SVD & PCA – The Scrabble Board Problem

1 Introduction

We extend the ideas of **Principal Component Analysis (PCA)** and **Singular Value Decomposition (SVD)** to an applied image recognition task, the automatic detection and classification of letters on a Scrabble board.

1.1 Motivation

- Automate board-state entry to save time and reduce human error.
- Extract tiles and letters from a single photo with minimal user input.
- Use PCA/SVD (**eigenletters**) to capture dominant stroke patterns for compact, interpretable features.
- Analogous to Eigenfaces: represent each letter as a linear combination of principal components for efficient recognition.

1.2 Approach

1. **Image Acquisition:** Capture a Scrabble board image under typical lighting conditions and surrounding environment.
2. **Image Preprocessing:** Convert the board image to grayscale, suppress noise, and enhance contrast.
3. **Board Detection and Alignment:** Identify the board contour, find the four corners, and apply a homography transform to obtain a top-down view.
4. **Grid Segmentation:** Divide the aligned board into a 15×15 grid of square tiles.
5. **Letter Extraction and Normalization:** Isolate each tile, normalize size and intensity, and convert the image to a vector form suitable for PCA.
6. **PCA and Eigenletters:** Compute the principal components of the training letter set to form a compact basis for letter representation.
7. **Classification:** Project new letter images into the PCA subspace and classify them using distance-based comparison by k-NN.

2 Background and Related Work

2.1 From Eigenfaces to Eigenletters

- The original **Eigenfaces** method used PCA to represent faces as linear combinations of basis images.
- Each face could be reconstructed from a few principal components, capturing most of the variation.
- Extends to letters. Instead of face features (eyes, nose, etc.), PCA captures key stroke patterns like curves, vertical bars, or cross lines.

2.2 Key Papers

Turk & Pentland (1991) - "Eigenfaces for Recognition" The original eigenface formulation that introduced PCA-based recognition for faces, forming the conceptual foundation extended later to text and letters.

Müller & Herbst (1998) - "The Use of Eigenpictures for OCR" Showed that the eigenface technique could be applied to optical character recognition by treating each letter as its own image class. They demonstrated that only a few principal components were needed to represent different characters efficiently, proving PCA's potential for compact feature extraction.

Van Hecke (2011) - "Teaching PCA Through Letter Recognition" Applied PCA to a dataset of all 26 capital letters and found that the first ten components captured almost 90% of total variance. Her results also showed that similar letters (e.g., O and Q, or E and F) appeared close together in PCA space, confirming that PCA can reveal structural similarity among characters.

Mudrová (2005) - "Automatic Rectification of Perspective Distortion from a Single Image Using Plane Homography" Provided methods for correcting perspective distortion in images, crucial for accurately segmenting the Scrabble board into tiles before letter recognition.

3 PCA Overview

Principal Component Analysis (PCA) is a statistical and machine-learning technique for *dimensionality reduction*. It transforms a dataset with possibly correlated variables into a smaller set of *uncorrelated variables* called **principal components (PCs)** that capture the maximum variance (information) in the data.

PCA simplifies large datasets while preserving their essential structure, **making visualization and computation more efficient**.

3.1 Goal

- PCA finds p orthogonal principal components ordered so that PC_1 captures the most variance, PC_2 the next most, and so on.
- Geometrically: principal components are the *directions of maximum data spread* (they maximize variance or the sum of squared distances along the axis).
- Numerically: principal components are linear combinations of the original features that best represent the data's variability.

3.2 Mathematical Definition

Let X be an $n \times p$ data matrix (rows = observations, columns = variables), centered so each column has mean 0.

Covariance Matrix

We need to underliestand how variables relate to each other. The covariance matrix is a $p \times p$ symmetric matrix defined as:

$$\Sigma = \frac{1}{n-1} X^T X$$

where Σ_{ij} measures how variables i and j vary together.

Example Convariance Matrix:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \text{Cov}(x_1, x_2) & \cdots & \text{Cov}(x_1, x_p) \\ \text{Cov}(x_2, x_1) & \sigma_2^2 & \cdots & \text{Cov}(x_2, x_p) \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}(x_p, x_1) & \text{Cov}(x_p, x_2) & \cdots & \sigma_p^2 \end{bmatrix}$$

The $Cov(x_i, x_i)$ is the variance of variable $x_i = \sigma_i^2$.

We are interested in the sign of the covariance. Positive covariance means the variables increase together, while negative covariance means they vary inversely.

Eigen Decomposition

By solving the optimization problem of maximizing variance along new axes using Lagrange multipliers, as follows:

$$\max_v \; v^T \Sigma v \quad \text{subject to } v^T v = 1$$

we find that the directions v_i that maximize variance are the eigenvectors of the covariance matrix Σ.

The principal components are obtained from the eigen-decomposition

$$\Sigma v_i = \lambda_i v_i$$

where λ_i is the variance explained by eigenvector v_i. Eigenvectors (v_i) define the new axes (principal directions), and eigenvalues (λ_i) quantify how much variance each direction captures.

3.3 Step-by-Step PCA Procedure

Step 1: Standardization

Since PCA is sensitive to scale, each variable must contribute equally:

$$z_{ij} = \frac{x_{ij} - \bar{x}_j}{s_j}$$

where $\bar{x}_j$ and s_j are the mean and standard deviation of variable j. After standardization, all features have mean 0 and variance 1.

Step 2: Compute the Covariance Matrix

$$\Sigma = \frac{1}{n-1} Z^T Z$$

reveals relationships between features. If $\text{Cov}(x_i, x_j) > 0$, they increase together; if negative, they vary inversely.

Step 3: Eigen Decomposition

Solve for eigenvalues λ_i and eigenvectors v_i of Σ.

- $v_i \rightarrow$ principal direction.
- $\lambda_i \rightarrow$ variance captured by that direction.

Sort eigenvectors by descending λ_i to rank importance.

Step 4: Feature Selection and Feature Vector

Form the **feature vector** W by choosing the k most significant eigenvectors:

$$W = [\, v_1 \; v_2 \; \ldots \; v_k \,]$$

Choosing $k < p$ reduces dimensionality while preserving most variance. The proportion of variance explained by component i is:

$$\text{PVE}_i = \frac{\lambda_i}{\sum_{j=1}^{p} \lambda_j}$$

Step 5: Projection of Data

Project the standardized data onto the new axes:

$$Y = ZW$$

The matrix Y represents the data in the reduced k-dimensional space of principal components.

4 Image Preprocessing

With PCA as the thoeretical foundation, we now describe the practical steps to prepare Scrabble board images for letter recognition.

4.1 Goal

Prepare the raw board image for reliable contour and corner detection. Preprocessing removes noise, corrects lighting, and emphasizes the Scrabble board boundaries. We tried different orders of filters and found this sequence worked best.

4.2 Step 1. Grayscale Conversion

Simplifies processing by reducing the image to intensity values. Color is not needed for edge or contour detection.

4.3 Step 2. Gaussian Blur

Applies a 2D Gaussian filter:

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

Smooths high-frequency noise and small variations in brightness. Replaces each pixel with a weighted average of its neighbors following a normal distribution.

We found that applying a Gaussian blur in two steps, first at the start and then after applying a contrast boost, worked best to reduce noise while preserving edges.

4.4 Step 3. Contrast Enhancement

Uses `adapthisteq` to boost local contrast implementsed via CLAHE (Contrast Limited Adaptive Histogram Equalization). This method divides the image into tiles and for each:

1. Computes a histogram of pixel intensities
2. The histogram is "clipped" to limit noise amplification and excess values are redistributed to other bins evenly.
3. Then smoothed using a cumulative distribution function (CDF) that finds how much of the tile's total pixel count is below each intensity level.
4. This reamps each dark pixel value to slightly brighter ones (and vice vers) to fill the entire 0-255 range of that loca ltile. Basically, compute the intensity distrubtion and spread it out to use the full range.
5. Then it blends the new reampped tiles together using bilinear interpolation with neighboring tiles to avoid abrupt changes at tile borders.

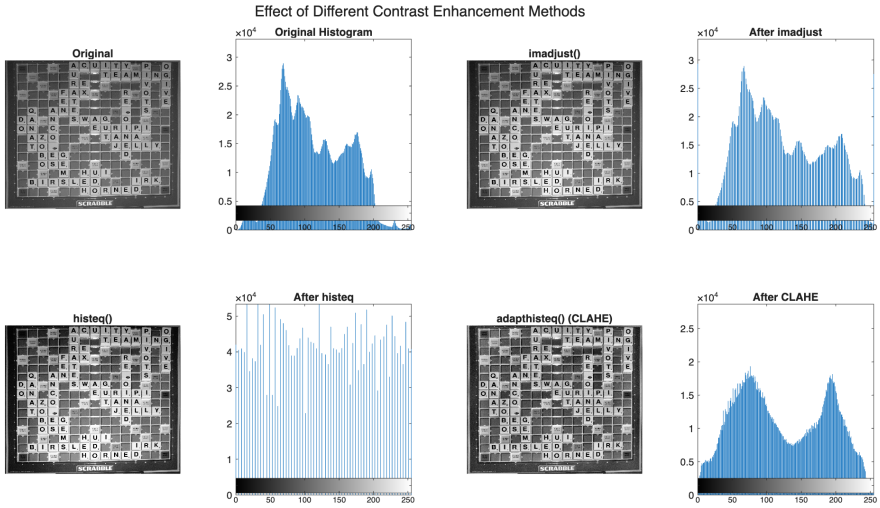


Figure 1: Comparison of contrast enhancement methods.

- Original image
  - The original image and histogram has strong lighting differences with brightness on one side and darker on the other and clustered into a few peaks.
  - The image is empty near the lower and upper intensity ranges, meaning the image doens't use the full brightness range. **low contrast, uneven lighting**.

`imadjust`

  - Stretches the histogram horizontally to use the full range, but does not fix local lighting issues.
  - Can see slightly strong intesities but roughly the same uneven lighting problem. **Increases global contrast only**.

`histeq`

  - Flattens the histogram to make all intensity levels equally likely, but it makes some parts harder to read and dark and light areas still get exaggerated.
  - Maximises global contrast but **over-equalizes and introduces noise** again.

`adapthisteq`

  - Locally enhances contrast in small regions, preserving details across the board. Perfect for images with uneven lighting by adjusting constrat locally not globally.
  - The histogram is wider and more balanced. There's two borad peaks not sharp but smooth. And visually the image is much more evenly lit. We wanted to amplify contrast but done evenly for the features to stand out
  - **Best local contrast enhancement without over-amplifying noise**.

4.5 Step 4. Median Filter

Non-linear smoothing filter that replaces each pixel with the median of its local neighborhood. Removes isolated specks or salt-and-pepper noise while preserving edges. Implemented as `medfilt2(img, [5 5])`.

While Gaussian blur smooths overall texture, acting as the first pass to avergae things out, the median filter specifically targets small, isolated noise points without blurring edges. The image at this point doesn't need to be sharp. We're not preserving letters but smoothing out the main board area for contour detection.

4.6 Step 5. Morphological Opening

Performs erosion followed by dilation:

$$I_{\text{open}} = (I \ominus B) \oplus B$$

where I is the image and B is a structuring element (e.g., a disk), $\ominus$ is erosion, and $\oplus$ is dilation.

It means first shrink (erode) the brightner foreground regions by B, then expand (dilate) what remains by B again.

Effect

- Removes small bright spots smaller than B
- Preserves larger structures that can contain B, making them more distinct.
- Smooths the boundaries of foreground objects and eliminates thin protrusion like glare or reflections.

We apply `imopen(img, strel('disk', 4))`. Any bright region or "blob" smaller than that disk will be removed or greatly reduced but large surfaces like the board remain mostly intact.

Why this helps

- After smoothing and median filtering, there may still be small bright specks (reflections, letters, highlights) or textural blobs that are not part of the main board region (which is large and uniform).
- Morphological opening removes these small bright artifacts while keeping the overall board shape intact.
- It also helps to flatten both the board region and surrounding objects, making the board boundary more dominant and easier to detect such as in cases with carpet or clutter around the board, that would otherwise dominate the edge map, causing the edge detector to pick up too many irrelevant contours.
- Because the board surface is large compared to the structuring element, the opening doesn't erode it away, but removes the smaller unwanted foreground artifacts.

4.7 Result

After preprocessing, the board has clean edges, uniform lighting, and clear intensity transitions. This prepares the image for reliable contour detection and warping in the next step.

5 Board Detection

5.1 Goal

After preprocessing, we have tried to make it easier to detect the playable board boundary. Now we extract edges, identify the board contour, and compute a convex hull to refine it.

5.2 Step 1: Edge Detection

We binarize the image using the Canny edge detector:

```
edges = edge(img, 'Canny', [lowThreshold highThreshold])
```

This method looks for the gradient or how steeply brightness changes at every pixel. If you imagine x and y plane of the image, its brigthness is like a topographic map. The Canny detector finds the steepest slopes (edges) by looking for places where the brightness changes rapidly.

Let the image be $I(x, y)$. The partials in the x and y directions are computed as:

$$G_x = \frac{\partial I}{\partial x}, \quad G_y = \frac{\partial I}{\partial y}$$

- Then the gradient magnitude and direction are:

$$M(x, y) = \sqrt{G_x^2 + G_y^2}$$

- Along the gradient, it is looking for local maxima where the brightness changes most sharply. It uses two thresholds to classify strong and weak edges, and only keeps weak edges that are connected to strong ones (hysteresis).
- We set a threshold range so anything above the high threshold is definitely an edge, anything below the low threshold is definitely not an edge, and anything in between is only kept if it connects to a strong edge.
- Finally, MATLAB follows all connected pixels that meet these conditions, keeping the continuous ones and deleting the rest. The result is a binary image where white = edge pixels and black = everything else.

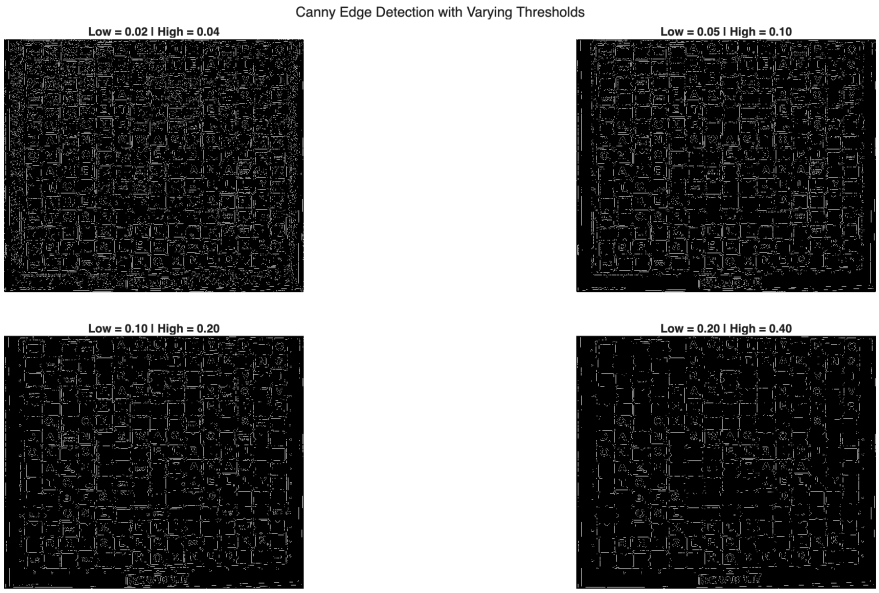


Figure 2: Image with different levels of canny strength applied.

5.3 Step 2: Dilation

After detecting edges, some may be broken or thin. We'll need as strong and continuous edges as possible to find the board contour reliably. To strengthen edges, we apply morphological dilation:

$$I_{\text{dilated}} = I \oplus B(x, y) = \max_{(s,t)\in B} I(x - s, y - t)$$

with 0 = background, 1 = foreground, and B is a structuring element.

We apply `imdilate(bw, strel('rectangle', [3,3]))`. Imagine the structuring element B sliding over the image. Wherever it overlaps with a white pixel (edge), the center pixel becomes white. This thickens edges and fills small gaps.

5.4 Step 3: Filling Holes

Hole filling identifies enclosed background (black) regions within foreground (white) areas and flips them to foreground:

$$I_{\text{filled}} = I \cup \text{holes}(I)$$

where holes(I) finds all background pixels completely surrounded by foreground pixels. Conceptually, we're finding the connected components of background pixels that do not touch the image.

We apply `imfill(bw, 'holes')`. This ensures that any gaps or holes inside the detected board contour are filled in, producing a solid shape for the board.

Why this helps

- After dilation, you have the thickened edge mask but the interior board boundary is almost always "windowed" by edges.

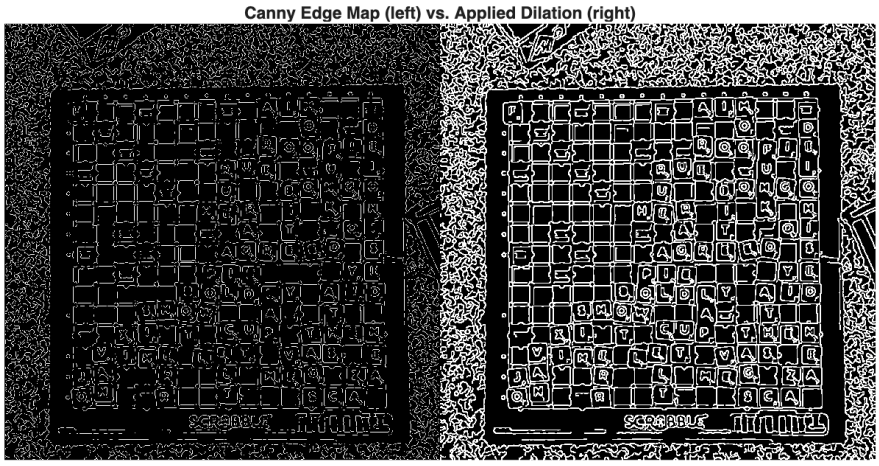


Figure 3: Showing the effects Canny edge detection and dilation without hole filling

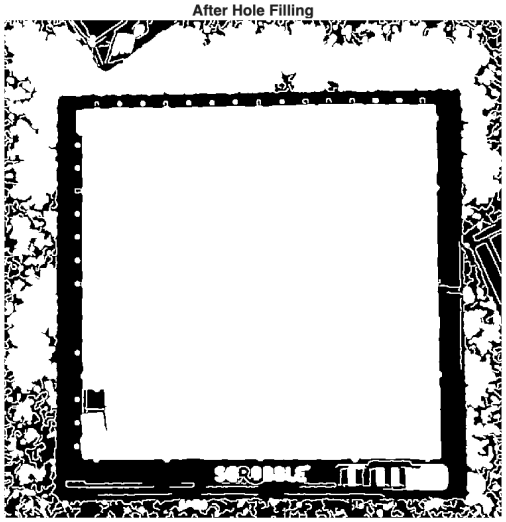


Figure 4: Image after hole filling, showing a complete board contour.

- Hole-filling converts that interior region to foreground, producing a relatively clean blob representign the board area.
- Without this, the contour detection step may find multiple disconnected board pieces (or not even the board at all) instead of a single continuous board shape.
- In Figures 3 and 4, wihtout hole filling the next steps determined the largest contour was the carpet area, not the board. Afterwards, the carept is a clean white shape that can be ignored and the largest contour is the board.

5.5 Step 4: Largest Contour

Labeling Regions

From the filled binary mask, we label each region using `bwlabel`.

- Creates a matrix that scans the image row by row, comparing each pixel to its neighbors.
- If a pixel is white (foreground) and connected to an already labeled region, it gets the same label. If not, it starts a new label.
- The outputted matrix has integers representing different connected components i.e. all the hwite pixels in one region will be labeled "1", the next region "2", etc.

Finding the Largest Region

1. Compute the area of each labeled region using `regionprops`, which counts the number of pixels in each connected component.
2. Find the index of the label with the largest area using `max([stats.Area])`
3. Ceate a binary mask that keeps only that largest region with `ismember(labeled, idx)`, which goes over the labeled matrix and checks if it matches that index.

5.6 Step 5: Convex Hull

The binary mask of the largest region could be left as is, but we found this step helped with corner detection later on to best apprxoimate the area of the board in the case where it's jagged, dented, or still has small holes.

We apply a convex hull, which is a shape that wraps around the outermost points of the region, forming the smallest convex polygon that contains it:

$$I_{\text{hull}} = (\text{convhull}\,\{(x, y) : I_{\text{largest}}(x, y) = 1\})$$

In MATLAB, this is implemented with `bwconvhull`, which takes the binary mask and returns a new binary mask representing the convex hull of the foreground region.

5.7 Result

We should now have a clean binary mask containing only the largest connected component, which should correspond to the Scrabble board. If it chooses something else or includes more than the playable area the next steps will diverge and it means the pre-processing steps failed to differentiate the board from the background well enough.

6 Corner Detection

6.1 Goal

Identify the four outer corners of the Scrabble board by computing the convex hull of the detected board region and applying PCA to find its principal directions.

6.2 Step 1: Hull Boundary

From the convex hull binary mask, we extract the boundary pixels using `bwboundaries`, which returns an array of (x, y) coordinates sets for each connected component's edge.

We select the first element `boundary = B{1}` because we only have one region (the board) and where boundary is of size $n \times 2$.

It uses a neighbor tracking algorithm that, without going into too much detail, can be described as following contours like so:

If R is a connected set of foreground pixels then:

$$\text{boundary} = \{(x, y) \in R : \exists (x', y') \in N(x, y) \text{ such that } B(x', y') = 0\}$$

where $N(x, y)$ are the 8-connected neighbors of pixel (x, y).

That is, the boundary consists of all pixels in R that have at least one neighboring pixel belonging to the background. **Before we had a region, now we have a path**.

6.3 Reducing to Vertices

- We found that the boudnary returned a very large set, about as many points as pixel edges.
- The bw boundary was too detailed because while it is a convex polygon and worked great for fully containing the desired region, for computing the corners, it through off the corner detection sometimes.
- All it does it include the neighboring outlined pixels of the largest region. We want the smooth outermost subset, so we apply `convhull` explcitly on the boundary points set.

.

```
k = convhull(boundary(:,2), boundary(:,1))
hullPoints = [boundary(k,2), boundary(k,1)]
```

MATLAB, it applies the Quickhull algorithm, which is a divide-and-conquer method that finds the convex hull of a set of points in 2D space. It works by recursively finding the farthest points from a line segment and partitioning the remaining points into subsets until all points are processed.

Another approach I was fimilar with was Graham's scan, which sorts the points by angle from a reference point and then constructs the hull by iterating through the sorted list and maintaining a stack of candidate hull points. However, Quickhull is generally more efficient for larger datasets.

6.4 Step 3: Center the Points

We have the coodinates of the hull points $P = \in \mathbb{R}^{N\times 2}$.

We compute the centroid (mean point) of these hull points:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} P_i$$

then center the points by subtracting the centroid: $P_0 = P - \mu$.

This ensures that the origin is at the center of the hull, the center of the board, so that when we compute covariance in the next step, the spread is relative to the center of the board and not the image.

6.5 Step 4: PCA on Hull Points

Initially corner detection used

```
regionprops(bw_hull, 'BoundingBox', 'Extrema')
```

which would create a bounding box and return the 8 extreme points. We'd then pick the four corners from those. However, this often failed when the board was skewed, as images often are.

Instead, we apply PCA to the centered hull points P_0. This application of PCA is more traditional than the image-based PCA later on.

Covariance Matrix

Right now we have a spread of points resembling the outline of a board and the covariance matrix gives how these points vary in x and y:

$$\Sigma = \frac{1}{N-1}P_0^T P_0 = \begin{bmatrix}\sigma_{xx} & \sigma_{xy}\\ \sigma_{yx} & \sigma_{yy}\end{bmatrix}$$

Eigen Decomposition

Computing the eigendecomposition gives us:

$$CV = VD$$

where V contains the eigenvectors (principal directions) and D contains the eigenvalues (variances along those directions).

We sort the eigenvalues in descending order and reorder the eigenvectors accordingly. The first eigenvector u corresponds to the direction of maximum variance (longest axis of the board), and the second eigenvector v is orthogonal to it.

Why this helps

- The principal directions give the orientation of the board in the image.
- By projecting the hull points onto these directions, we can find the extreme points along each axis, which correspond to the corners of the board.
- So here even if the board is rotated or skewed, PCA finds its dominant orientation and helps us locate the corners accurately.

6.6 Step 5: Projecting the Points

We specify the four corners in terms of the principal directions in the u-v basis such that thye form the diagonal extents:

$$d_1 = u + v \quad d_2 = u - v \quad d_3 = -u + v \quad d_4 = -u - v$$

Then we normalize these direction vectors to unit length:

$$\hat{d}_i = \frac{d_i}{\|d_i\|}$$

Next, we project all hull points onto each of these four directions:

$$p_{ij} = P_0 \cdot \hat{d}_i$$

where p_{ij} is the projection of hull point j onto direction i. The point with the maximum score in each direction lies furthest along that axis, meaning it corresponds to a corner of the board.

Finally, we retrieve the original coordinates of these corner points and order them clockwise so that when we warp the board later, the mapping is consistent (or else a rotation could occur).

6.7 Result

The output is the four corner coordinates of the Scrabble board in the original image space.

7 Grid Segmentation

7.1 Goal

Warp the board contour to a clean, top-down view of the Scrabble board and divide it into its 15×15 grid of tiles. This ensures each tile can later be processed individually for letter recognition.

7.2 Step 1: Projective Transformation

The four outer corners of the convex hull define the board boundary. Using these corners, we compute a 3×3 **homography matrix** H such that:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \quad H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

Here, we go from the Cartesian coordinates (x_i, y_i) to homogeneous coordinates, $(x_i, y_i, 1)$, we multiply by H to get (x'_i, y'_i, w_i). Therefore:

$$(x_{new}, y_{new}) = \left(\frac{x'_i}{w_i}, \frac{y'_i}{w_i} \right)$$

The coordinate mapping here is equal up to scale, so we must later normalize by dividing by the third coordinate.

What Homography Does

H defines a mapping between two planes that captures translations, rotations, scalings, shears, and perspective distortions. Since the board lies on a plane, any perspective image of it can be connected to a flat top-down view via such a homography.

Solving for H

We know the 4 target (corner) points. Each point correspondence gives two equations, so with 4 points we have 8 equations to solve for the 8 unknowns in H (since h_{33} can be set to 1).

$$x'_i(h_{31}x_i + h_{32}y_i + h_{33}) = h_{11}x_i + h_{12}y_i + h_{13}$$
$$y'_i(h_{31}x_i + h_{32}y_i + h_{33}) = h_{21}x_i + h_{22}y_i + h_{23}$$

Rearranging these gives a homogeneous linear system $Ah = 0$ where h is the vector of unknowns in H. We solve this using SVD, taking the right singular vector corresponding to the smallest singular value.

$$\begin{bmatrix} -x_i & -y_i & -1 & 0 & 0 & 0 & x'_i x_i & x'_i y_i & x'_i \\ 0 & 0 & 0 & -x_i & -y_i & -1 & y'_i x_i & y'_i y_i & y'_i \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = 0$$

This is done via the Direct Linear Transformation (DLT) algorithm, which does the following:

- Constructs the matrix A from the point correspondences.
- Computes the SVD of A: $A = U\Sigma V^T$.
- The solution h is the last column of V (corresponding to the smallest singular value).
- Reshapes h into the 3×3 matrix H.

We are only interested in the MATLAB implementation of this step as the full details of DLT are beyond the scope of this project.

MATLAB Implementation

First we define an array `target` with the desired corner coordinates in the warped image:

```
= (0, 0), (size, 0), (size, size), (0, size)
```

where `size` is the desired width/height of the squre warped board image. Then we use `fitgeotrans` to compute the homography and `imwarp` to apply it:

```
tform = fitgeotrans(corners, target, 'projective');
warped = imwarp(gray, tform, 'OutputView', imref2d(outputSize));
```

7.3 Step 3: Grid Division

Once the board is flattened, divide int into 15x15 tiles for the standard Scrabble layout.

7.4 Result

The output is a perfectly aligned, cropped board image, divided into 225 uniform tiles. Each tile can now be analyzed independently for letter detection and classification. Assumes the correct largest contour and corners were found or otherwise the warping will be off.

8 Letter Normalization

After the board is warped and divided into its 15×15 grid, each tile must be evaluated and converted into a standardized binary format suitable for PCA. The updated pipeline consists of four stages:

1. Filter 1 - raw grayscale tile filter
2. Normalization — contrast boost, thresholding, component extraction, centering
3. Filter 2 — shape-based letter filter
4. Vectorization into column vectors

This replaces the earlier simple brightness/threshold approach.

8.1 Raw Tile Filter (Filter 1)

Given a grayscale tile T of size $h \times w$, we compute:

Mean brightness

$$0.10 \leq \text{mean}(T) \leq 0.95$$

Proportion of dark pixels

$$0.01 \leq p_{\text{dark}} = \frac{\#\{T < 0.55\}}{hw} \leq 0.60$$

Edge density

$$0.003 \leq E = \text{mean}(\text{sobel}(T)) \leq 0.35$$

Only tiles satisfying all three criteria pass to normalization. This rejects blank tiles, bonus squares, extremely bright/dark tiles, and non-letter noise.

8.2 Normalization Pipeline

For each tile passing Filter 1, we apply the full normalization routine:

1. **Contrast enhancement**

$$T_{\text{boost}} = \text{imadjust}(T)$$

2. **Thresholding using scaled Otsu threshold**

$$T_{\text{thresh}}(i,j) = \begin{cases} 1, & T_{\text{boost}}(i,j) < 0.9\,\tau \\ 0, & \text{otherwise} \end{cases} \quad \text{where } \tau \text{ is Otsu's threshold}$$

3. **Connected component selection and rejection**

   For each component we compute:

$$\text{Area}, \quad \text{Height}, \quad \text{Aspect ratio}, \quad \text{Eccentricity}$$

   A component is rejected if:

$$\text{Area} < 0.04hw, \quad \text{Height} < 0.40h, \quad \text{Eccentricity} > 0.995, \quad \text{Aspect} \notin [0.2, 3.5]$$

   or if it touches the border of the tile.

   Among remaining components, we select:

$$\text{argmax}_i \ \text{Area}_i \cdot (\text{Height}_i)^{0.3}$$

4. **Cropping and centering**

   The selected component is cropped tightly and pasted into the center of a new $h \times w$ binary tile.

Let the final normalized tile be denoted by T_{final}.

8.3 Shape Filter (Filter 2)

We compute:

Fraction of white pixels

$$p_w = \frac{\#\{T_{\text{final}} = 1\}}{hw}$$

We require:

$$p_w \geq 0.045$$

Vertical height Let the vertical support be:

$$H = \max(rows) - \min(rows) + 1$$

We require:

$$H > 16$$

This eliminates fragments, noise blobs, and garbage resulting from centering.

8.4 Vectorization

Accepted tiles are flattened into column vectors:

$$x_i = \text{vec}(T_{\text{final}}) \in \mathbb{R}^{4096}$$

and stacked into:

$$X_{\text{tiles}} = [\, x_1 \; x_2 \; \ldots \; x_m \,].$$

8.5 Result

This updated pipeline:

- rejects almost all non-letter tiles,
- centers and normalizes glyph shapes consistently,
- removes border noise and multi-component artifacts,
- produces high-quality input for PCA projection and classification.

9 Letter Classification Using PCA and k-NN

9.1 Projection into PCA Space

Let U_k be the matrix of the top k PCA components (eigenletters), and let μ be the mean training image.

A centered image x is projected as:

$$z = U_k^T(x - \mu)$$

The full sets are:

$$Z_{\text{train}} = U_k^T(X_{\text{train}} - \mu), \quad Z_{\text{test}} = U_k^T(X_{\text{test}} - \mu).$$

9.2 Automatic Selection of k

We evaluate odd values:

$$k \in \{1, 3, 5, \ldots, 15\}$$

A 5-fold stratified cross-validation is performed on the training set. For each candidate k, we compute the prediction error. The chosen value is:

$$k^* = \arg\min_k \text{CVError}(k)$$

9.3 k-Nearest Neighbor Classification

For each projected test vector z_{test}, distances to all training vectors are computed:

$$d_i = \|z_{\text{test}} - z_i\|_2$$

The labels of the k^* nearest training vectors vote to determine the predicted letter.

9.4 Classification of Real Board Tiles

After preprocessing:

$$X_{\text{board}} = [\, x_1, x_2, \ldots]$$

each tile is projected and classified identically.

Predicted letters are inserted into a 15×15 textual grid:

$$G_{r,c} = \begin{cases} \text{predicted letter}, & \text{if tile accepted}, \\ \text{blank}, & \text{otherwise}. \end{cases}$$

9.5 Result

The PCA + k-NN system provides:

- strong performance on synthetic test splits,
- good generalization to real Scrabble board images,
- robustness due to the normalization and filtering pipeline,
- fast classification using only tens of PCA dimensions.

10 Experiment Setup and Results

10.1 Training Dataset

The updated synthetic dataset consists of:

- 26 capital letters (A–Z),
- 100 samples per class,
- 2600 images total,
- each image of size $64 \times 64 = 4096$ pixels,
- variations in rotation, brightness, and contrast.

The resulting data matrix is:

$$X \in \mathbb{R}^{4096 \times 2600}$$

with label vector:

$$L \in \{1, \ldots, 26\}^{2600}.$$

10.2 PCA Findings

Applying SVD to centered data:

$$X_c = X - \mu, \qquad X_c = U \Sigma V^T$$

The top 60 singular vectors (U) capture nearly all meaningful structure. The first several eigenletters represent:

- vertical strokes,
- loops and circular structures (O, Q),
- crossbars (E, F),
- diagonal structure (A, K, M, W).

Clusters in PCA space reveal visually similar groups such as:

$$\text{O–Q}, \quad \text{E–F}, \quad \text{P–R}.$$

10.3 Classification Performance

Synthetic test split

$$\text{Accuracy} \approx 90\% - 94\%$$

Real Scrabble board tiles Accuracy depends on lighting and tile condition, yet the filtering pipeline removes nearly all non-letter tiles, and classification of the remaining tiles is highly reliable.

10.4 Board Reconstruction

Predicted letters are reconstructed into the Scrabble layout using a 15×15 grid. Visualization consists of:

- a mosaic of processed tile images,
- a corresponding textual prediction grid,
- side-by-side comparison with synthetic samples.

11 Conclusion

This project showed how concepts from SVD and PCA can be applied end-to-end — from image preprocessing and homography alignment to eigenletter computation and classification. The method achieved strong accuracy while remaining mathematically transparent. It serves as a clear example of how linear algebra directly supports real-world computer vision tasks.

The system is simple and interpretable since each eigenletter has a clear visual meaning and works wlel on small datasets to illustrate core linear algebra ideas. However, the board detection has limitations as dicussed above and whose process can be streamlined or optimized further.

Results Summary

The PCA + k-NN system effectively recognized Scrabble letters using a compact, low-dimensional representation. The first few principal components captured dominant letter structures such as vertical strokes, loops, and crossbars, showing that PCA can represent visual data efficiently. Misclassifications mainly occurred between letters with similar geometry (e.g., E/F, O/Q, P/R), where shape differences were subtle or lighting inconsistent.

Modern Context

Modern deep learning and OCR methods would handle nonlinear distortions and variable lighting automatically, but they require far more data and computation. PCA remains valuable as an interpretable and data-efficient baseline that demonstrates how linear transformations alone can achieve meaningful image recognition.