Automating Lateral Flow Tests

This project automates the analysis of lateral flow test (LFT) images to determine results ("Positive", "Negative", or "Void") with confidence scores. The pipeline includes quality checks, background removal, red channel highlighting, cropping, and line detection.

In [ ]:

```python
#importing libraries required
import os
import cv2
import numpy as np
from rembg import remove #deep learning based background removal
from PIL import Image
from pathlib import Path
import matplotlib.pyplot as plt
from collections import Counter
from tqdm import tqdm
```

Step 1: Image Quality Assessment and Background Removal

Images are assessed for brightness and sharpness to ensure reliable LFT analysis. Poor-quality images (too dim or blurry) are rejected, and backgrounds are removed to isolate the test strip.

- Brightness: Normalized histogram average (0-1).
- Sharpness: Laplacian variance (higher = sharper).
- Cropping: Uses largest contour to focus on LFT strip.

In [ ]:

```python
#function to calculate brightness (using average pixel intensity)
#calculates the normalized average pixel intensity of the image to assess its brightness.

def calculate_brightness(image):
    grayscale = image.convert("L") #Converts image to grayscale to simplify brightness calculation.
    histogram = grayscale.histogram()
    pixels = sum(histogram)
    brightness = sum(i * pixel for i, pixel in enumerate(histogram)) / pixels
    return brightness / 255  #normalize to [0, 1]

#function to calculate sharpness (using laplacian variance)
#calculates the variance of the Laplacian, which measures the sharpness of edges in the image
def calculate_sharpness(image):
    #measures edge sharpness using laplacian variance
    image_np = np.array(image.convert("L"))
    laplacian_var = cv2.Laplacian(image_np, cv2.CV_64F).var()
    return laplacian_var  #higher values indicate sharper images

#function to crop the image to remove black background
#crops the image by isolating the largest contour, typically representing the region of interest (lft strip)
def crop_to_lft(image):
    #isolates largest contour (lft strip) for cropping
    image_np = np.array(image)
    gray = cv2.cvtColor(image_np, cv2.COLOR_RGB2GRAY) #convert to grayscale for contour detection.
    _, binary = cv2.threshold(gray, 1, 255, cv2.THRESH_BINARY)#create a binary image for contour finding.
    contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE) #find contours in the binary image.
    largest_contour = max(contours, key=cv2.contourArea) #select the largest contour, assumed to be the lft region.
    x, y, w, h = cv2.boundingRect(largest_contour)
    cropped_image = image.crop((x, y, x + w, y + h))
    return cropped_image  #returns focused lft region

#paths to folders
input_folder = "/Users/saniyakhan/Desktop/Option 1 data"
output_folder = "/Users/saniyakhan/Desktop/Option 1 data_no_bg"
processed_folder = "/Users/saniyakhan/Desktop/Processed images"

#create output folders if they don't exist
if not os.path.exists(output_folder):
    os.makedirs(output_folder)  #for background removed images
if not os.path.exists(processed_folder):
    os.makedirs(processed_folder)  #for quality checked images

#brightness and sharpness thresholds
brightness_threshold = 0.1
sharpness_threshold = 5

#lists to track rejected images
rejected_images = []

#loop through all images in the input folder
for filename in os.listdir(input_folder):
    if filename.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.gif')):
        #full file path
        input_path = os.path.join(input_folder, filename)
        output_path = os.path.join(output_folder, filename)
        processed_path = os.path.join(processed_folder, filename)

        #open the image
        with open(input_path, 'rb') as i:
            #remove the background using rembg
            input_image = i.read()
            output_image = remove(input_image)

            #save the output image
            with open(output_path, 'wb') as o:
                o.write(output_image)

        #open the background removed image for further processing
        image = Image.open(output_path)

        #convert rgba to rgb if necessary
        if image.mode == 'RGBA':
            image = image.convert('RGB')  #ensures consistent format

        #crop the image to remove black background
        image = crop_to_lft(image)

        #check brightness
        brightness = calculate_brightness(image)
        if brightness < brightness_threshold:
            print(f"Rejected {filename}: Too dim (Brightness: {brightness:.2f})")
            rejected_images.append((filename, f"Too dim (Brightness: {brightness:.2f})"))
            continue

        #check sharpness
        sharpness = calculate_sharpness(image)
        if sharpness < sharpness_threshold:
            print(f"Rejected {filename}: Too blurry (Sharpness: {sharpness:.2f})")
            rejected_images.append((filename, f"Too blurry (Sharpness: {sharpness:.2f})"))
            continue

        #save the processed image
        image.save(processed_path)
        print(f"Accepted {filename} (Brightness: {brightness:.2f}, Sharpness: {sharpness:.2f})")

#print rejected images at the end
if rejected_images:
    print("\nRejected Images:")
    print("=" * 30)
    for filename, reason in rejected_images:
        print(f"{filename}: {reason} - Please retake image.")
else:
    print("\nAll images passed the checks. No rejected images.")

print("\nProcessing complete!")
```

Step 2: Highlighting Red Channels

LFT lines are red, so this step isolates them using HSV color space, which is more robust than RGB for color detection.

- Why HSV: Separates hue from lighting, helping us improving accuracy.
- Range: Tuned to LFT red hues (127-179).

In [ ]:

```python
#function to highlight red channels in an image using HSV color segmentation.
#the processed image is saved in the specified output folder.
def highlight_red_channels(image_path, output_folder):
    #load the image
    image = cv2.imread(image_path)
    if image is None:
        print(f"Error: Could not load image {image_path}")
        return

    #convert to HSV for better red color segmentation
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    #define the range for red color in HSV (tuned for lateral flow test lines)
    lower_red = np.array([127, 63, 0])  #lower bounds for red hue
    upper_red = np.array([179, 255, 255])  #uupper bounds for red hue

    #create a binary mask for red hues within the defined range
    red_mask = cv2.inRange(hsv, lower_red, upper_red)

    #apply the mask to retain only the red regions in the image
    red_highlighted = cv2.bitwise_and(image, image, mask=red_mask)

    #save the image with highlighted red regions in the output folder
    output_path = os.path.join(output_folder, os.path.basename(image_path))
    cv2.imwrite(output_path, red_highlighted)

    print(f"Highlighted red channels for {os.path.basename(image_path)}")

#path to the folder of processed images
processed_folder = "/Users/saniyakhan/Desktop/Processed images"

#create the output folder for highlighted images if it doesn't exist
highlighted_folder = "/Users/saniyakhan/Desktop/Highlighted images"
if not os.path.exists(highlighted_folder):
    os.makedirs(highlighted_folder)  #ensure the folder exists

#loop through all processed images in the folder and highlight red channels
for filename in os.listdir(processed_folder):
    if filename.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.gif')):
        image_path = os.path.join(processed_folder, filename)
        highlight_red_channels(image_path, highlighted_folder)

print("Red channel highlighting completed!")
```

Step 3: Cropping to Line Region

Images are cropped to the central region where LFT lines typically appear, reducing noise and focusing analysis.

- Parameters: 60% width, 60% height, centered.
- Helps us prevent errors and false positives.

In [ ]:

```python
#function to crop images to middle region
def crop_to_middle_region(input_folder, output_folder, crop_width_percentage=0.6,
                          crop_height_percentage=0.6, vertical_offset_percentage=0.0):
    #create output folder if it doesn't exist
    os.makedirs(output_folder, exist_ok=True)

    #get list of image files
    image_extensions = ['.jpg', '.jpeg', '.png', '.bmp', '.tif', '.tiff']
    image_files = []

    for file in os.listdir(input_folder):
        ext = os.path.splitext(file)[1].lower()
        if ext in image_extensions:
            image_files.append(file)

    print(f"Found {len(image_files)} images in folder")

    #process each image
    for image_file in image_files:
        input_path = os.path.join(input_folder, image_file)
        output_path = os.path.join(output_folder, f"cropped_{image_file}")

        #read image
        image = cv2.imread(input_path)

        if image is None:
            print(f"Could not read image: {input_path}")
            continue

        #get dimensions
        height, width = image.shape[:2]

        #calculate horizontal crop boundaries
        crop_width = int(width * crop_width_percentage)
        left_boundary = (width - crop_width) // 2
        right_boundary = left_boundary + crop_width

        #calculate vertical crop boundaries
        crop_height = int(height * crop_height_percentage)
        vertical_offset = int(height * vertical_offset_percentage)
        top_boundary = (height - crop_height) // 2 + vertical_offset
        bottom_boundary = top_boundary + crop_height

        # ensure boundaries are within image dimensions
        top_boundary = max(0, top_boundary)
        bottom_boundary = min(height, bottom_boundary)

        #crop image to middle region
        cropped_image = image[top_boundary:bottom_boundary, left_boundary:right_boundary]

        #save cropped image
        cv2.imwrite(output_path, cropped_image)
        print(f"Cropped and saved: {output_path}")

#folder paths
input_folder = "/Users/saniyakhan/Desktop/Highlighted images"
output_folder = "/Users/saniyakhan/Desktop/Cropped images"

#percentage of middle region to keep
crop_width_percentage = 0.6   #60% of width focuses on lines
crop_height_percentage = 0.6  #60% of height captures line region
vertical_offset_percentage = 0  #center vertically

#process images
crop_to_middle_region(input_folder, output_folder,
                      crop_width_percentage,
                      crop_height_percentage,
                      vertical_offset_percentage)

print("Cropping completed!")
```

Step 4: Line Detection and Result Classification

This step detects red lines in LFT images and classifies results.

- Method: Red channel row sums, smoothed with convolution, peak detection.
- Noise reduction via smoothing; the minimum distance (40 pixels) between peaks.
- Confidence: Based on peak intensity and sharpness, capped at 95%.
- Results: "Positive" (2 lines), "Negative" (1 line), "Void" (0 lines).

In [ ]:

```python
#function to analyze lateral flow test
def analyze_lateral_flow_test(image_path, min_line_distance=40):
    #read the image
    image = cv2.imread(image_path)
    if image is None:
        return "Error: Could not read image", 0, None

    #create visualisation copy
    visualization = image.copy()

    #extract the red channel
    _, _, red_channel = cv2.split(image)

    #calculate the sum of red intensities
    row_sums = np.sum(red_channel, axis=1)

    #smooth the sums to reduce noise
    smoothed_sums = np.convolve(row_sums, np.ones(5)/5, mode='same')

    #find peaks above average
    avg_sum = np.mean(smoothed_sums)
    threshold = avg_sum * 2  #twice average better accuracy

    #find potential peaks
    raw_peaks = []
    window_size = 10

    for i in range(window_size, len(smoothed_sums) - window_size):
        window = smoothed_sums[i-window_size:i+window_size]
        if (smoothed_sums[i] == max(window) and smoothed_sums[i] > threshold):
            raw_peaks.append((i, smoothed_sums[i]))

    #sort peaks by intensity
    raw_peaks.sort(key=lambda x: x[1], reverse=True)

    #filter peaks by minimum distance
    peaks = []
    for pos, intensity in raw_peaks:
        if all(abs(pos - existing_pos) >= min_line_distance for existing_pos in peaks):
            peaks.append(pos)  #ensures distinct lines

    #sort peaks by position (top to bottom)
    peaks.sort()

    #calculate confidence based on intensity and sharpness
    confidence = 0
    peak_intensities = []
    peak_sharpness = []

    #get peak intensities relative to threshold
    for peak in peaks:
        rel_intensity = smoothed_sums[peak] / threshold
        peak_intensities.append(rel_intensity)

        #calculate peak sharpness
        if peak > window_size and peak < len(smoothed_sums) - window_size:
            local_bg = (smoothed_sums[peak-window_size] + smoothed_sums[peak+window_size]) / 2
            sharpness = smoothed_sums[peak] / max(local_bg, 1)  #avoids division by zero
            peak_sharpness.append(sharpness)

    #determine result and confidence
    line_count = len(peaks)

    if line_count == 0:
        result = "Void: No lines detected"
        confidence = 0  #no peaks = zero confidence
    elif line_count == 1:
        result = "Negative: Control line only"
        if peak_intensities and peak_sharpness:
            intensity_factor = min(2.0, peak_intensities[0]) / 2.0
            sharpness_factor = min(3.0, peak_sharpness[0]) / 3.0
            confidence = min(95, 50 * intensity_factor + 45 * sharpness_factor)
    elif line_count == 2:
        result = "Positive: Control and test lines"
        if len(peak_intensities) >= 2 and len(peak_sharpness) >= 2:
            intensity1 = min(2.0, peak_intensities[0]) / 2.0
            sharpness1 = min(3.0, peak_sharpness[0]) / 3.0
            intensity2 = min(2.0, peak_intensities[1]) / 2.0
            sharpness2 = min(3.0, peak_sharpness[1]) / 3.0
            conf1 = 50 * intensity1 + 45 * sharpness1
            conf2 = 50 * intensity2 + 45 * sharpness2
            confidence = min(95, min(conf1, conf2))
    else:
        result = f"Warning: {line_count} lines detected"
        confidence = 20  #low confidence for unexpected cases

    #draw detected lines with arrows on visualisation
    for i, peak in enumerate(peaks):
        if i == 0:
            color = (0, 255, 0)  #green for control line
            label = "Control"
        else:
            color = (0, 0, 255)  #red for test line
            label = "Test"

        #draw arrow pointing to the line
        start_point = (10, peak)
        end_point = (50, peak)
        cv2.arrowedLine(visualization, start_point, end_point, color, 2, tipLength=0.3)

        #add label text
        cv2.putText(visualization, label, (55, peak + 5),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, color, 2)

    #add result text with confidence
    result_text = f"{result}"
    confidence_text = f"Confidence: {confidence:.1f}%"

    cv2.putText(visualization, result_text, (10, 30),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
    cv2.putText(visualization, confidence_text, (10, 60),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)

    return result, confidence, visualization

#function to process all images
def process_images(folder_path, output_folder=None):
    #check if folder exists
    if not os.path.exists(folder_path):
        print(f"Error: Folder {folder_path} does not exist")
        return

    #create output folder if specified
    if output_folder and not os.path.exists(output_folder):
        os.makedirs(output_folder)  #for result images

    #get all image files
    image_extensions = ['.jpg', '.jpeg', '.png', '.bmp', '.tif', '.tiff']
    image_files = []

    for file in os.listdir(folder_path):
        ext = os.path.splitext(file)[1].lower()
        if ext in image_extensions:
            image_files.append(file)

    #sort files for consistent output
    image_files.sort()

    #process each image
    results = {}

    print("\nLateral Flow Test Results:")
    print("="*60)
    print(f"{'Filename':<25} {'Result':<30} {'Confidence'}")
    print("="*60)

    plt.figure(figsize=(10, 8))

    for image_file in image_files:
        image_path = os.path.join(folder_path, image_file)
        result, confidence, visualization = analyze_lateral_flow_test(image_path)

        #store the results
        results[image_file] = {
            "result": result,
            "confidence": confidence
        }

        #print the result
        print(f"{image_file:<25} {result:<30} {confidence:.1f}%")

        #display image with detected lines
        plt.clf()
        plt.imshow(cv2.cvtColor(visualization, cv2.COLOR_BGR2RGB))
        plt.title(f"{image_file}: {result} (Confidence: {confidence:.1f}%)")
        plt.axis('off')
        plt.tight_layout()
        plt.show()

        #save result image if output folder is specified
        if output_folder:
            output_path = os.path.join(output_folder, f"result_{image_file}")
            cv2.imwrite(output_path, visualization)

    plt.close()

    #print summary
    positives = sum(1 for r in results.values() if "Positive" in r["result"])
    negatives = sum(1 for r in results.values() if "Negative" in r["result"])
    voids = sum(1 for r in results.values() if "Void" in r["result"])
    warnings = sum(1 for r in results.values() if "Warning" in r["result"])

    print("\nSummary:")
    print(f"Total images: {len(results)}")
    print(f"Positive results: {positives}")
    print(f"Negative results: {negatives}")
    print(f"Void results: {voids}")
    print(f"Warnings: {warnings}")

    #calculate average confidence by result type
    if positives > 0:
        avg_pos_conf = sum(r["confidence"] for r in results.values() if "Positive" in r["result"]) / positives
        print(f"Average confidence for positive results: {avg_pos_conf:.1f}%")

    if negatives > 0:
        avg_neg_conf = sum(r["confidence"] for r in results.values() if "Negative" in r["result"]) / negatives
        print(f"Average confidence for negative results: {avg_neg_conf:.1f}%")

    return results

#specify the folder path
input_folder = "/Users/saniyakhan/Desktop/Cropped images"

#output folder to save final results images
output_folder = "/Users/saniyakhan/Desktop/Results"

#process all images in the folder
results = process_images(input_folder, output_folder)
```

Interpretation of Results

Using our results, we can produce graphs to visualise the results for better interpretation.

Below we have include:

1. Bar graph - to present the outcome in a visual and easy to read way.
2. Scatter plot - Image wise variation of confidence percentages to visualise how accurate the pipeline is.

In [ ]:

```python
#extracting result types by creating lists
classifications = []
confidences = []
filenames = []

for filename, data in tqdm(results.items(), desc="Loading result data"):
     #extract the classification label ("Positive", "Negative", "Void") from the result string
    label = data['result'].split(':')[0].strip()
    confidence = data['confidence'] #extract the confidence score for the image

    classifications.append(label)
    confidences.append(confidence)
    filenames.append(filename)

#generate a bar chart to visualize the classification distribution (Positive, Negative, Void, etc.)
#'counts' is a Counter object that will count the occurrences of each classification type.
plt.figure(figsize=(6, 4))
counts = Counter(classifications)

plt.bar(counts.keys(), counts.values(), color=['#6baed6', '#74c476', '#fd8d3c', '#d73027'])
plt.title('Classification Results')
plt.xlabel('Result Type')
plt.ylabel('Number of Images')
plt.grid(axis='y', linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()

#generate a scatter plot to visualize the confidence scores for each image.
#this plot helps assess how confident the model is for each image, from 0% to 100%.
plt.figure(figsize=(10, 5))
plt.scatter(range(len(confidences)), confidences, color='#2c7fb8')
plt.xticks(range(len(filenames)), filenames, rotation=90)
plt.ylim(0, 100)
plt.title('Confidence Scores per Image')
plt.xlabel('Image Filename')
plt.ylabel('Confidence (%)')
plt.grid(True, linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

In [ ]: