Based on my analysis of your signature service classes, analyzers, and DTOs in the `forgery-detection` project, I'll provide a detailed breakdown of the methodology and algorithm used for signature verification detection. I'll focus on the core logic from `SignatureVerificationService.java`, the supporting analyzers (`StructuralSimilarityAnalyzer`, `StrokeConsistencyAnalyzer`, `PressurePatternAnalyzer`), and the `SignatureVerificationResult` DTO. Note that the implementation has some inconsistencies or potential bugs (e.g., in how analyzers are invoked), which I'll highlight for completeness. The overall approach is a multi-feature fusion method combining structural similarity, stroke analysis, and pressure pattern analysis to determine if a test signature matches a reference signature. ### **Overall Methodology** The signature verification system uses a **hybrid, feature-based forgery detection approach** inspired by biometric verification techniques for handwritten signatures. It does not rely on machine learning models but instead fuses multiple heuristic-based analyses to compute a similarity score. The process involves: - **Preprocessing** both reference and test signature images to standardize them (grayscale conversion and binarization). - **Feature extraction and analysis** using three specialized analyzers to quantify similarity across different aspects of the signature (structural, stroke-based, and pressure-based). - **Score fusion** via weighted averaging to produce a final forgery probability score. - **Decision threshold** to classify the signature as genuine or forged. This is suitable for offline signature verification (static images), focusing on image-level comparisons rather than dynamic features like pen speed or timing. The methodology assumes signatures are provided as OpenCV `Mat` objects (e.g., from uploaded images). Key assumptions: - Signatures are binary or grayscale images where black pixels represent ink. - Genuine signatures should have high similarity in structure, stroke consistency, and pressure patterns. - Weights are empirically set (SSIM: 0.5, Stroke: 0.3, Pressure: 0.2), prioritizing structural similarity. - Threshold for genuineness is 0.6 (scores range from 0 to 1, where 1 is perfect match). Potential limitations: - Sensitive to image quality, rotation, scaling, or noise (no alignment or normalization beyond basic preprocessing). - The stroke analyzer usage in the service appears buggy (see below), which may lead to inaccurate results. - No handling for multi-stroke or complex signatures; treats images as 2D arrays. ### **Algorithm Step-by-Step** The main algorithm is implemented in `SignatureVerificationService.verifySignature(Mat refSig, Mat testSig)`, which returns a `SignatureVerificationResult` DTO containing individual scores, a final score (forgery probability), and a boolean for genuineness. 1. **Input Validation**:   - Check if either reference (`refSig`) or test (`testSig`) image is empty. If so, return a result with all scores set to 0 and `likelyGenuine = false`. 2. **Preprocessing**:   - Convert both images to grayscale if they have multiple channels (using `Imgproc.cvtColor` with `COLOR_BGR2GRAY`).   - Apply binary thresholding using Otsu's method (`Imgproc.threshold` with `THRESH_BINARY | THRESH_OTSU`) to create black-and-white versions. This standardizes the images for analysis, emphasizing edges and shapes while reducing noise. 3. **Feature Extraction and Analysis** (using three analyzers):   - **Structural Similarity (SSIM) Analysis** (via `StructuralSimilarityAnalyzer`):   - Convert preprocessed images to normalized double arrays (0-1 range).   - Compute SSIM on overlapping patches of sizes 4x4, 8x8, and 16x16 pixels.   - Average SSIM across patches and scales, clamp to [0, 1], then apply a power adjustment (raise to 0.6) to tune sensitivity (higher values favor similarity).   - Result: A score (0-1) where 1 indicates identical structure. This captures overall shape and pixel-level similarity.   - **Stroke Consistency Analysis** (via `StrokeConsistencyAnalyzer`):   - Extract "stroke lengths" from each image: Apply Canny edge detection (thresholds 30-100), morphological closing (3x3 kernel), and count edge pixels per row (horizontal strokes assumed).   - **Bug/Issue**: The service calls `strokeAnalyzer.analyze(refStrokes, testStrokes)`, but the `analyze` method expects `(double[] strokes, double[] thicknesses)` for a single signature. It computes internal consistency (mean/variance of strokes and thicknesses) but not direct comparison. Thicknesses are not extracted in the service, so this likely produces incorrect results. Intended logic probably should compute

consistency scores for ref and test separately, then compare (e.g., via correlation or difference). As implemented, it may treat `testStrokes` as thicknesses, leading to nonsense scores. - If fixed, the score would quantify stroke length/thickness variability (lower variance = higher consistency score). Genuine signatures should have similar consistency profiles. - **Pressure Pattern Analysis** (via `PressurePatternAnalyzer`): - Extract "pressure" data: Use grayscale pixel intensities (0-255) as a proxy for pressure (darker = higher pressure). - Compute a "pressure score" for each image: Divide into 16x16 blocks, calculate per-block variance (standard deviation), average across blocks, and derive a score (1 / (1 + avg_std/255)), clamped to [0,1]. Higher variance (inconsistent pressure) lowers the score. - Compare ref and test: Compute `pressureScore = 1 - |refPressureScore - testPressureScore|`, so identical pressure patterns yield 1. 4. **Score Fusion and Decision**: - Compute final score as a weighted sum: - `finalScore = (ssimScore * 0.5) + (strokeScore * 0.3) + (pressureScore * 0.2)` - Classify as genuine if `finalScore >= 0.6`; otherwise, forged. - Populate the `SignatureVerificationResult` DTO with: - `structuralSimilarity`: SSIM score. - `strokeConsistencyScore`: Stroke score. - `pressurePatternScore`: Pressure similarity score. - `forgeryProbability` (mapped to `finalScore` in DTO, though field is `finalScore`—minor naming inconsistency). - `likelyGenuine`: Boolean result. - `details`: A map for additional breakdowns (not populated in current code). 5. **Output**: - Return the DTO for API responses (e.g., via `DetectionController`). ### **Key Components and Roles** - **DTO (`SignatureVerificationResult`)**: Extends `DetectionResult` to hold scores, final probability, and genuineness flag. Provides getters/setters for serialization (e.g., JSON responses). Fields like `details` allow extensibility for breakdowns. - **Analyzers**: - `StructuralSimilarityAnalyzer`: Implements a custom SSIM variant for patch-based comparison. Efficient for detecting tampering or replication errors. - `StrokeConsistencyAnalyzer`: Focuses on stroke dynamics (lengths via edges, thicknesses via dilation). Aims to detect unnatural variations in forged signatures. - `PressurePatternAnalyzer`: Simulates pressure analysis via intensity variance. Assumes genuine signatures have consistent "pressure" profiles. - **Service (`SignatureVerificationService`)**: Orchestrates preprocessing, analysis calls, and fusion. Injected analyzers ensure modularity. ### **Potential Improvements and Bugs** - **Bugs in Implementation**: - Stroke analyzer invocation is incorrect (passes two stroke arrays instead of strokes + thicknesses). This likely causes runtime errors or invalid scores. Fix: Extract thicknesses separately and compute/compare consistency scores properly (e.g., `strokeScore = 1 - |refConsistency - testConsistency|`). - DTO field mismatch: Code sets `setForgeryProbability(finalScore)`, but DTO has `setFinalScore()`. Update for consistency. - **Enhancements**: - Add image alignment/rotation invariance (e.g., via feature matching). - Incorporate more features (e.g., curvature, slant) or ML (e.g., SVM on features). - Dynamic thresholds or user-calibrated weights. - Better error handling for mismatched image sizes. - **Performance**: OpenCV operations are efficient, but large images may slow down patch-based SSIM. If you'd like me to fix the bugs, add features, or run tests on sample data, let me know! For example, I can use the provided test images (`sign_ref.jpeg`, `sign_test.png`) to demonstrate.