UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
CAMPUS DE PONTA PORÃ

PEDRO ANTONIO TRINDADE DOS SANTOS

CONTAGEM DE LARANJAS EM IMAGENS EM DIFERENTES MÉTODOS

PONTA PORÃ-MS
2024

UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
CAMPUS DE PONTA PORÃ

CONTAGEM DE LARANJAS EM IMAGENS EM DIFERENTES MÉTODOS

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Mato Grosso do Sul, como parte dos requisitos para conclusão do curso de sistemas de informação.

Orientador: Prof. Dr. Robson Soares Silva

PONTA PORÃ-MS
2024

SUMÁRIO

Lista de Tabelas

Lista de Figuras

Resumo

Este trabalho de conclusão de curso apresenta uma análise comparativa de métodos para contagem automática de laranjas em imagens, utilizando tanto algoritmos de visão computacional clássicos quanto modelos de inteligência artificial. A abordagem de visão computacional, implementada com as bibliotecas *OpenCV* e *scikit-image*, envolveu pré-processamento das imagens para realçar a cor laranja e remover ruídos, seguido da detecção de componentes conectados representando as laranjas. Em contraponto, um modelo de detecção de objetos baseado em inteligência artificial, o YOLO, foi treinado utilizando um conjunto de dados extensivo, construído e anotado com auxílio da plataforma Roboflow. Este conjunto de dados, composto por centenas de imagens, permitiu o treinamento e a avaliação de diferentes versões do modelo YOLO, dentre elas a versão 5, 8, 9 e 11. A acurácia da contagem serviu como métrica principal para a comparação dos métodos. Os resultados obtidos demonstram a superioridade do modelo YOLO em relação às técnicas de visão computacional, onde a acurácia foi de 52,23% para o YOLOv5 e chegando a 58,08% para o YOLOv9, assim evidenciando a eficácia inteligência artificial em tarefas de contagem de objetos em imagens. A pesquisa identifica, contudo, a necessidade de otimizar parâmetros como o limiar de confiança e o número de épocas de treinamento para maximizar o desempenho do modelo YOLO.

Palavras-chave: Visão computacional, Inteligência artificial, Contagem de objetos, Laranja, YOLO.

Abstract

This undergraduate thesis presents a comparative analysis of methods for the automatic counting of oranges in images, utilizing both classical computer vision algorithms and artificial intelligence models. The computer vision approach, implemented with the OpenCV and scikit-image libraries, involved image preprocessing to enhance the orange color and remove noise, followed by the detection of connected components representing the oranges. In contrast, an object detection model based on artificial intelligence, YOLO, was trained using an extensive dataset built and annotated with the help of the Roboflow platform. This dataset, comprising hundreds of images, enabled the training and evaluation of different YOLO model versions, including versions 5, 8, 9, and 11. Counting accuracy served as the primary metric for comparing the methods. The results demonstrate the superiority of the YOLO model compared to computer vision techniques, with an accuracy of 52.23% for YOLOv5 and reaching 58.08% for YOLOv9, highlighting the effectiveness of artificial intelligence in object counting tasks in images. However, the research identifies the need to optimize parameters such as confidence thresholds and the number of training epochs to maximize YOLO's performance.

Keyworks: Computer vision, Artificial intelligence, Object counting, Orange, YOLO

1. Introdução

A precisão na contagem de produtos agrícolas é um fator crucial para a eficiência e lucratividade em diversas etapas da cadeia produtiva, desde a colheita até a comercialização. A contagem manual, método tradicionalmente empregado, apresenta limitações significativas, como a lentidão, a subjetividade e a alta probabilidade de erros, especialmente quando lidamos com grandes volumes de produção. Essas desvantagens geram impactos econômicos negativos, incluindo perdas financeiras devido a estimativas imprecisas de estoque, dificuldades no planejamento logístico e na precificação adequada dos produtos. A busca por soluções automatizadas e mais eficientes, portanto, se torna imperativa para garantir a competitividade no mercado agrícola.

Este trabalho se propõe a investigar e comparar diferentes métodos para a contagem automática de laranjas em imagens, explorando tanto as técnicas tradicionais de visão computacional quanto as abordagens mais modernas baseadas em inteligência artificial. A escolha das laranjas como objeto de estudo justifica-se por sua importância econômica no contexto da fruticultura, e pela complexidade da tarefa de contagem, considerando as variações na iluminação, sombras e aglomeração das frutas. A comparação entre algoritmos de processamento de imagens e detecção de objetos (usando *OpenCV* e *scikit-image*) e modelos de detecção de objetos baseados em inteligência artificial (utilizando diferentes versões do modelo *YOLO*, v5, v8, v9, v11) permitirá avaliar a eficácia e a robustez de cada abordagem. A acurácia da contagem, avaliada através de um conjunto de imagens de teste, servirá como principal métrica para a comparação, permitindo a identificação da técnica mais promissora para a automatização do processo de contagem de laranjas e, por extensão, a sua aplicação em outros contextos da agricultura e indústria. Os resultados obtidos fornecem insights relevantes para o desenvolvimento de sistemas de contagem automatizados, contribuindo para a otimização de recursos e o aumento da rentabilidade na produção e comercialização de frutas.

2. Referencial teórico

2.1. Conjunto de dados

(GOOGLE) Um conjunto de dados é vários dados organizados usados frequentemente para análise pesquisa ou treinamento de modelos de inteligência artificial.

2.2. Inteligência artificial

"A inteligência artificial é um ramo da Ciência da Computação cujo interesse é fazer com que os computadores pensem ou se comportem de forma inteligente." (GOMES, 2010)

2.3. Python

"Python é uma linguagem de programação interpretada, orientada a objetos e de alto nível com semântica dinâmica." (PYTHON)

2.4. Roboflow

(ROBOFLOW) É uma plataforma que tem um conjunto de ferramentas que ajudam a desenvolver conjunto de dados e modelos de visão computacional mais facilmente. Algumas ferramentas presentes são:

- Anotação de dados: permite definir a classificar as imagens assim são criados os conjuntos de dados
- Treinamento de modelos: permite que seja feito o treinamento do modelo de visão computacional
- Implantação: facilita a implantação do modelo na nuvem, dispositivos móveis e etc.

2.5. YOLO

YOLO é a sigla para *You only look once* (Você só olha uma vez, tradução livre), é um modelo para detecção e classificação de objetos em imagens ou vídeos em tempo real.

YOLO foi criado por Joseph Redmon, Ali Farhadi e Adam Bewley em 2016, (Redmon et al, 2016) utiliza uma rede neural convolucional para prever várias caixas delimitadoras e probabilidades de classe para essas caixas na imagem inteira de uma só vez diferente de técnicas que utilizam caixas deslizantes que precisam testar um classificador em diferentes áreas da imagem. Assim utilizando essa técnica o YOLO é capaz de processar 45 imagens por segundo em uma placa de vídeo NVIDIA Titan X.

(Redmon et al, 2016) *YOLO* usa um sistema que divide a imagem de entrada em uma grade de S × S onde cada célula dessa grande fica responsável de detectar objetos em seu interior e prever as caixas delimitadoras e a probabilidade de pertencer a alguma classe.

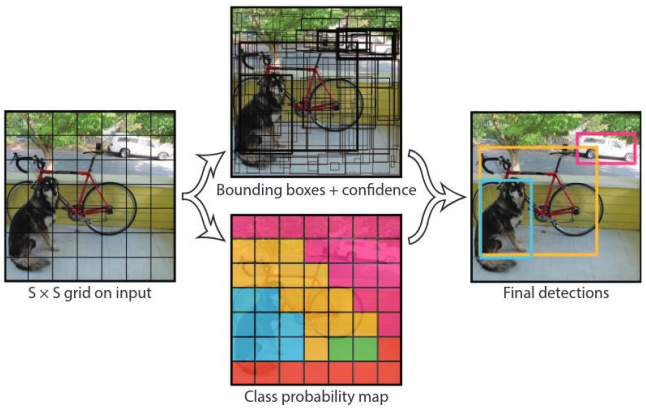


Figura 1: Estrutura do modelo YOLO.

Fonte: Redmon et al, You Only Look Once: Unified, Real-Time Object Detection. 2016.

Atualmente existem duas versões populares, uma mantida por Joseph, onde se encontra na versão 7, e outra mantida pela empresa *Ultralyrics* que está na versão 11.

2.6. Ultralytics

É uma empresa voltada para inteligência artificial sendo uns dos seus principais produtos é uma bifurcação do modelo *YOLO* assim criando suas próprias versões deste modelo. Também é responsável pela criação e atualização da biblioteca para linguagem de programação *python*, *ultralytics*, que interage de forma fácil com o modelo *YOLO* distribuído pela empresa.

2.7. OpenCV

"OpenCV (Open Source Computer Vision Library) é uma biblioteca de software de código aberto de visão computacional e aprendizado de máquina." (OPENCV)

2.8. Scikit-image

Scikit-image ou *skimage* é uma biblioteca para linguagem de programação *python* com várias funções para processamento de imagem e visão computacional.

3. Matérias e métodos

3.1. Versão do *YOLO* utilizadas

A versão utilizada foi a oferecida pela *Ultralyrics*, pois possui uma grande gama de documentação sobre o seu uso, ser mais simples a sua implementação e por conta do seu desenvolvimento estar mais ativo em relação a versão oferecida por Joseph, onde a última atualização do código fonte no repositório do GitHub é dia 17 de julho de 2022 (no momento da escrita deste artigo).

Além disso foram utilizadas as versões número 5, 8, 9 e 11 por conta da sua disponibilidade no repositório de versões do modelo da empresa *ultralyrics*.

3.2. Images utilizadas no conjunto de dados

Para este trabalho, foram selecionadas 402 imagens provenientes de dois conjuntos de dados: o *Kaggle* e o images.cv, onde todas as imagens são de uso livre. As imagens obtidas a partir do *Kaggle* têm como origem o artigo científico *Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks* (ZHU et al., 2017). Já as imagens do images.cv foram extraídas diretamente do repositório oficial da plataforma (IMAGES.CV).

Não foi restringida a seleção a uma única espécie de laranja; em vez disso, optou-se por incluir apenas laranjas com coloração laranja uniforme, que apresentassem sua forma íntegra, sem partes faltantes, como cortes ao meio ou segmentos removidos. Além disso, foram priorizadas frutas com formato arredondado, excluindo aquelas com deformidades significativas, a fim de manter a consistência visual dos dados analisados.

3.3. Criação do conjunto de dados

Foi necessário classificar as imagens criando os rótulos, que são as informações sobre onde estão localizadas as laranjas nas imagens, para isso foi utilizada a plataforma *roboflow*, a seguir é demonstrado os passos de como foi rotulada cada imagem e criado o conjunto de dados:

1. Criar a sua conta na plataforma.
2. Criar um novo projeto clicando em *create new projec*t.
3. Definir os atributos do projeto, crie um nome para o projeto, selecione uma licença de distribuição de software, adicione o nome dos objetos que irá ser detectado em *annotation group*, e o principal, selecione o tipo do projeto, neste caso como é detecção de laranjas selecione *object detection*.

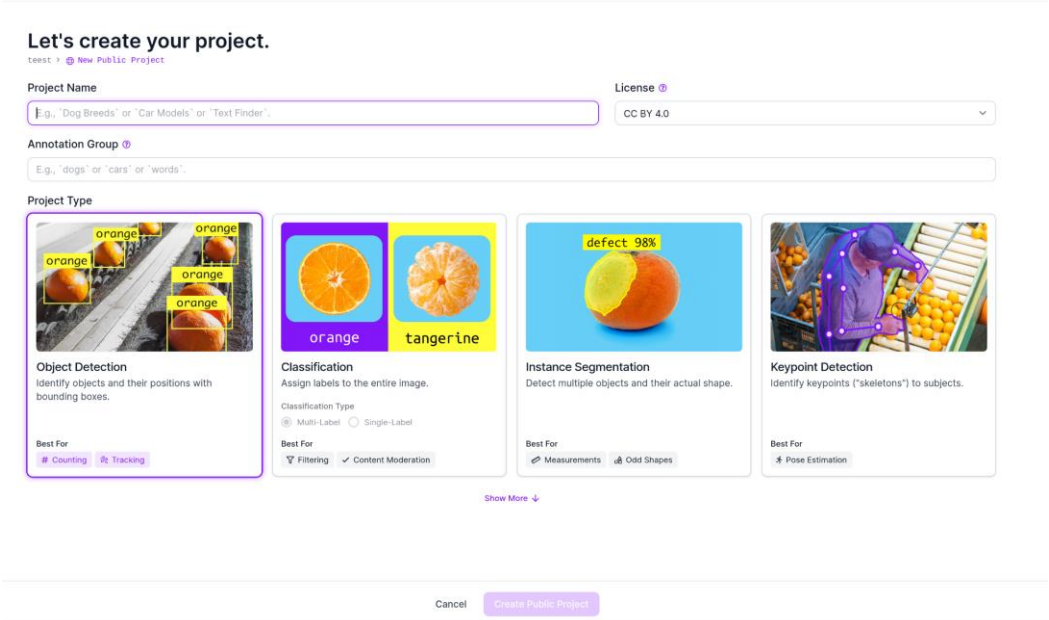


Figura 2: Definição dos atributos do projeto na plataforma roboflow.

Fonte: Autor.

4. Enviar as imagens para a plataforma.

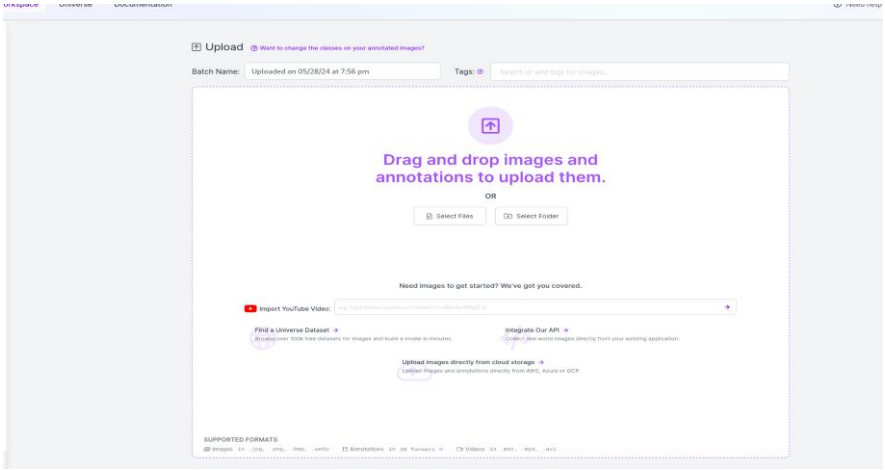


Figura 3: Página web para envio das imagens na plataforma roboflow.

Fonte: Autor.

5. Classificar as imagens, existem 3 opções de como pode ser feita a classificação, *Auto Label*, será usado um modelo de classificação genérico para classificar as imagens, *Roboflow Labeling*, será contratada uma equipe da plataforma para fazer a classificação das imagens e por último *Manual Labeling*, onde deverá ser feita a classificação forma manual dos dados pelo o autor do projeto, neste projeto foi selecionada a opção de *Manual Labeling.*

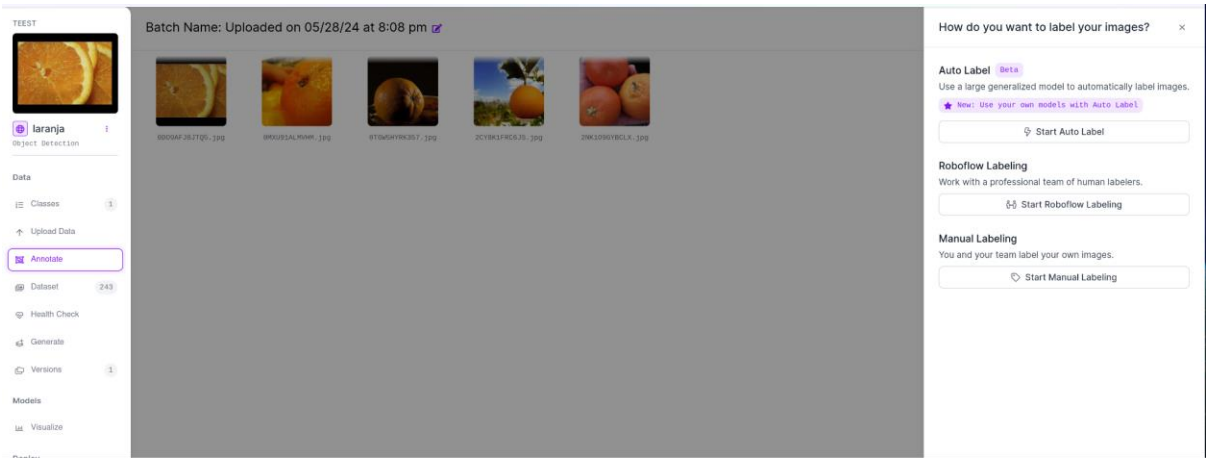


Figura 4: Página para seleção do método de classificação.

Fonte: Autor.

6. Adicionar pessoas ao projeto, essa etapa é opcional pois é necessário selecionar a opção *Manual Labeling* para classificação de imagens. A plataforma irá solicitar se irá adicionar alguém para ajudar a classificar as imagens, caso tenha clique em “invite teammate” e após isso clique em “assign images” para designar a essa pessoa as imagens que ela irá classificar.

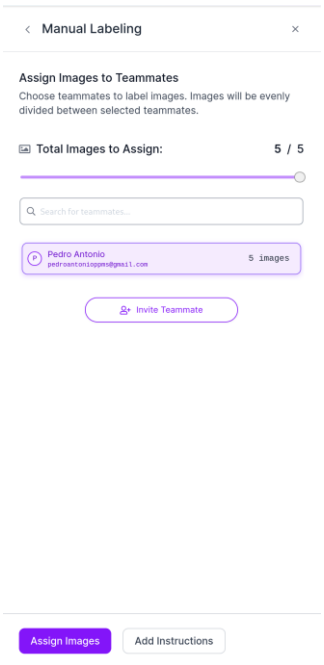


Figura 5: Página para adicionar colaboradores no projeto

Fonte: Autor.

7. Começar a classificar a imagem, depois de selecionar as pessoas que irão classificar as imagens, é redirecionado a uma página de visão geral mostrando as imagens que serão classificadas, clique em *start annotating*.

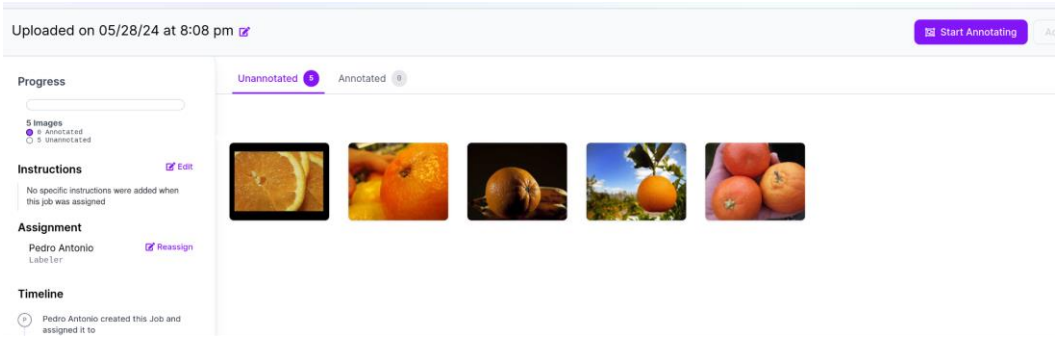


Figura 6: Visão geral das imagens a serem classificadas.

Fonte: autor.

8. Classificar uma imagem, será aberta uma página para classificar as imagens, selecione a opção *bounding box tool* e desenhe a caixa envolta do objeto desejado, nesse caso uma laranja, assim será aberto um *popup* para inserir a classe do objeto, depois disso clique em “save”.

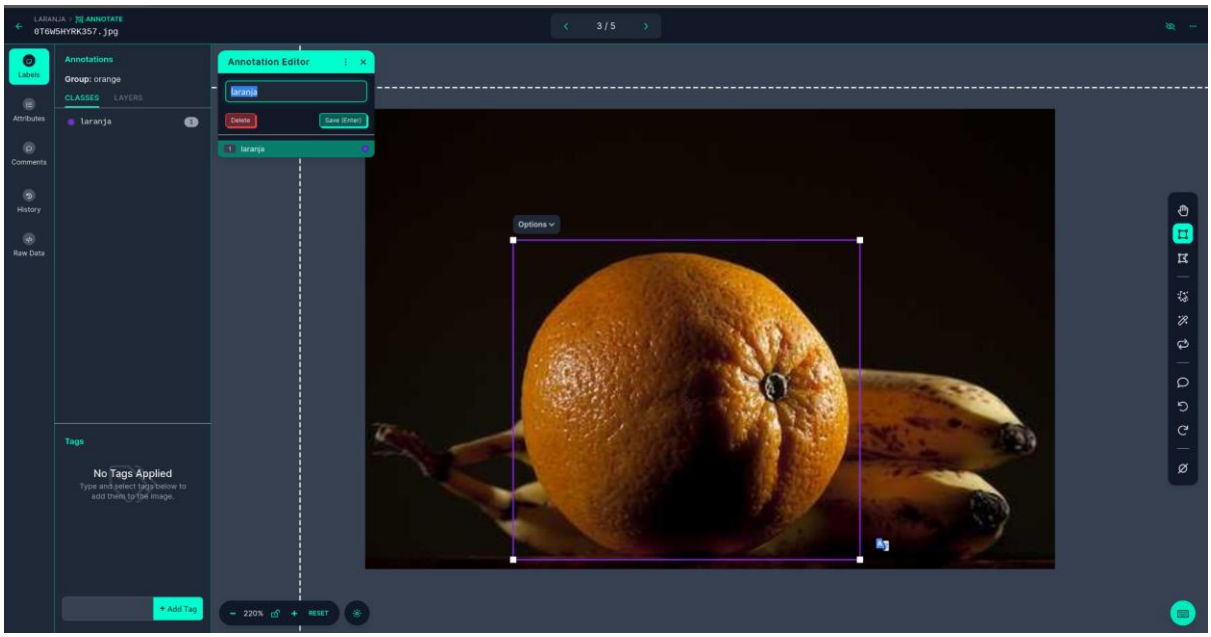


Figura 7: Exemplo de classificação de uma laranja

Fonte: Autor.

9. Criar uma nova versão do conjunto de dados, após classificar todas as imagens será gerado um *dataset (*termo em inglês para conjunto de dados), com esse *dataset* será necessário criar uma nova versão do mesmo, para isso clique em *versions* na aba na esquerda, depois em *create new version*.

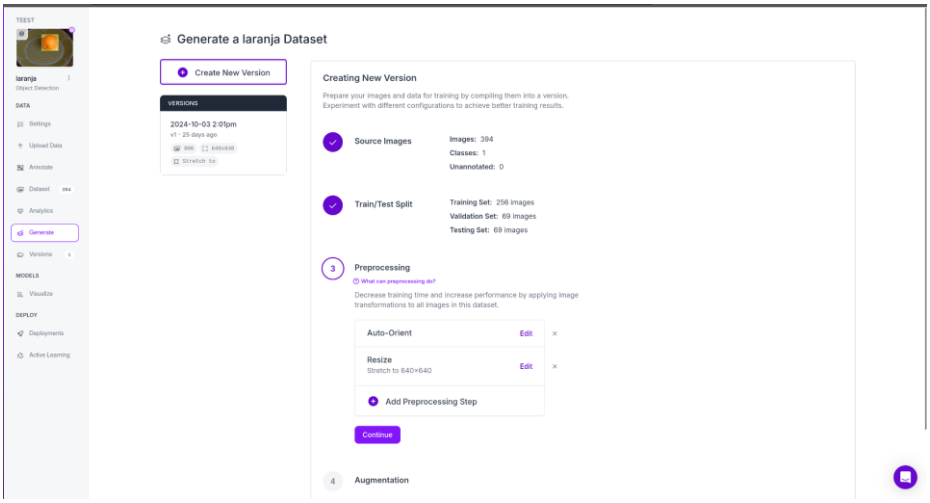


Figura 8: Página para criar uma nova versão do conjunto de dados.

Fonte: Autor.

9.1. Existem alguns passos a serem seguidos, sendo eles:

9.1.1. Selecionar a origem das imagens, após isso clique em “continue”.

9.1.2. *Train/Test Split* nessa opção será selecionada a proporção de imagens que serão separadas para o conjunto de treinamento, validação e teste.

9.1.3. *Preprocessing*, nesta etapa serão aplicadas algumas transformações no conjunto de imagens a fim de melhorar o desempenho do treinamento.

9.1.4. *Augmentation* é uma opção onde são criados novos exemplos para o treinamento aplicado alguns filtros ou transformações na imagem como inverter a imagem horizontalmente. Foram selecionadas as opções *flip horizontal* e *vertical* onde serão invertidas as imagens na horizontal e vertical, *crop* onde serão cortadas algumas partes da imagem, e a última opção foi *noise*, onde são criados ruídos em algumas imagens de até 1% dos *pixels*.

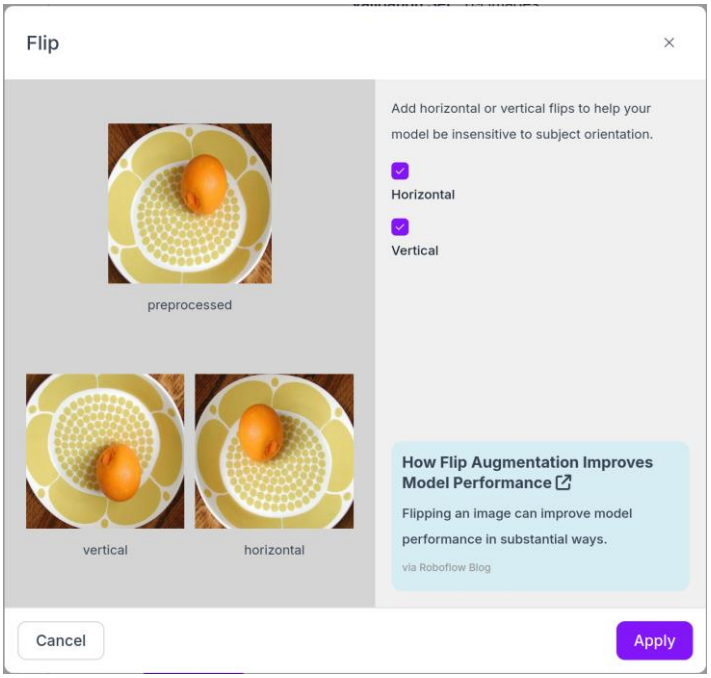


Figura 9: Exemplo de inversão na imagem para criar novos dados no conjunto de dados.

Fonte: Autor.

9.1.5. E por fim a última opção é criar uma nova versão.

10. Exportar o conjunto de dados, após criar uma nova versão clique em *download dataset*, selecione o formato *YOLO v5 PyTorch* e clique em continue assim será feito do *download* o conjunto de dados compactado no formato *zip*.

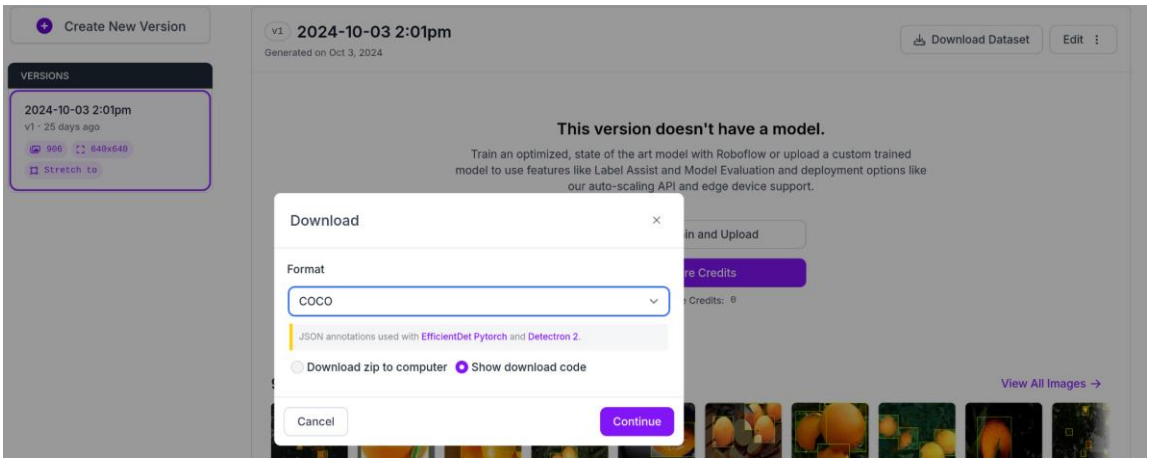


Figura 10: Página para seleção do formato e exportar o conjunto de dados.

Fonte: Autor.

3.4. Treinamento

Para realizar o treinamento do modelo foi necessário fazer o *download* do conjunto de dados, instalar a biblioteca *ultralytics* e a importar no código *python*, definir um arquivo com extensão *YAML* com os detalhes como caminho para as imagens de treino, validação e teste, e as classes que os objetos serão classificados. Se o conjunto de dados foi criado pela plataforma

roboflow, esse arquivo é criado automaticamente apenas necessitando corrigir o caminho para os diretórios contendo as imagens do conjunto de treino, validação e teste. Abaixo o código na linguagem *python* para realizar treinamento:

```python
import os
from ultralytics import YOLO, settings

def main():
    # Carrega o modelo pretreinado do YOLO
    # YOLO 5 - yolov5nu.pt      You, 5 days ago • corrigido o nome dos modelos e aumenta
    # YOLO 8 - yolov8n.pt
    # YOLO 9 - yolov9c.pt
    # YOLO 11 - yolo11n.pt
    model = YOLO("yolov9c.pt")

    settings.update({"datasets_dir": os.path.join(os.path.abspath("."), "dataset")})

    # Treina o modelo usando o dataset de laranjas
    model.train(data="./data.yaml", epochs=100)


if __name__ == "__main__":
    main()
```

Figura 11: Código usado para o treinamento do modelo.

Fonte: Autor.

Após finalizar a execução o código será gerado uma pasta *train* no caminho *runs/detect/train* onde o modelo está no diretório *weights*. Nesse diretório existem dois arquivos, o arquivo *last.pt* contém o modelo com os pesos da última época treinada, o arquivo *best.pt* contém modelo com os pesos da melhor época que obteve a melhor pontuação nos testes. Esses testes são definidos pela biblioteca *ultralyrics*, são usadas métricas como matriz de confusão.

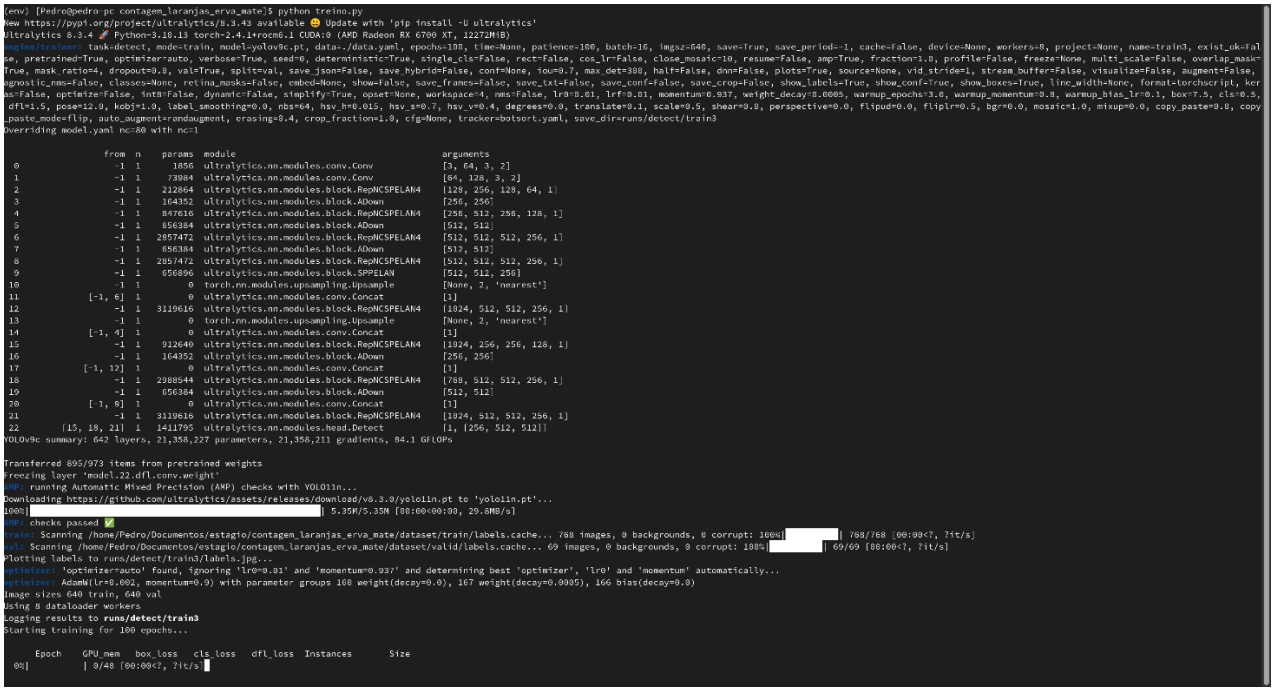


Figura 12: Execução do código para treinamento do novo modelo

Fonte: Autor.

3.5. Detecção

Para a detecção é carregado o modelo treinado pela classe *YOLO*, onde primeiro parâmetro será o caminho para o modelo treinado e a seguir é utilizada a função *predict* presente nesta classe junto com a imagem para fazer as predições.

A função *predict* precisa de dois parâmetros sendo primeiro a origem da imagem, e o segundo chamado *conf* que é taxa de confiança mínima que um determinado objeto precisa ter para seja detectado e classificado, essa variável deve ser em decimal onde por exemplo 0.1 representa 10% de confiança. Abaixo contém o código utilizado para contagem:

```python
import io
from PIL import Image
from ultralytics import YOLO

def contar_objetos(modelo: str, imagem: str, conf=0.5):
    # Carrega o modelo
    model = YOLO(modelo)

    # Faz as predições
    results = model.predict(source=imagem, conf=conf, show_labels=False, show_boxes=True)

    # Pega as caixas detectadas e o nome das classes que tem no modelo
    boxes = results[0].boxes # objeto com as caixas delimitadoras
    names = results[0].names # dicionarios com id e nome de cada classe

    count: dict[int, int] = {}
    if boxes:
        # Dicionario para contar quantos objetos de uma classe foram detectados
        # Enche o dicionarios com 0 para cada classe
        for key in names.keys():
            count[key] = 0

        # Conta os objetos de cada classe e adiciona no dicionario count
        classes_objetos = boxes.cls.cpu().numpy()
        for classe in classes_objetos:
            classe = int(classe)
            count[classe] = count[classe] + 1

    imagem_bgr = results[0].plot(labels=False)
    im_rgb = Image.fromarray(imagem_bgr[..., ::-1])

    # Salva a imagem como bytes
    buffer = io.BytesIO()
    im_rgb.save(buffer, format="JPEG")
    imagem_final = buffer.getvalue()
    buffer.close()

    contagem_final: dict[str, int] = {}
    for key, value in count.items():
        name = names[key]
        contagem_final[name] = value

    return contagem_final, imagem_final

if __name__ == "__main__":
    # Seleciona modelo usado e a imagem para ser detectada
    contagem, imagem = contar_objetos("./treinamento/yolov5/weights/best.pt", "1.jpg")

    print(contagem)
    with open("imagem_teste.jpg", "wb") as file:
        file.write(imagem)
```

Figura 13: Código usado para contagem de laranjas com YOLO.

Fonte: Autor.

Após a execução do código é exibido a contagem de laranjas da imagem na saída de dados e a imagem com as caixas de detecção será salva em um arquivo chamado “imagem_teste.jpg”.

```
(env) [Pedro@pedro-pc contagem_laranjas_erva_mate]$ python deteccao_yolo.py

image 1/1 /home/Pedro/Documentos/estagio/contagem_laranjas_erva_mate/docs/laranja.png: 640x640 3 laranjas, 6.3ms
Speed: 2.6ms preprocess, 6.3ms inference, 113.9ms postprocess per image at shape (1, 3, 640, 640)
Contagem de laranjas: {'laranja': 3}
```

Figura 14: Execução do código de contagem de laranjas e resultados da contagem com YOLO na versão 5

Fonte: Autor.

A figura 15 apresenta um resultado da contagem onde foram detectadas 3 laranjas com a confiança que são da classe laranja de 82%, 92% e 88%.

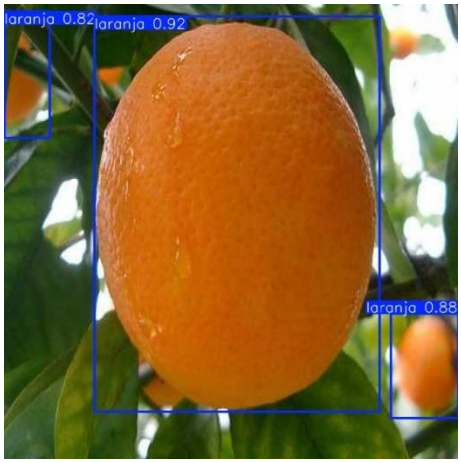
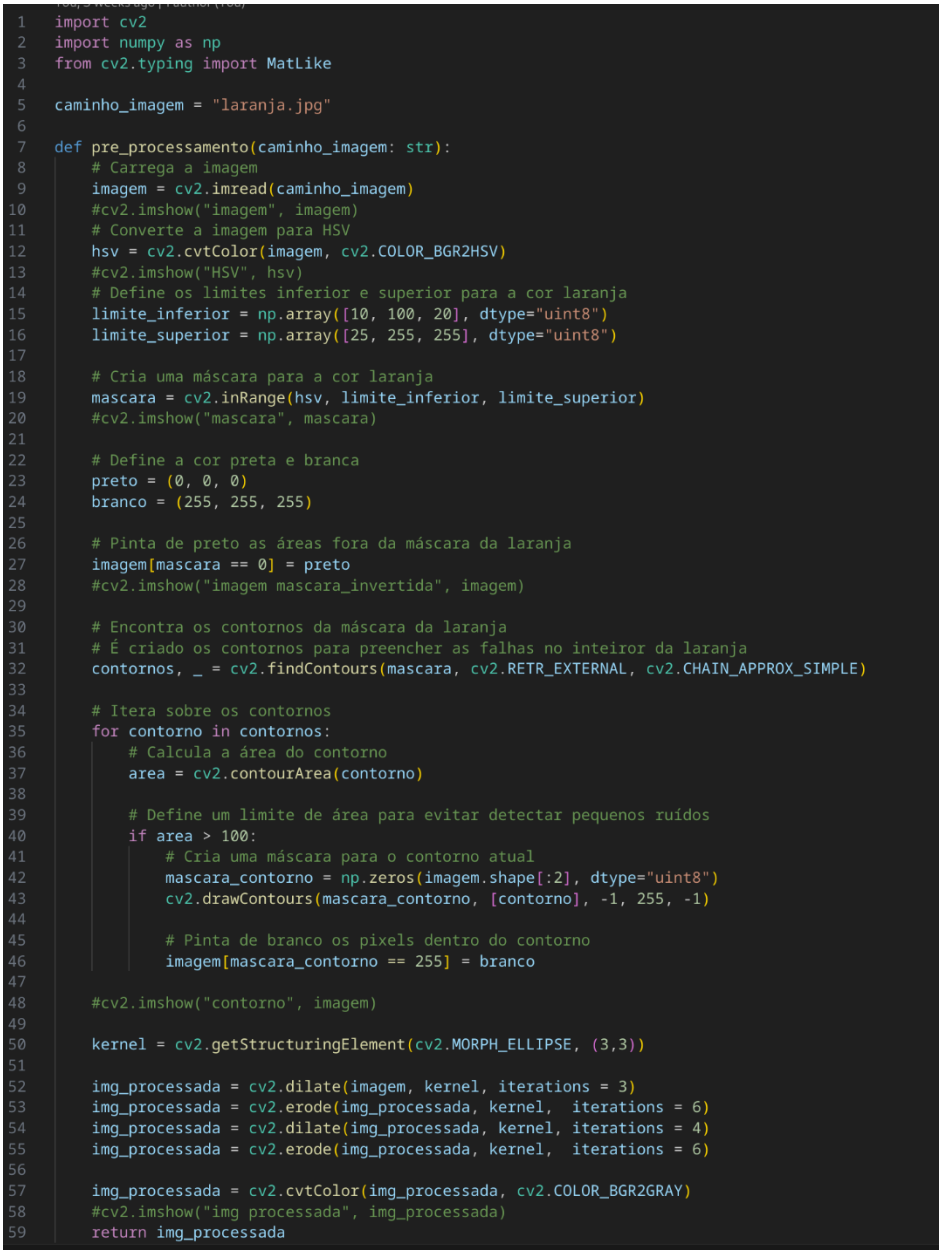


Figura 15: Resultado da detecção após utilizar YOLO.

Fonte: Autor.

3.6. Algoritmos de visão computacional

Foram utilizadas bibliotecas para a linguagem de programação Python, OpenCV e scikit-image. A biblioteca OpenCV foi utilizada para filtrar a cor laranja, remover ruídos da imagem e, em conjunto com a scikit-image, realizar a contagem e detecção dos objetos. (FIRIGATO) O código-base para este estudo foi adaptado a partir do disponibilizado por João Otávio Firigato no ambiente do *google collab*. Diversas alterações foram realizadas no código original, como a utilização da escala de cores HSV e modificações nas funções e bibliotecas empregadas. Entre as principais mudanças, destacam-se a aplicação da escala HSV para detecção de cor, ajustes nos valores dos limites inferior e superior, a redefinição de funções de dilatação e erosão, e a alteração dos parâmetros relacionados à área mínima dos objetos detectados e às funções de contorno. As figuras 16 e 17 apresentam o código criado para o pré-processamento e contagem de objetos usando as bibliotecas OpenCV e scikit-image.



```python
import cv2
import numpy as np
from cv2.typing import MatLike

caminho_imagem = "laranja.jpg"

def pre_processamento(caminho_imagem: str):
    # Carrega a imagem
    imagem = cv2.imread(caminho_imagem)
    #cv2.imshow("imagem", imagem)
    # Converte a imagem para HSV
    hsv = cv2.cvtColor(imagem, cv2.COLOR_BGR2HSV)
    #cv2.imshow("HSV", hsv)
    # Define os limites inferior e superior para a cor laranja
    limite_inferior = np.array([10, 100, 20], dtype="uint8")
    limite_superior = np.array([25, 255, 255], dtype="uint8")

    # Cria uma máscara para a cor laranja
    mascara = cv2.inRange(hsv, limite_inferior, limite_superior)
    #cv2.imshow("mascara", mascara)

    # Define a cor preta e branca
    preto = (0, 0, 0)
    branco = (255, 255, 255)

    # Pinta de preto as áreas fora da máscara da laranja
    imagem[mascara == 0] = preto
    #cv2.imshow("imagem mascara_invertida", imagem)

    # Encontra os contornos da máscara da laranja
    # É criado os contornos para preencher as falhas no inteiror da laranja
    contornos, _ = cv2.findContours(mascara, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Itera sobre os contornos
    for contorno in contornos:
        # Calcula a área do contorno
        area = cv2.contourArea(contorno)

        # Define um limite de área para evitar detectar pequenos ruídos
        if area > 100:
            # Cria uma máscara para o contorno atual
            mascara_contorno = np.zeros(imagem.shape[:2], dtype="uint8")
            cv2.drawContours(mascara_contorno, [contorno], -1, 255, -1)

            # Pinta de branco os pixels dentro do contorno
            imagem[mascara_contorno == 255] = branco

    #cv2.imshow("contorno", imagem)

    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))

    img_processada = cv2.dilate(imagem, kernel, iterations = 3)
    img_processada = cv2.erode(img_processada, kernel,  iterations = 6)
    img_processada = cv2.dilate(img_processada, kernel, iterations = 4)
    img_processada = cv2.erode(img_processada, kernel,  iterations = 6)

    img_processada = cv2.cvtColor(img_processada, cv2.COLOR_BGR2GRAY)
    #cv2.imshow("img processada", img_processada)
    return img_processada
```

Figura 16: Primeira parte do código contendo a função que realiza o pré-processamento

Fonte: Autor.

```python
def contar_objetos_opencv(imagem: MatLike, caminho_imagem: str):
    # Encontrar componentes conectados
    _num_obj, _, stats, _ = cv2.connectedComponentsWithStats(imagem, connectivity=4, ltype=cv2.C

    # Abre a imagem original para criar as caixas de detecção
    img_com_caixa = cv2.imread(caminho_imagem)

    # Como é filtrado alguns objetos é necessario criar uma variavel nova para contar
    # o numero de objetos detectados
    num_obj = 0
    for i in range(_num_obj):
        # Pega o ponto x e y do canto superior esquerdo do objeto
        x = stats[i, cv2.CC_STAT_LEFT]
        y = stats[i, cv2.CC_STAT_TOP]
        # Pega a largunta e altura do objeto
        w = stats[i, cv2.CC_STAT_WIDTH]
        h = stats[i, cv2.CC_STAT_HEIGHT]
        # Pega area
        area = stats[i, cv2.CC_STAT_AREA]

        height, width, _ = img_com_caixa.shape

        # Algumas imagens tem objetos detectados como a imagem inteira então o if abaixo ignora
        # Filtra objetos com a area maior que 100 pixeis
        if area >= 100 and w * h < height * width:
            # Desenha a caixa de detecção
            cv2.rectangle(img_com_caixa, (x, y), (x + w, y + h), (0, 255, 0), 2)
            num_obj += 1

    return num_obj, img_com_caixa

def contar_objetos_skimage(imagem: MatLike, caminho_imagem: str):
    from skimage.measure import label, regionprops

    # Encontrar componentes conectados
    label_image = label(imagem)

    # Abre a imagem original para criar as caixas de detecção
    img_com_caixa = cv2.imread(caminho_imagem)

    num_obj = 0
    for region in regionprops(label_image):
        # Filtra objetos com a area maior que 100 pixeis
        if region.area >= 100:
            # Obtem as coordenadas da região
            minr, minc, maxr, maxc = region.bbox

            # Desenha a caixa de detecção
            cv2.rectangle(img_com_caixa, (minc, minr), (maxc, maxr), (0, 255, 0), 2)  # Verde, 2

            num_obj += 1

    return num_obj, img_com_caixa

if __name__ == "__main__":
    img_processada = pre_processamento(caminho_imagem)

    num_obj_cv2, img_cv2 = contar_objetos_opencv(img_processada, caminho_imagem)

    num_obj_skimage, img_skimage = contar_objetos_skimage(img_processada, caminho_imagem)
```

Figura 17: Segunda parte do código contendo as funções que fazem a contagem com OpenCV e scikit-image

Fonte: Autor.

O para fazer a contagem foi necessário primeiro fazer um pré-processamento da imagem, onde consiste converter a cor laranja para branco e as outras cores restantes para preto, e também eliminar os ruídos gerados por esse processo. Abaixo contém a imagem utilizada para a exemplificação dos passos seguidos no pré-processamento:

Figura 18: Imagem utilizada para exemplificação dos algoritmos de visão computacional.

Fonte: Autor.

O primeiro passo do pré-processamento da imagem foi converter a imagem do espaço de cores RGB para HSV (*Hue*, *Saturation*, *Value*), (MOTA) onde *Hue* é a cor, Saturation é a saturação da cor e Value é o brilho da cor. HSV é bastante utilizado para detecção de cores pois não depende da luminosidade e saturação da cor, sendo assim basta olhar para o intervalo que representa a cor.

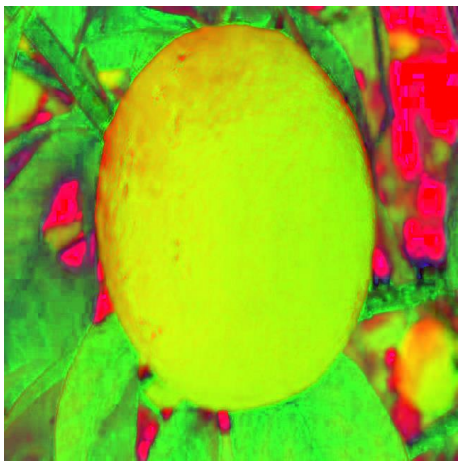


Figura 19: Imagem após a conversão para espaço HSV.

Fonte: Autor.

O segundo passo foi definir um valor do limite inferior e superior para aplicarmos uma máscara onde será preto as partes de outras coisas diferentes de laranja, nesse caso o valor

escolhido foi 10 para *Hue*, 100 *Saturation*, 20 *Value* para limite inferior e 25 *Hue*, 255 *Saturation*, 255 *Value* para limite superior.

Com os valores definidos foi criada a máscara usando a função *inRange* da biblioteca *OpenCV* onde os *pixels* que estão dentro do intervalo do limite inferior e superior são brancos e outros *pixels* são pretos. Na próxima etapa foi utiliza a função *findContours* junto com a máscara criada para encontrarmos os contornos dentro da imagem para removermos pontes com falhas dentro dos objetos de cor branca para isso foi iterado sobre cada contorno calculando sua área com a função *contourArea* e caso sua área seja maior que 100 *pixels* é preenchido de branco no interior desse contorno, as figuras 20 e 21 fazem a comparação de antes e depois de aplicarmos este método:

Figura 20: Máscara gerada antes do preenchimento interno dos contornos.

Fonte: Autor.

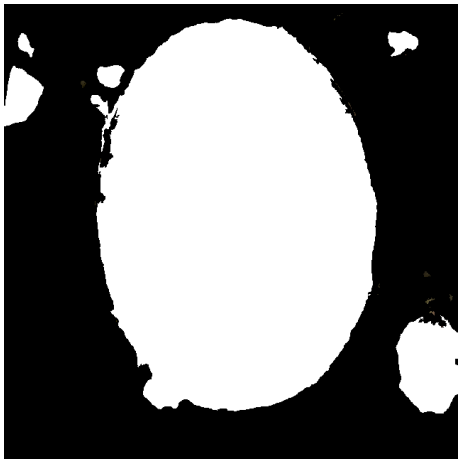


Figura 21: Máscara gerada após o preenchimento interno dos contornos.

Fonte: Autor.

Com a remoção das falhas no interior dos objetos são aplicados os processos de dilatação com a função *dilate* e erosão com a função *erode* para remover ruídos na imagem,

esse processo pode ser repetido de várias vezes podendo ser intercalo ou continuo. Nas figuras 22 e 23 é demonstrada a dilatação e erosão sendo usadas duas vezes intercalando o seu uso:

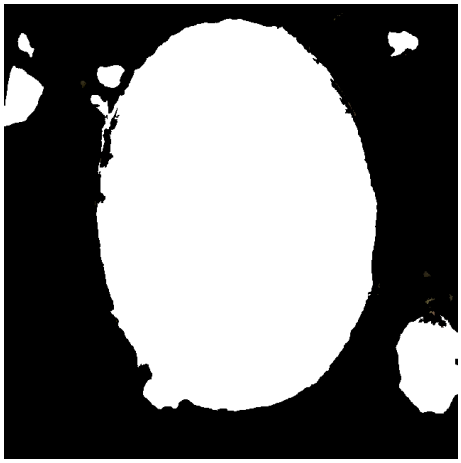


Figura 22: Imagem antes da dilatação e erosão.

Fonte: Autor.

Figura 23: Imagem após a dilatação e erosão.

Fonte: Autor.

Com o pré-processamento da imagem pronto existem duas formas para criar as caixas delimitadoras envolvendo as laranjas, com *OpenCV* utilizando a função *connectedComponentsWithStats* que busca componentes conectados na imagem assim retornando a caixa do componente, a sua área em *pixels* e o centro dele e a outra maneira é utilizar a biblioteca scikit-image com as funções, *label* que rotula os componentes conectados através de *pixels* que são vizinhos e tem o mesmo valor e *regionprops* que tem a função de medir a regiões rotuladas assim retornando quase as mesmas propriedades da função do *OpenCV*.

3.6.1. Utilizando *OpenCV*

(ROSEBROCK) Foi utilizado a função já mencionada passando o objeto que representa a imagem como parâmetro, 4 para *connectivity* onde os 4 *pixels* são considerados como

vizinhos e *cv2.CV_32S* para *ltype* definindo o tipo da matriz de entrada. A função retorna uma tupla onde o primeiro parâmetro é o número de objetos detectado, o segundo parâmetros a *labels* ou rótulos, terceiro parâmetro é *status* que contém a altura e largura do objeto detectado e x e y que são as coordenadas do ponto superior à esquerdo da caixa de detecção. Abaixo tem a imagem da detecção das laranjas.

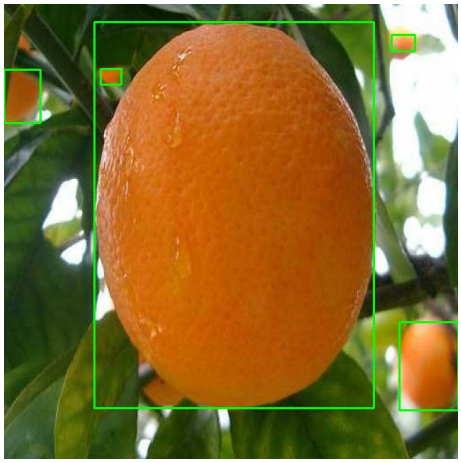


Figura 24: Imagem gerada pela biblioteca OpenCV com as caixas delimitadoras.

Fonte: Autor.

3.6.2. Utilizando *scikit-image*

É necessário a função *label* com a imagem como parâmetro, assim o valor retornado pela função será utilizado pela função *regionprops*. Com isso o valor do ponto superior esquerdo e do ponto inferior direito está presente na variável *bbox*. Abaixo a imagem com o resultado da detecção.

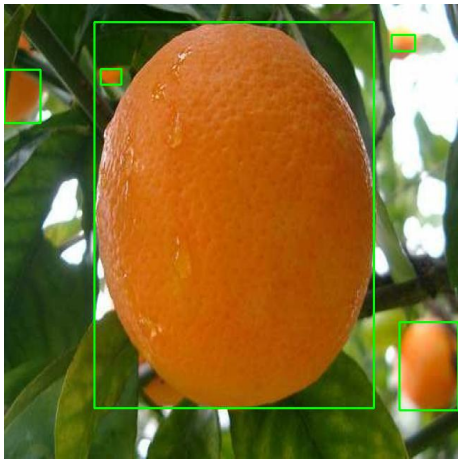


Figura 25: Imagem gerada pela biblioteca scikit-image com as caixas delimitadoras.

Fonte: Autor.

Os resultados da execução, com as imagens com as caixas delimitadoras geradas por ambos os métodos, são apresentados nas Figuras 26.

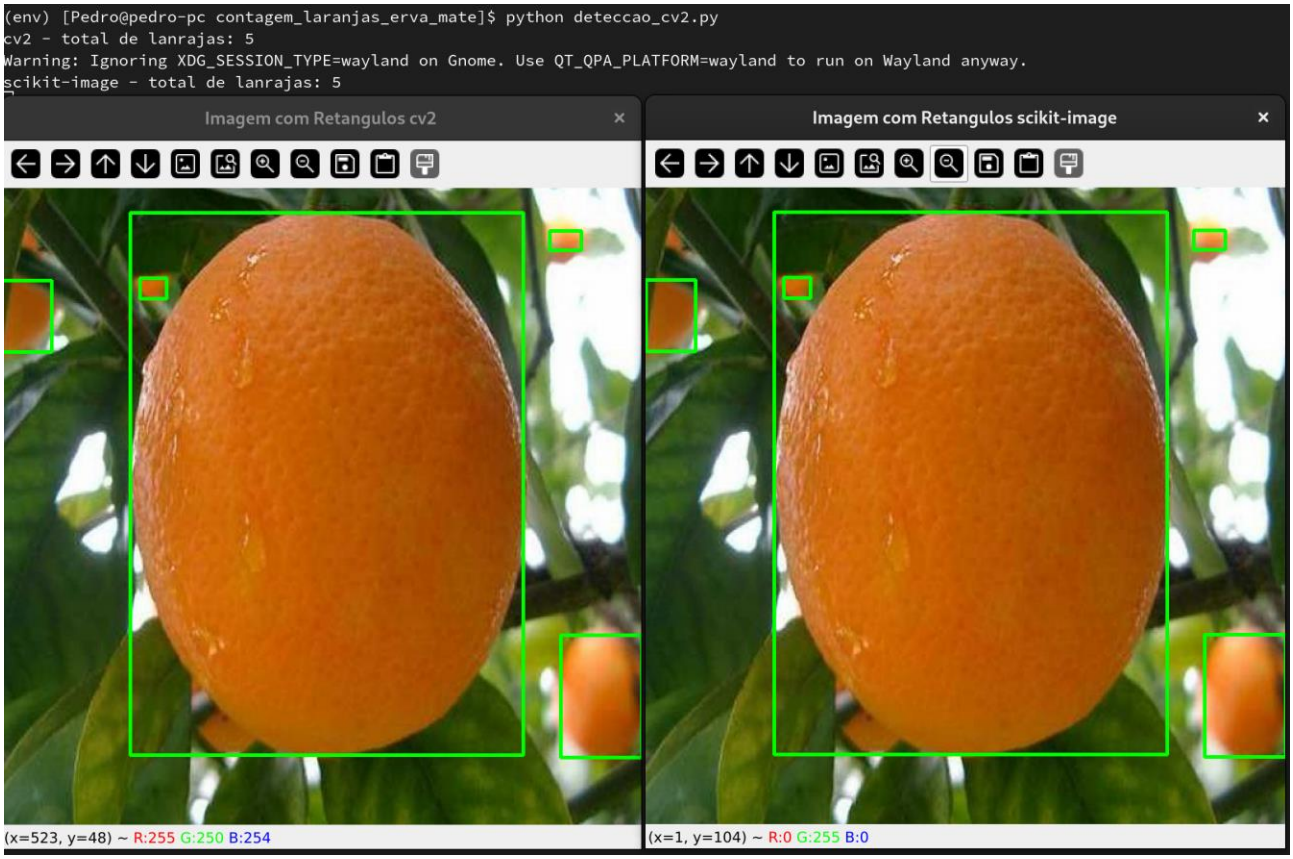


Figura 26: Execução do código de contagem de laranjas e resultados da contagem de ambas as técnicas

Fonte: Autor.

3.7. Dificuldades na implementação

A implementação e o treinamento do modelo YOLO apresentaram diversos desafios técnicos e metodológicos ao longo do processo. Um dos principais obstáculos foi a necessidade de configurar o ambiente de execução para suportar placas de vídeo da fabricante AMD. Para isso, foi preciso instalar o *ROCm* (*Radeon Open Compute*), um conjunto de ferramentas específico para o suporte a placas de vídeo AMD, além de uma versão específica da biblioteca *PyTorch* compatível com essa plataforma. Esse processo demandou tempo e esforço devido à necessidade de seguir instruções específicas, solucionar problemas de compatibilidade e ajustar configurações do sistema.

Outro desafio significativo foi determinar o número mínimo de épocas necessário para o treinamento eficiente do modelo. Treinamentos com poucas épocas resultavam em um modelo subajustado, enquanto muitas épocas aumentavam o tempo de treinamento sem garantias de melhora substancial. Encontrar o equilíbrio ideal exigiu experimentação cuidadosa e análise dos resultados intermediários.

A seleção de imagens de melhor qualidade para compor o conjunto de dados também se mostrou crucial. Imagens com baixa resolução ou condições de iluminação inadequadas

comprometiam a precisão do modelo. Assim, foi necessário revisar o conjunto de dados e remover imagens que poderiam introduzir ruído ou dificultar o aprendizado.

Além disso, a definição de limites inferiores e superiores na escala HSV para a detecção da cor laranja destacou as dificuldades na implementação de algoritmos de visão computacional clássicos. A escolha precisa desses limites é sensível a variações de iluminação, sombras e diferenças nos tons de laranja, tornando o ajuste um processo complexo e iterativo.

Outros parâmetros exigiram atenção cuidadosa para otimizar o desempenho do algoritmo, como a área mínima dos contornos detectados, o número de iterações para as funções de dilatação e erosão, e a área mínima dos objetos identificados como relevantes.

4. Resultado e discussão

Para determinar a eficácia dos algoritmos foram selecionadas 55 imagens para o teste que não estão no conjunto de dados, as imagens do conjunto de dados estão disponíveis nos sites *images.cv* e *kaggle*. Foi utilizada a plataforma *robloflow* com as imagens para criar o conjunto de dados. Foi feita a contagem manual das laranjas nas imagens do conjunto de teste para ser feita a comparação com as técnicas.

O computador utilizado para ser feito os testes foi com processador AMD Ryzen™ 9 5900X, 32 gigabytes de memória *ram*, placa de vídeo AMD Radeon™ RX 6700 XT e sistema operacional *Fedora Linux 40 (Workstation Edition)* e a versão utilizada da linguagem de programação *python* foi 3.10.13.

As versões das bibliotecas utilizadas foram: *ultralytics* versão 8.3.4, *torch* versão 2.4.1+rocm6.1(*rocm* é um software para fazer uma ponte entre um software com a placa de vídeo AMD de forma mais fácil para treinamento de inteligência artificial), *OpenCV-python* versão 4.10.0.84 e *pillow* versão 11.0.0, todas instaladas via o instalador de pacotes *pip*.

Foram treinados 4 modelos *YOLO* em versões diferentes, um na versão 5, versão 8, versão 9 e versão 11, todos treinados no computador já mencionado acima e treinados com 100 épocas. O conjunto de dados foi dividido em 768 imagens no conjunto de treinamento, 69 imagens no conjunto de teste e validação.

Para os algoritmos de visão computacional foram ignorados objetos com área menor que 100 *pixels* e também foram ignorados objetos detectados via a biblioteca *OpenCV* com o tamanho total igual ao tamanho da imagem original.

4.1. Problemas encontrados

4.1.1. *YOLO*

No exemplo abaixo o modelo gerado foi utilizado imagens muito semelhantes assim o modelo não conseguiu capturar diferença entre os objetos criando apenas uma caixa de detecção em volta de todos objetos.

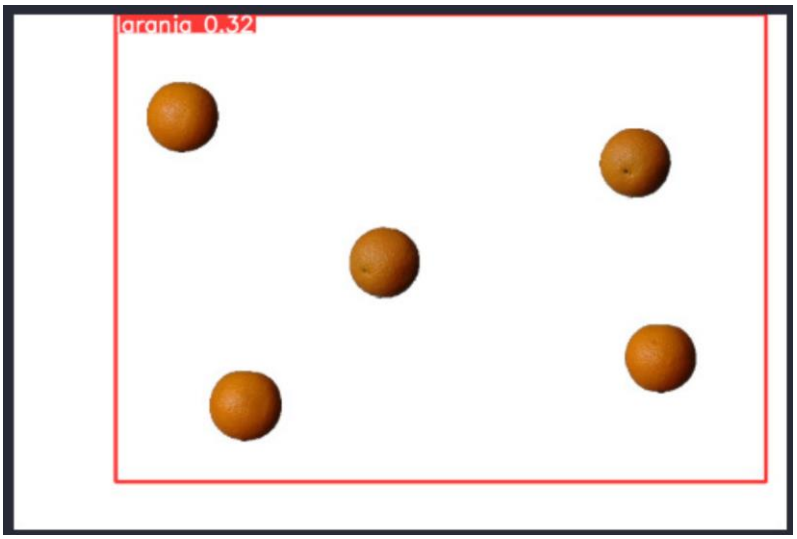


Figura 27: Imagem com a caixa de delimitadora envolvendo todas as laranjas.

Fonte: Autor.

Outro problema como mostrado no exemplo abaixo, o modelo gerado não foi treinado com um número satisfatório de épocas assim o modelo não conseguiu generalizar as laranjas detectando-as mais de uma vez.

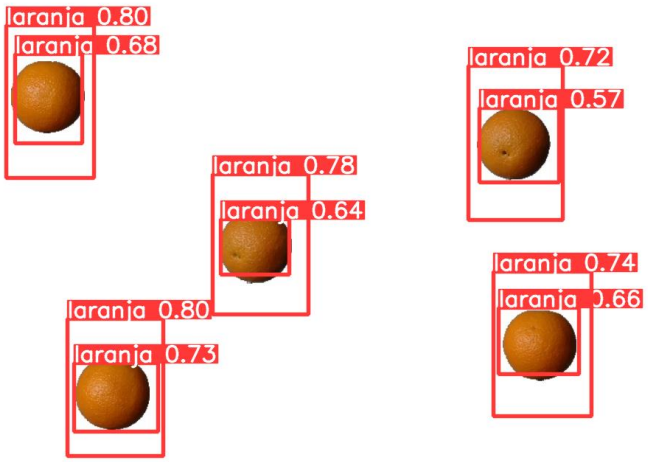


Figura 28: Imagem com várias caixas delimitadoras em cada laranja.

Fonte: Autor.

O grau de confiança definido também pode ocasionar problemas, grau de confiança baixo faz com que objetos que não sejam os desejados sejam detectados e o grau de confiança alto faz o inverso. Na figura número 25 o grau de confiança foi defino muito baixo, para 1%, nela podemos ver que foram detectados 2 objetos que não são desejados.

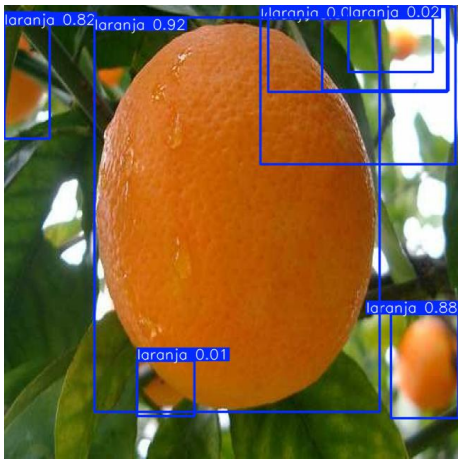


Figura 29: Imagem com um grau de confiança muito baixo.

Fonte: Autor.

Na figura número 26 o grau de confiança foi definido para 90% fazendo com que 2 laranjas não fossem detectadas.

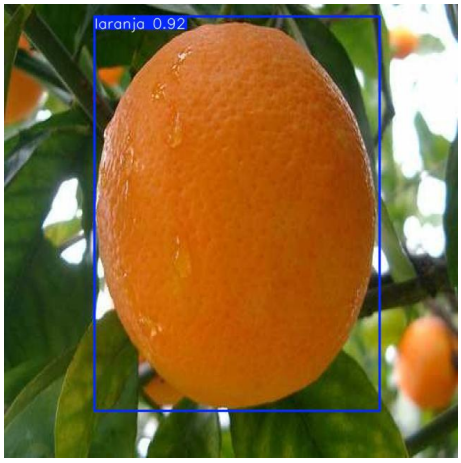


Figura 30: Imagem com um grau de confiança muito alto.

Fonte: Autor.

4.1.2. Algoritmos de visão computacional

Alguns problemas aconteceram com algoritmos de visão computacional, o primeiro quando existem vários objetos com a cor semelhante conectados faz com o que o algoritmo entenda que todos esses objetos são único objeto.

Nas próximas imagens contém a imagem original, processada e os resultados. A imagem original é de várias laranjas juntas fazendo com que usando o algoritmo das bibliotecas *scikit-image* e *OpenCV* tenham dificuldade em criar as caixas delimitadoras.

Figura 31: Imagem com várias laranjas.

Fonte: Autor.

Figura 32: Imagem com várias laranjas depois de processada.

Fonte: Autor.

Figura 33: Imagem com a caixa delimitadora utilizando a biblioteca scikit-image.

Fonte: Autor.

Figura 34: Imagem com caixa delimitadora utilizando a biblioteca OpenCV.

Fonte: Autor.

O segundo problema é se existem outros objetos com uma cor semelhante ao objeto desejado, esse algoritmo o detecta como um dos objetos detectados. Nas imagens abaixo tanto na biblioteca scikit-image e OpenCV detectaram 3 laranjas sendo que só existe 1 laranja na imagem original, ambos algoritmos tiveram dificuldade em diferenciar a laranja em relação a mesa assim caixa delimitadora envolveu a mesa. As sombras também fizeram diferença, foi criada regiões mais escuras onde por conta de fecharem um objeto os algoritmos detectaram essas regiões com um novo objeto.

Figura 35: Imagem original de um laranja em uma mesa.

Fonte: Autor.

Figura 36: Imagem processada de um laranja em uma mesa.

Fonte: Autor.

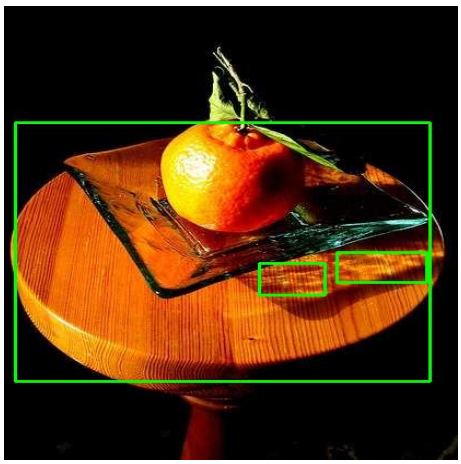


Figura 37: Imagem de laranja em uma mesa com caixa delimitadora utilizando OpenCV.

Fonte: Autor.

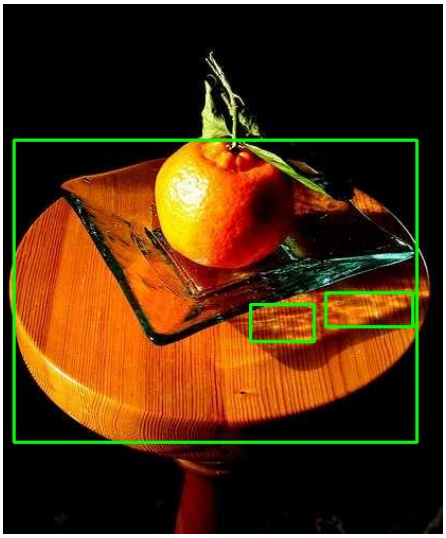


Figura 38: Imagem laranja em uma mesa com caixa delimitadora utilizando scikit-image.

Fonte: Autor.

4.2. Resultados

A métrica de avaliação utilizada foi através da fórmula de acurácia onde (Silva, 2016, P.137) VP é o verdadeiro positivo sendo a classificação correta do item da classe positiva, FP é chamado de Falso positivo, classificação incorreta do item da classe negativa, VN é o verdadeiro negativo sendo a classificação correta do item da classe negativa, e por fim FN sendo o falso negativo onde é classificação incorreta do item da classe negativa.

$$Acurácia = \frac{VP + VN}{VP + VN + FN + FP}$$

Formula de acurácia. Fonte (Mariano, 2021)

Foi escolhida esta fórmula por conta de a contagem ser um item binário, pertence a contagem ou não pertence, sendo VP os acertos na contagem, representado pela variável A, VN é contante no valor de 0 tendo esse valor pois o foco é em contar os objetos pertencentes a classe de laranjas e não em contar os objetos que são verdadeiramente da classe que não pertencem a classe laranja, FN a quantidade de laranjas que deveriam ser contadas representado pela variável F, e FP os erros, objetos que não deveriam ser contados ou caixa de detecção inválidas (sendo essas caixas selecionando objetos errados ou selecionando múltiplas laranjas) onde é a variável E. Assim a fórmula final da acurácia é o somatório de acertos para cada imagem em cada técnica (A), dividido pelo somatório de acertos somado com o somatório de laranjas que deveria ser contada (F) e o somatório de erros (E) multiplicado por 100 para obtermos a acurácia em porcentagem.

$$Acurácia = \frac{\Sigma A}{\Sigma A + \Sigma F + \Sigma E} x100$$

Formula final para cálculo da acurácia

Abaixo estão as duas tabelas comparativas entre a contagem manual das laranjas e 3 técnicas de contagem automática, onde, a coluna contagem manual contém o número de laranjas que foram contadas manualmente, *C* é o número de laranjas contadas pela técnica, *A* o número de objetos contados de forma correta, *F* o número de objetos que faltaram a ser contados e *E* que contém o número de erros, objetos contados de forma incorreta.

imagem número	contagem manual	OpenCV				scikit-image				YOLO v5			
		C	A	F	E	C	A	F	E	C	A	F	E
1	1	12	1	0	11	12	1	0	11	1	1	0	0
2	0	11	0	0	11	7	0	0	7	0	0	0	0
3	8	4	1	7	3	5	1	7	4	8	8	0	0
4	3	28	0	3	28	28	0	3	28	3	3	0	0
5	17	5	4	13	1	5	4	13	1	9	9	8	0
6	11	8	7	4	1	6	5	5	1	7	7	4	0
7	3	6	1	2	5	6	1	2	5	3	3	0	0
8	3	5	3	0	2	5	3	0	2	3	2	1	1
9	4	12	0	4	12	10	0	4	10	5	3	1	2
10	1	1	1	0	0	1	1	0	0	1	1	0	0
11	4	6	0	4	6	4	0	4	4	4	4	0	0
12	1	3	1	0	2	3	1	0	2	2	1	0	1
13	0	5	0	0	5	6	0	0	6	3	0	0	3
14	0	2	0	0	2	0	0	0	0	1	0	0	1

imagem número	contagem manual	OpenCV				scikit-image				YOLO v5			
		C	A	F	E	C	A	F	E	C	A	F	E
15	0	4	0	0	4	3	0	0	3	1	0	0	3
16	2	2	0	2	2	2	0	2	2	3	2	0	1
17	0	0	0	0	0	1	0	0	1	2	0	0	2
18	0	1	0	0	1	1	0	0	1	1	0	0	1
19	3	1	0	3	1	1	0	3	1	1	0	3	1
20	0	4	0	0	4	0	0	0	0	6	0	0	6
21	0	0	0	0	0	0	0	0	0	1	0	0	1
22	1	1	1	0	0	1	1	0	0	2	1	0	1
23	0	2	0	0	2	7	0	0	7	7	0	0	7
24	1	1	1	0	0	1	1	0	0	1	1	0	0
25	0	5	0	0	5	6	0	0	6	3	0	0	3
26	0	7	0	0	7	7	0	0	7	6	0	0	6
27	0	2	0	0	2	1	0	0	2	0	0	0	0
28	0	0	0	0	0	0	0	0	0	4	0	0	4
29	0	4	0	0	4	4	0	0	4	2	0	0	2
30	0	1	0	0	1	1	0	0	1	1	0	0	1
31	1	2	0	1	2	2	0	1	2	3	1	0	2
32	0	4	0	0	4	2	0	0	2	6	0	0	6
33	1	0	0	1	1	1	1	0	0	1	1	0	0
34	3	4	2	1	1	4	2	1	1	3	3	0	0
35	1	0	0	1	0	1	0	0	1	0	0	1	0
36	1	3	0	1	3	2	0	1	2	3	1	0	2
37	4	6	2	2	4	6	2	2	4	5	4	0	1
38	1	1	1	0	0	1	1	0	0	1	1	0	0
39	1	4	0	1	4	6	1	0	0	1	1	0	0

imagem número	contagem manual	OpenCV				scikit-image				YOLO v5			
		C	A	F	E	C	A	F	E	C	A	F	E
40	2	2	0	2	2	2	0	2	2	3	2	0	1
41	1	3	1	0	2	2	1	0	1	1	1	0	0
42	2	1	1	1	0	1	1	1	0	2	2	0	0
43	4	3	0	4	3	4	0	4	4	11	4	0	7
44	1	2	1	0	1	2	1	0	1	1	1	0	0
45	2	5	2	0	3	5	2	0	3	1	1	1	0
46	3	4	0	3	4	4	0	3	4	3	3	0	0
47	1	2	1	0	1	1	1	0	0	0	0	1	0
48	6	6	3	3	3	6	3	3	3	5	5	1	0
49	3	1	0	3	3	1	0	3	3	1	1	2	0
50	0	57	0	0	57	57	0	0	57	7	0	0	7
51	20	0	0	20	0	5	1	19	4	16	16	4	0
52	7	2	1	6	1	2	1	6	1	5	5	2	0
53	9	5	3	6	2	5	3	6	2	10	9	0	1
54	4	1	0	4	0	1	0	4	0	0	0	4	0
55	9	3	0	9	3	3	0	9	3	9	9	0	0

Tabela 1: Comparativo entre contagem original, OpenCV, scikit-image e YOLO v5

Fonte: Autor.

imagem número	contagem manual	YOLO v8				YOLO v9				YOLO v11			
		C	A	F	E	C	A	F	E	C	A	F	E
1	1	1	1	0	0	1	1	0	0	1	1	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0
3	8	8	8	0	0	8	8	0	0	8	8	0	0

imagem número	contagem manual	YOLO v8				YOLO v9				YOLO v11			
		C	A	F	E	C	A	F	E	C	A	F	E
4	3	3	3	0	0	2	2	1	0	3	3	0	0
5	17	9	9	8	0	10	10	7	0	8	8	9	0
6	11	7	7	4	0	6	6	5	0	4	4	7	0
7	3	3	3	0	0	3	3	0	0	3	3	0	0
8	3	2	2	1	0	3	3	0	0	2	2	0	0
9	4	4	3	1	1	5	4	0	1	7	4	0	3
10	1	1	1	0	0	1	1	0	0	1	1	0	0
11	4	4	4	0	0	4	4-	0	0	4	4	0	0
12	1	2	1	0	1	2	1	0	1	1	1	0	0
13	0	2	0	0	2	2	0	0	2	3	0	0	3
14	0	1	0	0	1	1	0	0	1	1	0	0	1
15	0	2	0	0	2	1	0	0	1	1	0	0	1
16	2	2	1	1	1	2	1	1	1	3	2	0	1
17	0	1	0	0	1	2	0	0	2	1	0	0	1
18	0	1	0	0	1	1	0	0	1	1	0	0	1
19	3	0	0	3	0	3	3	0	0	2	2	1	0
20	0	3	0	0	3	5	0	0	5	3	0	0	3
21	0	1	0	0	1	0	0	0	0	1	0	0	1
22	1	2	1	0	1	2	1	0	1	2	1	0	1
23	0	7	0	0	5	5	0	0	5	5	0	0	5
24	1	1	1	0	0	1	1	0	0	1	1	0	0

imagem número	contagem manual	YOLO v8				YOLO v9				YOLO v11			
		C	A	F	E	C	A	F	E	C	A	F	E
25	0	3	0	0	3	3	0	0	3	6	0	0	6
26	0	6	0	0	6	6	0	0	6	6	0	0	6
27	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	3	0	0	3	2	0	0	2	5	0	0	5
29	0	2	0	0	2	2	0	0	2	2	0	0	2
30	0	1	0	0	1	1	0	0	1	1	0	0	1
31	1	2	1	0	1	2	1	0	1	3	1	0	2
32	0	5	0	0	5	5	0	0	5	6	0	0	6
33	1	1	1	0	0	1	1	0	0	1	1	0	0
34	3	3	3	0	0	3	3	0	0	3	3	0	0
35	1	0	0	1	0	0	0	1	0	1	1	0	0
36	1	3	1	0	2	2	1	0	1	3	1	0	2
37	4	5	4	0	1	5	4	0	1	4	4	0	0
38	1	1	1	0	0	1	1	0	0	1	1	0	0
39	1	1	1	0	0	1	1	0	0	1	1	0	0
40	2	2	2	0	0	2	2	0	0	2	2	0	0
41	1	1	1	0	0	1	1	0	0	1	1	0	0
42	2	2	2	0	0	2	2	0	0	1	1	1	0
43	4	11	4	0	7	11	4	0	7	12	4	0	8
44	1	1	1	0	0	1	1	0	0	1	1	0	0
45	2	1	1	1	0	1	1	1	0	1	1	1	0
46	3	3	3	0	0	2	2	1	0	3	3	0	0
47	1	1	1	0	0	1	1	0	0	0	0	1	0
48	6	5	5	1	0	5	5	1	0	5	5	1	0
49	3	1	1	2	0	2	2	1	0	1	1	2	0

imagem número	contagem manual	YOLO v8				YOLO v9				YOLO v11			
		C	A	F	E	C	A	F	E	C	A	F	E
50	0	4	0	0	4	2	0	0	2	3	0	0	3
51	20	14	14	6	0	16	16	4	0	15	15	5	0
52	7	6	6	1	0	2	2	5	0	5	5	2	0
53	9	9	9	0	0	9	9	0	0	10	9	0	1
54	4	3	2	2	1	1	1	3	0	0	0	4	0
55	9	9	9	0	0	9	9	0	0	9	9	0	0

Tabela 2: Comparativo entre contagem original, YOLO v8, YOLO v9 e YOLO v11

Fonte: Autor.

Na tabela 3 contem comparação da acurácia de todas as técnicas, onde é possível visualizar que *YOLO* se saiu melhor que as técnicas tradicionais de visão computacional:

Técnica	OpenCV	scikit-image	YOLO v5	YOLO v8	YOLO v9	YOLO v11
Acurácia	10,37%	10,99%	52,23%	57,28%	58,08%	54,25%

Tabela 3: Comparativo entre a acurácia das diferentes técnicas abordadas em porcentagem

Fonte: Autor.

Foi demonstrada a superioridade dos modelos *YOLO* em relação aos algoritmos clássicos de visão computacional para a tarefa de contagem automática de laranjas em imagens. A acurácia dos modelos *YOLO* variou entre 52,23% (*YOLOv5*) e 58,08% (*YOLOv9*), enquanto os métodos de visão computacional alcançaram apenas cerca de 10% de acurácia. Essa diferença significativa ressalta o potencial da Inteligência Artificial em tarefas de contagem de objetos, especialmente quando lidamos com variações na iluminação, sombras e aglomeração de objetos.

No entanto, os resultados também expõem limitações importantes em ambas as abordagens, necessitando de melhorias significativas:

4.2.1. Problemas e Soluções para os Modelos *YOLO*:

- A inconsistência na acurácia entre as diferentes versões do *YOLO* (v5, v8, v9, v11): Essa inconsistência pode ser atribuída, em parte, às diferenças arquitetônicas e às melhorias implementadas em cada versão. Embora todas sejam desenvolvidas pela *Ultralytics* e compartilhem princípios fundamentais, cada iteração introduz

modificações significativas que afetam o desempenho em conjuntos de dados específicos.

- Conjunto de Dados: A qualidade e a diversidade do conjunto de dados são cruciais. A solução passa por:
    - Aumentar o tamanho do conjunto de dados: Um conjunto maior, com maior variação de cenários (iluminação, ângulos, etc.) e diferentes tipos de laranjas, ajudaria a generalizar melhor o modelo.
    - Melhorar a qualidade das anotações: Erros nas caixas durante a anotação impactam diretamente na performance.
    - Aumentar a variedade de imagens: Incluir imagens com outras frutas similares seria crucial para aumentar a capacidade do modelo de diferenciar laranjas de objetos semelhantes.
- Limiar de Confiança: A escolha do limiar de confiança é um parâmetro crucial. Valores muito baixos geram falsos positivos, enquanto valores muito altos levam a falsos negativos. A otimização desse limiar, ajudaria a encontrar o ponto de equilíbrio ótimo entre precisão e recall.

4.2.2. Problemas e Soluções para os Algoritmos de Visão Computacional:

- Segmentação de Cores: A dependência da segmentação de cores na escala HSV, embora útil, é sensível a variações de iluminação e sombras. A simples alteração dos limites inferior e superior não garante uma solução robusta. A proposta é explorar técnicas mais avançadas de segmentação, como:
    - Ajusta o limiar de forma dinâmica, levando em conta as variações de intensidade na imagem.
    - Explorar outros modelos de cores como que sejam menos sensíveis a variações de iluminação.
    - Utilizar combinações mais complexas de dilatação e erosão, ou operações morfológicas como abertura e fechamento, para melhor remover ruídos e melhorar a conectividade dos objetos.
- Objetos conectados: A detecção de objetos conectados, utilizada nos algoritmos de visão computacional, mostrou-se problemática devido à sua sensibilidade à proximidade entre as laranjas. Em situações onde as laranjas estão próximas ou em contato, o algoritmo tende a agrupá-las como um único objeto. Uma possível solução para mitigar esse problema seria ajustar o parâmetro de conectividade na função

connectedComponentsWithStats do *OpenCV* ou na função equivalente da biblioteca scikit-image. Ao aumentar o número de vizinhos considerados (de 4 para 8, por exemplo), o algoritmo poderia ser mais sensível a pequenas separações entre as laranjas, melhorando a distinção entre objetos individualmente. No entanto, isso pode levar a um aumento no número de falsos positivos, exigindo um ajuste cuidadoso desse parâmetro e, possivelmente, a integração de outras técnicas de pós-processamento para refinar a segmentação. Outra alternativa para superar essas limitações seria substituir completamente o método atual por outras técnicas mais sofisticadas como algoritmos baseados em aprendizado de máquina (além do *YOLO*) ou outros métodos tradicionais.

5. Conclusão

Este trabalho investigou a contagem automática de laranjas em imagens, comparando métodos de visão computacional (*OpenCV* e scikit-image) com modelos de detecção de objetos baseados em inteligência artificial (*YOLOv5*, *YOLOv8*, *YOLOv9* e *YOLOv11*). Embora os modelos *YOLO* tenham apresentado problemas como detecções múltiplas e baixa confiança em algumas instâncias, consistentemente superaram as abordagens de visão computacional clássica em termos de acurácia (52,23% para o *YOLOv5*, chegando a 58,08% para o *YOLOv9*). As técnicas de visão computacional, apesar de mais simples, apresentaram acurácias significativamente mais baixas (10,37% para *OpenCV* e 10,99% para *scikit-image*), demonstrando a superioridade da inteligência artificial nesta tarefa específica. A dificuldade em diferenciar laranjas de outros objetos esféricos similares (maçãs, peras) e a sensibilidade a variações de iluminação e sombras nos modelos *YOLO* destacam a importância da qualidade e diversidade do conjunto de dados de treinamento. A utilização de um conjunto de dados para teste contendo apenas imagens de laranjas, sem outras frutas, poderia melhorar significativamente o desempenho dos modelos *YOLO*, reduzindo a confusão entre classes. Por outro lado, o desempenho dos algoritmos de visão computacional pode ser otimizado através da alteração de parâmetros como os aumentar a quantidade de vezes que é executada dilatação e erosão, a faixa de cores utilizada na segmentação, e a exploração de técnicas de detecção de objetos além da análise de componentes conectados. Em conclusão, enquanto os modelos *YOLO* se mostraram mais eficazes para a contagem de laranjas em imagens, o aprimoramento dos conjuntos de dados e a otimização dos parâmetros em ambos os tipos de abordagem são cruciais para obter resultados mais robustos e precisos.

6. Referências bibliográficas

ULTRALYTCS. **YOLOv5**. Disponível em: https://github.com/ultralytics/YOLOv5. Acesso em: 24 março de 2024.

ULTRALYTCS. **ultralytics**. Disponível em: https://github.com/ultralytics/ultralytics. Acesso em: 24 março de 2024.

REDMON, Joseph. **darknet**. Disponível em: https://github.com/pjreddie/darknet. Acesso em: 24 março de 2024.

ULTRALYTCS. **Ultralytics YOLO Docs**. Disponível em: https://docs.ultralytics.com/pt. Acesso em: 24 março de 2024.

REDMON, Joseph; FARHADI, Ali . **YOLO: Real-Time Object Detection**. Disponível em: https://pjreddie.com/darknet/YOLO/. Acesso em: 24 março de 2024.

ALMEIDA, Pedro. Detectando objetos com YOLO e Darknet. Disponível em: https://medium.com/@argonalyst/detectando-objetos-com-YOLO-e-darknet-cdb9a17fbc3f. Acesso em: 24 março de 2024.

AI_Pioneer. **Object Detection with YOLO and OpenCV: A Practical Guide**. Disponível em: https://medium.com/@tejasdalvi927/object-detection-with-YOLO-and-OpenCV-a-practical-guide-cf7773481d11. Acesso em: 24 março de 2024.

ASHWATH, Balraj. **Apple2orange Dataset**. Disponível em: https://www.kaggle.com/datasets/balraj98/apple2orange-dataset. Acesso em: 25 maio de 2024.

IMAGES.CV**. 1.4K Orange Labeled Image Dataset**. Disponível em: https://images.cv/dataset/orange-image-classification-dataset. Acesso em: 25 maio de 2024.

ROBOFLOW. **Our Company**. Disponível em: https://roboflow.com/about. Acesso em: 25 maio de 2024.

FIRIGATO, João Otávio. **Como realizar uma estimativa de contagem de plantas em uma imagem de Drone usando Python?**. Youtube. Disponível em: https://www.youtube.com/watch?v=zw_T28zF_rQ. Acesso em: 04 de junho de 2024

FIRIGATO, João Otávio. **Usando algoritmos de Visão Computacional para detecção e contagem de plantas**. Youtube. Disponível em: https://colab.research.google.com/drive/18WrfHLeLEhYrhWxHa6AaZJJhNDB4RJOa?usp=sharing. Acesso em: 04 de junho de 2024.

MOTA, Suzana. **Como identificar cores no mundo real utilizando Visão Computacional?.** Disponível em: https://suzana-svm.medium.com/vis%C3%A3o-computacional-

detec%C3%A7%C3%A3o-de-cores-em-tempo-real-utilizando-python-e-OpenCV-a466444d40e. Acesso em: 16 de outubro de 2024

ROSEBROCK, Adrian. **OpenCV Connected Component Labeling and Analysis**. Disponível em: https://pyimagesearch.com/2021/02/22/OpenCV-connected-component-labeling-and-analysis/. Acesso em: 16 de outubro de 2024.

SCIKIT-IMAGE. **scikit-image.measure**. Disponível em: https://scikit-image.org/docs/stable/api/scikit-image.measure.html#scikit-image.measure.label. Acesso em: 16 de outubro de 2024.

SCIKIT-IMAGE. **Measure region properties**. Disponível em: https://scikit-image.org/docs/stable/auto_examples/segmentation/plot_regionprops.html. Acesso em: 16 de outubro de 2024.

REDMON, Joseph; DIVVALA, Santosh; GIRSHICK, Ross; FARHADI, Ali. **You Only Look Once: Unified, Real-Time Object Detection**. 2016. Disponível em: https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Redmon_You_Only_Look_CVPR_2016_paper.pdf. Acesso em: 23 de outubro de 2024.

SILVA, Leandro Augusto da; PERES, Sarajane M.; BOSCARIOLI, Clodis. **Introdução à Mineração de Dados - Com Aplicações em R**. Rio de Janeiro: GEN LTC, 2016. *E-book.* p.137. ISBN 9788595155473. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9788595155473/. Acesso em: 15 de novembro 2024.

GOOGLE**. Conjuntos de dados: características dos dados**. Disponível em: https://developers.google.com/machine-learning/crash-course/overfitting/data-characteristics?hl=pt-br. Acesso em: Acesso em: 20 de novembro de 2024.

GOMES, Dennis dos Santos. Inteligência Artificial: conceitos e aplicações. **Revista Olhar Científico**, v. 1, n. 2, p. 234-246, 2010. Disponível em: https://d1wqtxts1xzle7.cloudfront.net/51841234/49-148-1-PB-libre.pdf?1487358168=&response-content-disposition=inline%3B+filename%3DInteligencia_Artificial_Conceitos_e_Apli.pdf&Expires=1733498673&Signature=Tszl4jG~uJy-jI1Ge1YQViCzhd03pE4aaug3cLIiuFpy5vwAIEwqXVfhERmx7fwXV9vWX5cu6soqZ~-mws8~LMXXSie~rKCKbZ7kBffTrQS3RZtS75ArELE6eOBUAbWgTgNhR9qcpNBefAkL7ijknFKChbm6Hw0Wu1EgPmbDdbib-HVHGLhYQ9V288MIRR2pUH-mpmMT4tz~rlLGlpYOSK0AMV-

ZTXhbiy3RoviV8U3ldXAoL5BFrWo9sKakgvXDyNUuO4NByYv8EHxEzO-AHq0~M0AiyZnTpfN3VTxOB4U8YXvZgJS59i~cSYh3UCNeL1B8zhJrYi5JWfEKoVCa3g__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA. Acesso em: 01 de dezembro de 2024.

PYTHON. **What is Python? Executive Summary**. Disponível em: https://www.python.org/doc/essays/blurb/. Acesso em: 20 de novembro de 2024.

ULTRALYTICS. **Isto é Ultralytics**. Disponível em: https://www.ultralytics.com/pt/about. Acesso em: 20 de novembro de 2024.

OPENCV. **About**. Disponível em: https://OpenCV.org/about/. Acesso em: 20 de novembro de 2024.

SCIKIT-IMAGE. **Image processing in Python**. Disponível em: https://scikit-image.org/. Acesso em: 20 de novembro de 2024.

DOS SANTOS, Pedro Antonio Trindade. Contagem de laranjas. Disponível em: https://github.com/prodesert22/contagem_laranjas. Acesso em: 20 de novembro de 2024.

ZHU, Jun-Yan; PARK, Taesung; ISOLA, Phillip; EFROS, Alexei A. *Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks*. In: **INTERNATIONAL CONFERENCE ON COMPUTER VISION**, 2017, Venice, Italy. Proceedings [...]. IEEE, 2017. p. 2242-2251. DOI: 10.1109/ICCV.2017.244. Disponível em: https://arxiv.org/pdf/1703.10593. Acesso em: 4 de dezembro de 2024.