**Source Codes:**

**detect.py:**

import argparse

import csv

import os

import platform

import sys

from pathlib import Path

import torch

FILE = Path(\_\_file\_\_).resolve()

ROOT = FILE.parents[0] # YOLOv5 root directory

if str(ROOT) not in sys.path:

sys.path.append(str(ROOT)) # add ROOT to PATH

ROOT = Path(os.path.relpath(ROOT, Path.cwd())) # relative

from ultralytics.utils.plotting import Annotator, colors, save\_one\_box

from models.common import DetectMultiBackend

from utils.dataloaders import IMG\_FORMATS, VID\_FORMATS, LoadImages, LoadScreenshots, LoadStreams

from utils.general import (

LOGGER,

Profile,

check\_file,

check\_img\_size,

check\_imshow,

check\_requirements,

colorstr,

cv2,

increment\_path,

non\_max\_suppression,

print\_args,

scale\_boxes,

strip\_optimizer,

xyxy2xywh,

)

from utils.torch\_utils import select\_device, smart\_inference\_mode

@smart\_inference\_mode()

def run(

weights=ROOT / "best.pt", # model path or triton URL

source=ROOT / "scanned\_imgs", # file/dir/URL/glob/screen/0(webcam)

data=ROOT / "data.yaml", # dataset.yaml path

imgsz=(416, 416), # inference size (height, width)

conf\_thres=0.70, # confidence threshold

iou\_thres=0.45, # NMS IOU threshold

max\_det=1000, # maximum detections per image

device="", # cuda device, i.e. 0 or 0,1,2,3 or cpu

view\_img=False, # show results

save\_txt=False, # save results to \*.txt

save\_csv=False, # save results in CSV format

save\_conf=False, # save confidences in --save-txt labels

save\_crop=False, # save cropped prediction boxes

nosave=True, # do not save images/videos

classes=None, # filter by class: --class 0, or --class 0 2 3

agnostic\_nms=False, # class-agnostic NMS

augment=False, # augmented inference

visualize=False, # visualize features

update=False, # update all models

project=ROOT / "runs/detect", # save results to project/name

name="exp", # save results to project/name

exist\_ok=False, # existing project/name ok, do not increment

line\_thickness=3, # bounding box thickness (pixels)

hide\_labels=False, # hide labels

hide\_conf=False, # hide confidences

half=False, # use FP16 half-precision inference

dnn=False, # use OpenCV DNN for ONNX inference

vid\_stride=1, # video frame-rate stride

):

source = str(source)

save\_img = not nosave and not source.endswith(".txt") # save inference images

is\_file = Path(source).suffix[1:] in (IMG\_FORMATS + VID\_FORMATS)

is\_url = source.lower().startswith(("rtsp://", "rtmp://", "http://", "https://"))

webcam = source.isnumeric() or source.endswith(".streams") or (is\_url and not is\_file)

screenshot = source.lower().startswith("screen")

if is\_url and is\_file:

source = check\_file(source) # download

# Directories

save\_dir = increment\_path(Path(project) / name, exist\_ok=exist\_ok) # increment run

(save\_dir / "labels" if save\_txt else save\_dir).mkdir(parents=True, exist\_ok=True) # make dir

# Load model

device = select\_device(device)

model = DetectMultiBackend(weights, device=device, dnn=dnn, data=data, fp16=half)

stride, names, pt = model.stride, model.names, model.pt

imgsz = check\_img\_size(imgsz, s=stride) # check image size

# Dataloader

bs = 1 # batch\_size

if webcam:

view\_img = check\_imshow(warn=True)

dataset = LoadStreams(source, img\_size=imgsz, stride=stride, auto=pt, vid\_stride=vid\_stride)

bs = len(dataset)

elif screenshot:

dataset = LoadScreenshots(source, img\_size=imgsz, stride=stride, auto=pt)

else:

dataset = LoadImages(source, img\_size=imgsz, stride=stride, auto=pt, vid\_stride=vid\_stride)

vid\_path, vid\_writer = [None] \* bs, [None] \* bs

# Run inference

model.warmup(imgsz=(1 if pt or model.triton else bs, 3, \*imgsz)) # warmup

seen, windows, dt = 0, [], (Profile(device=device), Profile(device=device), Profile(device=device))

for path, im, im0s, vid\_cap, s in dataset:

with dt[0]:

im = torch.from\_numpy(im).to(model.device)

im = im.half() if model.fp16 else im.float() # uint8 to fp16/32

im /= 255 # 0 - 255 to 0.0 - 1.0

if len(im.shape) == 3:

im = im[None] # expand for batch dim

if model.xml and im.shape[0] > 1:

ims = torch.chunk(im, im.shape[0], 0)

# Inference

with dt[1]:

visualize = increment\_path(save\_dir / Path(path).stem, mkdir=True) if visualize else False

if model.xml and im.shape[0] > 1:

pred = None

for image in ims:

if pred is None:

pred = model(image, augment=augment, visualize=visualize).unsqueeze(0)

else:

pred = torch.cat((pred, model(image, augment=augment, visualize=visualize).unsqueeze(0)), dim=0)

pred = [pred, None]

else:

pred = model(im, augment=augment, visualize=visualize)

# NMS

with dt[2]:

pred = non\_max\_suppression(pred, conf\_thres, iou\_thres, classes, agnostic\_nms, max\_det=max\_det)

# Second-stage classifier (optional)

# pred = utils.general.apply\_classifier(pred, classifier\_model, im, im0s)

# Define the path for the CSV file

csv\_path = save\_dir / "predictions.csv"

# Create or append to the CSV file

def write\_to\_csv(image\_name, prediction, confidence):

"""Writes prediction data for an image to a CSV file, appending if the file exists."""

data = {"Image Name": image\_name, "Prediction": prediction, "Confidence": confidence}

with open(csv\_path, mode="a", newline="") as f:

writer = csv.DictWriter(f, fieldnames=data.keys())

if not csv\_path.is\_file():

writer.writeheader()

writer.writerow(data)

# Process predictions

for i, det in enumerate(pred): # per image

seen += 1

if webcam: # batch\_size >= 1

p, im0, frame = path[i], im0s[i].copy(), dataset.count

s += f"{i}: "

else:

p, im0, frame = path, im0s.copy(), getattr(dataset, "frame", 0)

p = Path(p) # to Path

save\_path = str(save\_dir / p.name) # im.jpg

txt\_path = str(save\_dir / "labels" / p.stem) + ("" if dataset.mode == "image" else f"\_{frame}") # im.txt

s += "%gx%g " % im.shape[2:] # print string

gn = torch.tensor(im0.shape)[[1, 0, 1, 0]] # normalization gain whwh

imc = im0.copy() if save\_crop else im0 # for save\_crop

annotator = Annotator(im0, line\_width=line\_thickness, example=str(names))

if len(det):

# Rescale boxes from img\_size to im0 size

det[:, :4] = scale\_boxes(im.shape[2:], det[:, :4], im0.shape).round()

# Print results

for c in det[:, 5].unique():

n = (det[:, 5] == c).sum() # detections per class

s += f"{n} {names[int(c)]}{'s' \* (n > 1)}, " # add to string

# Write results

for \*xyxy, conf, cls in reversed(det):

c = int(cls) # integer class

label = names[c] if hide\_conf else f"{names[c]}"

confidence = float(conf)

confidence\_str = f"{confidence:.2f}"

if save\_csv:

write\_to\_csv(p.name, label, confidence\_str)

if save\_txt: # Write to file

xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / gn).view(-1).tolist() # normalized xywh

line = (cls, \*xywh, conf) if save\_conf else (cls, \*xywh) # label format

with open(f"{txt\_path}.txt", "a") as f:

f.write(("%g " \* len(line)).rstrip() % line + "\n")

if save\_img or save\_crop or view\_img: # Add bbox to image

c = int(cls) # integer class

label = None if hide\_labels else (names[c] if hide\_conf else f"{names[c]} {conf:.2f}")

annotator.box\_label(xyxy, label, color=colors(c, True))

if save\_crop:

save\_one\_box(xyxy, imc, file=save\_dir / "crops" / names[c] / f"{p.stem}.jpg", BGR=True)

# Stream results

im0 = annotator.result()

if view\_img:

if platform.system() == "Linux" and p not in windows:

windows.append(p)

cv2.namedWindow(str(p), cv2.WINDOW\_NORMAL | cv2.WINDOW\_KEEPRATIO) # allow window resize (Linux)

cv2.resizeWindow(str(p), im0.shape[1], im0.shape[0])

cv2.imshow(str(p), im0)

cv2.waitKey(0) # Wait indefinitely until a key is pressed

# Save results (image with detections)

if save\_img:

if dataset.mode == "image":

cv2.imwrite(save\_path, im0)

else: # 'video' or 'stream'

if vid\_path[i] != save\_path: # new video

vid\_path[i] = save\_path

if isinstance(vid\_writer[i], cv2.VideoWriter):

vid\_writer[i].release() # release previous video writer

if vid\_cap: # video

fps = vid\_cap.get(cv2.CAP\_PROP\_FPS)

w = int(vid\_cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))

h = int(vid\_cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))

else: # stream

fps, w, h = 30, im0.shape[1], im0.shape[0]

save\_path = str(Path(save\_path).with\_suffix(".mp4")) # force \*.mp4 suffix on results videos

vid\_writer[i] = cv2.VideoWriter(save\_path, cv2.VideoWriter\_fourcc(\*"mp4v"), fps, (w, h))

vid\_writer[i].write(im0)

# Print time (inference-only)

LOGGER.info(f"{s}{'' if len(det) else '(no detections), '}{dt[1].dt \* 1E3:.1f}ms")

import yaml

# Load data.yaml

with open('data.yaml', 'r') as file:

data = yaml.safe\_load(file)

# Get the names of all classes

class\_names = data.get('names', [])

print(f"- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -")

print("Class names:", class\_names)

# Initialize counts for good beans and total beans

good\_bean\_count = 0

total\_bean\_count = 0

# Iterate through predictions to count good beans

for det in pred:

for detection in det:

total\_bean\_count += 1

class\_index = int(detection[5])

print("Class index:", class\_index)

print("Class name:", class\_names[class\_index])

if class\_names[class\_index] == 'good-bean':

good\_bean\_count += 1

# Calculate the percentage of good beans relative to the total count of all beans

percentage\_good = (good\_bean\_count / total\_bean\_count) \* 100 if total\_bean\_count > 0 else 0

# Calculate the percentage of defective beans

percentage\_defective = 100 - percentage\_good

# Determine the resulting class based on the percentage of defective beans

resulting\_class = 'class A: high grade' if percentage\_defective < 9 else 'class B: low grade'

# Print the resulting class and the total percentage of defective beans

print(f"Resulting class: {resulting\_class}")

print(f"Total percentage of good beans: {percentage\_good:.2f}%")

print(f"Total percentage of defective beans: {percentage\_defective:.2f}%")

# Print results

t = tuple(x.t / seen \* 1e3 for x in dt) # speeds per image

LOGGER.info(f"Speed: %.1fms pre-process, %.1fms inference, %.1fms NMS per image at shape {(1, 3, \*imgsz)}" % t)

if save\_txt or save\_img:

s = f"\n{len(list(save\_dir.glob('labels/\*.txt')))} labels saved to {save\_dir / 'labels'}" if save\_txt else ""

LOGGER.info(f"Results saved to {colorstr('bold', save\_dir)}{s}")

if update:

strip\_optimizer(weights[0]) # update model (to fix SourceChangeWarning)

def parse\_opt():

"""Parses command-line arguments for YOLOv5 detection, setting inference options and model configurations."""

parser = argparse.ArgumentParser()

parser.add\_argument("--weights", nargs="+", type=str, default=ROOT / "best.pt", help="model path or triton URL")

parser.add\_argument("--source", type=str, default=ROOT / "scanned\_imgs", help="file/dir/URL/glob/screen/0(webcam)")

parser.add\_argument("--data", type=str, default=ROOT / "data.yaml", help="(optional) dataset.yaml path")

parser.add\_argument("--imgsz", "--img", "--img-size", nargs="+", type=int, default=[416], help="inference size h,w")

parser.add\_argument("--conf-thres", type=float, default=0.25, help="confidence threshold")

parser.add\_argument("--iou-thres", type=float, default=0.45, help="NMS IoU threshold")

parser.add\_argument("--max-det", type=int, default=1000, help="maximum detections per image")

parser.add\_argument("--device", default="", help="cuda device, i.e. 0 or 0,1,2,3 or cpu")

parser.add\_argument("--view-img", action="store\_true", help="show results")

parser.add\_argument("--save-txt", action="store\_true", help="save results to \*.txt")

parser.add\_argument("--save-csv", action="store\_true", help="save results in CSV format")

parser.add\_argument("--save-conf", action="store\_true", help="save confidences in --save-txt labels")

parser.add\_argument("--save-crop", action="store\_true", help="save cropped prediction boxes")

parser.add\_argument("--nosave", action="store\_true", help="do not save images/videos")

parser.add\_argument("--classes", nargs="+", type=int, help="filter by class: --classes 0, or --classes 0 2 3")

parser.add\_argument("--agnostic-nms", action="store\_true", help="class-agnostic NMS")

parser.add\_argument("--augment", action="store\_true", help="augmented inference")

parser.add\_argument("--visualize", action="store\_true", help="visualize features")

parser.add\_argument("--update", action="store\_true", help="update all models")

parser.add\_argument("--project", default=ROOT / "runs/detect", help="save results to project/name")

parser.add\_argument("--name", default="exp", help="save results to project/name")

parser.add\_argument("--exist-ok", action="store\_true", help="existing project/name ok, do not increment")

parser.add\_argument("--line-thickness", default=3, type=int, help="bounding box thickness (pixels)")

parser.add\_argument("--hide-labels", default=False, action="store\_true", help="hide labels")

parser.add\_argument("--hide-conf", default=False, action="store\_true", help="hide confidences")

parser.add\_argument("--half", action="store\_true", help="use FP16 half-precision inference")

parser.add\_argument("--dnn", action="store\_true", help="use OpenCV DNN for ONNX inference")

parser.add\_argument("--vid-stride", type=int, default=1, help="video frame-rate stride")

opt = parser.parse\_args()

opt.imgsz \*= 2 if len(opt.imgsz) == 1 else 1 # expand

print\_args(vars(opt))

return opt

def main(opt):

"""Executes YOLOv5 model inference with given options, checking requirements before running the model."""

check\_requirements(ROOT / "requirements.txt", exclude=("tensorboard", "thop"))

run(\*\*vars(opt))

if \_\_name\_\_ == "\_\_main\_\_":

opt = parse\_opt()

main(opt)

**scan.py:**

import os

import subprocess

import logging

import tkinter as tk

from tkinter import ttk

def scan\_images(folder\_path, weight\_file, data\_file):

output\_text = "" # Initialize variable to store output

high\_grade = False # Flag to indicate high grade

# Check if folder exists

if not os.path.exists(folder\_path):

logging.error("Folder '{}' does not exist.".format(folder\_path))

return output\_text

total\_defective\_bean\_percentage = 0

total\_good\_bean\_percentage = 0

num\_images = 0

# Iterate through all files in the folder

for filename in os.listdir(folder\_path):

if filename.endswith(".jpg") or filename.endswith(".png"): # Check if the file is an image

image\_path = os.path.join(folder\_path, filename)

# Run the detection command for the image

command = [

"python",

"detect.py",

"--weights",

weight\_file,

"--img",

"416",

"--source",

image\_path,

"--nosave",

"--view-img",

"--data",

data\_file

]

try:

result = subprocess.run(command, capture\_output=True, text=True, check=True)

logging.info("Detection completed for image: {}".format(filename))

# Parse detection results

lines = result.stdout.strip().split('\n')

for line in lines:

output\_text += line + "\n" # Collect output

if line.startswith("Total percentage of defective beans:"):

percentage\_defective = float(line.split(":")[-1].strip()[:-1])

total\_defective\_bean\_percentage += percentage\_defective

num\_images += 1

elif line.startswith("Total percentage of good beans:"):

percentage\_good = float(line.split(":")[-1].strip()[:-1])

total\_good\_bean\_percentage += percentage\_good

except subprocess.CalledProcessError as e:

logging.error("Error running detection for image {}: {}".format(filename, e))

# Calculate average percentage of defective beans

if num\_images > 0:

average\_defective\_percentage = total\_defective\_bean\_percentage / num\_images

else:

average\_defective\_percentage = 0

# Determine grade

grade = "HIGH GRADE" if average\_defective\_percentage < 9 else "LOW GRADE"

# Save output to a file

with open('output.txt', 'w') as file:

file.write(output\_text)

return output\_text, grade

def run\_scan\_and\_display\_output():

output\_text, grade = scan\_images("scanned\_imgs", "best.pt", "data.yaml")

display\_output\_in\_tkinter(output\_text, grade)

def display\_output\_in\_tkinter(output\_text, grade):

# Create a tkinter window

root = tk.Tk()

root.title("Scan Output")

root.geometry("400x500")

# Create a label for grade

grade\_label = tk.Label(root, text=grade, font=("Arial", 24, "bold"), fg="green" if grade == "HIGH GRADE" else "red")

grade\_label.pack(pady=10)

# Create a text widget to display output

output\_text\_widget = tk.Text(root, wrap="word", font=("Arial", 10))

output\_text\_widget.pack(expand=True, fill="both", padx=10, pady=(0, 10))

# Highlight good bean percentage in blue and defective bean percentage in red

for line in output\_text.split("\n"):

if line.startswith("Total percentage of good beans:"):

good\_percentage = float(line.split(":")[-1].strip()[:-1])

formatted\_line = f"{line.split(':')[0]}: {good\_percentage:.2f}%\n"

output\_text\_widget.insert(tk.END, formatted\_line, "blue")

elif line.startswith("Total percentage of defective beans:"):

defective\_percentage = float(line.split(":")[-1].strip()[:-1])

formatted\_line = f"{line.split(':')[0]}: {defective\_percentage:.2f}%\n"

output\_text\_widget.insert(tk.END, formatted\_line, "red")

else:

output\_text\_widget.insert(tk.END, line + "\n")

# Configure tag styles

output\_text\_widget.tag\_configure("blue", foreground="blue")

output\_text\_widget.tag\_configure("red", foreground="red")

root.mainloop()

# Run the scanning process and display output in tkinter window

run\_scan\_and\_display\_output()

**percentage.py:**

import tkinter as tk

from datetime import datetime

def count\_good\_and\_defective\_indices(file\_path):

total\_class\_indices = 0

good\_bean\_count = 0

with open(file\_path, 'r') as file:

lines = file.readlines()

for line in lines:

if line.startswith("Class index:"):

total\_class\_indices += 1

class\_index = int(line.split(":")[-1].strip())

if class\_index == 1: # Assuming 1 represents 'good-bean'

good\_bean\_count += 1

defective\_count = total\_class\_indices - good\_bean\_count

return total\_class\_indices, good\_bean\_count, defective\_count

def display\_output():

file\_path = 'output.txt'

total\_class\_indices, good\_bean\_count, defective\_count = count\_good\_and\_defective\_indices(file\_path)

# Calculate the percentage of 'good-bean' indices relative to total class indices

good\_bean\_percentage = (good\_bean\_count / total\_class\_indices) \* 100

# Calculate the percentage of 'defective-bean' indices relative to total class indices

defective\_percentage = (defective\_count / total\_class\_indices) \* 100

# Create a Tkinter window

window = tk.Tk()

window.title("Bean Classification Result")

# Calculate the screen width and height

screen\_width = window.winfo\_screenwidth()

screen\_height = window.winfo\_screenheight()

# Set the window size and position it at the center of the screen

window\_width = 400 # Adjust as needed

window\_height = 300 # Adjust as needed

x = (screen\_width - window\_width) // 2

y = (screen\_height - window\_height) // 2

window.geometry(f"{window\_width}x{window\_height}+{x}+{y}")

# Create a frame for the receipt-style design

receipt\_frame = tk.Frame(window, padx=20, pady=20)

receipt\_frame.pack()

# Add date and time on the top

date\_time\_label = tk.Label(receipt\_frame, text=datetime.now().strftime("%Y-%m-%d %H:%M:%S"), font=("Arial", 12, "bold"))

date\_time\_label.pack()

# Add caption "RESULT"

result\_label = tk.Label(receipt\_frame, text="RESULT", font=("Arial", 16, "bold"), pady=10)

result\_label.pack()

# Display the output in labels with larger text

tk.Label(receipt\_frame, text="Total Bean Count: {}".format(total\_class\_indices), font=("Arial", 14)).pack()

tk.Label(receipt\_frame, text="Percentage of Good Beans: {:.2f}%".format(good\_bean\_percentage), font=("Arial", 14), fg="blue").pack()

tk.Label(receipt\_frame, text="Percentage of Defective Beans: {:.2f}%".format(defective\_percentage), font=("Arial", 14), fg="red").pack()

# Add grading

if defective\_percentage < 9:

grading\_label = tk.Label(receipt\_frame, text="GRADE 1", font=("Arial", 14, "bold"), fg="green")

else:

grading\_label = tk.Label(receipt\_frame, text="GRADE 2", font=("Arial", 14, "bold"), fg="purple")

grading\_label.pack()

window.mainloop()

# Call the display\_output function to show the results in a Tkinter window

display\_output()

**capture.py:**

import os

import cv2

import tkinter as tk

from PIL import Image, ImageTk

class ImageCaptureApp:

def \_\_init\_\_(self):

self.num\_images = 0

self.image\_count = 0

self.root = tk.Tk()

self.root.title("Image Capture")

self.num\_images\_label = tk.Label(self.root, text="Enter the number of images to capture:")

self.num\_images\_label.pack()

self.num\_images\_entry = tk.Entry(self.root)

self.num\_images\_entry.pack()

self.proceed\_button = tk.Button(self.root, text="Proceed", command=self.proceed\_capture)

self.proceed\_button.pack()

self.root.protocol("WM\_DELETE\_WINDOW", self.on\_close)

def proceed\_capture(self):

try:

self.num\_images = int(self.num\_images\_entry.get())

self.root.destroy() # Close the initial tkinter window

self.capture\_app = CaptureApp(self.num\_images) # Start the CaptureApp with the specified number of images

self.capture\_app.root.mainloop()

except ValueError:

print("Please enter a valid number of images.")

def on\_close(self):

self.root.destroy()

class CaptureApp:

def \_\_init\_\_(self, num\_images):

self.num\_images = num\_images

self.image\_count = 0

self.cap = cv2.VideoCapture(0)

if not self.cap.isOpened():

print("Error: Unable to access webcam.")

return

self.root = tk.Tk()

self.root.title("Image Capture")

self.label = tk.Label(self.root)

self.label.pack()

self.capture\_button = tk.Button(self.root, text="Capture", command=self.capture\_image)

self.capture\_button.pack()

self.root.protocol("WM\_DELETE\_WINDOW", self.on\_close)

self.show\_frame()

def show\_frame(self):

ret, frame = self.cap.read()

if ret:

frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

image = Image.fromarray(frame)

photo = ImageTk.PhotoImage(image=image)

self.label.config(image=photo)

self.label.image = photo

self.root.after(10, self.show\_frame)

def capture\_image(self):

if self.image\_count < self.num\_images:

self.image\_count += 1

print(f"Image {self.image\_count} captured.")

ret, frame = self.cap.read()

if ret:

folder\_path = "scanned\_imgs"

if not os.path.exists(folder\_path):

os.makedirs(folder\_path)

cv2.imwrite(os.path.join(folder\_path, f"scan{self.image\_count}.jpg"), frame)

if self.image\_count >= self.num\_images:

print("All images captured.")

self.root.destroy()

def on\_close(self):

self.cap.release()

self.root.destroy()

def main():

app = ImageCaptureApp()

app.root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**dashboard.py:**

import os

import tkinter as tk

from tkinter import ttk, messagebox

import subprocess

import threading

def browse\_files():

folder\_path = "scanned\_imgs"

if os.path.exists(folder\_path):

subprocess.Popen(["xdg-open", folder\_path])

else:

messagebox.showerror("Folder Not Found", "The 'scanned\_imgs' folder does not exist.")

def open\_saved\_results\_folder():

folder\_path = "Percentage\_Results"

if os.path.exists(folder\_path):

subprocess.Popen(["xdg-open", folder\_path])

else:

messagebox.showerror("Folder Not Found", "The 'Percentage\_Results' folder does not exist.")

def clear\_memory():

folder\_path = "scanned\_imgs"

if os.path.exists(folder\_path):

image\_files = [f for f in os.listdir(folder\_path) if f.endswith('.jpg')]

if not image\_files:

messagebox.showinfo("No Images", "There are no images to delete.")

return

confirmation = messagebox.askyesno("Confirmation", f"Are you sure you want to delete {len(image\_files)} image(s)?")

if confirmation:

for image\_file in image\_files:

file\_path = os.path.join(folder\_path, image\_file)

os.remove(file\_path)

messagebox.showinfo("Deletion Complete", "All image files have been deleted.")

else:

messagebox.showerror("Folder Not Found", "The 'scanned\_imgs' folder does not exist.")

def open\_manual():

confirmation = messagebox.askyesno("Confirmation", "Do you want to open the manual?")

if confirmation:

manual\_path = "manual.pdf" # Replace with the actual path to your manual.pdf

if os.path.exists(manual\_path):

subprocess.Popen(["xdg-open", manual\_path])

else:

messagebox.showerror("File Not Found", "The 'manual.pdf' file does not exist.")

def dashboard():

global root

root = tk.Tk()

root.title("Admin Dashboard")

root.attributes('-fullscreen', True)

sidebar = tk.Frame(root, bg="#333", width=200)

sidebar.pack(side=tk.LEFT, fill=tk.Y)

style = ttk.Style()

style.theme\_use('clam') # Use 'clam' theme which supports more customization

style.configure("TButton", font=("Arial", 12), padding=25)

style.configure("CapturedImages.TButton", background="#FF5733", foreground="white")

style.configure("SavedResults.TButton", background="#FF5733", foreground="white")

style.configure("Manual.TButton", background="#FF5733", foreground="white")

style.configure("Logout.TButton", background="#FF0000", foreground="black")

style.configure("ClearMemory.TButton", background="#FF5733", foreground="white")

style.configure("Refresh.TButton", background="#00FF00", foreground="black")

style.configure("Capture.TButton", background="#B76A24", foreground="white", font=("Arial", 18))

style.configure("Scan.TButton", background="#B76A24", foreground="white", font=("Arial", 18))

style.map("CapturedImages.TButton", foreground=[("active", "black")])

style.map("SavedResults.TButton", foreground=[("active", "black")])

style.map("Manual.TButton", foreground=[("active", "black")])

style.map("ClearMemory.TButton", foreground=[("active", "black")])

style.map("Capture.TButton", background=[("active", "#FB8518")])

style.map("Scan.TButton", background=[("active", "#FB8518")])

# Fix for button background colors in 'clam' theme

style.element\_create("Custom.TButton", "from", "clam")

style.layout("TButton", [

("Button.border", {"children": [

("Button.padding", {"children": [

("Button.label", {"side": "left", "expand": 1})

]})

]})

])

links = ["Captured Images", "Saved Results", "Manual", "Clear Memory", "Refresh", "Logout"]

for link in links:

if link == "Logout":

link\_button = ttk.Button(sidebar, text=link, style="Logout.TButton", width=20, command=logout)

elif link == "Captured Images":

link\_button = ttk.Button(sidebar, text=link, style="CapturedImages.TButton", width=20, command=browse\_files)

elif link == "Saved Results":

link\_button = ttk.Button(sidebar, text=link, style="SavedResults.TButton", width=20, command=open\_saved\_results\_folder)

elif link == "Clear Memory":

link\_button = ttk.Button(sidebar, text=link, style="ClearMemory.TButton", width=20, command=clear\_memory)

elif link == "Manual":

link\_button = ttk.Button(sidebar, text=link, style="Manual.TButton", width=20, command=open\_manual)

elif link == "Refresh":

link\_button = ttk.Button(sidebar, text=link, style="Refresh.TButton", width=20, command=refresh)

else:

link\_button = ttk.Button(sidebar, text=link, style="TButton", width=20)

link\_button.pack(pady=5, padx=10, anchor="w")

content = tk.Frame(root, bg="#EDC6A0") # Set background color to light gradient orange

content.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

welcome\_label = tk.Label(content, text="Cacao Bean Classifier", font=("Arial", 40, 'bold'), background="#EDC6A0", foreground="#583311")

welcome\_label.pack(pady=90)

global capture\_button

capture\_button = ttk.Button(content, text="Capture", style="Capture.TButton", command=capture\_webcam)

capture\_button.config(width=20, padding=(20, 20))

capture\_button.pack(pady=20)

global scan\_button

scan\_button = ttk.Button(content, text="Scan", style="Scan.TButton", command=run\_scan)

scan\_button.config(width=20, padding=(20, 20))

scan\_button.pack(pady=20)

root.mainloop()

def logout():

confirmation = messagebox.askyesno("Confirmation", "Are you sure you want to logout?")

if confirmation:

root.destroy()

def capture\_webcam():

subprocess.Popen(["python", "v4.py"])

def run\_scan():

# Disable the Scan button

scan\_button.config(state=tk.DISABLED)

scanning\_var = tk.StringVar()

scanning\_label = tk.Label(root, textvariable=scanning\_var, font=("Arial", 20), background="#EDC6A0")

scanning\_label.place(relx=0.418, rely=0.8)

scanning\_var.set("- - - SCANNING PLEASE WAIT - - -")

def scan\_thread():

# Start the fack.py process

scan\_process = subprocess.Popen(["python", "fack.py"], stdout=subprocess.PIPE, stderr=subprocess.PIPE)

# Wait for the fack.py process to finish

scan\_process.communicate()

scanning\_var.set("") # Clear the scanning message

# Re-enable the Scan button after scanning is complete

scan\_button.config(state=tk.NORMAL)

threading.Thread(target=scan\_thread, daemon=True).start()

def refresh():

confirmation = messagebox.askyesno("Confirmation", "Are you sure you want to refresh?")

if confirmation:

root.destroy()

dashboard()

dashboard()