Feature-Based Logo Detection in Images: A SIFT Approach for Brand Recognition

Alberto Di Maio

14 June 2025

Contents

List of Figures

Listings

Abstract

This paper presents a comprehensive analysis of a computer vision system designed for automatic logo detection in images, specifically focusing on Starbucks brand recognition. The system employs Scale-Invariant Feature Transform (SIFT) algorithm combined with feature matching techniques to identify and localize logos within input images. Through detailed code examination and mathematical foundation analysis, we explore the complete pipeline from feature extraction to geometric validation using homography estimation. The approach demonstrates robust performance in detecting logos across various scales, rotations, and lighting conditions, making it suitable for real-world brand monitoring applications.

1 Introduction and System Architecture

1.1 Overall System Design

This research analyzes a practical implementation of logo detection using the SIFT (Scale-Invariant Feature Transform) algorithm, specifically designed for Starbucks logo recognition. The system represents a classical approach to object recognition in computer vision, leveraging local feature descriptors and geometric consistency checks to achieve reliable detection.

The logo detection system follows a classical computer vision pipeline consisting of four main stages:

1. **Reference Template Processing**: Extract and store features from a reference logo image
2. **Query Image Analysis**: Extract features from input images to be analyzed
3. **Feature Matching**: Compare and match features between template and query images
4. **Geometric Validation**: Verify matches using homography estimation and spatial consistency

The system architecture is encapsulated in the `StarbucksLogoDetector` class, which provides both single-image and batch processing capabilities.

2 SIFT Feature Extraction and Scale-Space Theory

2.1 Keypoint Detection through Difference of Gaussians

The core of the detection system relies on SIFT features, which are particularly well-suited for logo detection due to their invariance properties. The mathematical foundation begins with the creation of a scale-space representation by convolving the image with Gaussian kernels of increasing standard deviation:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{1}$$

where the Gaussian kernel is defined as:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2}$$

SIFT keypoints are then detected as local extrema in the Difference of Gaussians (DoG) pyramid:

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma) \tag{3}$$

where k is typically set to $\sqrt{2}$ to ensure adequate sampling across scales.

Mathematical Explanation: The Difference of Gaussians operation creates a scale-space representation by subtracting two Gaussian-blurred versions of the same image at different scales. This operation is mathematically equivalent to approximating the Laplacian of Gaussian (LoG), which is excellent at detecting blob-like structures. In our logo detection context, the Starbucks circular logo acts as a "blob" - the DoG operation will respond strongly to such circular features at the appropriate scale.

The scale parameter σ controls the size of features detected: - Small σ values (e.g., 1.0): Detect fine details like small text - Large σ values (e.g., 4.0): Detect larger structures like the overall logo shape

The implementation initializes the SIFT detector with specific parameters:

```
self.detector = cv2.SIFT.create(nfeatures=500)
```

Listing 1: SIFT Detector Initialization

The `nfeatures=500` parameter limits the number of strongest keypoints to prevent computational overhead while maintaining detection accuracy.

2.2 Feature Description and Gradient Computation

Each detected keypoint is described by a 128-dimensional vector that captures the local image gradient information. The gradient at each pixel is calculated as:

$$\nabla I = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right) \tag{4}$$

The magnitude and orientation are computed as:

$$|\nabla I| = \sqrt{\left(\frac{\partial I}{\partial x} \right)^2 + \left(\frac{\partial I}{\partial y} \right)^2} \tag{5}$$

$$\theta = \arctan \left(\frac{\partial I / \partial y}{\partial I / \partial x} \right) \tag{6}$$

The final SIFT descriptor is a 128-dimensional vector:

$$\mathbf{d} = [d_1, d_2, \ldots, d_{128}]^T \tag{7}$$

where each d_i represents the magnitude of gradients in specific orientation bins around the keypoint.

Mathematical Explanation: The SIFT descriptor acts as a sophisticated "fingerprint" for each keypoint. Around each detected keypoint, SIFT examines a 16×16 pixel neighborhood and divides it into a 4×4 grid of subregions. For each subregion, it computes a histogram of gradient orientations with 8 bins, resulting in 4×4×8 = 128 values.

For example, if we detect a keypoint on the edge of the Starbucks logo, the descriptor will capture the pattern of edges around that point - perhaps strong vertical gradients on one side and horizontal gradients on another. This creates a unique "signature" that can be matched even if the logo appears at different scales or rotations.

2.3 Reference Logo Processing

The system begins by processing the reference logo image to extract stable features:

```
def load_reference_logo(self, logo_path):
    # Load and convert to grayscale
    logo_img = cv2.imread(logo_path)
    self.reference_logo = cv2.cvtColor(logo_img, cv2.COLOR_BGR2GRAY)

    # Extract SIFT features
    self.reference_kp, self.reference_desc = self.detector.
   detectAndCompute(
        self.reference_logo, None
    )
```

Listing 2: Reference Logo Feature Extraction

This preprocessing step establishes the template against which all query images will be compared. The grayscale conversion eliminates color variations while preserving structural information essential for feature matching.

3 Feature Matching and Distance Metrics

3.1 Euclidean Distance for Descriptor Comparison

The system uses Euclidean distance to compare SIFT descriptors between the reference logo and query images:

$$d(\mathbf{f}_1, \mathbf{f}_2) = \sqrt{\sum_{i=1}^{128} (f_{1i} - f_{2i})^2} \tag{8}$$

where $\mathbf{f}_1$ and $\mathbf{f}_2$ are 128-dimensional SIFT descriptors.

Mathematical Insight: This distance measures how similar two feature vectors are in the 128-dimensional space. Smaller distances indicate more similar local image patterns: - Distance ≈ 0.2: Very similar features (likely the same point) - Distance ≈ 0.8: Moderately similar features (possible match) - Distance ≈ 1.5: Dissimilar features (unlikely match)

3.2 Brute Force Matching with Ratio Test

The system employs a brute force matcher with k-nearest neighbors (k=2) to find potential correspondences:

```
bf = cv2.BFMatcher()
matches = bf.knnMatch(self.reference_desc, input_desc, k=2)

# Apply ratio test
good_matches = []
for match_pair in matches:
    if len(match_pair) == 2:
        m, n = match_pair[0], match_pair[1]
        if m.distance < self.ratio_threshold * n.distance:
            good_matches.append(m)
```

Listing 3: Feature Matching Implementation

The ratio test, proposed by Lowe, filters matches based on the distance ratio between the first and second nearest neighbors:

$$\frac{d_1}{d_2} < \tau \tag{9}$$

where d_1 and d_2 are the Euclidean distances to the first and second nearest neighbors, respectively, and τ is the ratio threshold (default 0.7).

Mathematical Explanation: The ratio test eliminates ambiguous matches by comparing the quality of the best match against the second-best match. If the best match (distance d_1) is significantly better than the second-best match (distance d_2), then d_1/d_2 will be small, indicating a confident match.

Consider these examples: - **Good match**: $d_1 = 0.3$, $d_2 = 0.8 \rightarrow$ ratio $= 0.375$ ¡ 0.7 (Accept) - **Ambiguous match**: $d_1 = 0.6$, $d_2 = 0.65 \rightarrow$ ratio $= 0.923$ ¿ 0.7 (Reject)

The threshold $\tau = 0.7$ is empirically determined to be strict enough to reject ambiguous matches while accepting clearly distinctive ones.

4 Geometric Validation through Homography

4.1 Homography Matrix and Perspective Transformation

Once sufficient good matches are found (minimum 10 by default), the system estimates a homography matrix to validate the geometric consistency of the matches. The homography matrix is defined as:

$$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{10}$$

The homography relates points in the reference image to points in the query image through the transformation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \mathbf{H} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{11}$$

In practice, this becomes:

$$\begin{bmatrix} x'w \\ y'w \\ w \end{bmatrix} = \mathbf{H} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{12}$$

and the final coordinates are obtained by:

$$x_{\text{final}} = \frac{x'w}{w}, \quad y_{\text{final}} = \frac{y'w}{w} \tag{13}$$

Mathematical Explanation: A homography describes how one planar surface appears when viewed from different perspectives. Each element of the matrix has specific geometric meaning: - h_{11}, h_{22}: Scaling factors in x and y directions - h_{12}, h_{21}: Rotation and shear components - h_{13}, h_{23}: Translation components - h_{31}, h_{32}: Perspective distortion parameters - h_{33}: Normalization factor (typically set to 1)

4.2 RANSAC Algorithm for Robust Estimation

The RANSAC algorithm robustly estimates the homography in the presence of outliers. The reprojection error for each point pair is calculated as:

$$e_i = \sqrt{(x'_i - \hat{x}'_i)^2 + (y'_i - \hat{y}'_i)^2} \tag{14}$$

where $(\hat{x}'_i, \hat{y}'_i)$ is the predicted location using the current homography estimate:

$$\begin{bmatrix} \hat{x}'_i \\ \hat{y}'_i \\ 1 \end{bmatrix} = \mathbf{H} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \tag{15}$$

```
# Extract matched points
src_pts = np.float32([self.reference_kp[m.queryIdx].pt for m in
    good_matches])
dst_pts = np.float32([input_kp[m.trainIdx].pt for m in good_matches])

# Find homography using RANSAC
M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
```

Listing 4: Homography Estimation with RANSAC

The RANSAC process works iteratively:

1. Randomly select 4 point pairs (minimum needed for homography)
2. Compute a candidate homography **H** from these 4 pairs
3. Test all other matches: if $e_i < 5.0$ pixels, mark as inlier
4. Count inliers for this homography
5. Repeat steps 1-4 many times
6. Choose the homography with the most inliers

Mathematical Rationale: The 5.0-pixel threshold accounts for small localization errors due to image noise or discretization effects while rejecting gross outliers that would indicate false matches.

4.3 Logo Localization and Corner Detection

After successful homography estimation, the system localizes the logo by transforming the reference logo corners to the query image space:

```
# Define corners of reference logo
h, w = self.reference_logo.shape
logo_corners = np.float32([[0, 0], [0, h-1], [w-1, h-1], [w-1, 0]])

# Transform corners to input image
detected_corners = cv2.perspectiveTransform(logo_corners.reshape
    (-1,1,2), M)
```

Listing 5: Logo Corner Transformation

This transformation maps the rectangular boundary of the reference logo to its detected location in the query image, which may appear as a rotated or perspective-distorted quadrilateral.

5 Confidence Scoring and Performance Analysis

5.1 Quantitative Confidence Measurement

The system calculates detection confidence based on the geometric consistency of matches:

$$\text{Confidence} = \frac{\text{Number of Inliers}}{\text{Total Good Matches}} \tag{16}$$

where inliers are matches that satisfy the homography constraint within the RANSAC threshold.

```
inliers = np.sum(mask)
confidence = inliers / len(good_matches)
```

Listing 6: Confidence Calculation

Probabilistic Interpretation: The confidence score quantifies detection reliability. Assuming false matches are randomly distributed, the probability of achieving a high inlier ratio by chance is extremely low. For confidence c with n total matches, the probability of random occurrence is approximately:

$$P(\text{random}) \approx \binom{n}{cn} \times (p_{\text{inlier}})^{cn} \times (1 - p_{\text{inlier}})^{n(1-c)} \tag{17}$$

where $p_{\text{inlier}} \approx 0.1$ for random matches. For example, with confidence = 0.8 and n = 40 matches:

$$P(\text{random}) \approx \binom{40}{32} \times (0.1)^{32} \times (0.9)^{8} \approx 10^{-25} \tag{18}$$

This confirms that high confidence scores indicate genuine detections rather than coincidences.

5.2 Detection Criteria and Thresholds

A logo is considered detected if all the following conditions are met:

- Number of good matches $\geq$ minimum threshold (default: 10)

- Homography estimation succeeds (sufficient inliers found)
- Confidence score indicates geometric consistency

The mathematical foundation ensures robust detection while minimizing false positives through multiple validation layers.

6 Implementation Considerations and Error Handling

6.1 Computational Optimizations

The system includes several optimizations for practical deployment:

1. **Feature Limitation**: The 500-feature limit balances accuracy with computational efficiency
2. **Early Termination**: Processing stops if insufficient features are detected
3. **Efficient Matching**: Brute force matching with optimized distance calculations

6.2 Robust Error Handling

The implementation handles common failure scenarios:

```
if input_desc is None or len(input_desc) == 0:
    return {"detected": False, "error": "No features found in input image"}

if len(good_matches) < self.min_matches:
    return {"detected": False, "error": f"Insufficient matches: {len(good_matches)}"}

try:
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    if M is None:
        return {"detected": False, "error": "Homography estimation failed"}
except cv2.error as e:
    return {"detected": False, "error": f"OpenCV error: {str(e)}"}
```

Listing 7: Comprehensive Error Handling

7 Results and Applications

7.1 System Strengths

The mathematical foundation provides several key advantages:

- **Scale Invariance**: Multi-scale SIFT features detect logos of varying sizes
- **Rotation Robustness**: Gradient-based descriptors handle arbitrary rotations
- **Perspective Tolerance**: Homography estimation accommodates viewpoint changes
- **Partial Occlusion Handling**: Local features allow detection with incomplete visibility

7.2 Limitations and Considerations

Mathematical and practical limitations include:

- **Computational Complexity**: $O(n^2)$ matching complexity with large feature sets
- **Texture Dependency**: SIFT requires sufficient image gradients for keypoint detection
- **Extreme Deformations**: Homography assumes planar transformations
- **False Positive Potential**: Similar patterns may trigger incorrect detections

8 Conclusion

This analysis demonstrates how classical computer vision techniques, grounded in solid mathematical foundations, provide robust solutions for logo detection tasks. The integration of scale-space theory, gradient-based feature description, statistical matching validation, and geometric consistency checking creates a comprehensive system suitable for real-world applications.

The mathematical rigor underlying each component - from DoG keypoint detection through RANSAC-based homography estimation - ensures reliable performance across diverse imaging conditions. While modern deep learning approaches may offer enhanced accuracy, the analyzed SIFT-based system provides interpretable, parameter-controlled detection with clear mathematical foundations.

Future work could explore hybrid approaches combining classical mathematical rigor with modern learning-based enhancements, maintaining the interpretability and theoretical grounding demonstrated in this analysis.

Code and Repository

The complete implementation of the logo detection system discussed in this paper is publicly available at:

- **GitHub Repository:** `github.com/albedim/SIFT-logo-detector`
- **License:** MIT License
- **Contributors:** Alberto Di Maio

The repository includes Python source code, sample images, usage instructions, and test results.