Extracting Numbers from Ishihara Color Blindness Tests Using K-means Clustering and Color Space Analysis

Comprehensive Guide

May 4, 2025

Abstract

This document provides a comprehensive explanation of how to extract hidden numbers from Ishihara color blindness test images using K-means clustering and color space transformations. We cover the fundamentals of Ishihara tests, color spaces, and the K-means algorithm, followed by a detailed walkthrough of the implementation process. This approach combines computer vision, unsupervised learning, and color theory to automatically identify patterns that are designed to be difficult for color-blind individuals to perceive.

Contents

1 Introduction to Ishihara Color Blindness Tests

1.1 What Are Ishihara Tests?

Ishihara tests are a series of colored plates designed to diagnose color vision deficiencies. They were created by Dr. Shinobu Ishihara, a Japanese ophthalmologist, in 1917.

Each plate contains a circle of dots in various colors and sizes. Within these dots, numbers or shapes are embedded that are visible to people with normal color vision but difficult or impossible to see for those with color blindness.

1.2 How Do Ishihara Tests Work?

The tests work by using specific color combinations that exploit the differences between normal color vision and color-deficient vision:

- **Red-Green Color Blindness Tests**: These use red and green dots to create patterns. People with red-green color blindness struggle to distinguish these colors.
- **Blue-Yellow Color Blindness Tests**: Less common, these tests target blue-yellow color vision deficiencies.
- **Total Color Blindness Tests**: Some plates can detect total color blindness (seeing only in grayscale).

1.3 Characteristics of Ishihara Test Images

A typical Ishihara test image has these key characteristics:

- A circular pattern of colored dots
- Dots of different sizes and colors
- A hidden number or pattern formed by dots of a specific color
- The background consisting of dots of another color or colors
- Designed so that the color contrast between the number and background is difficult for color-blind individuals to perceive

1.4 Why Is Computer Analysis Needed?

Computer analysis of Ishihara tests is valuable for:

- Automated screening and diagnosis
- Educational purposes (understanding how color perception works)
- Image processing research
- Developing tools for the color blind to "see" the hidden patterns

2 Understanding Color Spaces

2.1 What Is a Color Space?

A color space is a specific organization of colors that allows for consistent representation across different devices and applications. Think of it as a coordinate system for colors.

2.2 Common Color Spaces

2.2.1 RGB (Red, Green, Blue)

- The standard color model used in electronic displays
- Three channels representing intensity of red, green, and blue light
- Values typically range from 0-255 for each channel
- When working with Ishihara tests, the red or green channel alone sometimes provides good discrimination

2.2.2 HSV (Hue, Saturation, Value)

- Hue: The color type (red, blue, etc.) - measured in degrees (0-360)
- Saturation: The color intensity or purity - from 0% to 100%
- Value: The brightness - from 0% to 100%
- For Ishihara tests, the hue channel can be useful for isolating color differences

2.2.3 Lab (CIELAB)

- L: Lightness (0-100)
- a: Green-Red component (-128 to +127)
- b: Blue-Yellow component (-128 to +127)
- Designed to be perceptually uniform
- The 'a' channel is particularly effective for Ishihara tests since it specifically represents the red-green axis

2.2.4 YCrCb

- Y: Luminance (brightness)
- Cr: Red-difference chroma component
- Cb: Blue-difference chroma component
- Similar to Lab, the Cr channel is effective for Ishihara tests

2.3 Why Different Color Spaces Matter

Ishihara tests rely on color differences that are hard for color-blind individuals to perceive. By converting images to different color spaces, we can:

- Isolate the specific color components that contain the hidden pattern
- Enhance the differences between the number/pattern and the background
- Find the optimal representation for automated extraction of the hidden information

The Lab and YCrCb color spaces are particularly useful for Ishihara tests because:

- The 'a' channel in Lab explicitly encodes the red-green color axis
- Ishihara tests often use red-green contrast to hide numbers
- By isolating the 'a' channel, we directly capture this critical contrast
- Similarly, the 'Cr' channel in YCrCb encodes red-difference information

3 K-means Clustering Algorithm

3.1 What Is K-means Clustering?

K-means is an unsupervised machine learning algorithm that partitions data into k distinct groups (clusters) based on similarity. The algorithm aims to minimize the distance between points within the same cluster.

3.2 How K-means Works: Step by Step

1. **Initialization**: Randomly select k points from the dataset as initial cluster centroids
2. **Assignment**: Assign each data point to the nearest centroid, forming k clusters
3. **Update**: Recalculate the position of each centroid as the mean of all points in its cluster
4. **Repeat**: Continue steps 2-3 until centroids no longer change significantly or a maximum number of iterations is reached

3.3 Mathematical Foundation

The objective of K-means is to minimize the within-cluster sum of squares (WCSS):

$$WCSS = \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2 \tag{1}$$

Where:

- S_i is the i-th cluster
- μ_i is the centroid of cluster S_i
- x is a data point in cluster S_i
- $\|x - \mu_i\|^2$ is the squared Euclidean distance between x and μ_i

3.4 Applying K-means to Images

When applying K-means to images:

- **Image as Data**: Each pixel is treated as a data point
- **Features**: Pixel color values (e.g., R, G, B) become the features for clustering
- **Clusters**: K-means groups similar-colored pixels together
- **Segmentation**: After clustering, the image is segmented by replacing each pixel with its cluster's centroid color

4 Detailed Implementation: Extracting Numbers from Ishihara Tests

4.1 The Complete Process Overview

1. **Image Preprocessing**:
   - Load the image
   - Convert to the desired color space (RGB, HSV, Lab, YCrCb)
   - Extract specific channel(s) if needed
2. **K-means Clustering**:
   - Reshape the image data into a format suitable for clustering
   - Apply K-means with an appropriate number of clusters (typically 2-3)

- Assign each pixel to its corresponding cluster

3. **Segmentation**:
   - Create a segmented image where each pixel is represented by its cluster's color
   - This helps visualize the separation achieved by clustering
4. **Number Extraction**:
   - Identify which cluster represents the hidden number
   - Create a binary mask highlighting only that cluster
   - Apply post-processing (morphological operations) to clean up the result
5. **Visualization**:
   - Display the original image, segmented result, and extracted number

4.2 Detailed Step-by-Step Implementation

4.2.1 Step 1: Loading the Image

```
# Load the image using OpenCV
image = cv2.imread("ishihara_74.png")
```

This loads the image in BGR (Blue-Green-Red) format, which is OpenCV's default.

4.2.2 Step 2: Color Space Conversion

```
# Convert to Lab color space
lab_image = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)
```

For Ishihara tests which often use red-green contrast, the 'a' channel in the Lab color space is particularly useful because it specifically encodes the red-green difference:

- Negative values indicate green
- Positive values indicate red

4.2.3 Step 3: Channel Extraction

```
# Extract the 'a' channel (red-green component)
height, width = lab_image.shape[:2]
a_channel = lab_image[:, :, 1]  # 0=L, 1=a, 2=b

# Reshape for K-means input
features = a_channel.reshape(-1, 1)  # Flatten to 1D array with 1
    feature per pixel
```

By extracting just the 'a' channel, we reduce the dimensionality of our data from 3D (RGB values) to 1D (a single color component), which makes the clustering problem simpler.

4.2.4 Step 4: K-means Clustering Implementation

```
# Initialize K-means with 2 clusters
kmeans = KMeansClustering(k=2, max_iterations=100)
```

We typically use k=2 for Ishihara tests because we want to separate:

1. The dots forming the number
2. The background dots

```
def initialize_centroids(self, X):
    n_samples, n_features = X.shape
    centroids = np.zeros((self.k, n_features))
    random_indices = np.random.choice(n_samples, self.k, replace=
    False)
    centroids = X[random_indices]
    return centroids
```

This function:

1. Randomly selects k pixels from the image
2. Uses their color values as our initial centroids
3. Provides starting points for our clusters

```
def compute_distance(self, X, centroids):
    n_samples = X.shape[0]
    distances = np.zeros((n_samples, self.k))
    for i in range(self.k):
        distances[:, i] = np.sqrt(np.sum((X - centroids[i])**2,
    axis=1))
    return distances
```

For each pixel in our image:

1. We calculate its Euclidean distance to each centroid
2. This gives us a measure of how similar each pixel is to each cluster

```python
def assign_clusters(self, distances):
    return np.argmin(distances, axis=1)
```

Each pixel is assigned to the cluster with the minimum distance:

- If a pixel's 'a' value is closer to centroid 0, it's assigned to cluster 0
- If it's closer to centroid 1, it's assigned to cluster 1

```python
def update_centroids(self, X, labels):
    n_features = X.shape[1]
    new_centroids = np.zeros((self.k, n_features))
    for i in range(self.k):
        cluster_points = X[labels == i]
        if len(cluster_points) > 0:
            new_centroids[i] = np.mean(cluster_points, axis=0)
        else:
            new_centroids[i] = X[np.random.randint(X.shape[0])]
    return new_centroids
```

For each cluster:

1. We find all pixels assigned to that cluster
2. We compute the mean 'a' value of those pixels
3. This mean becomes the new centroid

```python
def fit(self, X):
    self.centroids = self.initialize_centroids(X)
    for _ in range(self.max_iterations):
        distances = self.compute_distance(X, self.centroids)
        labels = self.assign_clusters(distances)
        old_centroids = self.centroids.copy()
        self.centroids = self.update_centroids(X, labels)
        if self.has_converged(old_centroids, self.centroids):
            break
    return self
```

The algorithm repeats the previous steps until:

1. The centroids stop changing significantly (convergence)
2. Or we reach the maximum number of iterations

After convergence, each pixel in the image is assigned to one of our k clusters.

4.2.5 Step 5: Image Segmentation

```
# Reshape labels to original image dimensions
labels = kmeans.predict(features)
labels = labels.reshape(height, width)
```

The cluster assignments are reshaped back to the original image dimensions, creating a 2D array where each position contains the cluster ID (0 or 1) for that pixel.

```
def segment_image(kmeans, features, image_shape, channel=None):
    # Predict clusters
    labels = kmeans.predict(features)

    # Reshape labels to original image shape
    labels = labels.reshape(image_shape[0], image_shape[1])

    # Create a grayscale segmented image
    # Sort clusters by size
    unique, counts = np.unique(labels, return_counts=True)
    cluster_sizes = dict(zip(unique, counts))
    sorted_clusters = sorted(cluster_sizes.items(), key=lambda x: x
    [1], reverse=True)

    # Map clusters to grayscale values (largest = black, smallest =
     white)
    cluster_map = {}
    for i, (cluster, _) in enumerate(sorted_clusters):
        gray_value = 0 if i == 0 else 255  # Binary: black and
    white
        cluster_map[cluster] = gray_value

    # Create segmented image
    segmented_image = np.zeros(image_shape[:2], dtype=np.uint8)
    for cluster, gray_value in cluster_map.items():
        segmented_image[labels == cluster] = gray_value

    return labels, segmented_image
```

This function:

1. Creates a new empty image
2. Fills each pixel with a color based on its cluster:
    - Larger cluster (usually background) = black (0)
    - Smaller cluster (usually the number) = white (255)

4.2.6 Step 6: Number Extraction

```python
def extract_number(labels, k, invert=False):
    # Find cluster sizes
    unique, counts = np.unique(labels, return_counts=True)
    cluster_sizes = dict(zip(unique, counts))

    # Sort clusters by size (largest first)
    sorted_indices = np.argsort([cluster_sizes[i] for i in range(k)
    ])[::-1]

    # For k=2, the smaller cluster is usually the number
    if k == 2:
        number_cluster = sorted_indices[-1]  # Smallest cluster
    else:
        # For k>2, often the second smallest is the number
        number_cluster = sorted_indices[-2]

    # Get binary mask for the number cluster
    number_mask = (labels == number_cluster).astype(np.uint8) * 255

    # Invert if requested
    if invert:
        number_mask = 255 - number_mask

    return number_mask
```

Key insights about this step:

1. For Ishihara tests, the number typically occupies fewer pixels than the background
2. So we identify the cluster with the second-largest number of pixels
3. We create a binary mask where pixels in the number cluster are white (255) and others are black (0)

```python
# Post-process the mask to clean noise
kernel = np.ones((5, 5), np.uint8)
number_mask = cv2.morphologyEx(number_mask, cv2.MORPH_OPEN, kernel)
number_mask = cv2.morphologyEx(number_mask, cv2.MORPH_CLOSE, kernel
    )
```

Morphological operations help clean up the extracted number:

1. Opening (erosion followed by dilation) removes small isolated dots
2. Closing (dilation followed by erosion) fills small holes in the number

4.3 Cluster Selection Logic

The logic for identifying which cluster represents the number is critical:

```
# For k=2 (typical case)
if k == 2:
    number_cluster = sorted_indices[-1]  # Smallest cluster
else:
    # For k>2, often the second smallest is the number
    number_cluster = sorted_indices[-2]
```

This handles different scenarios:

1. With k=2: We assume the larger cluster is the background and the smaller is the number
2. With k¿3: We assume the largest is background, smallest might be noise, and second-smallest is the number

4.4 Optimizing the Process with Different Color Spaces

For challenging Ishihara tests, we can implement parameter optimization:

```
def recommend_color_space(image):
    # Convert to all color spaces
    rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)
    ycrcb = cv2.cvtColor(image, cv2.COLOR_BGR2YCrCb)

    # Calculate variance for each channel (higher variance \
    textrightarrow{} better discrimination)
    variances = {
        'lab_1': np.var(lab[:,:,1]),  # a channel
        'ycrcb_1': np.var(ycrcb[:,:,1]),  # Cr channel
        'hsv_0': np.var(hsv[:,:,0]),  # Hue channel
        'rgb_0': np.var(rgb[:,:,0]),  # Red channel
        # ... other channels
    }

    # Return recommendations sorted by variance
    return sorted_recommendations
```

This function:

1. Converts the image to multiple color spaces
2. Measures the variance in each channel (higher variance suggests better discrimination potential)
3. Recommends the channels with highest variance
4. Prioritizes Lab a-channel and YCrCb Cr-channel based on empirical effectiveness

5 Technical Considerations

5.1 Feature Dimensionality

Using a single channel reduces dimensionality:

- RGB: 3 dimensions
- Lab (a-channel only): 1 dimension

This makes clustering:

1. Faster (fewer calculations)
2. More stable (less sensitive to initialization)
3. More effective for this specific task

5.2 Euclidean Distance vs. Other Metrics

We use Euclidean distance because:

1. It works well in 1D space (which we have after channel extraction)
2. It's computationally efficient
3. It gives intuitive results for color clustering

5.3 Results Interpretation

The final result is a binary mask where:

- White pixels (255) represent the extracted number
- Black pixels (0) represent the background

From this mask:

1. The number becomes clearly visible
2. It's much more distinct than in the original image
3. Even someone with color blindness could now see the number

6 Practical Applications

6.1 Medical and Vision Testing

- Automated screening for color blindness
- Analysis of test results for research
- Development of personalized vision tests

6.2 Educational Tools

- Interactive demonstrations of color perception
- Learning tools for ophthalmology students
- Visual explanations of color spaces and image processing

6.3 Accessibility Applications

- Tools to help color-blind individuals see hidden patterns
- Image enhancement for accessibility

6.4 Computer Vision Research

- Benchmark for image segmentation algorithms
- Study of color-based feature extraction
- Development of more sophisticated clustering methods

7 Advanced Concepts and Extensions

7.1 Beyond Basic K-means

- **K-means++**: Better initialization for more consistent results
- **Fuzzy K-means**: Allowing partial membership in multiple clusters
- **Hierarchical Clustering**: Building nested clusters

7.2 Advanced Color Processing

- **Color Space Transformations**: Custom transformations to enhance specific features
- **Adaptive Thresholding**: Dynamic thresholds based on image properties
- **Color Correction**: Preprocessing to standardize colors across different images

7.3 Machine Learning Integration

- **Supervised Classification**: Training models to recognize extracted numbers
- **Deep Learning**: Using convolutional neural networks for end-to-end extraction
- **Ensemble Methods**: Combining results from multiple algorithms

8 Conclusion

The extraction of numbers from Ishihara color blindness test images using K-means clustering is a fascinating application that brings together computer vision, unsupervised learning, and color theory. By understanding the properties of color spaces and how they interact with the design of Ishihara tests, we can develop effective algorithms to automatically extract the hidden patterns.

This approach effectively "decodes" the Ishihara test, making the hidden information accessible to everyone regardless of color vision ability. The technique can be extended to handle a wide variety of Ishihara tests by optimizing the color space, channel selection, and clustering parameters.