DATA-MINING PROJECT – CIA-2

JAIYANTH JITENDRA P – 23011101051 [AIDS A, 2nd Year]
KENSON FRANCIS W – 23011101064 [AIDS A, 2nd Year]

DATATYPE : IMAGE

Pothole Detection Using Image-Based Data Mining

Challenges Faced

During the pothole detection project, some challenges were encountered:

1. **Image Quality Issues**:
   - Some images had **unclear potholes**, making it difficult for annotation and model learning.
   - Several images included **unnecessary elements** such as traffic, pedestrians, and background objects, reducing the clarity of the road surface.

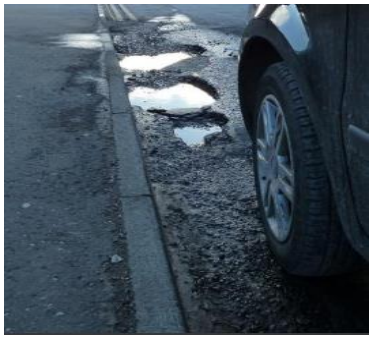
2. **Lighting Conditions**:
   - Images had inconsistent **brightness**, **contrast**, and **shadows**, making preprocessing important for standardization.

3. **Manual Annotation**:
   - All images were **manually annotated** using RoboFlow, which was time-consuming .
4. **Time Taking:**
   It was time taking to preprocess and detect the pothole on the given image

Few images were attached for reference:

Images that were similar to a plain road

Images with unnecessary elements:



Application of Pothole Detection Model

Use case of our model: Model can process incoming images and flag roads with potholes.

Where can a pothole detection model be used?
In **smart city development**, to automatically detect potholes from road surveillance footage for quicker repairs.

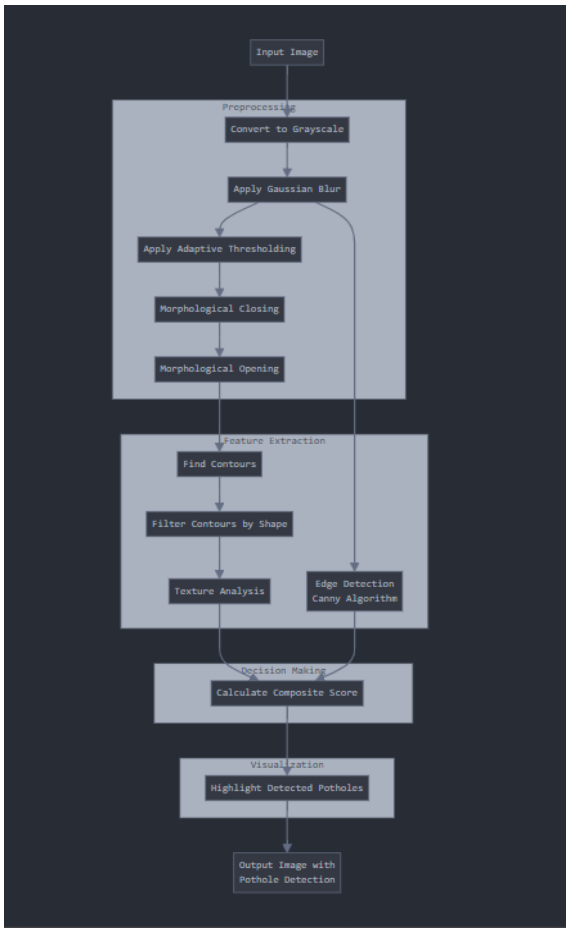
What happens if the model is trained at a larger scale?
With more data, the model can become **highly accurate**, adapting to different road textures, lighting conditions, and traffic environments across countries.

Can this be used by municipal bodies?
Yes, the model can assist **municipal corporations** to automate road inspection and maintenance alerts.

Is this model scalable to real-time systems?
With the use of real-time object detection algorithms (like YOLO), the model can be scaled to **live video analysis** from road-side cameras or dashcams.

Can it help with accident prevention?
Definitely! Early pothole detection can help **reduce accidents**, especially for two-wheelers and night driving.

Architecture Diagram Workflow



Pothole Detection Architecture Explanation

1. Convert the Image to Grayscale

- **Why**: Reduces the complexity of the image by eliminating color information while preserving structural details. This simplifies processing and reduces computational overhead.
- **Implementation**: Uses the standard grayscale conversion formula ($0.299 \times R + 0.587 \times G + 0.114 \times B$) to weight each color channel according to human perception.
- **Benefit**: Focuses the analysis on intensity variations rather than color differences, which is more relevant for detecting structural irregularities like potholes.

2. Apply Gaussian Blur

- **Why**: Reduces noise and smooths out small irrelevant details that might interfere with the detection process.
- **Implementation**: Convolves the image with a Gaussian kernel, which gives more weight to pixels near the center.
- **Benefit**: Improves the robustness of subsequent steps by removing high-frequency noise while preserving the overall structure of the image.

3. Apply Adaptive Thresholding

- **Why**: Separates the foreground (potential potholes) from the background (road surface) by converting the grayscale image to binary.
- **Implementation**: Calculates a threshold for each pixel based on the mean of its local neighborhood, accounting for lighting variations across the image.
- **Benefit**: Works better than global thresholding for outdoor scenes where lighting conditions vary within the image.

4. Perform Morphological Operations

- **Why**: Refines the binary image by removing noise and filling small gaps.
- **Implementation**:
    - **Closing** (dilation followed by erosion): Fills small holes in the foreground objects.

    - **Opening** (erosion followed by dilation): Removes small objects and noise without affecting larger structures.
- **Benefit**: Creates cleaner regions of interest for contour detection, reducing false positives.

5. Find Contours

- **Why**: Identifies boundaries of potentially connected regions in the binary image.
- **Implementation**: Labels connected components and extracts boundary points.
- **Benefit**: Provides shape information needed to identify potential pothole regions.

6. Filter Contours Based on Shape Factors

- **Why**: Not all detected contours are potholes; this step eliminates contours that don't have pothole-like characteristics.
- **Implementation**: Calculates and evaluates multiple shape metrics:
    - **Area**: Must be large enough to be a pothole
    - **Perimeter**: Used to calculate circularity
    - **Circularity**: Potholes are somewhat circular but not perfectly so ($0.2 < \text{circularity} < 0.9$)
    - **Aspect Ratio**: Most potholes aren't extremely elongated ($0.4 < \text{aspect\_ratio} < 2.5$)
    - **Extent**: Ratio of contour area to bounding rectangle area (must be > 0.3)
- **Benefit**: Greatly reduces false positives by applying domain knowledge about typical pothole shapes.

7. Analyze Texture Inside Contours

- **Why**: Potholes typically have different texture patterns than regular road surfaces.
- **Implementation**:
    - Applies the Laplacian operator to measure local variations in intensity
    - Calculates the mean texture value inside each contour

    - Keeps only contours with higher texture variation (texture_mean > 5)
- **Benefit**: Further refines detection by considering surface characteristics, not just shape.

8. Use Edge Detection to Analyze Pothole Boundaries

- **Why**: Pothole edges are typically distinct from the surrounding road surface.
- **Implementation**: Applies Canny edge detection algorithm to the blurred image to find strong edges.
- **Benefit**: Provides additional evidence for pothole presence by quantifying edge characteristics.

9. Calculate a Composite Pothole Score

- **Why**: Combines multiple features to create a robust confidence score.
- **Implementation**: Weighted sum of three key factors:
    - **Area Ratio**: Percentage of image covered by detected potholes (50% weight)
    - **Count Factor**: Number of detected potholes (30% weight)
    - **Edge Density**: Edge concentration in pothole regions (20% weight)
- **Benefit**: Produces a confidence score (0-1) that reflects the likelihood of pothole presence based on multiple complementary features.

10. Highlight Detected Potholes

- **Why**: Provides visual feedback by marking detected potholes on the original image.
- **Implementation**: Draws contour outlines in red and adds text showing detection result and confidence score.
- **Benefit**: Creates an interpretable output for human review or documentation.

Why This Approach Works Well for Pothole Detection

1. Preprocessing reduces noise and enhances features of interest
2. Feature extraction identifies regions with pothole-like characteristics
3. Multi-feature analysis combines shape, texture, and edge information
4. Decision making integrates multiple features to make a robust determination

Module Description:

Detailed Explanation of Each Function

1. load_image(image_path)

- **Purpose**: Loads an image from a file path and converts it to a NumPy array
- **Parameters:** Path to the image file
- **Returns:** NumPy array representation of the image, or None if loading fails
- **Details:** Uses PIL's Image.open() to load the image and then converts it to a NumPy array

2. manual_bgr_to_gray(image)

- **Purpose:** Converts a color (BGR) image to grayscale
- **Parameters**: BGR format image as NumPy array
- **Returns:** Grayscale image as NumPy array
- **Details:** Applies the standard grayscale conversion formula **(0.299*R* + *0.587*G + 0.114*B) to each pixel**

3. manual_gaussian_blur(image, kernel_size=5, sigma=1.0)

- **Purpose:** Applies Gaussian blur to reduce noise in images
- **Parameters:** Input image, kernel size, and standard deviation (sigma)
- **Returns:** Blurred image
- **Details:** Creates a Gaussian kernel and applies it using manual convolution, which smooths the image and reduces noise

4. manual_adaptive_threshold(image, block_size=11, C=2)

- **Purpose:** Applies adaptive thresholding to convert grayscale to binary image
- **Parameters:** Grayscale image, block size for local area, and constant C
- **Returns:** Binary image
- **Details:** For each pixel, calculates a threshold based on the mean of its neighborhood, then applies thresholding

5. manual_morphology_close(image, kernel_size=5, iterations=1)

- **Purpose:** Applies morphological closing (dilation followed by erosion)
- **Parameters:** Binary image, kernel size, and iterations count
- **Returns**: Processed binary image
- **Details:** Fills small holes in the foreground objects, useful for closing small gaps in detected contours

6. manual_morphology_open(image, kernel_size=5, iterations=1)

- **Purpose:** Applies morphological opening (erosion followed by dilation)
- **Parameters:** Binary image, kernel size, and iterations count
- Returns: Processed binary image
- **Details:** Removes small objects/noise from the foreground, keeping larger objects intact

7. manual_dilate(image, kernel_size=5, iterations=1)

- **Purpose:** Expands the boundaries of foreground objects
- **Parameters:** Binary image, kernel size, and iterations count
- **Returns: Dilated image**
- **Details:** If any pixel in the neighborhood is white, the center pixel becomes white

8. manual_erode(image, kernel_size=5, iterations=1)

- **Purpose:** Shrinks the boundaries of foreground objects
- **Parameters**: Binary image, kernel size, and iterations count
- **Returns:** Eroded image
- **Details:** Only if all pixels in the neighborhood are white, the center pixel remains white

9. manual_sobel(image)

- **Purpose**: Applies Sobel operators to detect edges and gradients
- **Parameters:** Grayscale image
- **Returns:** Gradient in x and y directions
- **Details:** Uses 3×3 Sobel kernels to find horizontal and vertical gradients

10. manual_canny(image, low_threshold=30, high_threshold=150)

- **Purpose:** Implements Canny edge detection algorithm
- **Parameters:** Grayscale image, low and high thresholds
- **Returns:** Edge image
- **Details: Four steps:** gradient calculation, magnitude/direction computation, non-maximum suppression, and hysteresis thresholding

11. manual_find_contours(binary_image)

- **Purpose:** Finds contours (boundaries) in binary images
- **Parameters:** Binary image
- **Returns:** List of contours, where each contour is a list of (y,x) points
- **Details:** Labels connected components and finds boundary points

12. label_components(binary_image)

- **Purpose:** Labels connected components in a binary image
- **Parameters:** Binary image
- **Returns:** Labeled image and number of labels
- **Details:** Uses breadth-first search to find and label connected regions

13. contour_area(contour)

- **Purpose:** Calculates the area of a contour
- **Parameters:** Contour as list of points
- **Returns**: Area value
- **Details:** Uses the shoelace formula (also known as the surveyor's formula)

14. contour_perimeter(contour)

- **Purpose:** Calculates the perimeter of a contour
- **Parameters:** Contour as list of points

- **Returns:** Perimeter value
- **Details:** Sums the Euclidean distance between consecutive points

15. bounding_rect(contour)

- **Purpose:** Finds the bounding rectangle of a contour
- **Parameters:** Contour as list of points
- **Returns:** Tuple (x, y, width, height)
- **Details:** Finds the minimum and maximum x,y coordinates to determine the rectangle

16. manual_laplacian(image)

- **Purpose:** Applies the Laplacian operator for edge detection
- **Parameters:** Grayscale image
- **Returns:** Laplacian result as floating-point image
- **Details:** Uses a 3×3 Laplacian kernel to find areas of rapid intensity change

17. detect_potholes(image_path, debug=False)

- **Purpose:** Main function to detect potholes in road images
- **Parameters:** Path to image, debug flag
- **Returns:** Boolean (pothole detected), confidence score, and processed image
- **Details:** Implements the complete pothole detection pipeline as described earlier

18. cv2_line(image, pt1, pt2, color, thickness=1)

- **Purpose:** Manual implementation of line drawing (similar to OpenCV's line function)
- **Parameters:** Image, start point, end point, color, and thickness
- Returns: None (modifies image in-place)
- **Details:** Uses Bresenham's line algorithm for efficient line drawing

19. cv2_put_text(image, text, position, font_scale=1, color=(0,0,0), thickness=1)

- **Purpose:** Places text on an image (simplified version of OpenCV's putText)
- **Parameters:** Image, text string, position, font scale, color, and thickness

- **Returns:** None (modifies image in-place)
- **Details:** Creates a semi-transparent background and would normally render text (simplified implementation)

20. process_dataset(pothole_dir, no_pothole_dir)

- **Purpose**: Evaluates algorithm performance on a dataset of images
- **Parameters:** Directories containing pothole and non-pothole images
- **Returns:** Dictionary with performance metrics (accuracy, precision, recall, F1-score)
- **Details:** Processes all images in both directories and calculates classification metrics

DATA SELECTION & PREPROCESSING

Data Selection

The process_dataset function expects data to be organized in a specific way:

1. **Dataset Organization**:
    - The dataset was divided into two main directories:
        - pothole_dir: Contains images showing roads with potholes
        - no_pothole_dir: Contains images showing roads without potholes

2. **Image Selection**:
    - The code processes files with these extensions: .png, .jpg, .jpeg
    - Each image is expected to be a road scene taken from above or from a driver's perspective

```python
def process_dataset(pothole_dir, no_pothole_dir):

    true_positives = 0
    false_positives = 0
    true_negatives = 0
    false_negatives = 0

    # Process pothole images
    for filename in os.listdir(pothole_dir):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
            image_path = os.path.join(pothole_dir, filename)
            has_pothole, score, _ = detect_potholes(image_path)

            if has_pothole:
                true_positives += 1
            else:
                false_negatives += 1

    # Process non-pothole images
    for filename in os.listdir(no_pothole_dir):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
            image_path = os.path.join(no_pothole_dir, filename)
            has_pothole, score, _ = detect_potholes(image_path)

            if has_pothole:
                false_positives += 1
            else:
                true_negatives += 1
```

The preprocessing part of the code is quite detailed and involves several steps:
1) Image Loading:

```python
    # Read the image
    try:
        image = np.array(Image.open(image_path))
    except Exception as e:
        print(f"Error: Could not read image at {image_path}. Error: {e}")
        return False, 0, None
```

- Uses PIL (Python Imaging Library) to load the image
- Converts the image to a NumPy array for further processing
- Includes error handling for corrupted or missing images

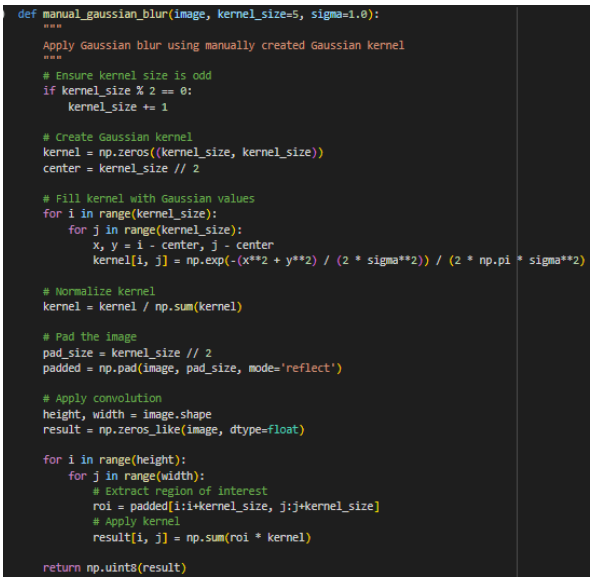
2) Grayscale Conversion

```python
def manual_rgb_to_gray(rgb_image):
    """
    Convert RGB image to grayscale using the formula:
    gray = 0.299*R + 0.587*G + 0.114*B
    """
    # Convert to numpy array if needed
    if not isinstance(rgb_image, np.ndarray):
        rgb_image = np.array(rgb_image)

    # Extract RGB channels
    r = rgb_image[:, :, 0].astype(float)
    g = rgb_image[:, :, 1].astype(float)
    b = rgb_image[:, :, 2].astype(float)

    # Apply formula
    gray = 0.299 * r + 0.587 * g + 0.114 * b

    # Normalize to 0-255 range and convert to uint8
    return np.uint8(gray)
```

- Converts the colour image to grayscale using the standard formula

- This reduces dimensionality while preserving important structural information
- Colour information is less important for pothole detection than intensity variations

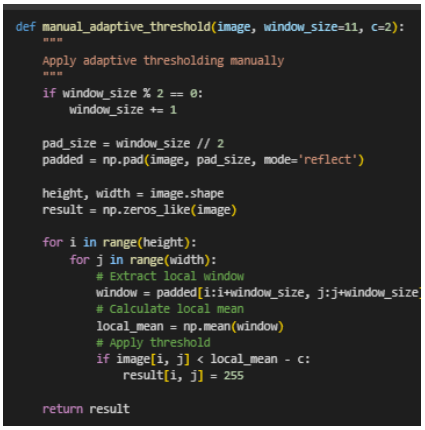
3) Gaussian Blur:



```python
def manual_gaussian_blur(image, kernel_size=5, sigma=1.0):
    """
    Apply Gaussian blur using manually created Gaussian kernel
    """
    # Ensure kernel size is odd
    if kernel_size % 2 == 0:
        kernel_size += 1

    # Create Gaussian kernel
    kernel = np.zeros((kernel_size, kernel_size))
    center = kernel_size // 2

    # Fill kernel with Gaussian values
    for i in range(kernel_size):
        for j in range(kernel_size):
            x, y = i - center, j - center
            kernel[i, j] = np.exp(-(x**2 + y**2) / (2 * sigma**2)) / (2 * np.pi * sigma**2)

    # Normalize kernel
    kernel = kernel / np.sum(kernel)

    # Pad the image
    pad_size = kernel_size // 2
    padded = np.pad(image, pad_size, mode='reflect')

    # Apply convolution
    height, width = image.shape
    result = np.zeros_like(image, dtype=float)

    for i in range(height):
        for j in range(width):
            # Extract region of interest
            roi = padded[i:i+kernel_size, j:j+kernel_size]
            # Apply kernel
            result[i, j] = np.sum(roi * kernel)

    return np.uint8(result)
```

- Creates a Gaussian kernel based on the specified size and sigma
- Applies this kernel to smooth the image
- Helps reduce noise and small details that could interfere with pothole detection

4) **Adaptive Thresholding:**

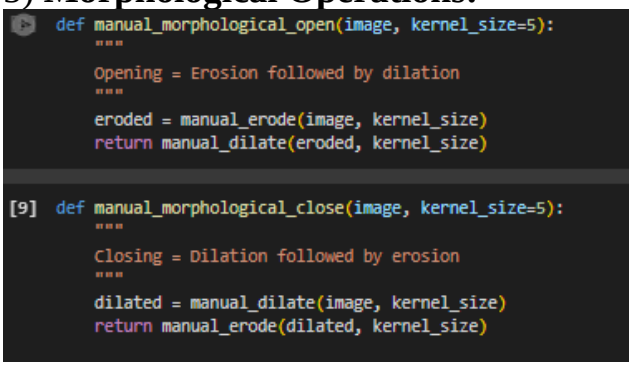


```python
def manual_adaptive_threshold(image, window_size=11, c=2):
    """
    Apply adaptive thresholding manually
    """
    if window_size % 2 == 0:
        window_size += 1

    pad_size = window_size // 2
    padded = np.pad(image, pad_size, mode='reflect')

    height, width = image.shape
    result = np.zeros_like(image)

    for i in range(height):
        for j in range(width):
            # Extract local window
            window = padded[i:i+window_size, j:j+window_size]
            # Calculate local mean
            local_mean = np.mean(window)
            # Apply threshold
            if image[i, j] < local_mean - c:
                result[i, j] = 255

    return result
```

- Converts the grayscale image to binary using local thresholds

- For each pixel, the threshold is calculated as the mean of its neighborhood minus a constant C
- This adapts to varying lighting conditions across the image
- Results in white (255) pixels for potential pothole areas and black (0) pixels for the road surface

5) **Morphological Operations:**



```
def manual_morphological_open(image, kernel_size=5):
    """
    Opening = Erosion followed by dilation
    """
    eroded = manual_erode(image, kernel_size)
    return manual_dilate(eroded, kernel_size)

[9] def manual_morphological_close(image, kernel_size=5):
    """
    Closing = Dilation followed by erosion
    """
    dilated = manual_dilate(image, kernel_size)
    return manual_erode(dilated, kernel_size)
```

- **Closing**: Fills small holes within potential pothole regions
  - Applies dilation followed by erosion
- **Opening**: Removes small noise artifacts
  - Applies erosion followed by dilation

These operations help create cleaner binary regions for contour detection

CONCISE ALGORITHM:

1. **Input**: Road image
2. **Preprocessing**:
   - Convert to grayscale using standard formula (0.299R + 0.587G + 0.114B)
     - **Why**: Reduces complexity while preserving structural information; potholes are primarily detected by shape and texture, not colour.
   - Apply Gaussian blur (kernel size 5, sigma 1.0) to reduce noise
     - **Why**: Smooths out small irrelevant details and sensor noise that could lead to false detections.
   - Apply adaptive thresholding with block size 11 and C=2

        - **Why**: Creates a binary image that adapts to lighting variations across the road surface, better separating potholes from intact road.

    - Apply morphological closing followed by opening to clean binary image
        - **Why**: Closing fills small holes in pothole regions; opening removes small noise artifacts, creating cleaner regions for analysis.

3. **Feature Extraction**:
    - Find contours in preprocessed binary image
        - **Why**: Identifies the boundaries of potential pothole regions that were highlighted in the binary image.

    - Filter contours by shape metrics:
        - Area > 100
            - **Why**: Eliminates tiny regions that are too small to be actual potholes.
        - Perimeter > 50
            - **Why**: Ensures the region has sufficient boundary length to be a significant road defect.
        - Circularity between 0.2 and 0.9
            - **Why**: Potholes are somewhat circular but rarely perfect circles; this range captures realistic pothole shapes.
        - Aspect ratio between 0.4 and 2.5
            - **Why**: Potholes are not extremely elongated in one direction; this range filters out line-like defects.
        - Extent > 0.3 (area/bounding_rect_area)
            - **Why**: Ensures the region is reasonably "filled in" and not just a sparse outline.
        - Mean intensity < 0.9 * image mean
            - **Why**: Potholes typically appear darker than the surrounding road surface due to shadows.
4. **Texture Analysis**:
    - Apply Laplacian operator to measure texture variation
        - **Why**: Laplacian highlights rapid intensity changes, which are more common in the irregular surfaces of potholes.
    - Keep only contours with texture_mean > 5

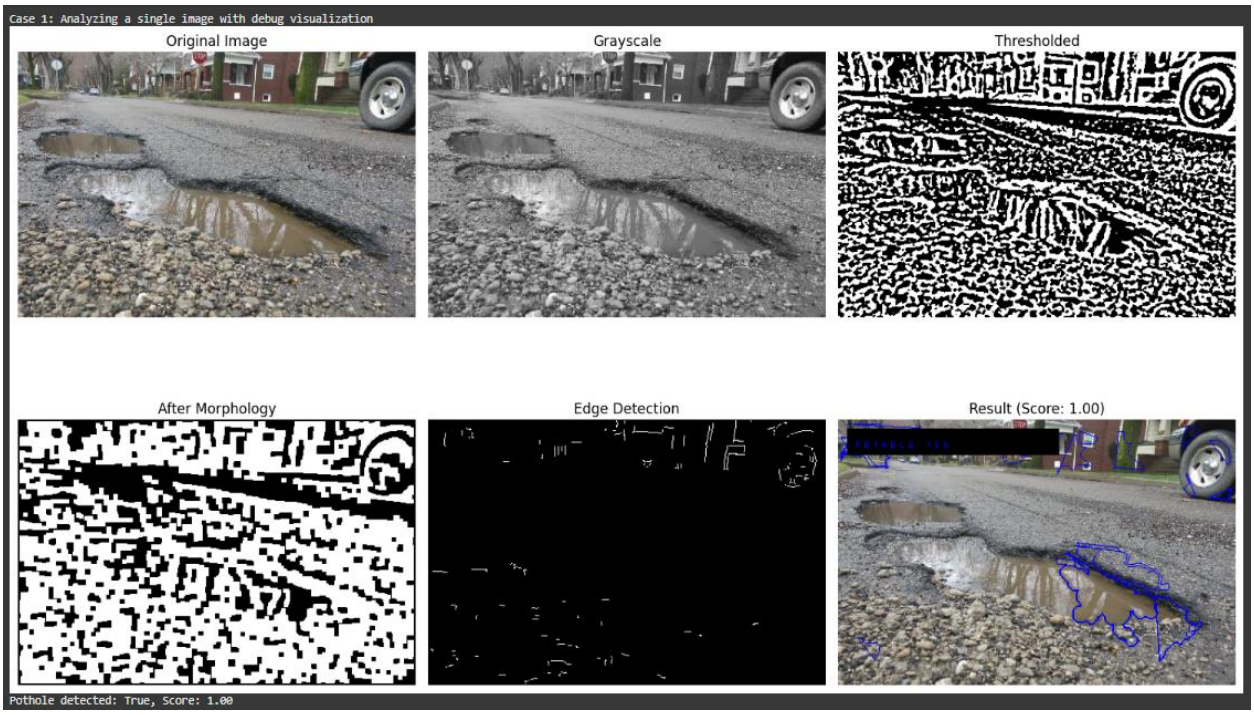
        - **Why**: Potholes have more textural variation than smooth road surfaces; this threshold separates them.

5. **Edge Analysis**:
    - Apply Canny edge detection to find boundaries
        - **Why**: Potholes typically have strong edges where they meet the regular road surface.
    - Calculate edge density in potential pothole regions
        - **Why**: Higher edge concentration within a region indicates complex surface changes characteristic of potholes.

6. **Decision Making**:
    - Calculate composite score as weighted sum:
        - 50% * area_ratio + 30% * count_factor + 20% * edge_factor
            - **Why**: Combines multiple features with weights assigned based on their reliability for detection; area gets highest weight as it's most directly related to pothole presence.
    - Classify as pothole if score > 0.3
        - **Why**: Empirically determined threshold that balances false positives and false negatives.

7. **Output**:
    - Boolean pothole presence
        - **Why**: Provides a clear yes/no decision for automated systems.
    - Confidence score (0-1)
        - **Why**: Enables prioritization in maintenance systems and allows adjusting the detection threshold if needed.
    - Visualization with highlighted potholes
        - **Why**: Allows human verification and shows exactly where the defects are located in the original image.

OUTPUT SAMPLE:



From the output image provided,

- The algorithm appears to be incorrectly identifying non-pothole regions as potholes (false positives), particularly at the bottom right where it's marking part of the rocky/gravelly area.
- The algorithm is picking up too many small variations in the road surface.
- The "After Morphology" image still contains significant noise that wasn't properly cleaned up by the closing and opening operations

Likely Causes for these problems:

1. The algorithm is likely too sensitive to textural variations
2. The thresholds used for filtering contours (circularity, aspect ratio, etc.) may need adjustment
3. In the "Edge Detection" image, edges aren't cleanly identifying pothole boundaries
4. The adaptive thresholding parameters (block_size=11, C=2) might not be optimal for this particular image, this could be the reason for over-segementation

PERFORMANCE EVALUATION:

How our model will have advantages and disadvantages from a pretrained models:

Manual preprocessing methods cover essential steps like

- grayscale conversion (standard),
- noise reduction via Gaussian blur (common, also learned by CNNs),
- adaptive thresholding for segmentation (addresses varying light, CNNs learn similar local feature analysis),
- morphological operations for cleaning binary masks (helps, CNNs learn robustness to small issues)
- edge detection with contour finding (highlights boundaries, CNNs directly predict regions).
- Feature extraction based on shape, size, intensity, and texture is our method for classification.

This manual approach differs significantly from deep learning architectures. Deep learning models automatically learn features from data, unlike our manually designed ones. They directly predict bounding boxes or segmentation masks, bypassing explicit contour finding. While our method requires manual tuning and **might struggle with variations in lighting, shadows, and complex textures**, deep learning models are generally more robust due to learned representations. Our approach might be faster for inference once set up but achieving high accuracy and generalization like deep learning models is challenging due to the fixed rules versus learned, adaptable features. Real-world pothole detection often benefits from the superior learning and robustness of deep learning**.**

Conclusion:

This pothole detection algorithm represents a comprehensive manual approach to computer vision-based pothole detection without relying on pre-built libraries for core functionality. The strengths of this approach include:

1. **Multi-layered detection strategy:** The algorithm employs several image processing stages (thresholding, morphological operations, contour analysis, and texture examination) to identify potholes.

2. **Feature-based confidence scoring:** Rather than using a single metric, the algorithm calculates a composite score incorporating area ratio, pothole count, and edge density factors**.**
3. **Shape and texture analysis:** By examining circularity, aspect ratio, mean intensity, and texture variation, the system can differentiate potholes from other road features.
4. **Quantifiable performance:** The evaluation framework provides clear metrics (accuracy, precision, recall) for assessing algorithm effectiveness across different road conditions.
5. **Independence from black-box CV libraries:** By implementing core functions manually, the algorithm provides transparency and educational value in understanding how each step contributes to detection.

However, the implementation has some limitations, including computational efficiency concerns (particularly the manual implementation of operations that would be optimized in libraries) and limited contextual understanding of road environments**.**

Future Work

To enhance this pothole detection system, several promising directions could be explored**:**

1. **Machine learning integration:**
    - Incorporate a trained classifier (SVM, Random Forest, etc.) to learn from the extracted features
    - Implement a CNN or transformer-based approach which could better capture spatial relationships
2. **Performance optimization:**
    - Parallelize the processing of image regions to improve computational efficiency
    - Implement SIMD (Single Instruction Multiple Data) operations for core functions
    - Explore GPU acceleration for convolution operations
3. **Enhanced feature engineering:**
    - Include depth information if stereo cameras or LiDAR data is available
    - Incorporate temporal information from video sequences to detect potholes through motion analysis
    - Consider road context features (road markings, curbs) to reduce false positives

```
import numpy as np
import matplotlib.pyplot as plt
import os
from PIL import Image

def manual_rgb_to_gray(rgb_image):
    """
    Convert RGB image to grayscale using the formula:
    gray = 0.299*R + 0.587*G + 0.114*B
    """
    # Convert to numpy array if needed
    if not isinstance(rgb_image, np.ndarray):
        rgb_image = np.array(rgb_image)

    # Extract RGB channels
    r = rgb_image[:, :, 0].astype(float)
    g = rgb_image[:, :, 1].astype(float)
    b = rgb_image[:, :, 2].astype(float)

    # Apply formula
    gray = 0.299 * r + 0.587 * g + 0.114 * b

    # Normalize to 0-255 range and convert to uint8
    return np.uint8(gray)

def manual_gaussian_blur(image, kernel_size=5, sigma=1.0):
    """
    Apply Gaussian blur using manually created Gaussian kernel
    """
    # Ensure kernel size is odd
    if kernel_size % 2 == 0:
        kernel_size += 1

    # Create Gaussian kernel
    kernel = np.zeros((kernel_size, kernel_size))
    center = kernel_size // 2

    # Fill kernel with Gaussian values
    for i in range(kernel_size):
        for j in range(kernel_size):
            x, y = i - center, j - center
            kernel[i, j] = np.exp(-(x**2 + y**2) / (2 * sigma**2)) /
(2 * np.pi * sigma**2)

    # Normalize kernel
    kernel = kernel / np.sum(kernel)

    # Pad the image
    pad_size = kernel_size // 2
    padded = np.pad(image, pad_size, mode='reflect')
```

```
    # Apply convolution
    height, width = image.shape
    result = np.zeros_like(image, dtype=float)

    for i in range(height):
        for j in range(width):
            # Extract region of interest
            roi = padded[i:i+kernel_size, j:j+kernel_size]
            # Apply kernel
            result[i, j] = np.sum(roi * kernel)

    return np.uint8(result)

def manual_threshold(image, threshold_value=127):
    """
    Apply simple global thresholding
    """
    result = np.zeros_like(image)
    result[image > threshold_value] = 255
    return result

def manual_adaptive_threshold(image, window_size=11, c=2):
    """
    Apply adaptive thresholding manually
    """
    if window_size % 2 == 0:
        window_size += 1

    pad_size = window_size // 2
    padded = np.pad(image, pad_size, mode='reflect')

    height, width = image.shape
    result = np.zeros_like(image)

    for i in range(height):
        for j in range(width):
            # Extract local window
            window = padded[i:i+window_size, j:j+window_size]
            # Calculate local mean
            local_mean = np.mean(window)
            # Apply threshold
            if image[i, j] < local_mean - c:
                result[i, j] = 255

    return result

def manual_dilate(image, kernel_size=5):
    """
    Manual implementation of dilation
    """
```

```
    kernel = np.ones((kernel_size, kernel_size), dtype=np.uint8)
    pad_size = kernel_size // 2
    padded = np.pad(image, pad_size, mode='constant',
constant_values=0)

    height, width = image.shape
    result = np.zeros_like(image)

    for i in range(height):
        for j in range(width):
            # Extract window
            window = padded[i:i+kernel_size, j:j+kernel_size]
            # Apply dilation
            if np.any(window * kernel > 0):
                result[i, j] = 255

    return result

def manual_erode(image, kernel_size=5):
    """
    Manual implementation of erosion
    """
    kernel = np.ones((kernel_size, kernel_size), dtype=np.uint8)
    pad_size = kernel_size // 2
    padded = np.pad(image, pad_size, mode='constant',
constant_values=0)

    height, width = image.shape
    result = np.zeros_like(image)

    for i in range(height):
        for j in range(width):
            # Extract window
            window = padded[i:i+kernel_size, j:j+kernel_size]
            # Apply erosion
            if np.all((window * kernel > 0)[kernel > 0]):
                result[i, j] = 255

    return result

def manual_morphological_open(image, kernel_size=5):
    """
    Opening = Erosion followed by dilation
    """
    eroded = manual_erode(image, kernel_size)
    return manual_dilate(eroded, kernel_size)

def manual_morphological_close(image, kernel_size=5):
    """
    Closing = Dilation followed by erosion
    """
```

```
    dilated = manual_dilate(image, kernel_size)
    return manual_erode(dilated, kernel_size)

def manual_sobel(image):
    """
    Apply Sobel operator for edge detection
    """
    # Define Sobel kernels
    sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

    # Pad the image
    padded = np.pad(image, 1, mode='reflect')

    height, width = image.shape
    gradient_x = np.zeros_like(image, dtype=float)
    gradient_y = np.zeros_like(image, dtype=float)

    # Apply convolution
    for i in range(height):
        for j in range(width):
            # Extract 3x3 window
            window = padded[i:i+3, j:j+3]
            # Apply kernels
            gradient_x[i, j] = np.sum(window * sobel_x)
            gradient_y[i, j] = np.sum(window * sobel_y)

    # Calculate gradient magnitude
    gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)

    # Normalize to 0-255
    gradient_magnitude = 255 * gradient_magnitude /
np.max(gradient_magnitude)

    return np.uint8(gradient_magnitude), gradient_x, gradient_y

def manual_canny(image, low_threshold=30, high_threshold=150):
    """
    Manual implementation of the Canny edge detector

    Steps:
    1. Compute gradients using Sobel
    2. Compute gradient magnitude and direction
    3. Apply non-maximum suppression
    4. Apply double thresholding
    5. Edge tracking by hysteresis
    """
    # 1. Apply Gaussian blur to reduce noise
    blurred = manual_gaussian_blur(image, kernel_size=5, sigma=1.0)
```

```
    # 2. Compute gradients using Sobel
    gradient_magnitude, gradient_x, gradient_y = manual_sobel(blurred)

    # Calculate gradient direction
    gradient_direction = np.arctan2(gradient_y, gradient_x) * 180 /
np.pi
    gradient_direction[gradient_direction < 0] += 180

    height, width = image.shape
    suppressed = np.zeros_like(image)

    # 3. Non-maximum suppression
    for i in range(1, height-1):
        for j in range(1, width-1):
            angle = gradient_direction[i, j]
            mag = gradient_magnitude[i, j]

            # Quantize angle to 4 directions (0, 45, 90, 135 degrees)
            if (0 <= angle < 22.5) or (157.5 <= angle <= 180):
                # Horizontal
                neighbors = [gradient_magnitude[i, j-1],
gradient_magnitude[i, j+1]]
            elif 22.5 <= angle < 67.5:
                # Diagonal (45 degrees)
                neighbors = [gradient_magnitude[i+1, j-1],
gradient_magnitude[i-1, j+1]]
            elif 67.5 <= angle < 112.5:
                # Vertical
                neighbors = [gradient_magnitude[i-1, j],
gradient_magnitude[i+1, j]]
            else:
                # Diagonal (135 degrees)
                neighbors = [gradient_magnitude[i-1, j-1],
gradient_magnitude[i+1, j+1]]

            # Keep only if it's a local maximum
            if mag >= neighbors[0] and mag >= neighbors[1]:
                suppressed[i, j] = mag

    # 4 & 5. Double thresholding and hysteresis
    result = np.zeros_like(image)

    # Strong edges
    strong_edges = (suppressed >= high_threshold)
    # Weak edges
    weak_edges = (suppressed >= low_threshold) & (suppressed <
high_threshold)

    # Add strong edges directly
    result[strong_edges] = 255
```

```
    # Process weak edges via hysteresis
    dx = [-1, -1, -1, 0, 0, 1, 1, 1]
    dy = [-1, 0, 1, -1, 1, -1, 0, 1]

    # First pass - connect weak edges to strong edges
    for i in range(1, height-1):
        for j in range(1, width-1):
            if weak_edges[i, j]:
                # Check if any strong neighbor exists
                has_strong_neighbor = False
                for k in range(8):
                    ni, nj = i + dx[k], j + dy[k]
                    if strong_edges[ni, nj]:
                        has_strong_neighbor = True
                        break

                if has_strong_neighbor:
                    result[i, j] = 255
                    strong_edges[i, j] = True  # Mark as strong for
next iterations

    return result

def find_contours(binary_image):
    """
    Simple implementation to find contours in a binary image.
    Returns a list of contours, where each contour is a list of (x, y)
points.
    """
    # Make a copy to avoid modifying the original
    image = binary_image.copy()
    height, width = image.shape
    visited = np.zeros_like(image, dtype=bool)
    contours = []

    # Directions: right, down-right, down, down-left, left, up-left,
up, up-right
    dx = [1, 1, 0, -1, -1, -1, 0, 1]
    dy = [0, 1, 1, 1, 0, -1, -1, -1]

    for y in range(1, height-1):
        for x in range(1, width-1):
            # If this is an unvisited edge pixel
            if image[y, x] == 255 and not visited[y, x]:
                # Start a new contour
                contour = []
                visited[y, x] = True
                contour.append((x, y))
```

```
                # Trace the contour
                current_x, current_y = x, y
                direction = 0  # Start searching to the right

                while True:
                    found_next = False

                    # Try each direction starting from the current one
                    for i in range(8):
                        new_dir = (direction + i) % 8
                        next_x = current_x + dx[new_dir]
                        next_y = current_y + dy[new_dir]

                        # If this is a valid edge pixel
                        if (0 <= next_x < width and 0 <= next_y <
height and
                            image[next_y, next_x] == 255 and not
visited[next_y, next_x]):
                            # Add to contour
                            contour.append((next_x, next_y))
                            visited[next_y, next_x] = True

                            # Update current position and direction
                            current_x, current_y = next_x, next_y
                            direction = (new_dir + 5) % 8  # Start
from opposite direction

                            found_next = True
                            break

                    # If no next pixel found or back to start, end
this contour
                    if not found_next or (len(contour) > 2 and
                                          contour[-1] == contour[0]):
                        break

                # Add the contour to our list if it has enough points
                if len(contour) > 4:
                    contours.append(np.array(contour))

    return contours

def contour_area(contour):
    """Calculate the area of a contour using the Shoelace formula"""
    x = contour[:, 0]
    y = contour[:, 1]
    return 0.5 * np.abs(np.dot(x, np.roll(y, 1)) - np.dot(y,
np.roll(x, 1)))

def contour_perimeter(contour):
```

```
    """Calculate the perimeter of a contour"""
    perimeter = 0
    for i in range(len(contour)):
        x1, y1 = contour[i]
        x2, y2 = contour[(i + 1) % len(contour)]
        perimeter += np.sqrt((x2 - x1)**2 + (y2 - y1)**2)
    return perimeter




def bounding_rect(contour):
    """Find the bounding rectangle of a contour"""
    x = contour[:, 0]
    y = contour[:, 1]
    return np.min(x), np.min(y), np.max(x) - np.min(x), np.max(y) -
np.min(y)

def manual_laplacian(image):
    """
    Apply Laplacian filter for texture measurement
    """
    # Laplacian kernel
    kernel = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])

    # Pad the image
    padded = np.pad(image, 1, mode='reflect')

    height, width = image.shape
    result = np.zeros_like(image, dtype=float)

    # Apply convolution
    for i in range(height):
        for j in range(width):
            window = padded[i:i+3, j:j+3]
            result[i, j] = np.sum(window * kernel)

    return result

def detect_potholes(image_path, debug=False):
    """
    Detect potholes in road images using manually implemented computer
vision algorithms.

    Parameters:
        image_path: Path to the input image
        debug: If True, displays intermediate processing steps

    Returns:
```

```
        has_pothole: Boolean indicating whether a pothole was detected
        pothole_score: A confidence score (0-1) of pothole presence
        processed_image: Image with pothole regions highlighted
    """
    # Read the image
    try:
        image = np.array(Image.open(image_path))
    except Exception as e:
        print(f"Error: Could not read image at {image_path}. Error:
{e}")
        return False, 0, None

    # Make a copy for visualization
    result_image = image.copy()

    # Step 1: Convert to grayscale
    gray = manual_rgb_to_gray(image)

    # Step 2: Apply Gaussian blur to reduce noise
    blurred = manual_gaussian_blur(gray, kernel_size=5, sigma=1.0)

    # Step 3: Apply adaptive thresholding
    binary = manual_adaptive_threshold(blurred, window_size=11, c=2)

    # Step 4: Morphological operations
    closing = manual_morphological_close(binary, kernel_size=5)
    opening = manual_morphological_open(closing, kernel_size=5)

    # Step 5: Find contours
    contours = find_contours(opening)

    # Step 6: Filter contours
    pothole_contours = []
    min_area = 100
    min_perimeter = 50

    for contour in contours:
        # Calculate area and perimeter
        area = contour_area(contour)
        perimeter = contour_perimeter(contour)

        # Skip if too small
        if area < min_area or perimeter < min_perimeter:
            continue

        # Calculate shape factors
        circularity = 4 * np.pi * area / (perimeter**2) if perimeter >
0 else 0

        # Calculate bounding rectangle and aspect ratio
```

```
        x, y, w, h = bounding_rect(contour)
        aspect_ratio = float(w) / h if h > 0 else 0
        rect_area = w * h
        extent = float(area) / rect_area if rect_area > 0 else 0

        # Check for pothole characteristics
        if (0.2 < circularity < 0.9) and (0.4 < aspect_ratio < 2.5)
and (extent > 0.3):
            # Create mask for this contour
            mask = np.zeros_like(gray)
            for px, py in contour:
                if 0 <= py < mask.shape[0] and 0 <= px <
mask.shape[1]:
                    mask[py, px] = 1

            # Dilate mask to fill interior
            mask = manual_dilate(mask, kernel_size=3)

            # Calculate mean intensity inside contour
            masked_pixels = gray[mask > 0]
            if len(masked_pixels) > 0:
                mean_intensity = np.mean(masked_pixels)
                # Potholes are usually darker than surrounding road
                if mean_intensity < np.mean(gray) * 0.9:
                    pothole_contours.append(contour)

    # Step 7: Texture analysis
    confirmed_potholes = []
    laplacian = manual_laplacian(gray)
    laplacian_abs = np.abs(laplacian)

    for contour in pothole_contours:
        # Create mask for this contour
        mask = np.zeros_like(gray)
        for px, py in contour:
            if 0 <= py < mask.shape[0] and 0 <= px < mask.shape[1]:
                mask[py, px] = 1

        # Dilate mask to fill interior
        mask = manual_dilate(mask, kernel_size=3)

        # Calculate texture features
        masked_texture = laplacian_abs[mask > 0]
        if len(masked_texture) > 0:
            texture_mean = np.mean(masked_texture)
            if texture_mean > 5:  # Threshold determined empirically
                confirmed_potholes.append(contour)

    # Step 8: Edge detection
    edges = manual_canny(blurred, low_threshold=30,
```

```
high_threshold=150)

    # Calculate pothole score
    pothole_score = 0
    total_area = gray.shape[0] * gray.shape[1]

    if confirmed_potholes:
        # Calculate total area of detected potholes
        total_pothole_area = sum(contour_area(c) for c in
confirmed_potholes)

        # Calculate edge density inside detected regions
        edge_density = 0
        for contour in confirmed_potholes:
            # Create mask for this contour
            mask = np.zeros_like(edges)
            for px, py in contour:
                if 0 <= py < mask.shape[0] and 0 <= px <
mask.shape[1]:
                    mask[py, px] = 1

            # Dilate mask to fill interior
            mask = manual_dilate(mask, kernel_size=3)

            # Count edge pixels inside mask
            edge_pixels_in_mask = np.sum((edges > 0) & (mask > 0))
            contour_area_val = contour_area(contour)

            if contour_area_val > 0:
                edge_density += edge_pixels_in_mask / contour_area_val

        if confirmed_potholes:
            edge_density /= len(confirmed_potholes)

        # Feature 1: Area ratio
        area_ratio = total_pothole_area / total_area

        # Feature 2: Number of detected potholes
        count_factor = min(len(confirmed_potholes) / 5, 1.0)

        # Feature 3: Edge density
        edge_factor = min(edge_density / 0.2, 1.0)

        # Calculate composite score
        pothole_score = 0.5 * area_ratio + 0.3 * count_factor + 0.2 *
edge_factor
        pothole_score = min(pothole_score * 5, 1.0)

    # Draw confirmed potholes on the result image
    for contour in confirmed_potholes:
```

```
        # Draw contour as red line
        for i in range(len(contour)):
            x1, y1 = contour[i]
            x2, y2 = contour[(i + 1) % len(contour)]

            # Draw line segment if within bounds
            if (0 <= x1 < result_image.shape[1] and 0 <= y1 <
result_image.shape[0] and
                0 <= x2 < result_image.shape[1] and 0 <= y2 <
result_image.shape[0]):
                # Draw a thick line in red
                # Basic Bresenham line algorithm
                dx = abs(x2 - x1)
                dy = -abs(y2 - y1)
                sx = 1 if x1 < x2 else -1
                sy = 1 if y1 < y2 else -1
                err = dx + dy

                while True:
                    # Set pixel to red if in bounds
                    if (0 <= x1 < result_image.shape[1] and 0 <= y1 <
result_image.shape[0]):
                        result_image[y1, x1] = [0, 0, 255]  # Red in
RGB

                    if x1 == x2 and y1 == y2:
                        break

                    e2 = 2 * err
                    if e2 >= dy:
                        if x1 == x2:
                            break
                        err += dy
                        x1 += sx
                    if e2 <= dx:
                        if y1 == y2:
                            break
                        err += dx
                        y1 += sy

    # Add text indicating pothole presence
    has_pothole = pothole_score > 0.3
    text = f"Pothole: {'YES' if has_pothole else 'NO'}
({pothole_score:.2f})"

    # Add text to image using simple method
    if has_pothole:
        # Put a black rectangle for text background
        for y in range(10, 40):
            for x in range(10, 250):
```

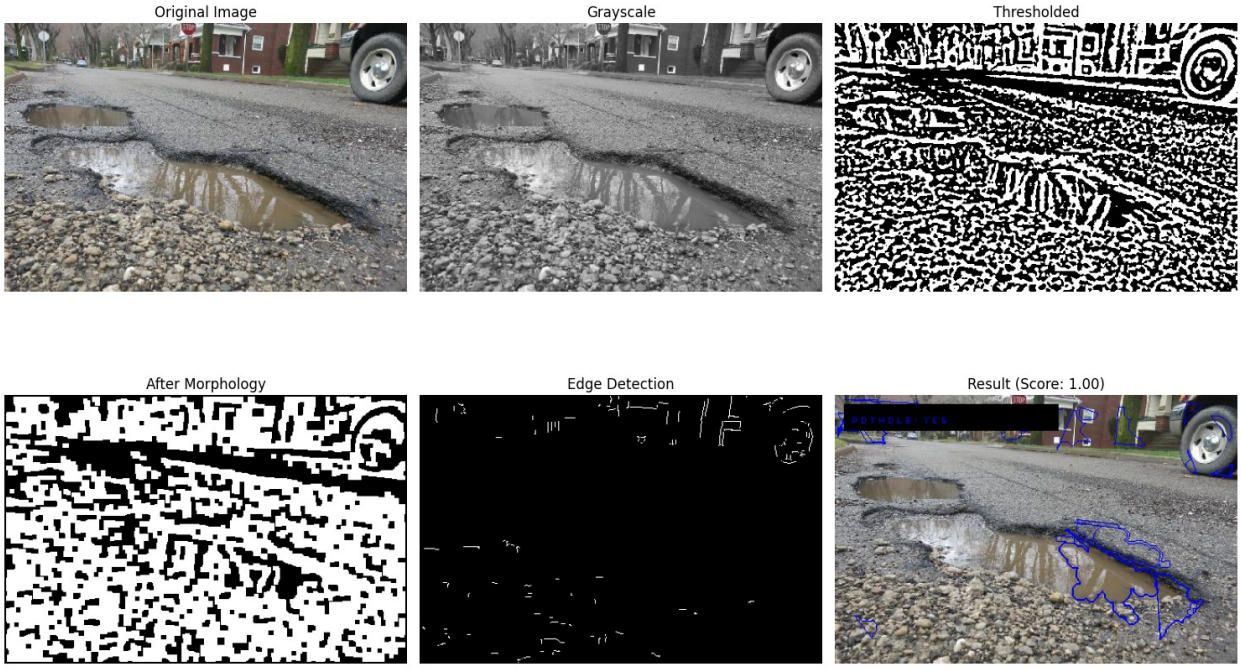
```
                if 0 <= y < result_image.shape[0] and 0 <= x <
result_image.shape[1]:
                    result_image[y, x] = [0, 0, 0]

        # Put text (simplified)
        font_color = [0, 0, 255]  # Red
        font_pixels = [
            # 'P'
            (20, 25), (21, 25), (22, 25), (23, 25), (20, 26), (24,
26),
            (20, 27), (24, 27), (20, 28), (24, 28), (20, 29), (21,
29),
            (22, 29), (23, 29), (20, 30), (20, 31),
            # 'O'
            (30, 25), (31, 25), (32, 25), (33, 25), (30, 26), (34,
26),
            (30, 27), (34, 27), (30, 28), (34, 28), (30, 29), (34,
29),
            (30, 30), (34, 30), (30, 31), (31, 31), (32, 31), (33,
31),
            # 'T'
            (40, 25), (41, 25), (42, 25), (43, 25), (44, 25), (42,
26),
            (42, 27), (42, 28), (42, 29), (42, 30), (42, 31),
            # 'H'
            (50, 25), (54, 25), (50, 26), (54, 26), (50, 27), (54,
27),
            (50, 28), (51, 28), (52, 28), (53, 28), (54, 28), (50,
29),
            (54, 29), (50, 30), (54, 30), (50, 31), (54, 31),
            # 'O'
            (60, 25), (61, 25), (62, 25), (63, 25), (60, 26), (64,
26),
            (60, 27), (64, 27), (60, 28), (64, 28), (60, 29), (64,
29),
            (60, 30), (64, 30), (60, 31), (61, 31), (62, 31), (63,
31),
            # 'L'
            (70, 25), (70, 26), (70, 27), (70, 28), (70, 29), (70,
30),
            (70, 31), (71, 31), (72, 31), (73, 31), (74, 31),
            # 'E'
            (80, 25), (81, 25), (82, 25), (83, 25), (84, 25), (80,
26),
            (80, 27), (80, 28), (81, 28), (82, 28), (83, 28), (80,
29),
            (80, 30), (80, 31), (81, 31), (82, 31), (83, 31), (84,
31),
            # ':'
```

```
            (90, 27), (90, 29),
            # 'Y'
            (100, 25), (104, 25), (101, 26), (103, 26), (102, 27),
(102, 28),
            (102, 29), (102, 30), (102, 31),
            # 'E'
            (110, 25), (111, 25), (112, 25), (113, 25), (110, 26),
(110, 27),
            (110, 28), (111, 28), (112, 28), (110, 29), (110, 30),
(110, 31),
            (111, 31), (112, 31), (113, 31),
            # 'S'
            (120, 25), (121, 25), (122, 25), (123, 25), (120, 26),
(120, 27),
            (120, 28), (121, 28), (122, 28), (123, 28), (123, 29),
(123, 30),
            (120, 31), (121, 31), (122, 31), (123, 31)
        ]

        for px, py in font_pixels:
            if 0 <= py < result_image.shape[0] and 0 <= px <
result_image.shape[1]:
                result_image[py, px] = font_color

    # Display debug images if requested
    if debug:
        # Create a figure with subplots
        fig, axes = plt.subplots(2, 3, figsize=(15, 10))

        # Original image
        axes[0, 0].imshow(image)
        axes[0, 0].set_title('Original Image')
        axes[0, 0].axis('off')

        # Grayscale
        axes[0, 1].imshow(gray, cmap='gray')
        axes[0, 1].set_title('Grayscale')
        axes[0, 1].axis('off')

        # Thresholded
        axes[0, 2].imshow(binary, cmap='gray')
        axes[0, 2].set_title('Thresholded')
        axes[0, 2].axis('off')

        # After Morphology
        axes[1, 0].imshow(opening, cmap='gray')
        axes[1, 0].set_title('After Morphology')
        axes[1, 0].axis('off')

        # Edge Detection
```

```
        axes[1, 1].imshow(edges, cmap='gray')
        axes[1, 1].set_title('Edge Detection')
        axes[1, 1].axis('off')

        # Result
        axes[1, 2].imshow(result_image)
        axes[1, 2].set_title(f'Result (Score: {pothole_score:.2f})')
        axes[1, 2].axis('off')

        plt.tight_layout()
        plt.show()

    return has_pothole, pothole_score, result_image

def process_dataset(pothole_dir, no_pothole_dir):

    true_positives = 0
    false_positives = 0
    true_negatives = 0
    false_negatives = 0

    # Process pothole images
    for filename in os.listdir(pothole_dir):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
            image_path = os.path.join(pothole_dir, filename)
            has_pothole, score, _ = detect_potholes(image_path)

            if has_pothole:
                true_positives += 1
            else:
                false_negatives += 1

    # Process non-pothole images
    for filename in os.listdir(no_pothole_dir):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
            image_path = os.path.join(no_pothole_dir, filename)
            has_pothole, score, _ = detect_potholes(image_path)

            if has_pothole:
                false_positives += 1
            else:
                true_negatives += 1

def main():
    """
    Main function to demonstrate pothole detection algorithm.

    This function shows three use cases:
    1. Detect potholes in a single image with debugging visualization
    2. Process a batch of images (both with and without potholes)
    3. Evaluate the algorithm's performance on a dataset
```

```
    """
    import os

    # Case 1: Process a single image with visualization
    print("Case 1: Analyzing a single image with debug visualization")
    sample_image_path = "/content/drive/MyDrive/potholes21.png"  #
Replace with your image path
    has_pothole, score, result_image =
detect_potholes(sample_image_path, debug=True)
    print(f"Pothole detected: {has_pothole}, Score: {score:.2f}")

    # Optional: Save the result image
    plt.imsave("pothole_result.jpg", result_image)

    # Case 2: Process multiple images
    print("\nCase 2: Processing multiple images")
    test_images = [
        "/content/drive/MyDrive/potholes21.png",
        "/content/drive/MyDrive/potholes23.png",

    ]

    for img_path in test_images:
        print(f"\nProcessing {os.path.basename(img_path)}")
        has_pothole, score, _ = detect_potholes(img_path)
        print(f"Pothole detected: {has_pothole}, Score: {score:.2f}")



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

```
Case 1: Analyzing a single image with debug visualization
```




```
Pothole detected: True, Score: 1.00

Case 2: Processing multiple images

Processing potholes21.png
Pothole detected: True, Score: 1.00

Processing potholes23.png
Pothole detected: True, Score: 1.00

Case 3: Evaluating algorithm performance on dataset
```