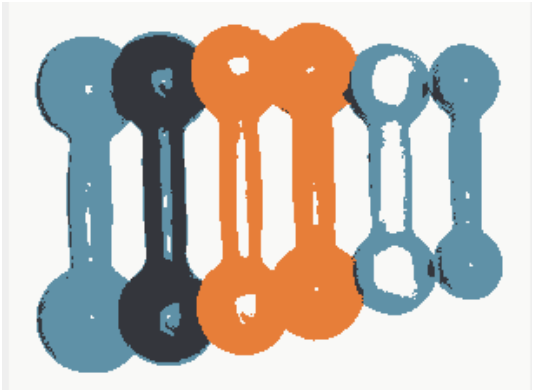

Figura 10 - k+1

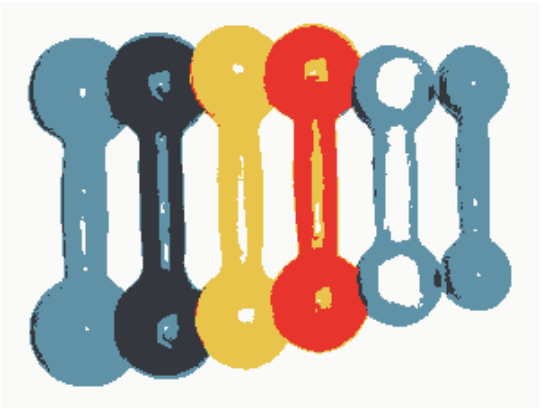

Figura 11 - k+2

2.1.7. Avaliação da Quantidade de Cores Utilizadas na Imagem Original:

Por fim, calculamos a quantidade de cores realmente utilizadas na imagem original, identificando o número de pares (b,g,r) distintos presentes na imagem antes do processo de segmentação. Essa análise fornece insights sobre a complexidade da imagem e a eficácia do algoritmo K-means na identificação das cores dos halteres.

```
tuplas_unicas = set(map(tuple, Z))
num_cor_pixel_utilizada = len(tuplas_unicas)
print("Esta imagem utilizou %d combinações de cores RGB" %
num_cor_pixel_utilizada)
Esta imagem utilizou 25389 combinações de cores RGB
```

Figura 12 - Combinações de cores RGB

3. Métodos para remoção de fundo usando segmentação através de limiriazição

A remoção de fundo (Figura 13) é uma etapa crucial em muitas aplicações de visão computacional e processamento de imagens, onde o foco principal é isolar objetos de interesse do restante da cena. Existem várias abordagens para realizar essa tarefa, cada uma com suas vantagens e limitações. Neste capítulo, exploramos diferentes métodos para a remoção de fundo, desde técnicas básicas até abordagens mais avançadas.

Figura 13 - Remoção de fundo. Fonte: [5]

Um dos métodos mais simples e amplamente utilizados para remoção de fundo é a extração por diferença de frames (Figura 13). Essa técnica baseia-se na comparação pixel a pixel entre frames consecutivos do vídeo, onde a diferença entre os valores de intensidade dos pixels é calculada para determinar áreas em movimento. A ideia subjacente é que os objetos em movimento serão destacados pela diferença significativa em seus valores de pixel em relação ao fundo estático.

Figura 14 - Extração por diferença de frames

A limiarização é um processo fundamental de processamento de imagens usado para segmentar uma imagem em regiões ou objetos de interesse, com base nos níveis de intensidade dos pixels. O vídeo **cars2.mp4** será usado como exemplo para aplicação de diversas técnicas a fim de extrair o fundo da imagem deixando apenas os carros em uma avenida.

3.1. Método Wellner

A função `**apply_welnner_threshold()**` (Figura 15) implementa o algoritmo de limiarização de **Wellner**, que é uma técnica de limiarização local baseada na média local dos pixels. Primeiramente, é calculada a média local da imagem usando uma janela deslizante de tamanho especificado. Em seguida, o limiar é calculado com base na **média local**, onde a intensidade do limiar é ajustada de acordo com a diferença entre cada pixel e sua média local. O parâmetro **k** controla o peso da diferença na intensidade do limiar final. Por fim, a imagem é binarizada usando o limiar calculado, onde os pixels com intensidade maior ou igual ao limiar são definidos como brancos (255) e os pixels abaixo do limiar são definidos como pretos (0).

No loop principal do código, cada frame do vídeo é processado da seguinte maneira: primeiro, a equalização de histograma é aplicada ao canal V (valor) da imagem para melhorar o contraste. Em seguida, o frame é convertido para tons de cinza e a diferença entre o frame atual e o frame anterior é calculada. Posteriormente, a função apply_welnner_threshold é chamada para aplicar a limiarização de Wellner à diferença calculada. O resultado binarizado é então exibido na tela (Figura 16).

```python
def apply_welnner_threshold(window_size, k):

    # Calcula a média local usando uma janela deslizante
    mean = cv2.boxFilter(gray, cv2.CV_32F, (window_size, window_size))

    # Aplica a fórmula de Wellner para calcular o limiar
    threshold = mean * (1 + k * (gray - mean) / mean)

    # Aplica o limiar na imagem
    thresholded_img = np.zeros_like(gray)
    thresholded_img[gray >= threshold] = 255

    return thresholded_img
```

Figura 15 - Função apply_welnner_threshold

Figura 16 - Resultado da implementação do método de Wellner

3.2. Método de Niblack

A função ´**apply_niblack_threshold()**´ (Figura 17) implementa a limiarização de Niblack, um método que calcula o limiar localmente para cada pixel da imagem, levando em consideração a média e o desvio padrão dos pixels vizinhos em uma janela definida. Inicialmente, a função calcula a média e a média do quadrado dos valores dos pixels dentro da janela especificada. Em seguida, é calculado o desvio padrão local a partir dessas médias.

Utilizando o fator de correção **k**, o limiar de **Niblack** é determinado como a média local mais o produto entre **k** e o desvio padrão local. Posteriormente, a função aplica o limiar de Niblack na imagem, onde os pixels com valores superiores ao limiar são considerados pertencentes ao primeiro plano e são definidos como brancos, enquanto os pixels abaixo do limiar são considerados pertencentes ao plano de fundo e são definidos como pretos. Isso resulta em uma imagem binária que destaca os objetos de interesse com base nas variações locais de intensidade. (Figura 18)

```
def apply_niblack_threshold(window_size, k):
    # Calcula a média e o desvio padrão locais usando uma janela deslizante
    mean = cv2.blur(gray, (window_size, window_size))
    mean_square = cv2.blur(gray * gray, (window_size, window_size))
    variance = mean_square - mean * mean

    # Calcula o limiar usando a fórmula de Niblack
    threshold = mean + k * np.sqrt(variance)

    # Aplica o limiar na imagem
    thresholded_img = np.zeros_like(gray)
    thresholded_img[gray >= threshold] = 255

    return thresholded_img
```

Figura 17 - Função apply_niblack_threshold

Figura 18 - Resultado da implementação do método de Niblack

3.3. Método Adaptativo Médio e Gaussiano

Neste tópico, são utilizadas duas técnicas de limiarização adaptativa, conhecidas como limiarização adaptativa com **média** e limiarização adaptativa com **Gaussiana**. Ambas são aplicadas para segmentar as diferenças entre os frames consecutivos de um vídeo.

A limiarização adaptativa com média, implementada na função ´**adaptive_mean_threshold()**´ (Figura 19), calcula um limiar local para cada pixel da imagem, levando em conta a média dos pixels vizinhos em uma janela definida. Esse limiar é calculado dinamicamente para cada região da imagem, permitindo adaptabilidade a diferentes condições de iluminação e contrastes locais (Figura 20).

Por outro lado, a limiarização adaptativa com Gaussiana, implementada na função ´**adaptive_gaussian_threshold()**´ (Figura 19), calcula o limiar local usando uma abordagem semelhante à anterior, porém levando em consideração uma ponderação Gaussiana para os pixels vizinhos. Isso significa que os pixels mais próximos do pixel de interesse têm uma contribuição maior para o cálculo do limiar, o que pode ser útil em casos onde a distribuição dos valores dos pixels não é uniforme (Figura 21).

```
def adaptive_mean_threshold(image, max_value=255, block_size=11,
constant=8):
   return cv2.adaptiveThreshold(image, max_value, cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY, block_size, constant)


def adaptive_gaussian_threshold(image, max_value=255, block_size=11,
constant=8):
                    return cv2.adaptiveThreshold(image, max_value,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, block_size,
                                 constant)
```

Figura 19 - Função adaptative_mean/gaussian_threshold

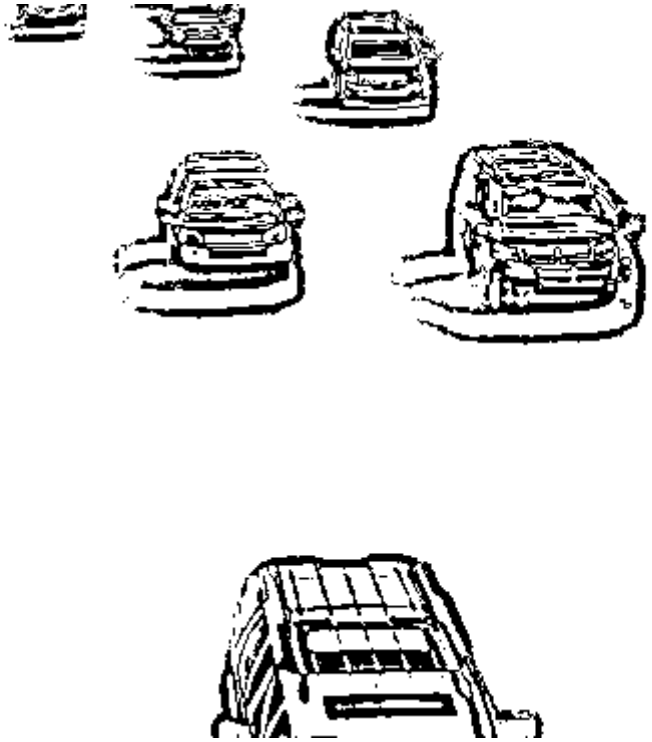

Figura 20 - Resultado da implementação do método adaptative_mean

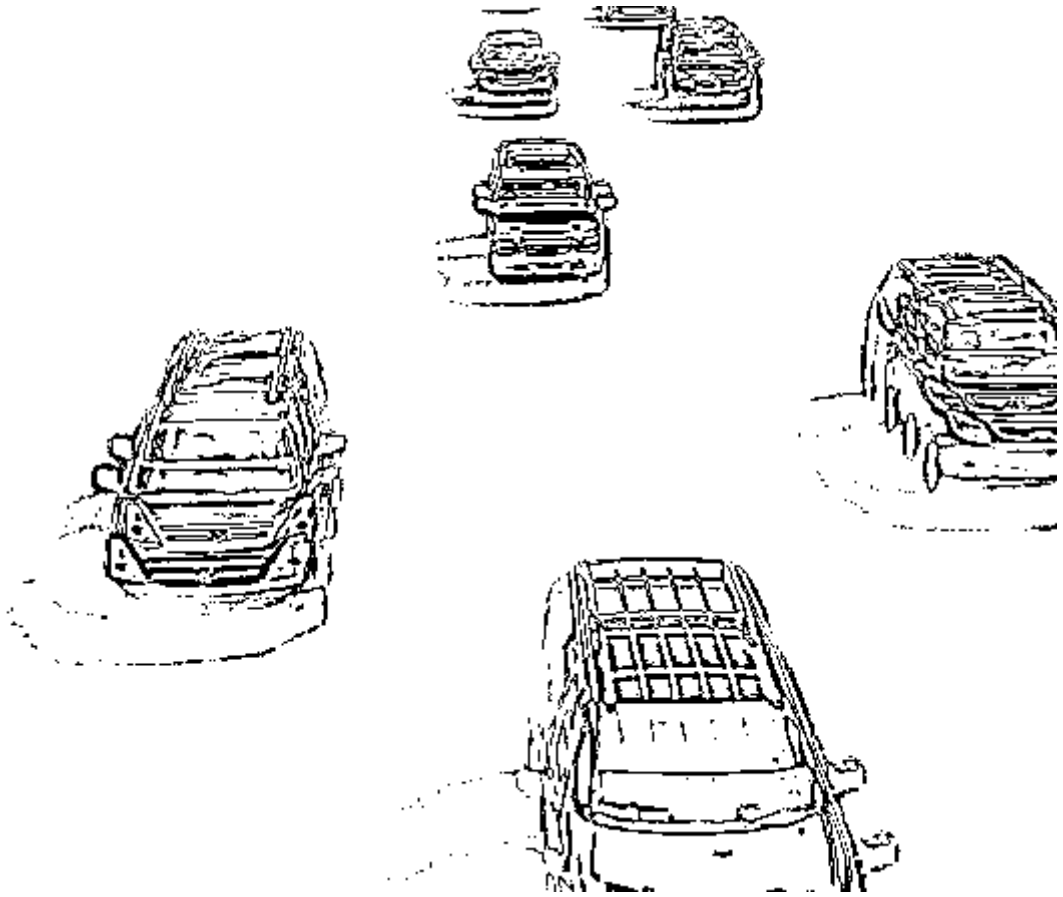

Figura 21 - Resultado da implementação do método adaptative_gaussian

3.4. Conclusões

Em conclusão, entre as técnicas de limiarização adaptativa utilizadas - **Adaptive Mean e Adaptive Gaussian** - ambas se destacam na segmentação das regiões de interesse nos frames do vídeo. Os carros são mais claramente destacados com ambas as técnicas, oferecendo resultados visuais satisfatórios. No entanto, a eficácia dessas técnicas depende significativamente dos parâmetros utilizados, como o tamanho do bloco e a constante de limiarização.

Por fim, é importante notar que outras técnicas como **Niblack e Wellner** também têm potencial para serem úteis, especialmente quando aplicadas com uma análise cuidadosa dos parâmetros. Com uma escolha adequada dos parâmetros e uma compreensão aprofundada do contexto da aplicação, essas técnicas podem oferecer definições ainda melhores e resultados mais precisos na segmentação de regiões de interesse em vídeos, tornando-se valiosas ferramentas em diversas aplicações de processamento de imagem e visão computacional.

4. Reconhecimento de placas de veículos com limiarização adaptativa e OCR

A detecção automática de placas de veículos desempenha um papel fundamental em várias aplicações de visão computacional e sistemas de segurança. A capacidade de identificar e registrar com precisão as placas de veículos em imagens é de suma importância em áreas como controle de tráfego, estacionamento inteligente, segurança pública e vigilância veicular.

Este capítulo descreve uma abordagem convencional para o reconhecimento de placas de veículos utilizando limiarização adaptativa e OCR. Embora não tenha sido capaz de identificar diretamente a placa em todos os casos, observou-se uma melhoria significativa no reconhecimento dos caracteres individuais.

Figura 22 - Placa de carro BRA0S17

Para fins de estudo da melhoria na aplicação da limiarização e clusterização, foi pedido que utilizássemos o reconhecimento **OCR** utilizando o **PyTesseract** anteriormente da aplicação das técnicas de processamento, o resultado dos testes está abaixo:

```
Placa do carro: BRASIL
```

Figura 23 - Resultado BRASIL do uso de OCR PyTesseract

```python
def ler_placa_sem_processamento(img, arquivo):
    result = pytesseract.image_to_string(img, config='--psm 6')
    cv2.imwrite(f"{arquivo}_ocr.png", img)
    return result.strip()
```

Figura 24 - OCR

4.1. OCR e Limiarização de Otsu

Neste capítulo, exploraremos os métodos empregados no processo de reconhecimento

de placas de veículos, com foco especial na limiarização adaptativa e no OCR (Reconhecimento Óptico de Caracteres). Estes métodos desempenham um papel importante na extração de informações precisas das imagens das placas.

4.1.1. OCR (Reconhecimento Óptico de Caracteres)

OCR, ou Reconhecimento Óptico de Caracteres, é uma tecnologia que permite a extração de texto de imagens ou documentos digitalizados. O OCR identifica padrões na imagem que representa letras, números e outros caracteres e os converte em texto legível por máquina. Esta tecnologia é essencial em aplicações onde é necessário extrair informações de documentos digitalizados, como reconhecimento de placas de veículos em imagens.

4.1.2. Limiarização de Otsu

A limiarização de Otsu é uma técnica de processamento de imagem utilizada para segmentar uma imagem em duas classes: pixels de primeiro plano e pixels de fundo. O objetivo é encontrar automaticamente um valor de limiar que maximize a variabilidade entre as duas classes. Isso é útil em casos onde a imagem tem um histograma bimodal (duas distribuições distintas de intensidades de pixels), como é comum em imagens de placas de veículos, onde a placa geralmente possui intensidades diferentes do fundo ao redor. A limiarização de Otsu ajuda a separar a placa do carro do resto da imagem, facilitando o reconhecimento dos caracteres.

4.2. Segmentação em placas de veículos

A segmentação s é o processo de identificar e isolar a região da imagem que contém a placa do veículo. Isso é feito usando técnicas de processamento de imagem, como detecção de bordas, limiarização e análise de contornos. O objetivo é extrair a placa do carro da cena ao redor, para que possa ser processada separadamente para reconhecimento de caracteres. A segmentação de placas é uma etapa importante no reconhecimento automático de placas de veículos, pois permite que o OCR se concentre apenas na região de interesse, melhorando a precisão do reconhecimento.

4.3. Leitura de placas

A atividade proposta para esse relatório consiste em várias etapas de processamento de imagem e análise de texto. Primeiramente, lendo a placa do carro com OCR. Depois, utilizou-se limiarização e clusterização para tentar obter somente a placa do carro. Um dos resultados obtidos foi escolhido para aplicação de um um OCR (**PyTesseract**) para ler a placa. Compare com a placa lida sem limiarização/clusterização.

Inicialmente, a imagem é pré-processada para remover ruídos e melhorar a qualidade da imagem. Em seguida, é aplicada a técnica de limiarização de Otsu para segmentar a região da placa de carro na imagem. A limiarização de Otsu é uma abordagem eficaz para a binarização automática da imagem, ajudando a separar a placa do carro do restante da cena.

```python
def aplicar_limiarizacao_otsu(img, arquivo):
    # Converte para escala de cinza
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Aplica um filtro de mediana para remover ruídos
    gray = cv2.medianBlur(gray, 5)

    # Aplica a limiarização de Otsu
    _, th_otsu = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    cv2.imwrite(f"{arquivo}_limiarizacao.png", th_otsu)
    return th_otsu
```

Figura 25 - Limiarização Ostu

Figura 26 - Resultado após a limiarização da imagem

Após a segmentação da placa, utilizamos técnicas de OCR para reconhecer os caracteres presentes na placa.

```python
def segmentar_placa(img, arquivo):
    # Encontrar contornos
            contours,     _     =     cv2.findContours(img,     cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    # Encontrar o maior contorno (placa do carro)
    max_contour = max(contours, key=cv2.contourArea)

    # Criar uma máscara para a placa do carro
    mask = np.zeros_like(img)
    cv2.drawContours(mask, [max_contour], -1, 255, thickness=cv2.FILLED)

    # Aplicar a máscara na imagem original
    segmented_plate = cv2.bitwise_and(img, img, mask=mask)
    cv2.imwrite(f"{arquivo}_segmentada.png", segmented_plate)
    return segmented_plate
```

Figura 27 - Segmentação da imagem

Figura 28- Resultado após clusterização da imagem

```
Placa do carro com processamento (antes da validação): BRAUST7
Placa do carro inválida ou não reconhecida.
```

Figura 29 - OCR na imagem Clusterizada indicando BRAUST7

Neste contexto, optamos por utilizar a biblioteca **PyTesseract**, que é uma ferramenta poderosa para realizar OCR em imagens com facilidade. Os resultados da OCR são então analisados para verificar se a placa reconhecida segue o padrão comum de placas de carros, consistindo de três letras seguidas, um número, uma letra e mais dois números. Os resultados apresentados na figura acima foram próximos do real 'BRA0S17', porém com um erro em 2 caracteres. É proposto a criação de algum tipo de algoritmo que aproxime letras de números para seguir o padrão de placas, que não fará parte do escopo deste projeto. Abaixo é proposta uma função para verificar a validade de uma placa gerada pelo OCR ou pelo algoritmo proposto.

```
def validar_placa(placa):
    # Expressão regular para verificar o padrão da placa
    pattern = r'^[A-Z]{3}\d[A-Z]\d{2}$'

    # Verifica se a placa corresponde ao padrão
    if re.match(pattern, placa):
        return True
    else:
        return False
```

Figura 30 - Função de verificação de placa.

5. Conclusões

Neste trabalho, exploramos diversas técnicas de processamento digital de imagens, com foco em limiarização e clusterização, especialmente aplicadas à segmentação e reconhecimento de objetos em imagens. Utilizamos o algoritmo **K-means** para segmentar uma imagem de halteres, demonstrando a importância do método Elbow na escolha do número ideal de clusters. Além disso, exploramos diferentes métodos de limiarização, como Niblack, Wellner e adaptativa, destacando suas aplicações e desempenho em cenários variados.

Na remoção de fundo em um vídeo, empregamos técnicas de limiarização adaptativa de Wellner, ressaltando a eficácia dessa abordagem em ambientes com variações significativas de iluminação. Além disso, abordamos a detecção e segmentação de placas de veículos, aplicando limiarização de Otsu e técnicas de OCR para reconhecer os caracteres das placas.

Os resultados obtidos demonstram a relevância e a eficácia das técnicas estudadas no contexto do processamento digital de imagens. No entanto, também identificamos desafios e limitações, como a necessidade de ajustes finos nos parâmetros dos algoritmos para obter resultados ideais e a necessidade de lidar com cenários complexos, como variações de iluminação e presença de ruídos..

6. Referências

[1] DABBURA, I. K-means clustering: Algorithm, applications, evaluation methods, and drawbacks. Disponível em: <https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a>.

[2] MIGUEL, T. K-Means Clustering (Agrupamento k-means). Disponível em: <https://aprenderdatascience.com/k-means-clustering-agrupamento-k-means/>.

[3] Image Segmentation using K Means Clustering. Disponível em: <https://www.geeksforgeeks.org/image-segmentation-using-k-means-clustering/>.

[4] Stop Using Elbow Method in K-means Clustering | Built In. Disponível em: <https://builtin.com/data-science/elbow-method>.

[5] OPERADORA, R. M. 5+ melhores ferramentas para remover plano de fundo de fotos [Guia 2021]. Disponível em: <https://www.minhaoperadora.com.br/2021/06/5-melhores-ferramentas-para-remover-plano-de-fundo-de-fotos-guia-2021.html>. Acesso em: 26 mar. 2024.