SIMC 2024 Report John Monash Science School
Janneke Delhey Peters, Douglas Shuttleworth, Joel Tan

Task 1

Task 1a Below are the rotated images;

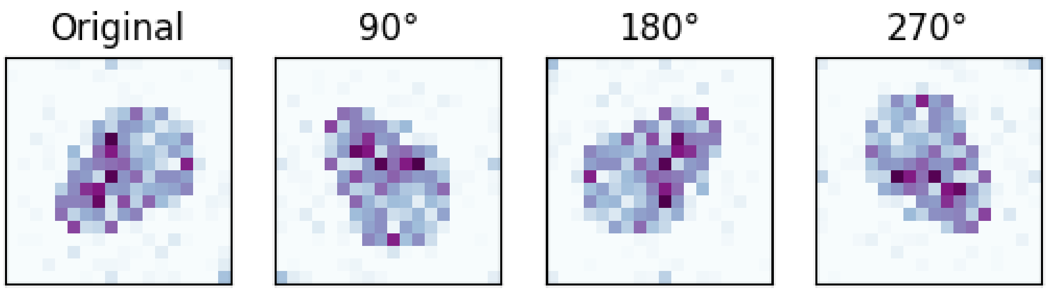


Figure 1: Orientations of Reference Image

Lemma 1 An algorithm to rotate a matrix by 90° clockwise involves the following two steps;
1. Swap the 1st and nth rows, 2nd and n-1st, etc.
2. Take the Transpose of the resulting matrix.

Operation 1 can be represented as multiplication by the anti-diagonal Identity Matrix.

$$T_1 = \begin{pmatrix} 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 1 & 0 \\ \vdots & \vdots & \iddots & \vdots & \vdots \\ 0 & 1 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

Therefore, the Matrix after rotation by 90° can be expressed as the following, with the final equality found as the transpose of the anti-diagonal Identity matrix is identical to itself;

$$A_{90^\circ} = (T_1 A)^T = A^T T_1^T = A^T T_1$$

Throughout our computation, the function `np.rot90()` (Harris et al., 2020) achieves the same result, with greater computational efficiency, whilst relying on the approach of applying Linear Transformations as outlined.

Task 1b

Orientation	0°	90°	180°	270°
# Cases	6	10	6	3

Task 1c Three copies of the master image were created by rotating the image 90°, 180° and 270° clockwise in order to create templates for the four orientation classes. All of the matrices in the set where compared to the four classes and sorted into the class corresponding to the template matrix they were identical to. The number of matrices in each class were counted to give the number of matrices with each orientation. (See Figure 2) Matrices were rotated by performing the Linear Transformations as outlined in Lemma 1, implemented in Python with `np.rot90()` throughout.

```
task_1_counts = [0, 0, 0, 0]

for i in task1:
    if np.array_equal(i, task_1_90):
        task_1_counts[1] += 1
    elif np.array_equal(i, task_1_180):
        task_1_counts[2] += 1
    elif np.array_equal(i, task_1_270):
        task_1_counts[3] += 1
    else:
        task_1_counts[0] += 1
```

Figure 2: Counting the Number of Patterns in Each Class

Task 2

Task 2a See Below

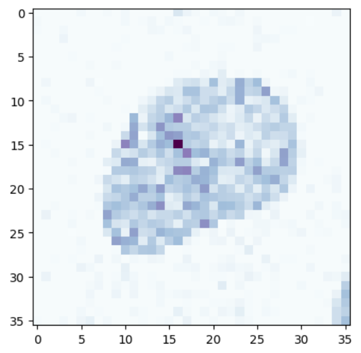


Figure 3: Rendered Master Image from Flattened Data

Task 2b

Orientation	0°	90°	180°	270°
# Cases	22	32	21	25

Task 2c The image was rendered by reshaping the flattened array into a 36x36 square matrix. This can be done by letting the first 36 elements be the first row of the matrix, the 2nd 36 elements be the second row and continuing in a similar fashion until 36 rows have been formed. In practice the `np.reshape` command was more efficient (See Figure 4).

```
task_2_original_img = np.reshape(task2[0], (36, 36))
plt.imshow(task_2_original_img, cmap="BuPu")
```

Figure 4: Reshaping Flattened Array

In order to find the number of patterns corresponding to each orientation class, each pattern was compared to flattened arrays of the rotated image at 0°, 90°, 180°and 270°, and the number of occurrences where a pattern was equal to each orientation class was counted - as per Task 1.

Task 3

Task 3a The image was a 33x33 square

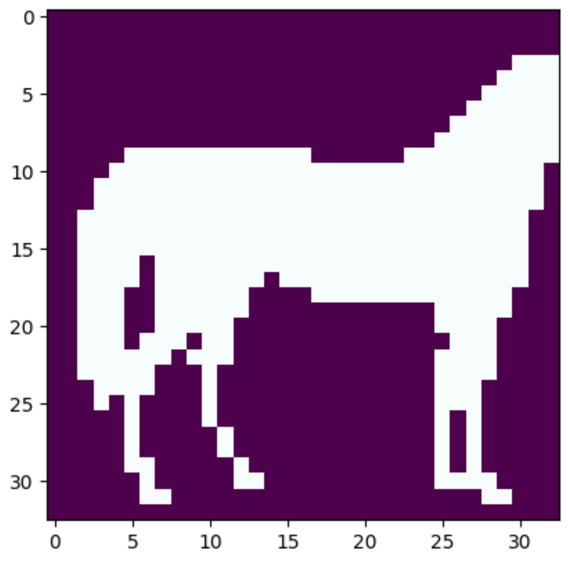


Figure 5: First image of task 3 - rendered

Task 3b

Orientation	0°	90°	180°	270°
# Cases	254	236	250	260

Task 3c See Below

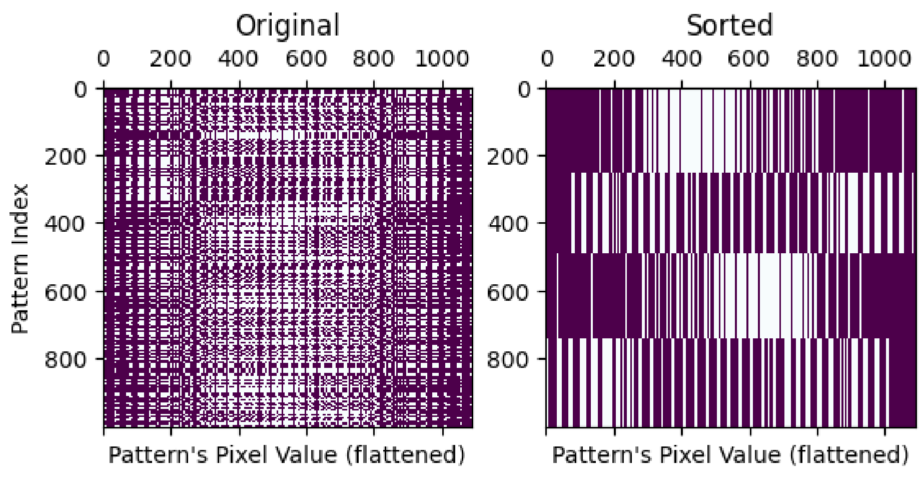


Figure 6: Unsorted and Sorted Design Matrices

Task 3d Using the factors of the size of the first element of the Task 3 array, several candidate shapes can be determined. Of these, the square 33x33 case returned an intelligible image. This image was then rotated into each orientation and flattened again in order to create four image classes, each with a different rotation with respect to the master image. Then the other row vectors were sorted into each of the classes and the number patterns in each class was counted. Finally, an array was created from each of the four image classes in order and plotted as above (Figure 6). The same clustering algorithm from Task 2 was used, instead of K-Means, as all patterns in each class were identical.

Task 4

Task 4a The average total pixel value is 1251.047.

```
task_4_row_sums = np.empty(1)
for i in task4:
    task_4_row_sums = np.append(task_4_row_sums, np.sum(i))
task_4_row_sums = task_4_row_sums[1:]
np.average(task_4_row_sums) # return
```

Figure 9: Average Row-Sum Code

Task 4b See below.

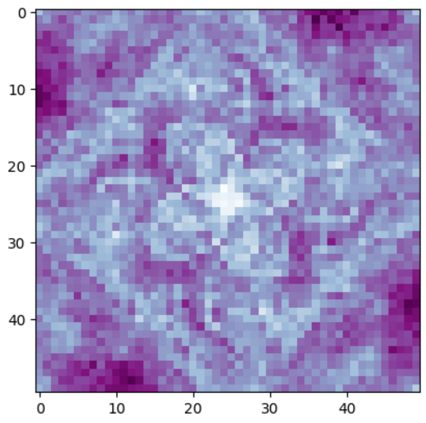


Figure 7: Rendered Image Assuming Same Orientation

Task 4c See below.

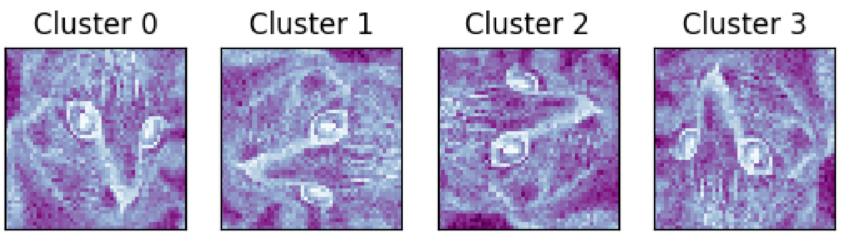


Figure 8: Each of the Four Orientation Classes - Rendered

Task 4d The sum of every row in the design matrix was appended to a list and the average value of the list was taken. This gives the average sum of pixel values for the image.

In order to render the image as if the orientations were identical, The average of each column of the design matrix was computed and the pixel corresponding to that column was assigned the average value. In order to sort the values, a K-Means algorithm was run on the set of all flattened row matrices in the design matrix. K-Means is a heuristic, unsupervised clustering algorithm which broadly functions by choosing a set number of initial points (in this case 4) at a maximum distance from each other (using `K-Means++` initialisation). Each datapoint is then assigned to its nearest cluster by Euclidean distance. The mean of each cluster is then recalculated to create a new 'centroid' and the algorithm repeats until the membership changes of each cluster become negligible.

```python
def sort_to_clusters(matrix: np.ndarray, labels: np.ndarray) -> list:
    sorting = [[], [], [], []]

    for i in range(matrix.shape[0]):
        sorting[labels[i]].append(matrix[i])

    orientations = []
    for i in range(4):
        orientations.append(np.array(sorting[i]))

    return orientations
```

Figure 10: Code for Sorting from K-Means Output

`numpy`'s K-Means class returns labels $\{0, 1, 2, 3\}$ after fitting. Each pattern is added to a list that represents its label. This list is converted back into `np.ndarray` for more efficient operations later.

Task 5

Task 5a Similarly to the likelihood of r = 0°, the likelihood of r = 90° can be expressed as the following;

$$\mathcal{L}(r = 90^\circ \mid K_{(4)}, \mu_{(4)}) \equiv Pr(k_1 \mid \beta\lambda)Pr(k_2 \mid \lambda)Pr(k_3 \mid \beta\lambda)Pr(k_4 \mid \beta\lambda) = \beta\lambda(1-\lambda)(1-\beta\lambda)^2$$

Task 5b From the results found in (a), and Eqn (3), the following expression can be derived simply

$$\frac{\mathcal{L}(r = 0^\circ \mid K_{(4)}, \mu_{(4)})}{\mathcal{L}(r = 90^\circ \mid K_{(4)}, \mu_{(4)})} = \frac{\lambda(1-\beta\lambda)^3}{\beta\lambda(1-\lambda)(1-\beta\lambda)^2} = \frac{1-\beta\lambda}{\beta(1-\lambda)} = \frac{1-\beta\lambda}{\beta-\beta\lambda}$$

Task 5c From setting the expression found in part (b) equal to 1, the value for β can be found via the following;

$$\frac{\mathcal{L}(r = 0^\circ \mid K_{(4)}, \mu_{(4)})}{\mathcal{L}(r = 90^\circ \mid K_{(4)}, \mu_{(4)})} = \frac{1-\beta\lambda}{\beta-\beta\lambda} = 1 \implies 1-\beta\lambda = \beta-\beta\lambda \implies \beta = \mathbf{1}$$

Task 5d Similarly to the 4 dimensional case, an expression can be obtained for the ratio of likelihoods in the 16 dimensional square case;

$$\frac{\mathcal{L}(aligned)}{\mathcal{L}(misaligned)} \equiv \frac{\mathcal{L}(r = 0^\circ \mid K_{(16)}, \mu_{(16)})}{\mathcal{L}(r = 90^\circ \mid K_{(16)}, \mu_{(16)})} = \frac{\lambda(1-\beta\lambda)^{15}}{\beta\lambda(1-\lambda)(1-\beta\lambda)^{14}} = \frac{1-\beta\lambda}{\beta-\beta\lambda}$$

An identical result to the 4 dimensional case, thus suggesting an equivalent ratio of probabilities between aligned, and misaligned, for a square of any n by n size.

Task 5e As there are no overlaps in the outlier tiles in the 0° and 90° cases, we can therefore form an expression similarly to as in the earlier parts of this question.

$$\frac{\mathcal{L}(aligned)}{\mathcal{L}(misaligned)} \equiv \frac{\mathcal{L}(r = 0^\circ \mid K_{(16)}, \mu_{(16)})}{\mathcal{L}(r = 90^\circ \mid K_{(16)}, \mu_{(16)})} = \frac{\lambda^3(1-\beta\lambda)^{13}}{(\beta\lambda)^3(1-\lambda)^3(1-\beta\lambda)^{10}} = \frac{(1-\beta\lambda)^3}{\beta^3(1-\lambda)^3} = \left(\frac{1-\beta\lambda}{\beta-\beta\lambda}\right)^3$$

Corollary Consider an n by n matrix with α outlier pixels, each with a probability of λ, with the remaining $n^2 - \alpha$ pixels having a probability of $\beta\lambda$. After rotating the matrix by 90° , let there be γ total overlaps. That is, outliers that exist in an identical location before, and after being rotated 90 degrees.

The assumption that all outliers will be filled at rotation 0° will follow throughout this corollary. As such, the following expression can be found.

$$\frac{\mathcal{L}(aligned)}{\mathcal{L}(misaligned)} \equiv \frac{\mathcal{L}(r=0^\circ \mid K_{(n^2)}, \mu_{(n^2)})}{\mathcal{L}(r=90^\circ \mid K_{(n^2)}, \mu_{(n^2)})} = \frac{\lambda^\alpha(1-\beta\lambda)^{n^2-\alpha}}{\lambda^\gamma(1-\lambda)^{\alpha-\gamma}(\beta\lambda)^{\alpha-\gamma}(1-\beta\lambda)^{n^2+\gamma-2\alpha}}$$

which simplifies to the following expression;

$$\frac{\mathcal{L}(aligned)}{\mathcal{L}(misaligned)} \equiv \frac{\mathcal{L}(r=0^\circ \mid K_{(n^2)}, \mu_{(n^2)})}{\mathcal{L}(r=90^\circ \mid K_{(n^2)}, \mu_{(n^2)})} = \left(\frac{1-\beta\lambda}{\beta-\beta\lambda}\right)^{\alpha-\gamma}$$

This aligns with the results as expected from the earlier elements of this task, as by simply setting $\alpha = (1,3)$ $\gamma = 0$, for parts (a) through (c), and (e) respectively, an identical result is achieved. Furthermore, it gains additional insight into the notion stated in part (c), that if the ratio between the likelihoods of 0 and 90 degrees occurring is equal to 1, does not necessarily imply that the confidence will be impacted. If we are to let α equal to γ, this expression is set to 1. Intuitively, this will occur when the image is symmetric under rotation, as all the pixels that are filled after rotating 90° are identical to those without rotation.

Task 5f As the order in which the pixels are arranged is not salient to the question, we will assume the first M pixels to have average value $\mu_i = \lambda$, and pixels $M+1$ through N to be $\mu_i = \beta\lambda$. Thus the expression for log-likelihood can be broken into two sums;

$$ln(\mathcal{L}(aligned \mid \vec{\mu}, \vec{k})) = \sum_{i=1}^{N} k_i ln(\mu_i) + (1-k_i) ln(1-\mu_i)$$

$$= \sum_{i=1}^{M} k_i ln(\lambda) + (1-k_i) ln(1-\lambda) + \sum_{i=M+1}^{N} k_i ln(\beta\lambda) + (1-k_i) ln(1-\beta\lambda)$$

As β, λ are constant, the terms involving β, λ can be taken out as factors from the sums, giving the following result.

$$= \sum_{i=1}^{M} (k_i(ln(\lambda) - ln(1-\lambda)) + ln(1-\lambda) + \sum_{i=M+1}^{N} (k_i(ln(\beta\lambda) - ln(1-\beta\lambda)) + ln(1-\beta\lambda)$$

$$= (ln(\lambda) - ln(1-\lambda)) \sum_{i=1}^{M} (k_i) + Mln(1-\lambda) + (ln(\beta\lambda) - ln(1-\beta\lambda)) \sum_{i=M+1}^{N} (k_i) + (N-M) ln(1-\beta\lambda)$$

As $\sum_{i=1}^{M} k_i$ is equivalent to $M * E(k)$, provided M, (and for the second summation, $N-M$) are sufficiently large, the simplified equation in terms of N, M, β, λ is given;

$$= (ln(\lambda) - ln(1-\lambda))(M\lambda) + Mln(1-\lambda) + (ln(\beta\lambda) - ln(1-\beta\lambda))(N-M)(\beta\lambda) + (N-M) ln(1-\beta\lambda)$$

For the purposes of plotting this result, arbitrarily let $N = 90$, $M = 10$

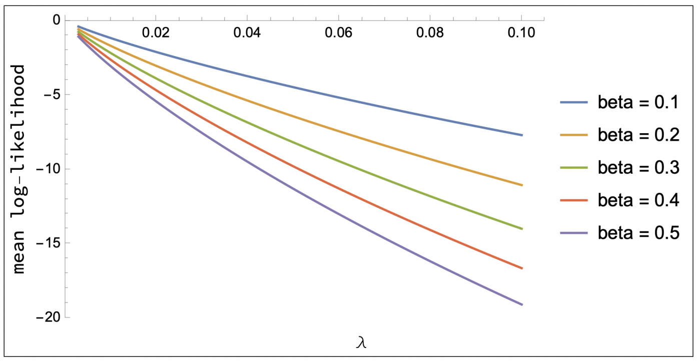


Figure 11: Plot of mean log likelihood with respect to λ

Consider the edge case $M = N$. Thus the expression for log-likelihood can be simplified to;

$$ln(\mathcal{L}(aligned \mid \vec{\mu}, \vec{k})) = Nln(1-\lambda) - \lambda Nln(1-\lambda) + \lambda Nln(\lambda)$$

This is independent of the value for β, which intuitively follows, as no pixels have average value $\mu_i = \beta\lambda$.

Task 6

Task 6a The average row-sum is found to be $\frac{1{,}556{,}153}{65{,}535}$, or approximately 23.7453727.

Task 6b As below

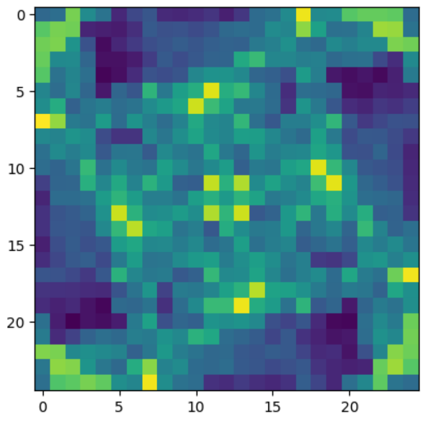


Figure 12: Average 2D pattern assuming identical orientation

Task 6c As below

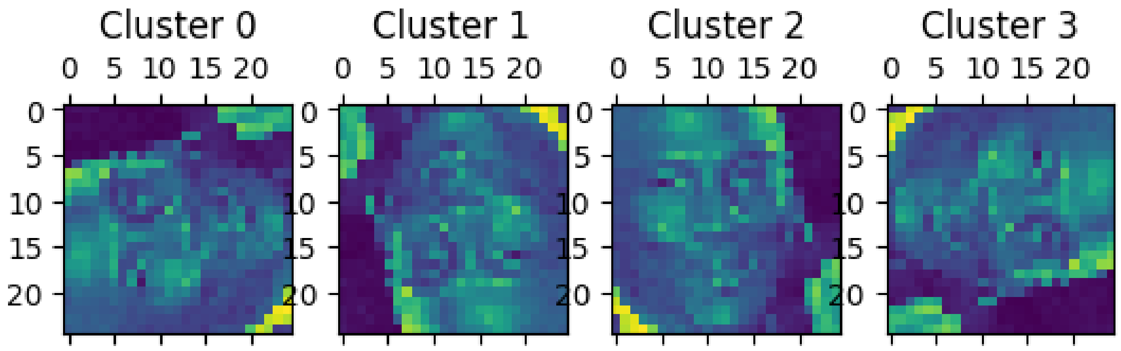


Figure 13: Representation of the four orientation class averages

Task 6d The average row-sum for the design matrix, can be simply calculated by the number of non-empty pixels, (as all empty pixels have value 0) divided by the number of patterns. The length of vector `task6a` (or equivalently `task6b`) represents the number of non-zero pixels, 1,556,153. Each component of vector `task6a` is an index describing the position of a pattern vector in the design matrix. Each component of vector `task6b` is an index describing the position of a 1 in each pattern. This effectively functions as a coordinate system that can be used to fill the design matrix.

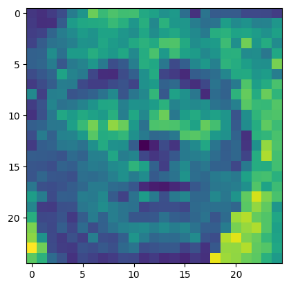
```
task_6_matrix = np.empty((65535, 625))
for i in range(task6a.shape[0]):
    task_6_matrix[task6a[i]][task6b[i]] = 1
```

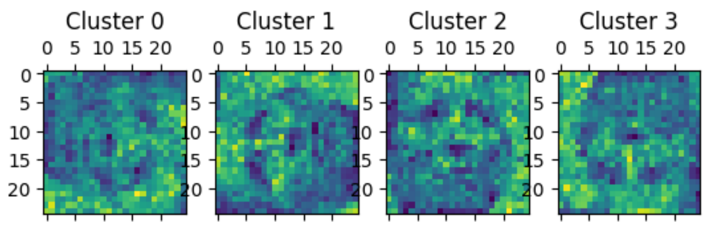
Figure 14: Reconstructing Design Matrix from Sparse Data

After the number of non-zero pixels are known, the column averages of the design matrix can be taken as per task 4b. In order to sort the orientation classes the same K-Means algorithm can be used. Returning the above plots (Figure 13).

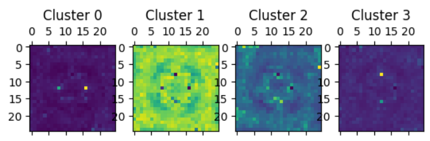
Task 7

Task 7a Multiple techniques were used to reconstruct the master image, the resulting images are shown below;

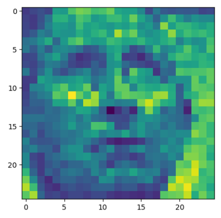

(a) Reconstruction using PCA and K-Means

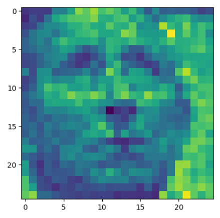


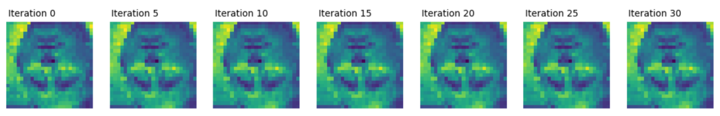
(b) Partial Reconstruction (4 Orientation Classes) using Factor Analysis



(c) Partial Reconstruction using PCA and K-Means to initialise a Gaussian Mixture Model


(d) Reconstruction using PCA and K-Means with Cosine Distance


(e) Reconstruction using PCA and Bisecting K-Means



(f) Iterations of the Heuristic Reconstruction Method

Figure 15: Results of Various Reconstruction Methods

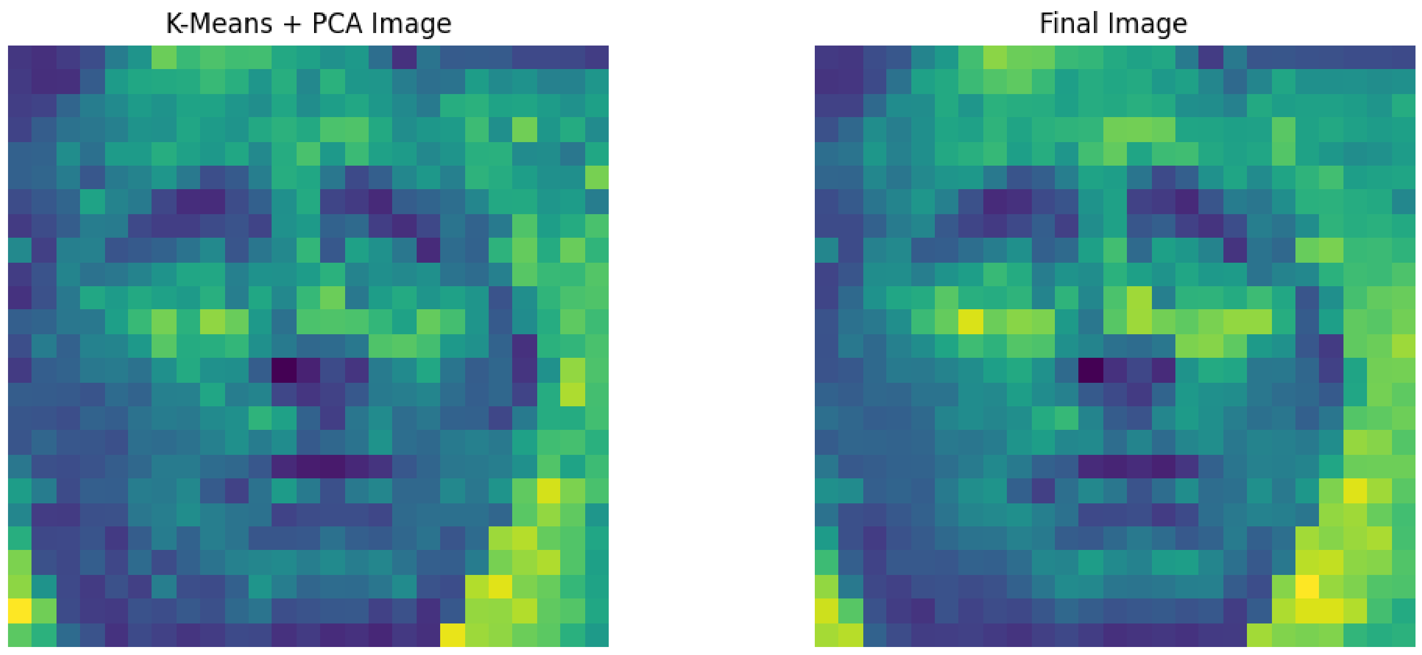


Figure 16: Original K-Means Reconstruction + Heuristic Reconstruction Method

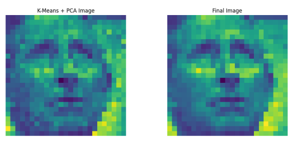


Figure 17: Scaled Down Version of Figure 16

We are still unable to identify this person.

Task 7b Solely employing K-Means resulted in a cluster that contained the majority of all samples, and the rest of the clusters contained very few samples. This potentially indicated a poor separation of data.

Principal Component Analysis (PCA) was then used to reduce the number of dimensions of the dataset, as K-Means performs poorly on data with high dimensions (Beyer et al., 1999). This is achieved by minimising the sum of the distances of the projection of each point onto a vector from the initial point. In practice this can be done by finding the eigenvectors of the covariance matrix with the highest eigenvalues (solutions to the characteristic equation of the covariance matrix) taken as highest priority. In this case the data was flattened into 25 dimensions, whilst preserving the maximum amount of relevant information.

Euclidean distance (the distance metric used by the K-Means algorithm) is a more viable distance measurement in 25 dimensions, mitigating the inaccuracies of high dimensionality. Therefore, using PCA to reduce the dimensionality of the data increases the separation between clusters and the efficacy of K-Means unsupervised classification.

Other Methods of Reconstruction This report is not exhaustive enough to outline each method in detail. The following are some (less effective) attempts, and why they were attempted.

- Factor Analysis - attempted using the orientation as the latent variable but failed to accurately separate patterns.
- Gaussian Mixture Model - assumed normal distribution of the clusters - this was too sensitive to initialisation and too computationally intensive to repeat frequently.

- K-Means with Cosine Distance - Matrices were normalised before computing Euclidean distance for K-Means (sachinruk, 2020), this returned similar result to Euclidean Distance when PCA was applied to both before clsutering. Without PCA being applied to either Cosine similarity returned a marginally better result than K-Means with Euclidean distance.
- Bisecting K-Means - assumed classes should be approximately equal in density and size.

The Heuristic Reconstruction Method After obtaining an 'image candidate' from one of the above processes (in this case, PCA + K-Means), the following procedure was followed:

1. Each pattern was rotated and compared to the 'candidate image'. The rotation with greatest similarity (closest by Euclidean distance) was preserved in a new dataset, as shown in code below;

```
def find_best_rotation(img_reshaped, candidate_image):
    best_similarity = np.inf
    best_image = candidate_image
    for j in range(4):
        rotated_image = np.rot90(candidate_image, j)
        similarity = np.linalg.norm(img_reshaped - rotated_image)
        if similarity < best_similarity:
            best_similarity = similarity
            best_image = rotated_image
    return best_image
```

2. The new dataset images were averaged to create a new 'candidate image'
3. This process was repeated until the 'candidate images' converged.

The image on which they converged was Figure 16, the final image.

The image following Heuristic reconstruction is included alongside the image obtained prior, noise was significantly reduced, but some detail may have been lost.

AI Use Report

GitHub Copilot was used for inline code completions (generating comments, filling in boilerplate, repetitive code from elsewhere in the notebook). Code was inspected before accepting completions.
ChatGPT-3.5 and **ChatGPT-4o** were used to generate LaTeX code in formatting the document.

References

Beyer, K., Goldstein, J., Ramakrishnan, R., & Shaft, U. (1999). When is "nearest neighbor" meaningful? In *Lecture notes in computer science* (pp. 217–235). Springer Berlin Heidelberg. http://dx.doi.org/10.1007/3-540-49257-7_15

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., . . . Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

sachinruk. (2020, April). Using k-means with cosine similarity - python. https://stackoverflow.com/a/61450691