WoundCare : Personalized wound treatment consultant driven by YOLO11 AI image recognition and Deepseek R1

G

Group Members: FONG KUN FAI, ZENG QIAO HUI
Advising Teacher: Mr. HO IO FAI
School: Fong Chong School of Taipa, Macao SAR CHINA
Email Address: roy24245@icloud.com

Table of Contents

I. Scene Analysis

(a) Investigation and Analysis Process

According to survey data from the Hong Kong Department of Health, injuries in daily life are highly prevalent. Incidents of injury frequently occur in a variety of settings, including homes, streets, commercial establishments, and sports venues. In situations where there are no medical professionals or individuals with adequate medical knowledge present, there is a significant risk of improper wound management. Failure to treat wounds correctly and promptly can lead to serious health consequences, such as sepsis or tetanus. This issue is particularly acute in regions with limited access to medical resources, where the probability of improper wound care is even higher. Therefore, the presence of a wound care assistant is especially crucial in such circumstances.

(b) Literature Review of Survey Research

In recent years, traumatic injuries have become a significant issue in the field of global public health. Among these, falls (39.4%), sprains (26.2%), and contusions (13.3%) constitute the primary causes of injury . Notably, 27.4% of injuries occur in the home environment, and the uneven distribution of medical resources exposes economically disadvantaged regions to a higher risk of wound-related complications, including infection, delayed healing, and even systemic threats such as sepsis . Against this backdrop, the development of intelligent wound assessment technologies is crucial for enhancing diagnostic efficiency and reducing the burden on healthcare systems.

Traditional clinical assessment relies on visual inspection combined with standardized scales, supplemented by digital planar measurements, alginate casting, and biochemical testing . However, these methods exhibit significant limitations: first, contact-based measurements may disrupt the wound microenvironment and increase the risk of cross-infection; second, manual interpretation is susceptible to subjective bias, especially when quantifying parameters such as exudate volume, necrotic tissue proportion, and granulation maturity in complex wounds. Studies have shown that wound area measurements based solely on visual assessment can have error rates of 15%–20%, and are insufficiently sensitive to deep tissue injuries .

Image analysis systems based on convolutional neural networks (CNNs) have enabled the automated extraction and classification of wound characteristics. For example, a multimodal imaging integration system released in 2024 combines color imaging, t hermal imaging, and 3D depth sensing data. Utilizing a residual network (ResNet) architecture, it extracts 136 characteristic parameters, including wound edges, exudate distribution, and peripheral skin temperature gradients . In clinical trials, this system demonstrated a wound staging accuracy rate of 92.3%, representing a 37 percentage point improvement over traditional methods .

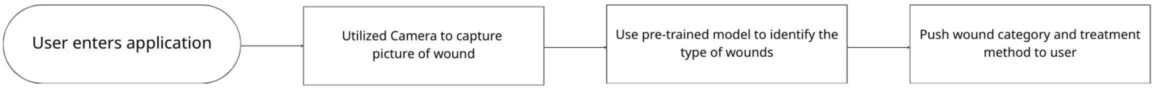
In the domain of area measurement, a laser-assisted deep learning model developed in 2022 innovatively integrated prior two-dimensional graphic calibration techniques. By establishing a nonlinear regression model between pixel density and shooting height, the system achieved wound area measurement errors of less than 2.5% without the need for reference objects, and was able to transmit data in real time to electronic medical record systems . Notably, this technology successfully overcomes measurement deviations caused by perspective distortion in curved wounds, which is particularly important for the accurate assessment of wounds on limbs .

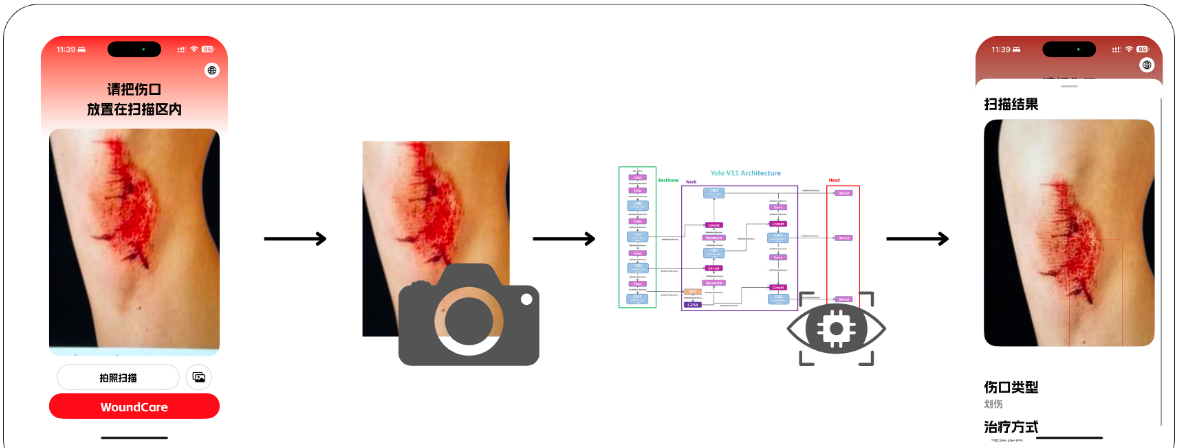
Prospective studies on postoperative incision healing monitoring have shown that AI systems can identify 87.6% of potential infection cases 3–5 days in advance. This is achieved by analyzing changes in microvascular density around the incision (ΔMVD 15%) and abnormalities in epidermal growth factor concentration gradients ($p<0.01$), enabling early warning . In resource-limited settings, mobile terminals equipped with this technology have increased the correct wound management rate among primary healthcare workers from 58.2% to 89.7%, while reducing the average assessment time from 23 minutes to 4.5 minutes .

Current technological bottlenecks focus on optimizing multimodal data fusion algorithms, particularly in effectively integrating near-infrared spectral tissue oxygenation data with visible light image features . The latest research in 2025 has attempted to introduce graph neural networks (GNNs) to establish dynamic models of the wound microenvironment, enabling prediction of healing trends within 72 hours ($R^2 = 0.89$). Furthermore, the development of embedded medical devices compliant with ISO-13485 standards, the establishment of cross-institutional wound image databases, and the refinement of FDA/CE certification processes will be key breakthroughs for the commercialization of these technologies .

LIDAR sensors, by emitting near-infrared laser pulses and receiving their reflected signals, can accurately measure the three-dimensional spatial distance of target objects, generating high-density point cloud data. Compared to traditional 2D image measurement, LIDAR technology overcomes the limitations of planar imaging, enabling precise capture of wound depth information and three-dimensional contours. This is especially suitable for measuring irregular and curved surface wounds.

II. Project Proposal





Credit：https://medium.com/@nikhil-rao-20/yolov11-explained-next-level-object-detection-with-enhanced-speed-and-accuracy-2dbe2d376f71

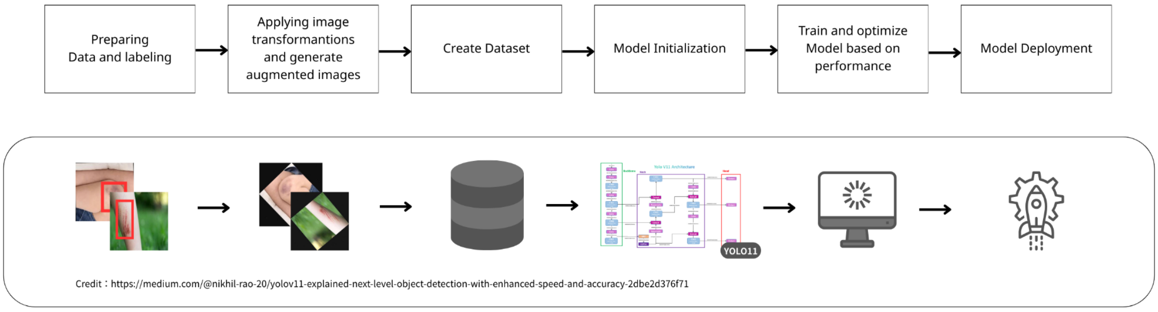
(a) Key Innovation

The project is divided into a mobile application and a physical device, both offering similar functionalities. First, the camera is used to capture images of the wound. The wound model, trained with YOLO v11, identifies the wound and provides appropriate treatment recommendations in either voice or text format based on the wound type. The application is deeply integrated with DeepSeek, allowing users to interact with the "e-Heal" chatbot while connected to the internet. Users can inquire about wound types, wound care methods, or other related questions. Based on the user’s specific situation, the system conducts wound assessments and offers more detailed diagnostic and treatment suggestions.

(b) Design Concept and Implementation Plan

The project is an AI image recognition-driven personalized wound management consultant. The entire solution consists of mobile application. The mobile version is an application that delivers results in text format.

(c) AI Model Training



The development process begins with the selection of an appropriate pre-trained model and preparation of a clearly annotated wound image dataset. Data augmentation techniques are employed to enhance the model's generalization capabilities. Subsequently, the final classification layer is removed and new layers are added to adapt the model specifically for wound classification tasks, while certain layers are frozen to preserve the pre-trained weights. Following model training, performance is evaluated on a validation dataset to ensure ac curate identification of various wound types. Finally, the model is deployed to the application, enabling real -time classification functionality

(1)Training Data Collection and Compilation

Due to the challenges in acquiring wound-related data in real-world environments, our projec t employs a diversified data collection strategy. Initially, we gathered relevant wound imagery through public internet channels, including existing public datasets and search engine querie s, to conduct supplementary data collection. Concurrently, this research collaborated with ou r school's medical staff to collect wound photographs from students and faculty members wh o sustained injuries, with appropriate consent, thereby establishing a proprietary wound imag e dataset specific to this project.

Currently, we have assembled 7,668 original wound images, categorized according to six wo und types plus normal skin classification. Considering the optimal input dimensions of 640×6 40 pixels for the YOLO11 model, all data has been standardized to these dimensions to enh ance training efficiency and effectiveness.

To further improve the model's accuracy and generalization capabilities, we implemented various data augmentation techniques, including:

- Angle adjustments (range: -15° to +15°)
- Horizontal and vertical flips
- Brightness adjustments (amplitude: ±15%)
- Addition of Gaussian noise (standard deviation 0.1 pixel)

Following these data augmentation procedures, the dataset expanded from the original 7,668 images to 45,271 images, significantly enhancing the model's ability to capture sample features and overall performance during the training process.

(2)Model Training

Given our project's high requirements for both accuracy and speed in wound recognition and detection, we selected YOLO11 as the core model. YOLO11 offers excellent object detection precision and efficient inference speed, effectively meeting the dual requirements of real-time performance and accuracy necessary for clinical wound image analysis.

YOLO11 surpasses previous generations of YOLO series models in architectural innovation, performance enhancement, and application flexibility. Its efficient feature extraction, optimized inference speed, and multi-task support capabilities make it one of the preferred models in the current field of image recognition.

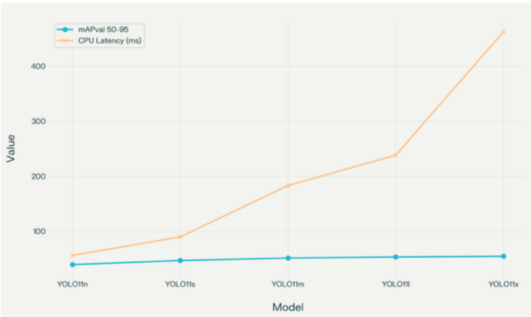
YOLO11 incorporates multiple optimizations in its network architecture. Both the Backbone and Neck networks have been redesigned, significantly enhancing feature extraction capabilities. Specifically, YOLO11 replaces the C2f structure from previous generations with C3K2 modules, strengthening the ability to capture image details. Additionally, the model introduces a C2PSA attention mechanism after the SPPF (Spatial Pyramid Pooling-Fast) module, further improving perception and selectivity of key visual information. The detection head section o ptimizes the convolutional structure, enhancing inference efficiency. These structural innovations make YOLO11's detection performance more robust in multi-object, occlusion, and complex scene scenarios.

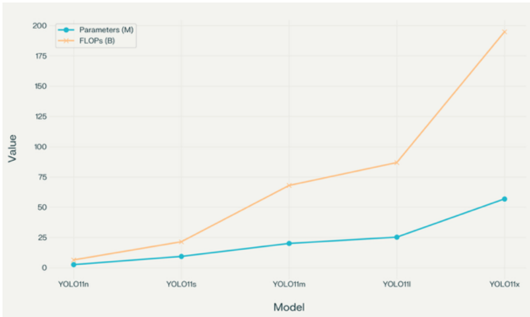
YOLO11 has achieved significant improvements in model efficiency and inference speed. According to official experimental results, YOLO11 maintains or even improves accuracy while significantly reducing parameter count and computational requirements. For example, the YOLO11m model achieves higher mean Average Precision (mAP) on the COCO dataset compared to YOLOv8m, with approximately 22% fewer parameters. Furthermore, YOLO11's inference speed is approximately 2% faster than YOLOv10, with particularly notable performance on CPU platforms, demonstrating its potential for application on resource-constrained devices. This high-efficiency, low-latency characteristic is particularly crucial for real-time image processing and edge computing scenarios.

YOLO11 extends support for various visual tasks, including object detection, instance segmentation, keypoint pose estimation, oriented object detection, classification, and object tracking. This unified multi-task framework simplifies application development processes and enhances the model's versatility and scalability. YOLO11 also offers multiple model scales (such as nano, small, medium, large, xlarge), allowing users to select appropriate model weights according to actual requirements, enabling flexible trade-offs between speed and precision.

Model	**Input Size**	mAP val 50-95	**CPU Inference Latency (ms)**	**GPU Inference Latency (ms, T4)**	**Parameters (M)**	FLOPs(B)
YOLO11n	640	39.5	56.1	1.5	2.6	6.5
YOLO11s	640	47.0	90.0	2.5	9.4	21.5

YOLO11m	640	51.5	183.2	4.7	20.1	68.0
YOLO11l	640	53.4	238.6	6.2	25.3	86.9
YOLO11x	640	54.7	462.8	11.3	56.9	194.9





Based on the comprehensive performance evaluation of the YOLO11 series models (as illustrated in the above graphs), this research conducted model selection tailored to the specific requirements of wound detection tasks. After weighing key indicators of computational efficiency against detection accuracy, the YOLO11l model was ultimately selected as the core architecture.

(d) Application Program

Operational Flow:

Upon first launching the application, the system requests camera, notification, and file access permissions. Once in the mobile application, users select their preferred language and position the smartphone camera toward the wound. By pressing the "Capture Scan" button or uploading an existing photo, users receive a textual identification of the wound type along with appropriate treatment methods within approximately 1.5 seconds.

Compatibility:

The application is compatible with both iOS and Android operating systems.

Model Loading Architecture:

- iOS application: The model loads when the user opens the application
- Android application: The model loads after the user captures a photograph

Development Languages:

- iOS version is compiled using Swift
- Android version is compiled using Kotlin

Advanced Integration:

The mobile version features deep integration with DeepSeek technology:

- In connected status, users can access the "e-Heal" chatroom via button press to inquire about wound types, treatment methods, or other relevant questions, providing services more tailored to user needs
- While online, users can enter the wound assessment section and select options that match their condition to receive more detailed wound diagnosis and treatment recommendations
-

Severity Measurement Capability:

On iOS devices equipped with LiDAR sensors, the application leverages distance detection data provided through ARKit to calculate wound surface area. This measurement helps determine wound severity based on size parameters.

(e) Product Advantages

- **User-Friendly Interface** (single-click operation)
- **Rapid Processing** (approximately 1.5 seconds loading time after image capture)
- **High Accuracy** (wound detection average accuracy rate of [value])
- **Offline Functionality** (both the application and physical device can operate without internet connectivity, allowing for use in various environments)
- **Enhanced Online Capabilities** (when connected to the internet, the iOS version integrates with DeepSeek to provide comprehensive wound assessment and AI chatroom functionality, delivering more detailed wound diagnosis and treatment recommendations)

III. Innovative Features

(a) Key Innovation Points

- AI-Powered Recognition Technology for efficient identification of wound types
- Pre-trained wound recognition model using the advanced YOLO11 architecture
- Intelligent medical device for wound detection and analysis
- Solution for underserved communities with limited access to medical resources
- Deep integration with DeepSeek as a virtual physician, enhancing user assessment of wound conditions
- LiDAR sensor technology for precise wound area calculation through distance measurement

(b) Process and Program code

(1) Process Overview

I. **User Operation (`ContentView.swift`):**
   i. User taps the "Photo Scan" button.
   ii. `ContentView` internally calls the `captureImage()` method.

II. **Image Capture (`CameraManager.swift`):**
   i. `ContentView.captureImage()` calls `cameraManager.capturePhoto { image, error in ... }`.
   ii. `CameraManager.capturePhoto()`: Uses the `capturePhoto(with:delegate:)` method of an `AVCapturePhotoOutput` instance to take a photo.
   iii. The captured `UIImage` (or error) is returned asynchronously via the `photoOutput(_:didFinishProcessingPhoto:error:)` delegate method of `AVCapturePhotoCaptureDelegate`.

III. **Image Classification (`ContentView.swift` -> `WoundClassifier.swift`):**
   i. `ContentView` receives the `UIImage` in the callback of `cameraManager.capturePhoto`.
   ii. Assigns the captured image to `@State var capturedImage`.
   iii. Calls the `classifier.classify(image)` method, passing the image to `WoundClassifier` for analysis.
   iv. `WoundClassifier.classify(image: UIImage)`:
      1. Internally, the image is first preprocessed: resized to the model's expected input size (e.g., 640x640), letterboxed to maintain aspect ratio, and then converted to `CVPixelBuffer`.

2. Creates a VNCoreMLRequest using a preloaded Core ML model (YOLOv8).
3. Creates a VNImageRequestHandler and performs model inference using handler.perform([request]).

v. WoundClassifier.processYOLOResults(request:error:) (as a callback for VNCoreMLRequest):

1. Parses the raw output of the model (usually a multi-dimensional array).
2. Iterates through the predictions, extracting the bounding box (box), confidence, and class index for each detected object.
3. Converts class indices to human-readable class names (e.g., "Abrasion", "Hematoma").
4. Selects the detection with the highest confidence from all detections above the confidence threshold as bestDetection.
5. Updates @Published properties such as self.classification, self.confidence, and self.bestDetection.
6. Finally, calls the self.onClassificationComplete?(self.classification, self.confidence) closure to notify observers (usually ContentView) that classification is complete.

IV. **Result Display and History (ContentView.swift):**

i. In ContentView, the classifier.onClassificationComplete callback is triggered (set up within the captureImage method).

ii. In this callback, historyManager.saveStandardScan(image:capturedImage, classification: ..., confidence: ...) is called to save the result to history.

iii. Simultaneously, a local notification sendNotification(...) is sent to inform the user of the identification result.

iv. Sets @State var showingResult = true, which triggers the display of the ResultView card.

V. **Result View (ResultView.swift):**

i. ResultView is presented, receiving data such as capturedImage and classifier (as an @ObservedObject).

ii. In its .onAppear or after listening for the ClassificationComplete notification via NotificationCenter, it calls the internal getWoundTreatmentAdvice(woundType: classifier.classification) method.

iii. ResultView.getWoundTreatmentAdvice() calls deepseekService.getWoundTreatmentAdvice(woundType: ..., language: languageManager.currentLanguage, ...) to asynchronously fetch AI-powered smart suggestions.

iv. ResultView internally uses classifier.drawAnnotations(on: image) to draw the bounding box and label of bestDetection on the passed image, and then displays this annotated image.

v. It also displays the wound type and local treatment suggestions obtained from classifier, as well as AI suggestions from deepseekService.

(2) Key Code

The following Swift code provides a more detailed look at the `WoundClassifier`, specifically expanding the `processYOLOResults` method to illustrate how raw model output from a YOLOv8 model might be parsed. It includes placeholder logic for parsing the tensor and mapping class indices to labels.

```swift
// WoundClassifier.swift
class WoundClassifier: ObservableObject {
    @Published var classification: String = "N/A"
    @Published var confidence: Float = 0.0
    @Published var bestDetection: WoundDetection? = nil // Updated type

    var onClassificationComplete: ((_ classification: String, _ confidence: Float) -> Void)?
    private var yoloRequest: VNCoreMLRequest?

    // Existing init() or a new one to setup the model request
    init() {
        // Load your Core ML model and create the VNCoreMLRequest
        // For example:
        // guard let modelURL = Bundle.main.url(forResource: "YOLOv8WoundModel", withExtension: "mlmodelc") else {
        //     fatalError("Failed to load Core ML model.")
        // }
        // do {
        //     let visionModel = try VNCoreMLModel(for: MLModel(contentsOf: modelURL))
        //     self.yoloRequest = VNCoreMLRequest(model: visionModel, completionHandler: processYOLOResults)
        //     // Set any specific request properties if needed
        //     // self.yoloRequest?.imageCropAndScaleOption = .scaleFill
        // } catch {
        //     fatalError("Failed to create VNCoreMLModel: \(error)")
        // }
    }

    func classify(_ image: UIImage) {
        guard let yoloRequest = self.yoloRequest else {
            print("YOLO request not initialized.")
            return
        }
        guard let pixelBuffer = image.toCVPixelBuffer(width: 640, height: 640) else { // Ensure correct dimensions for your model
            print("Failed to convert UIImage to CVPixelBuffer.")
            DispatchQueue.main.async {
                self.classification = "Preprocessing Failed"
                self.confidence = 0.0
                self.bestDetection = nil
                self.onClassificationComplete?("Preprocessing Failed", 0.0)
            }
            return
        }
        let handler = VNImageRequestHandler(cvPixelBuffer: pixelBuffer, orientation: .up) // Assuming image is upright
        do {
            try handler.perform([yoloRequest]) // Calls VNCoreMLRequest completion (processYOLOResults)
```

```swift
    } catch {
        print("Failed to perform Vision request: \(error.localizedDescription)")
        DispatchQueue.main.async {
            self.classification = "Inference Error"
            self.confidence = 0.0
            self.bestDetection = nil
            self.onClassificationComplete?("Inference Error", 0.0)
        }
    }
}

private func processYOLOResults(_ request: VNRequest, error: Error?) {
    guard error == nil else {
        print("Vision request failed with error: \(error!.localizedDescription)")
        DispatchQueue.main.async {
            self.classification = "Error"
            self.confidence = 0.0
            self.bestDetection = nil
            self.onClassificationComplete?("Error", 0.0)
        }
        return
    }

    // This part is highly dependent on the specific YOLOv8 model's output format.
    // Some models output VNRecognizedObjectObservation directly.
    // Others output raw feature values (MLMultiArray) that need manual parsing.

    // Option 1: If your model is post-processed to output VNRecognizedObjectObservation
    if let results = request.results as? [VNRecognizedObjectObservation] {
        var highestConfidenceObservation: VNRecognizedObjectObservation? = nil
        var maxConfidence: Float = 0.0

        for observation in results {
            // Assuming the first label is the most relevant one.
            if let firstLabel = observation.labels.first, firstLabel.confidence > maxConfidence {
                maxConfidence = firstLabel.confidence
                highestConfidenceObservation = observation
            }
        }

        DispatchQueue.main.async {
            if let bestObs = highestConfidenceObservation, let label = bestObs.labels.first {
                self.classification = label.identifier
                self.confidence = label.confidence
                self.bestDetection = WoundDetection(
                    boundingBox: bestObs.boundingBox, // This is normalized (0-1)
                    confidence: label.confidence,
                    label: label.identifier,
                    classIndex: 0 // Or map identifier to index if needed
                )
            } else {
```

```
                self.classification = "No Wound Detected"
                self.confidence = 0.0
                self.bestDetection = nil
            }
            self.onClassificationComplete?(self.classification, self.confidence)
        }
    }
    // Option 2: If your model outputs raw MLMultiArray (more common for unmodified YOLO)
    else if let results = request.results as? [VNCoreMLFeatureValueObservation],
            let outputTensor = results.first?.featureValue.multiArrayValue {

        let detections = parseYOLOv8Output(tensor: outputTensor)

        var highestConfidence: Float = 0.0
        var bestOverallDetection: WoundDetection? = nil

        for detection in detections {
            if detection.confidence > highestConfidence && detection.confidence > 0.5 { // Example threshold
                highestConfidence = detection.confidence
                bestOverallDetection = detection
            }
        }

        DispatchQueue.main.async {
            if let best = bestOverallDetection {
                self.classification = best.label
                self.confidence = best.confidence
                self.bestDetection = best
            } else {
                self.classification = "No Wound Detected"
                self.confidence = 0.0
                self.bestDetection = nil
            }
            self.onClassificationComplete?(self.classification, self.confidence)
        }
    } else {
        print("Failed to interpret Vision request results. Neither VNRecognizedObjectObservation nor VNCoreMLFeatureValueOb
servation found or parsable.")
        DispatchQueue.main.async {
            self.classification = "Processing Failed"
            self.confidence = 0.0
            self.bestDetection = nil
            self.onClassificationComplete?("Processing Failed", 0.0)
        }
    }
}

// Dummy parseYOLOv8Output function and WoundDetection struct for illustration
// Replace with your actual parsing logic and data structures.
private func parseYOLOv8Output(tensor: MLMultiArray) -> [WoundDetection] {
    // This is highly dependent on your YOLOv8 model's exact output format.
```

```
    // It might involve reshaping the tensor, iterating through detections,
    // applying non-maximum suppression, and converting coordinates.
    // Example: tensor shape could be (1, num_attributes, num_detections) or (1, num_detections, num_attributes)
    // num_attributes typically includes (x_center, y_center, width, height, object_confidence, class_probs...)

    var detectedObjects: [WoundDetection] = []
    // The following is a generic placeholder and needs to be adapted.
    // Let's assume output is (1, 8400, 5 + num_classes) for an example YOLO model
    // where 8400 is number of proposals, 5 is (cx, cy, w, h, obj_conf)
    let numProposals = tensor.shape[1].intValue
    let numAttributesPerProposal = tensor.shape[2].intValue
    let numClasses = numAttributesPerProposal - 5 // Assuming 5 for box + obj_conf

    for i in 0..<numProposals {
        let basePointer = UnsafeMutableBufferPointer<Float32>(striding: tensor.strides[1].intValue, count: numAttributesPerProposal, UnsafeMutableRawPointer(tensor.dataPointer).advanced(by: i * tensor.strides[1].intValue * MemoryLayout<Float32>.stride))

        let cx = basePointer[0]
        let cy = basePointer[1]
        let w = basePointer[2]
        let h = basePointer[3]
        let objConfidence = basePointer[4]

        if objConfidence < 0.5 { continue } // Object confidence threshold

        var maxClassProb: Float = 0.0
        var classIndex: Int = -1

        for j in 0..<numClasses {
            let classProb = basePointer[5+j]
            if classProb > maxClassProb {
                maxClassProb = classProb
                classIndex = j
            }
        }

        let finalConfidence = objConfidence * maxClassProb
        if finalConfidence > 0.5 { // Final confidence threshold
            // Convert YOLO center_x, center_y, width, height to top-left x,y, width, height
            // These coordinates are usually normalized to the input image size (e.g., 640x640)
            let x = CGFloat(cx - w/2)
            let y = CGFloat(cy - h/2)
            let boundingBox = CGRect(x: x, y: y, width: CGFloat(w), height: CGFloat(h))

            let label = mapClassIndexToLabel(classIndex)

            detectedObjects.append(WoundDetection(boundingBox: boundingBox, confidence: finalConfidence, label: label, classIndex: classIndex))
        }
    }
    // Non-Maximum Suppression (NMS) should ideally be applied here to filter overlapping boxes.
```

```swift
    // This is often a separate step or can be part of the model's custom layers if exported that way.
    return nms(detections: detectedObjects) // Placeholder for NMS
}

// Placeholder NMS function
private func nms(detections: [WoundDetection], iouThreshold: Float = 0.45) -> [WoundDetection] {
    // Implement Non-Maximum Suppression logic here
    // 1. Sort detections by confidence (descending).
    // 2. Iterate through sorted detections:
    //    - Take the current highest confidence detection.
    //    - Remove all other detections that have an IoU (Intersection over Union) with it above the threshold.
    return detections // Return filtered detections
}


private func mapClassIndexToLabel(_ index: Int) -> String {
    let labels = ["Abrasion", "Hematoma", "Laceration", "Puncture", "Burn"] // Example labels
    if index >= 0 && index < labels.count {
        return labels[index]
    }
    return "Unknown"
}

// Helper to convert UIImage to CVPixelBuffer (from Apple's sample code or similar)
// You might have this in an extension or utility class
func imageToCVPixelBuffer(image: UIImage, width: Int, height: Int) -> CVPixelBuffer? {
    // ... implementation for converting UIImage to CVPixelBuffer of specific size ...
    // This involves creating a CVPixelBuffer, drawing the image into it (potentially resizing/letterboxing)
    return image.toCVPixelBuffer(width: width, height: height) // Assuming an extension like below
}

func drawAnnotations(on image: UIImage, detection: WoundDetection?) -> UIImage {
    let imageSize = image.size
    UIGraphicsBeginImageContextWithOptions(imageSize, false, image.scale)
    image.draw(at: .zero)
    guard let context = UIGraphicsGetCurrentContext(), let detection = detection else {
        UIGraphicsEndImageContext()
        return image
    }

    // Convert normalized bounding box to image coordinates
    let boundingBox = detection.boundingBox // This should be normalized (0-1)
    let rect = CGRect(
        x: boundingBox.origin.x * imageSize.width,
        y: boundingBox.origin.y * imageSize.height,
        width: boundingBox.width * imageSize.width,
        height: boundingBox.height * imageSize.height
    )

    context.setStrokeColor(UIColor.red.cgColor)
    context.setLineWidth(max(imageSize.width / 200, 2.0)) // Dynamic line width
```

```
        context.stroke(rect)

        // Draw label and confidence
        let text = String(format: "%@: %.2f", detection.label, detection.confidence)
        let attributes: [NSAttributedString.Key: Any] = [
            .font: UIFont.systemFont(ofSize: max(imageSize.width / 40, 12.0)), // Dynamic font size
            .foregroundColor: UIColor.white,
            .backgroundColor: UIColor.red.withAlphaComponent(0.7)
        ]
        let textSize = text.size(withAttributes: attributes)
        let textRect = CGRect(x: rect.origin.x, y: rect.origin.y - textSize.height - 2, width: textSize.width, height: textSize.height)
        text.draw(in: textRect, withAttributes: attributes)

        let annotatedImage = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()
        return annotatedImage ?? image
    }
}

// UIImage extension for CVPixelBuffer conversion (example)
extension UIImage {
    func toCVPixelBuffer(width: Int, height: Int) -> CVPixelBuffer? {
        let attrs = [
            kCVPixelBufferCGImageCompatibilityKey: kCFBooleanTrue,
            kCVPixelBufferCGBitmapContextCompatibilityKey: kCFBooleanTrue
        ] as CFDictionary
        var pixelBuffer: CVPixelBuffer?
        let status = CVPixelBufferCreate(kCFAllocatorDefault, width, height, kCVPixelFormatType_32ARGB, attrs, &pixelBuffer)
        guard status == kCVReturnSuccess, let buffer = pixelBuffer else {
            return nil
        }

        CVPixelBufferLockBaseAddress(buffer, CVPixelBufferLockFlags(rawValue: 0))
        let pixelData = CVPixelBufferGetBaseAddress(buffer)

        let rgbColorSpace = CGColorSpaceCreateDeviceRGB()
        guard let context = CGContext(
            data: pixelData,
            width: width,
            height: height,
            bitsPerComponent: 8,
            bytesPerRow: CVPixelBufferGetBytesPerRow(buffer),
            space: rgbColorSpace,
            bitmapInfo: CGImageAlphaInfo.noneSkipFirst.rawValue
        ) else {
            CVPixelBufferUnlockBaseAddress(buffer, CVPixelBufferLockFlags(rawValue: 0))
            return nil
        }

        // Scale and draw the image to fit the pixel buffer (letterboxing/aspect fill as needed)
        // This example scales to fill, potentially distorting aspect ratio.
```

```
        // For production, use a more sophisticated scaling (e.g., Vision's VNImageCropAndScaleOption)
        // or implement letterboxing manually.
        context.translateBy(x: 0, y: CGFloat(height))
        context.scaleBy(x: 1.0, y: -1.0)
        UIGraphicsPushContext(context)
        self.draw(in: CGRect(x: 0, y: 0, width: width, height: height))
        UIGraphicsPopContext()

        CVPixelBufferUnlockBaseAddress(buffer, CVPixelBufferLockFlags(rawValue: 0))
        return buffer
    }
}


// It's recommended to define shared data structures like WoundDetection globally or in a shared file.
// For illustrative purposes, its structure might be:
// struct WoundDetection: Identifiable { // Conforming to Identifiable if used in ForEach
//     let id = UUID() // Useful for SwiftUI lists
//     let boundingBox: CGRect // Normalized coordinates (0.0 to 1.0) relative to the input image size.
//     let confidence: Float
//     let label: String
//     let classIndex: Int      // Original index from the model's classes
// }
// Ensure @Published bestDetection in WoundClassifier is updated to type WoundDetection?
// Example: @Published var bestDetection: WoundDetection? = nil
```

X.2 LiDAR Enhanced Scan

This function utilizes the LiDAR sensor to measure distance and combines it with image analysis to calculate wound area and assess severity.

Process Overview:

1. **User Operation (`ContentView`)**:
   * User taps the "Enhanced Scan" button (only displayed if LiDAR is available).
   * Calls `startEnhancedScan()`.
2. **Initiate Scan (`ContentView` -> `EnhancedScanManager`)**:
   * `ContentView.startEnhancedScan()`:
     * Sets `showingScanProgress = true` to display the progress UI.
     * Calls `enhancedScanManager.performScan()`.
   * `EnhancedScanManager.performScan()`:
     * Checks for LiDAR support.
     * Calls `setupARSession()` to start `ARSession` and request depth data.
     * Starts `startDepthStabilizationProcess()` to stabilize depth readings.
3. **Data Capture and Processing (`EnhancedScanManager`)**:
   * `startDepthStabilizationProcess()`: Stabilizes the acquisition of `ARFrame` and distance using a timer and `takeSingleFrameCapture`.
   * After stabilization, calls `captureFrame()`, which in turn calls `processFrame(frame: ARFrame)`.
   * `processFrame()`:
     * Gets the image (`capturedImage`) and average distance (`distanceCopy`) from `ARFrame`.

        *   Calls `updateCameraIntrinsics()` to update camera intrinsic parameters.
        *   Instantiates `WoundClassifier` and calls `classifier.classify(image)`.
4. **Classification and Area Calculation (`EnhancedScanManager` + `WoundClassifier`)**:
    *   `WoundClassifier.classify()` follows the same process as the standard scan to find `bestDetection`.
    *   `EnhancedScanManager` in the `classifier.onClassificationComplete` callback:
        *   If `bestDetection` exists:
            *   Calls `classifier.drawAnnotations(on: image)` to generate an annotated image.
            *   Calls `self.calculateArea(boundingBox: bestDetection.box, distance: distanceCopy, ...)` to calculate the area.
            *   Calls `self.assessSeverity(area: areaResult)` to assess severity.
            *   Prepares `EnhancedScanResult`.
        *   If `bestDetection` does not exist, prepares an `EnhancedScanResult` indicating "not detected".
        *   Calls `self.scanCompletion?(.success(finalResult))` or `.failure()`.
        *   Calls `self.cleanup()` to stop `ARSession`.
5. **Result Display and History (`ContentView`)**:
    *   The callback of `ContentView.startEnhancedScan()` is triggered.
    *   Hides the progress view.
    *   If successful, updates `enhancedScanResult` and sets `showingEnhancedScanResult = true` to display `EnhancedScanResultView`.
    *   Calls `historyManager.saveLiDARScan(...)` to save the record.
    *   If failed, displays an error message.
6. **Result View (`EnhancedScanResultView`)**:
    *   Receives `EnhancedScanResult`.
    *   Displays the image (with annotations generated by `EnhancedScanManager` or drawn by the view itself based on `woundBoundingBox`).
    *   Displays wound classification, distance, area, and severity.
    *   Displays basic treatment advice, AI suggestions, "e-Consult" button, and "Seek Immediate Help" button.

Key Code Snippets (Detailed Illustration):

The `EnhancedScanManager.swift` snippet below is expanded to demonstrate a more complete, albeit still illustrative, implementation for LiDAR-enhanced scanning. It includes:
*   Basic `ARSession` setup to access depth data and camera frames.
*   Callbacks for `ARSessionDelegate` to receive frame updates and potentially camera intrinsics.
*   A more detailed `processFrame` method to convert `ARFrame`'s `capturedImage` to `UIImage` and initiate classification using the `WoundClassifier`.
*   An expanded `handleClassificationCompletion` callback (triggered by `WoundClassifier`) to integrate classification results with depth information for placeholder area and severity calculations. This highlights where 3D geometry and depth map processing would occur.
*   Placeholder functions for `getAverageDepthForBoundingBox`, `calculatePhysicalAreaFromDetection`, and `assessSeverity` to underscore the complex calculations and logic required for accurate measurements.

```swift
// EnhancedScanManager.swift
import ARKit
import RealityKit // Often used with ARKit, though not strictly necessary for session management
import Combine   // For @Published properties and managing asynchronous operations
import UIKit     // For UIImage

class EnhancedScanManager: NSObject, ObservableObject, ARSessionDelegate {
    @Published var scanProgress: Double = 0.0 // Example: 0.0 to 1.0
    @Published var currentDepthString: String = "N/A" // For displaying live depth info
```

```swift
    @Published var annotatedImage: UIImage? // The image with wound annotation and measurements
    @Published var isScanning: Bool = false

    private var arSession: ARSession?
    private var imageClassifier: WoundClassifier?
    private var cancellables = Set<AnyCancellable>()

    // Store latest camera parameters and depth data for calculations
    private var currentFrame: ARFrame?
    private var cameraIntrinsics: simd_float3x3?
    private var cameraResolution: CGSize?

    // Completion handler for the scan result
    var scanCompletion: ((Result<EnhancedScanResult, Error>) -> Void)?

    override init() {
        super.init()
        self.imageClassifier = WoundClassifier() // Assuming WoundClassifier is defined as in X.1

        // Subscribe to the classifier's completion
        imageClassifier?.$bestDetection // Or use the onClassificationComplete closure
            .receive(on: DispatchQueue.main)
            .sink { [weak self] detection in
                guard let self = self, self.isScanning, let detection = detection else { return }
                // This is called when classification is done for a frame
                self.handleClassificationCompletion(detection: detection)
            }
            .store(in: &cancellables)
    }

    func performScan(completion: @escaping (Result<EnhancedScanResult, Error>) -> Void) {
        self.scanCompletion = completion
        self.isScanning = true
        self.annotatedImage = nil // Reset previous scan image

        guard ARWorldTrackingConfiguration.isSupported else {
            completeScan(.failure(ScanError.arNotSupported))
            return
        }
        // Check for LiDAR availability if strictly required (though sceneDepth can work on non-LiDAR with less accuracy)
        // guard ARWorldTrackingConfiguration.supportsSceneReconstruction(.mesh) else {
        //     completeScan(.failure(ScanError.lidarNotSupported))
        //     return
        // }

        arSession = ARSession()
        arSession?.delegate = self

        let configuration = ARWorldTrackingConfiguration()
        if ARWorldTrackingConfiguration.supportsFrameSemantics(.sceneDepth) {
            configuration.frameSemantics.insert(.sceneDepth) // Request scene depth data
```

```swift
    } else {
        completeScan(.failure(ScanError.sceneDepthNotSupported))
        return
    }
    // configuration.sceneReconstruction = .mesh // If you need to build a 3D mesh of the environment

    arSession?.run(configuration)

    // Simulate a scan duration or trigger frame capture based on stability/user action
    // For this example, we'll rely on the ARSessionDelegate to provide frames.
    // You might have a button "Capture" or a timer.
    // Here, let's assume we want to process a frame after a short delay for stabilization.
    DispatchQueue.main.asyncAfter(deadline: .now() + 1.5) { [weak self] in
        self?.captureAndProcessCurrentFrame()
    }
}

private func captureAndProcessCurrentFrame() {
    guard self.isScanning, let frame = self.currentFrame ?? arSession?.currentFrame else {
        // If no frame available yet, or scan was stopped
        if self.isScanning { // if still scanning but no frame, might be an error
            completeScan(.failure(ScanError.noFrameAvailable))
        }
        return
    }
    processFrameForWound(frame: frame)
}

// ARSessionDelegate method
func session(_ session: ARSession, didUpdate frame: ARFrame) {
    self.currentFrame = frame // Store the latest frame
    self.cameraIntrinsics = frame.camera.intrinsics
    self.cameraResolution = frame.camera.imageResolution

    // Optionally, update live depth string for UI, e.g., depth at screen center
    if let sceneDepth = frame.sceneDepth {
        let depthMap = sceneDepth.depthMap
        // This requires converting screen point to depth map coordinates
        // For simplicity, let's skip live display update here, focus on capture.
    }
}

func session(_ session: ARSession, didFailWithError error: Error) {
    print("ARSession failed with error: \(error.localizedDescription)")
    completeScan(.failure(ScanError.arSessionFailed(error)))
}

private func processFrameForWound(frame: ARFrame) {
    guard let pixelBuffer = frame.capturedImage else {
        completeScan(.failure(ScanError.noImageInFrame))
        return
```

```swift
    }

    let ciImage = CIImage(cvPixelBuffer: pixelBuffer)
    let context = CIContext(options: nil)
    guard let cgImage = context.createCGImage(ciImage, from: ciImage.extent) else {
        completeScan(.failure(ScanError.imageConversionFailed))
        return
    }
    let capturedUIImage = UIImage(cgImage: cgImage, scale: 1.0, orientation: .right) // ARKit frames are landscape right
    self.annotatedImage = capturedUIImage // Store initial image

    // Pass to classifier
    imageClassifier?.classify(capturedUIImage)
    // Result will be handled by the sink observing imageClassifier.$bestDetection
}

private func handleClassificationCompletion(detection: WoundDetection) {
    guard self.isScanning,
          let currentFrame = self.currentFrame, // Use the stored frame corresponding to the classification
          let depthData = currentFrame.sceneDepth,
          let camIntrinsics = self.cameraIntrinsics,
          let camResolution = self.cameraResolution else {
        // Create a result even if some data is missing, but indicate issues
        let result = EnhancedScanResult(
            annotatedImage: self.annotatedImage ?? UIImage(), // Use current annotated or raw image
            woundType: detection.label,
            confidence: detection.confidence,
            distance: nil, // Mark as nil or error value
            estimatedArea: nil,
            severity: "Unknown (incomplete data)",
            woundBoundingBox: detection.boundingBox
        )
        completeScan(.success(result)) // Or failure if essential data is missing
        return
    }

    // 1. Get average distance to the wound using the depth map
    let averageDistance = getAverageDepthForBoundingBox(
        detection.boundingBox, // Normalized 0-1 coordinates from classifier
        depthMap: depthData.depthMap,
        depthConfidenceMap: depthData.confidenceMap, // Use confidence if available
        camera: currentFrame.camera // For unprojection if needed
    )

    // 2. Calculate Physical Area
    var physicalArea: Float? = nil
    if let dist = averageDistance {
        physicalArea = calculatePhysicalAreaFromDetection(
            detection: detection,
            distanceToWound: dist,
            cameraIntrinsics: camIntrinsics,
```

```
            cameraImageResolution: camResolution // Original image resolution for normalization reference
        )
    }

    // 3. Assess Severity
    let severity = assessSeverity(area: physicalArea, type: detection.label)

    // 4. Draw final annotations (including area, distance if available) on the image
    if let baseImage = self.annotatedImage { // Start with the captured image
        self.annotatedImage = drawEnhancedAnnotations(on: baseImage, detection: detection, distance: averageDistance, area: physicalArea)
    }

    let finalResult = EnhancedScanResult(
        annotatedImage: self.annotatedImage ?? UIImage(), // Should be the fully annotated one
        woundType: detection.label,
        confidence: detection.confidence,
        distance: averageDistance,
        estimatedArea: physicalArea,
        severity: severity,
        woundBoundingBox: detection.boundingBox
    )
    completeScan(.success(finalResult))
}

// Placeholder: Get average depth for the wound's bounding box
private func getAverageDepthForBoundingBox(_ normalizedBoundingBox: CGRect, depthMap: CVPixelBuffer, depthConfidenceMap: CVPixelBuffer?, camera: ARCamera) -> Float? {
    // This is a complex function:
    // 1. Convert normalized bounding box to pixel coordinates in the depth map.
    // 2. Iterate pixels in this region of the depth map.
    // 3. Read depth values (Float32 for sceneDepth, meters).
    // 4. Optionally filter by confidence from depthConfidenceMap (if available and sceneDepth).
    // 5. Average valid depth values. Handle NaNs or very large/small values.
    // 6. Unprojection might be needed for more accuracy if the wound surface is not perpendicular to camera.

    // Simplified placeholder:
    // Get depth at the center of the bounding box
    let depthWidth = CVPixelBufferGetWidth(depthMap)
    let depthHeight = CVPixelBufferGetHeight(depthMap)

    let centerX = Int(normalizedBoundingBox.midX * CGFloat(depthWidth))
    let centerY = Int(normalizedBoundingBox.midY * CGFloat(depthHeight))

    guard centerX >= 0 && centerX < depthWidth && centerY >= 0 && centerY < depthHeight else { return nil }

    CVPixelBufferLockBaseAddress(depthMap, .readOnly)
    defer { CVPixelBufferUnlockBaseAddress(depthMap, .readOnly) }

    if let baseAddress = CVPixelBufferGetBaseAddress(depthMap) {
        let bytesPerRow = CVPixelBufferGetBytesPerRow(depthMap)
```

```swift
            let buffer = baseAddress.assumingMemoryBound(to: Float32.self)
            let depthValue = buffer[centerY * (bytesPerRow / MemoryLayout<Float32>.stride) + centerX]
            return depthValue.isNaN ? nil : depthValue
        }
        return nil // Placeholder
    }

    // Placeholder: Calculate physical area
    private func calculatePhysicalAreaFromDetection(detection: WoundDetection, distanceToWound: Float, cameraIntrinsics: simd_
float3x3, cameraImageResolution: CGSize) -> Float? {
        // This uses pinhole camera model principles.
        // Assumes boundingBox in detection is normalized to cameraImageResolution.
        let fx = cameraIntrinsics[0,0] // Focal length in x (pixels)
        let fy = cameraIntrinsics[1,1] // Focal length in y (pixels)

        // Denormalize bounding box width and height to pixels on the image sensor
        let boxWidthInPixels = detection.boundingBox.width * cameraImageResolution.width
        let boxHeightInPixels = detection.boundingBox.height * cameraImageResolution.height

        // Physical width = (BoxWidthInPixels / fx) * DistanceToWound
        // Physical height = (BoxHeightInPixels / fy) * DistanceToWound
        let physicalWidth = (Float(boxWidthInPixels) / fx) * distanceToWound // meters
        let physicalHeight = (Float(boxHeightInPixels) / fy) * distanceToWound // meters

        let areaInSquareMeters = physicalWidth * physicalHeight
        return areaInSquareMeters * 10000 // Convert m^2 to cm^2
    }

    // Placeholder: Assess severity
    private func assessSeverity(area: Float?, type: String) -> String {
        guard let area = area else { return "Unknown (area not calculated)" }
        // Example logic (cm^2)
        if area > 50.0 { return "High" }
        if area > 10.0 { return "Medium" }
        if area > 0 { return "Low" }
        return "Not classified"
    }

    private func drawEnhancedAnnotations(on image: UIImage, detection: WoundDetection, distance: Float?, area: Float?) -> UIIma
ge {
        UIGraphicsBeginImageContextWithOptions(image.size, false, image.scale)
        image.draw(at: .zero)
        guard let context = UIGraphicsGetCurrentContext() else {
            UIGraphicsEndImageContext()
            return image
        }

        let imageSize = image.size
        let boundingBox = detection.boundingBox // Normalized
        let rect = CGRect(
            x: boundingBox.origin.x * imageSize.width,
```

```
            y: boundingBox.origin.y * imageSize.height,
            width: boundingBox.width * imageSize.width,
            height: boundingBox.height * imageSize.height
        )

        context.setStrokeColor(UIColor.cyan.cgColor) // Different color for enhanced scan
        context.setLineWidth(max(imageSize.width / 180, 2.5))
        context.stroke(rect)

        var textLines: [String] = []
        textLines.append(String(format: "%@: %.2f", detection.label, detection.confidence))
        if let d = distance { textLines.append(String(format: "Dist: %.2f m", d)) }
        if let a = area { textLines.append(String(format: "Area: %.1f cm²", a)) }

        let text = textLines.joined(separator: "\n")

        let attributes: [NSAttributedString.Key: Any] = [
            .font: UIFont.systemFont(ofSize: max(imageSize.width / 45, 10.0)),
            .foregroundColor: UIColor.black,
            .backgroundColor: UIColor.cyan.withAlphaComponent(0.7)
        ]

        let paragraphStyle = NSMutableParagraphStyle()
        paragraphStyle.alignment = .left
        let finalAttributes = attributes.merging([.paragraphStyle: paragraphStyle], uniquingKeysWith: { (current, _) in current })


        let textSize = text.boundingRect(with: CGSize(width: imageSize.width, height: .greatestFiniteMagnitude),
                                         options: .usesLineFragmentOrigin,
                                         attributes: finalAttributes,
                                         context: nil).size

        var textRectY = rect.origin.y - textSize.height - 5
        if textRectY < 0 { textRectY = rect.origin.y + rect.height + 5 } // Position below if no space above
        if textRectY + textSize.height > imageSize.height { textRectY = imageSize.height - textSize.height - 5} // Ensure it's within b
ounds


        let textRect = CGRect(x: rect.origin.x, y: textRectY, width: textSize.width + 10, height: textSize.height + 5)

        // Draw background for text
        let backgroundPath = UIBezierPath(roundedRect: textRect, cornerRadius: 5)
        (finalAttributes[.backgroundColor] as? UIColor)?.setFill()
        backgroundPath.fill()

        // Draw text
        (text as NSString).draw(in: textRect.insetBy(dx: 5, dy: 2.5), withAttributes: finalAttributes)

        let annotatedImage = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()
        return annotatedImage ?? image
```

```
    }

    private func completeScan(_ result: Result<EnhancedScanResult, Error>) {
        self.isScanning = false
        arSession?.pause()
        // arSession = nil // Keep session if you might restart, or nil out if completely done
        scanCompletion?(result)
        scanCompletion = nil // Avoid multiple calls
    }

    func stopScan() {
        if self.isScanning {
            completeScan(.failure(ScanError.cancelled)) // Or a success with partial data if applicable
        }
    }

    // Define specific error types
    enum ScanError: Error, LocalizedError {
        case arNotSupported
        case lidarNotSupported
        case sceneDepthNotSupported
        case arSessionFailed(Error)
        case noFrameAvailable
        case noImageInFrame
        case imageConversionFailed
        case classificationFailed
        case depthProcessingFailed
        case cancelled

        var errorDescription: String? {
            switch self {
            case .arNotSupported: return "ARKit is not supported on this device."
            case .lidarNotSupported: return "LiDAR sensor is not available or supported."
            case .sceneDepthNotSupported: return "Scene depth is not supported on this device/OS version."
            case .arSessionFailed(let err): return "AR session failed: \(err.localizedDescription)"
            case .noFrameAvailable: return "No AR frame was available for processing."
            case .noImageInFrame: return "The AR frame contained no image data."
            case .imageConversionFailed: return "Failed to convert AR frame image."
            case .classificationFailed: return "Wound classification failed."
            case .depthProcessingFailed: return "Failed to process depth data for measurements."
            case .cancelled: return "Scan was cancelled by the user."
            }
        }
    }
}

// Define EnhancedScanResult struct (as referenced above and potentially in ContentView)
// This struct holds all the data gathered from an enhanced scan.
// struct EnhancedScanResult {
//     let annotatedImage: UIImage        // Image with visual annotations (bounding box, measurements)
//     let woundType: String              // Classified type of the wound (e.g., "Abrasion")
```

```
//     let confidence: Float               // Confidence score from the classifier (0.0 to 1.0)
//     let distance: Float?                // Estimated distance to the wound in meters (e.g., from LiDAR/depth map)
//     let estimatedArea: Float?           // Estimated area of the wound in square centimeters
//     let severity: String                // Assessed severity (e.g., "Low", "Medium", "High")
//     let woundBoundingBox: CGRect?       // Normalized coordinates of the detected wound on the original image
//     // You might also include:
//     // let timestamp: Date
//     // let depthDataUsed: Bool // To indicate if measurements are depth-assisted
// }
//
// And ensure WoundDetection (if used by classifier) is defined as shown in X.1.
```

X.3 AI Consultation (e-Consult)

This function allows users to have a more in-depth conversation with the AI about the current wound after viewing the scan results.

Process Overview:

1. **User Operation (`ResultView` or `EnhancedScanResultView`)**:
   * User taps the "e-Consult" button on the result card.
   * The `showingChat` state of the corresponding view becomes `true`.
2. **Present Chat View (`ResultView`/`EnhancedScanResultView` -> `ChatView`)**:
   * Presents `ChatView` using the `.sheet` modifier.
   * Passes the current `woundType`, `deepseekService` instance, and the `$showingChat` binding to `ChatView`.
   * Injects `languageManager` at the same time.
3. **Initialize Chat (`ChatView`)**:
   * `ChatView.onAppear`:
     * Generates an initial prompt message based on the incoming `woundType` (e.g., "My wound type is XX, please give me detailed treatment advice.").
     * Calls `sendInitialMessage()`.
4. **Send Message (`ChatView` -> `DeepseekService`)**:
   * `ChatView.sendMessage()` or `sendInitialMessage()`:
     * Adds the user input or initial message to the local `messages` list.
     * Sets `isLoading = true`.
     * Calls `deepseekService.sendChatMessage(message: ..., language: languageManager.currentLanguage, ...)`.
   * `DeepseekService.sendChatMessage()`:
     * Adds the user message to the internal `chatHistory`.
     * Constructs a request body containing the **complete chat history** and a system prompt with **language instructions**.
     * Sends a request to the Deepseek API.
5. **Receive and Display AI Reply (`DeepseekService` -> `ChatView`)**:
   * Callback of `DeepseekService.sendChatMessage()`:
     * If successful, adds the AI's reply to `chatHistory`.
     * Returns the AI's reply via `completion`.
   * `ChatView` receives the AI reply:
     * Sets `isLoading = false`.
     * Adds the AI reply to the local `messages` list.
     * `ChatBubble` is responsible for rendering the message; if the AI reply contains Markdown, the `Text` view will attempt to render it.
6. **User Interaction (`ChatView`)**:

* Users can type new questions in the input box and send them, repeating steps 4 and 5.
* The chat card height can be adjusted by sliding, or the chat can be closed by tapping the close button.

Key Code Snippets (Detailed Illustration):

The following Swift code provides a more detailed illustration of the `ChatView` structure and its interaction with the `DeepseekService` for handling the AI consultation. It includes state management for messages, user input, UI elements for displaying the chat, and logic for sending/receiving messages. This example uses SwiftUI.

```swift
// ChatView.swift
import SwiftUI

// Define ChatMessage and ChatBubble (can be in separate files or same file if small)
struct ChatMessage: Identifiable, Equatable { // Equatable for .onChange
    let id = UUID()
    let text: String
    let isUser: Bool
    var isLoadingIndicator: Bool = false // To show a "thinking..." bubble for AI
    // let timestamp: Date = Date() // Optional for sorting or display
}

struct ChatBubble: View {
    let message: ChatMessage

    var body: some View {
        HStack {
            if message.isUser { Spacer(minLength: 20) } // Push user messages to the right

            if message.isLoadingIndicator {
                ProgressView()
                    .padding(10)
                    .background(Color(UIColor.systemGray5))
                    .clipShape(RoundedRectangle(cornerRadius: 10))
            } else {
                Text(message.text)
                    .padding(12)
                    .background(message.isUser ? Color.blue.opacity(0.9) : Color(UIColor.systemGray4))
                    .foregroundColor(message.isUser ? .white : .primary)
                    .clipShape(RoundedRectangle(cornerRadius: 12))
                    .textSelection(.enabled) // Allow copying text from bubbles
            }

            if !message.isUser { Spacer(minLength: 20) } // Push AI messages to the left
        }
        .padding(.horizontal, 10)
        .padding(.vertical, 4)
    }
}
```

```swift
struct ChatView: View {
    @Binding var showingChat: Bool // To dismiss the sheet
    var woundType: String         // Passed from the result view
    @ObservedObject var deepseekService: DeepseekService // Assumed to be an ObservableObject
    @EnvironmentObject var languageManager: LanguageManager // For language settings

    @State private var userInput: String = ""
    @State private var messages: [ChatMessage] = []
    @FocusState private var isTextFieldFocused: Bool // To manage keyboard

    var body: some View {
        NavigationView {
            VStack(spacing: 0) {
                ScrollViewReader { scrollViewProxy in
                    ScrollView {
                        LazyVStack(spacing: 8) {
                            ForEach(messages) { msg in
                                ChatBubble(message: msg)
                                    .id(msg.id) // Assign ID for scrolling
                            }
                        }
                        .padding(.top, 10)
                    }
                    .onChange(of: messages) { _ in // Use Equatable ChatMessage for reliable onChange
                        // Scroll to the bottom for new messages
                        if let lastMessage = messages.last {
                            withAnimation {
                                scrollViewProxy.scrollTo(lastMessage.id, anchor: .bottom)
                            }
                        }
                    }
                    .onTapGesture {
                        isTextFieldFocused = false // Dismiss keyboard on tap outside
                    }
                }

                // Input area
                HStack(spacing: 12) {
                    TextField("Ask about \(woundType)...", text: $userInput, axis: .vertical) // Allow multi-line input
                        .lineLimit(1...5) // Limit lines for text field
                        .padding(EdgeInsets(top: 8, leading: 12, bottom: 8, trailing: 12))
                        .background(Color(UIColor.systemGray6))
                        .clipShape(RoundedRectangle(cornerRadius: 20))
                        .focused($isTextFieldFocused)
                        .onSubmit(sendMessage) // Send on return key

                    Button(action: sendMessage) {
                        Image(systemName: "arrow.up.circle.fill")
                            .resizable()
                            .frame(width: 32, height: 32)
                            .foregroundColor(userInput.trimmingCharacters(in: .whitespacesAndNewlines).isEmpty ? .gray : .blue)
```

```
                }
                .disabled(userInput.trimmingCharacters(in: .whitespacesAndNewlines).isEmpty || deepseekService.isLoading)
            }
            .padding()
            .background(.thinMaterial) // Material background for input area
        }
        .navigationTitle("AI Consultation")
        .navigationBarTitleDisplayMode(.inline)
        .toolbar {
            ToolbarItem(placement: .navigationBarTrailing) {
                Button("Done") {
                    showingChat = false
                }
            }
        }
        .onAppear(perform: sendInitialMessage)
        .alert("Error", isPresented: $deepseekService.hasError, presenting: deepseekService.errorMessage) { _ in
            Button("OK") { deepseekService.clearError() }
        } message: { errorMessage in
            Text(errorMessage)
        }
    }
}

private func sendInitialMessage() {
    // Only send if messages are empty (e.g., first time view appears)
    guard messages.isEmpty else { return }

    let initialPrompt = "I have a \(woundType). Can you provide detailed treatment advice, potential complications to watch for, a
nd when I should see a doctor?"
    let initialMessage = ChatMessage(text: initialPrompt, isUser: true)
    messages.append(initialMessage)

    // Show AI thinking indicator
    let thinkingMessageId = UUID() // Need a stable ID if we want to remove/replace it
    messages.append(ChatMessage(id: thinkingMessageId, text: "", isUser: false, isLoadingIndicator: true))

    // Construct history for the service
    let historyForService = messages.filter { !$0.isLoadingIndicator } // Don't send thinking bubble as history

    deepseekService.sendChatMessage(
        message: initialPrompt, // The service might just use the latest message + history
        language: languageManager.currentLanguage.rawValue, // Assuming Language enum has rawValue: String
        history: historyForService.map { $0.text } // Or a more structured history object
    ) { replyText, error in
        // Remove thinking indicator
        messages.removeAll { $0.id == thinkingMessageId && $0.isLoadingIndicator }

        if let error = error {
            let errorMessage = ChatMessage(text: "Sorry, I encountered an error: \(error.localizedDescription)", isUser: false)
            messages.append(errorMessage)
```

```swift
                // Optionally set deepseekService.hasError and errorMessage here if not handled by service
                return
            }
            if let replyText = replyText, !replyText.isEmpty {
                let aiMessage = ChatMessage(text: replyText, isUser: false)
                messages.append(aiMessage)
            } else if error == nil { // No error but empty reply
                 let emptyReplyMessage = ChatMessage(text: "I didn't receive a response. Please try asking again.", isUser: false)
                messages.append(emptyReplyMessage)
            }
        }
    }

    private func sendMessage() {
        let trimmedInput = userInput.trimmingCharacters(in: .whitespacesAndNewlines)
        guard !trimmedInput.isEmpty else { return }

        let userMessage = ChatMessage(text: trimmedInput, isUser: true)
        messages.append(userMessage)
        let textToSend = userInput
        userInput = "" // Clear input field immediately

        // Show AI thinking indicator
        let thinkingMessageId = UUID()
        messages.append(ChatMessage(id: thinkingMessageId, text: "", isUser: false, isLoadingIndicator: true))

        let historyForService = messages.filter { !$0.isLoadingIndicator && $0.id != thinkingMessageId }


        deepseekService.sendChatMessage(
            message: textToSend,
            language: languageManager.currentLanguage.rawValue,
            history: historyForService.map { $0.text } // Example history format
        ) { replyText, error in
            messages.removeAll { $0.id == thinkingMessageId && $0.isLoadingIndicator }

            if let error = error {
                let errorMessage = ChatMessage(text: "Error: \(error.localizedDescription)", isUser: false)
                messages.append(errorMessage)
                return
            }
            if let replyText = replyText, !replyText.isEmpty {
                let aiMessage = ChatMessage(text: replyText, isUser: false)
                messages.append(aiMessage)
            } else if error == nil {
                 let emptyReplyMessage = ChatMessage(text: "I received an empty response. Could you rephrase or try again?", isUser:
false)
                messages.append(emptyReplyMessage)
            }
        }
        isTextFieldFocused = true // Keep keyboard focus after sending, or set to false to dismiss
```

```
    }
}

// Assuming DeepseekService is an ObservableObject like this:
// class DeepseekService: ObservableObject {
//     @Published var isLoading: Bool = false
//     @Published var hasError: Bool = false
//     @Published var errorMessage: String? = nil
//
//     func sendChatMessage(message: String, language: String, history: [String], completion: @escaping (String?, Error?) -> Void)
{
//         self.isLoading = true
//         self.clearError()
//         // ... (API call logic) ...
//         // On completion:
//         // self.isLoading = false
//         // if error { self.hasError = true; self.errorMessage = error.localizedDescription }
//         // completion(reply, error)
//     }
//     func clearError() {
//         self.hasError = false
//         self.errorMessage = nil
//     }
// }
//
// Assuming LanguageManager and Language enum:
// class LanguageManager: ObservableObject {
//     @Published var currentLanguage: Language = .english
// }
// enum Language: String { case english = "en", chinese = "zh" /* ... other languages */ }

```
```

IV. Project Implementation Process

(a) Dataset

The wound classification test dataset comprises 7,686 independently collected wound image s, encompassing five distinct categories: normal skin, lacerations, incisions, abrasions, and h ematomas. These images simulate authentic clinical scenarios, incorporating variations in lig hting conditions, background noise, and diverse imaging angles to enhance model robustnes s. Following the implementation of data augmentation techniques, the dataset was significan
```

tly expanded from the original 7,668 images to 45,271 images, substantially enhancing the model's feature extraction capabilities and overall performance during the training process.

(b) Model Training

WoundCare model, based on the YOLO11l architecture, was trained on the NVIDIA A100 Tensor Core GPU platform. This training process utilized 7,686 meticulously a nnotated medical image samples and underwent 100 training epoch iterations, resulti ng in significant enhancement of wound detection accuracy.

The model training parameters were configured as follows:

- **Batch size (batch)**: -1 (Auto-batch sizing based on GPU memory capacity)
- **Cache**: None
- **Device**: None
- **Training epochs**: 100
- **Image size (imgsz)**: 640
- **Patience value**: 100
- **Time**: None

The NVIDIA A100 platform was selected for its superior deep learning performance c apabilities, which according to industry benchmarks, provides significantly faster train ing speeds compared to previous generation hardware. The auto-batch sizing param eter (-1) allowed the system to automatically determine the optimal batch size based on the available GPU resources, maximizing computational efficiency while preventin g memory overflow issues.

(c) Application Deployment

To ensure optimal performance and user experience across both iOS and Android mobile operating systems, our team implemented a platform-native development strategy. Given the significant differences between iOS and Android system architectures, development kits (SDKs), and application programming interfaces (APIs)—such as Apple's Core ML versus the predominantly used TensorFlow Lite on Android—platform-specific development became essential. Specifically, the iOS application was developed using Apple's official Swift programming language and native API suite, while the Android application utilized Google's supported Kotlin programming language.

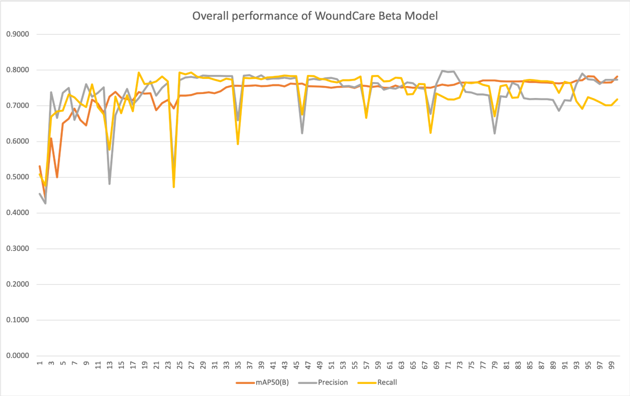
To further enhance the user experience, the application integrates the Deepseek large language model via API. After the system identifies a wound type through the application, this information is sent as a request to the Deepseek API to generate relevant professional wound care recommendations and content.

For the iOS platform, we developed a specialized wound area and severity assessment feature based on LiDAR technology. This functionality utilizes the LiDAR sensor built into select iPhone and iPad devices, in conjunction with the system-level ARKit framework, to perform high-precision distance detection. By combining the depth data with the wound bounding box dimensions detected by the YOLO model in the image, the system can calculate the estimated wound area in the physical world, thereby assisting in determining wound severity.

V. Project Outcomes

(a) Model Performance

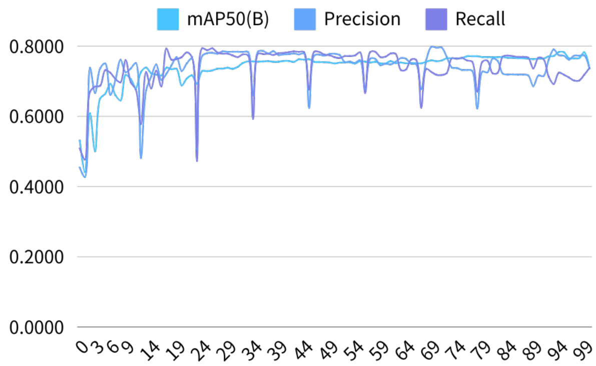
The data in the chart below represents the model's overall performance, where the X-axis indicatesthe total epochs of the model. An epoch refers to the state in the model training process where the algorithm has completely used every data point in the dataset. The Y-axis represents the maximum values.



mAP50 is the mean Average Precision calculated using a threshold value of 0.5 to measure the overlap degree between detection boxes and label boxes.

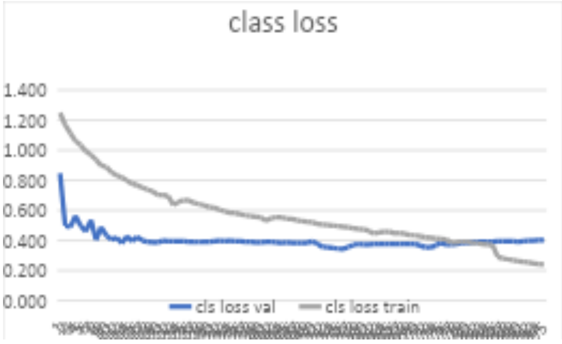
Recall is a metric that measures the model's prediction capability, particularly its ability to identify relevant instances.

Precision is the ratio of correctly predicted positive samples to the total number of samples predicted as positive.

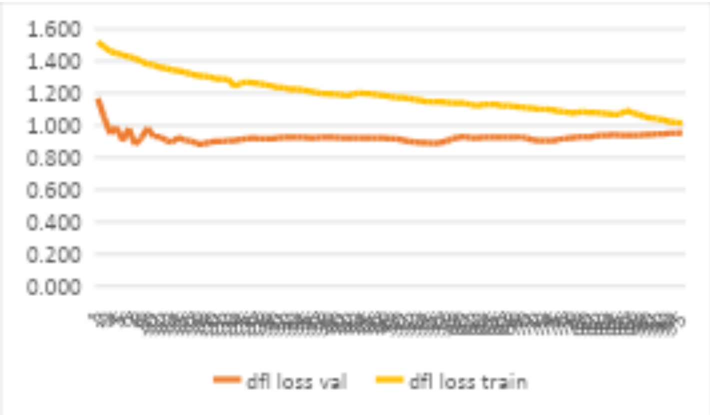


"Box loss" typically refers to the overall loss of boundary boxes, encompassing center position (x,y) and size (width, height). In YOLO11, this is usually calculated using IoU (Intersection over Union)-based loss functions, such as CIoU or SIoU, to measure the degree of overlap between predicted bounding boxes and ground truth bounding boxes. This component of the loss function ensures the model can correctly localize and adjust the size of objects.

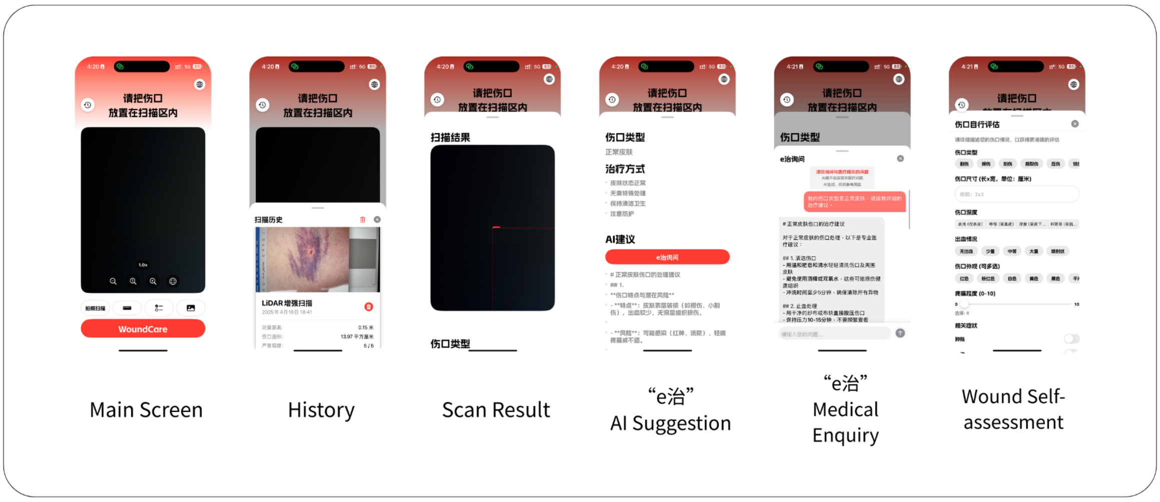
"Class loss" refers to the loss component associated with classification, which is responsible for predicting the category to which each detected object belongs. This loss is typically calculated using cross-entropy loss, measuring the difference between predicted class probabilities and actual class labels. This component is crucial for identifying different types of objects in images (such as cars, pedestrians, etc.).



"dfl loss" refers to Distribution Focal Loss, which is used in YOLO11 for precise prediction of bounding box center coordinates. Unlike traditional methods, DFL considers the potential distribution of center positions, particularly addressing scenarios involving small objects or size variations, helping the model better handle challenging detection situations. This is an advanced loss function designed to improve the prediction accuracy of center positions.



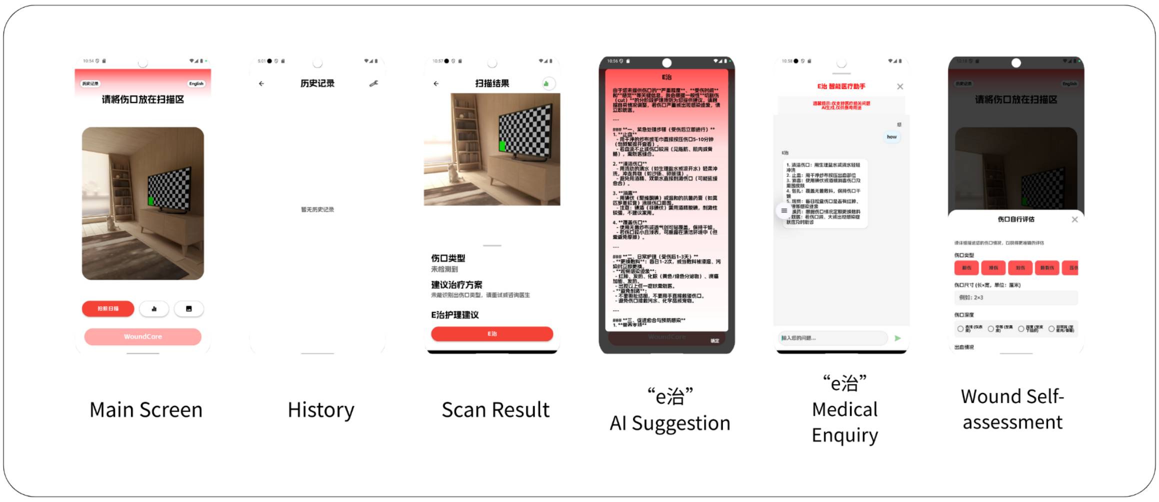
(b) Application Program



iOS Version Overall Interface

Based on the current trends in wound assessment applications, the iOS version of our application features a comprehensive user interface designed for optimal clinical functionality while maintaining user-friendly navigation. The interface incorporates intuitive design elements similar to leading medical applications in the wound care domain, with specialized components for wound image capture, analysis, and treatment recommendation display.

The application interface prioritizes accessibility and clear information presentation, following established patterns in successful medical imaging applications. Unlike many existing solutions that focus exclusively on healthcare practitioners, our interface is designed to be accessible to both medical professionals and general users, particularly in situations where immediate professional care may not be available.



The Android main interface design aims to provide an intuitive and efficient user experience. The main page features a clear layout of four core functional module buttons to accommodate diverse user needs.

First, the **"Photo Scan Detection"** button serves as the program's core functionality, enabling users to perform real-time wound detection through live camera scanning.

Second, the **"Self-Detection"** button innovatively employs a questionnaire format combined with generative artificial intelligence technology to guide users through preliminary wound condition self-assessment.

Finally, the **"Upload Photo Detection"** button facilitates convenient upload of existing wound photographs for analysis.

The overall program interface design maintains consistency with the iOS version, emphasizing usability, user-friendliness, and rapid response capabilities, ensuring users can conveniently and efficiently operate all detection functions.

VI. Project Testing Results

Wound Classification Accuracy: Ensuring that the YOLO11 model achieves an average accuracy rate of 80% or higher for wound type identification (including lacerations, incisions, abrasions, hematomas, etc.) on independent test datasets.

Area Measurement Precision: Validating that wound area measurement error rates remain below 5% on iOS devices equipped with LiDAR sensors.

Application Response Speed: Confirming that wound identification and treatment recommendation output times do not exceed 2 seconds on both iOS and Android platforms.

User Experience: Evaluating application stability in offline mode, as well as the practicality of the DeepSeek-integrated wound assessment and chatroom functionality.

VII. Conclusion and Future Prospects

With the rapid advancement of technology, particularly in artificial intelligence, we aspire to integrate technology with healthcare through this emergency medical a ssistance device. Our goal is to efficiently and accurately address injuries that occur in people's daily lives by providing real-time medical aid, thereby minimizing wound deterioration and the probability of secondary injuries to the greatest extent possible. We aim to create a safe environment that safeguards everyone's safety and health, with broader applications anticipated in the future.

We look forward to further enhancing the system's accuracy and practicality to better serve the public and reduce health risks resulting from improper wound management. This project not only demonstrates the potential of artificial intelligence in the medical field but also emphasizes the critical importance of combining technology with healthcare. Our ultimate objective is to create a safer and healthier living environment for everyone.

The integration of advanced AI algorithms with accessible mobile technology represents a significant step toward democratizing healthcare access, particularly in underserved areas where immediate professional medical attention may not be readily available. As we continue to refine and expand this technology, we envision a future where intelligent wound assessment becomes a standard component of first aid care, ultimately contributing to improved health outcomes and reduced healthcare disparities globally.

IX. Appendix

(a) Keywords

Core Project Attributes:

- Fast/Rapid
- Convenient
- Easy-to-use
- Accurate
- Safe/Safety

Technology and AI Components:

- Artificial Intelligence (AI)
- Image Recognition
- Wound Recognition/Identification
- Wound Treatment/Management
- High-efficiency/Efficient

Application Domains:

- Health/Healthcare

- Technology
- Medical
- Safety

(b) References

[1] 老人跌倒造成的傷害及如何預防

https://www.airitilibrary.com/Article/Detail/P20210804002-N202311070009-00007

[2] 護理指導對外科傷口照護之成效探討

https://ndltd.ncl.edu.tw/cgi-bin/gs32/gsweb.cgi/login?o=dnclcdr&s=id%3D"099CTC05743052".&searc hmode=basic

[3] 傷口癒合機轉

http://tnha.com.tw/web/images/ckfinder/files/20171010104559.pdf

[4] 慢性傷口之評估與測量原則

https://www.airitilibrary.com/Publication/alDetailedMesh?docid=0047262x-200704-54-2-62-67-a

[5] 人工智能伤口评估方法及智能终端

https://patents.google.com/patent/CN111523508A/zh

[6] 慢性傷口智慧照護

https://www.moea.gov.tw/Mns/doit/videos/Videos.aspx?menu_id=13596&video_id=174

[7]二零一八年非故意損傷統計調查報告書 https://www.chp.gov.hk/files/pdf/report_of_unintentional_injury_survey_2018_tc.pdf

[8]IWII-Consensus-2016_Chinese https://woundinfection-institute.com/wp-content/uploads/2021/06/IWII-Consensus-2016_Chinese.pdf

[9] 非接觸式慢性傷口分析最新方法的系統概述

https://www.mdpi.com/2076-3417/10/21/7613

[10] 人工智能在傷口數字化成像評估中的應用与前景 https://d.wanfangdata.com.cn/periodical/Ch9QZXJpb2RpY2FsQ0hJTmV3UzIwMjQxMTA1MTcxMzA0EhJsYW56eXh5eGIyMDI0MDMwMTMaCGZ5amhjZndi

[11] 一種基于人工智能的術後切口愈合狀態識別系統 https://d.wanfangdata.com.cn/patent/ChhQYXRlbnROZXdTMjAyNDExMjIxNjU4MjISEENOMjAyNDExMDAwOTEyLjkaCHN1ZG10amMz

[12] 一種基于深度學習技術的傷口識別與面積測量系統及方法 https://d.wanfangdata.com.cn/patent/ChhQYXRlbnROZXdTMjAyNDExMjIxNjU4MjISEENOMjAyMjEwNTI2MzA5LjkaCHN1ZG10amMz

[13]YOLO11 与YOLOv8：详细比较
https://docs.ultralytics.com/zh/compare/yolo11-vs-yolov8/