LAOML Project 1

Sathish Kumar RK, Hans Keur, Sofia-Eirini Kotti and Omar

May 2022

1 Exercise 1a — K-Means Clustering

The first step to our clustering algorithms was to collapse the images into 1D feature vectors. In this section, we report the results of K-means clustering of the entire 70000 samples in the MNIST dataset by finding the standard squared Euclidean distance (sqeuclidean). First we initialized 10 clusters with preferably 10 maximally different centroids. We did this by randomly choosing 10 samples from the whole dataset and calculated the sqeuclidean between them. Then we repeated this several times and chose the subset with the maximum sqeuclidean. An example of the initial centroids can be found in Figure 1. Using these initial centroids, we followed algorithm 22 of the course notes to clusterize the whole dataset and the resulting centroids can be found in Figure 2.

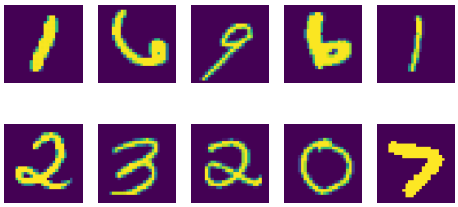

Figure 1: Initial centroids.

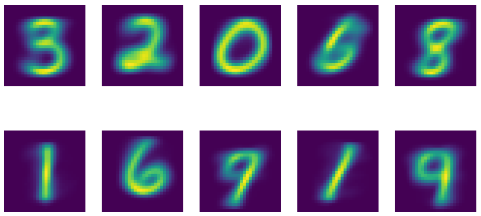

Figure 2: The centroids found by the K-Means clustering algorithm.

We computed the accuracy of the clustering using the procedure given in the assignment. In addition to accuracy, we have also calculated the *Homogeneity* and *Completeness* using `sklearn.metrics`. A clustering result satisfies homogeneity if all of its clusters contain only data points which are members of a single class and it satisfies completeness if all the data points that are members of a given class are elements of the same cluster. Then the score is 1.0, otherwise smaller.

In order to verify the performance of our implementation, we compared our results of accuracy, homogeneity, completeness and elapsed time against an existing Python implementation from `sklearn.cluster.KMeans`. Moreover, we used the correlation between two samples as another distance measure and compared our results computed using sqeuclidean against them. The algorithms ran 1x for the *whole data set* and the results were as follows

Algorithm	Homogeneity	Completeness	Accuracy	Time [s]
Standard K-Means	0.4971	0.5187	59.9%	240.966
K-Means correlation	0.5141	0.5213	60.1%	838.551
sklearn.cluster.KMeans	0.4965	0.5038	58.5%	61.181

From this single run of the algorithms, we have the indication that all algorithms have approximately the same accuracy. However, the K-means using correlation as a distance ran about 3.5x slower than the standard K-means we coded. As an improvement, we should have had more runs of the algorithms to have average results (they were all rather fast to run).

2 Exercise 1b — Kernelized K-Means Clustering (*1b.py*)

2.1 Kernelization as discussed in the class notes

For this question, Algorithm 23 of the notes was used. In particular, the matrix containing the pairwise kernel distances between samples $K(x_i, x_j)$, $i, j = 1, \dots N$ was calculated once and stored. In every iteration of the algorithm, the indices showing which cluster the sample belongs to are updated. That is how the cost function for every sample and every cluster is calculated. Two types of kernels were tested: polynomial and Gaussian.

2.1.1 Polynomial kernel

The polynomial kernels were of the form $K(x_i, x_j) = (1 + x_i^T x_j)^r$, for different parameter r values. When a polynomial kernel is used, it is possible to run the algorithm on the entire dataset of 70000 samples; in the specific machine used for this section, it took approximately 30 minutes. This is due to the matrix operations that are exploited.

2.1.2 Gaussian kernel

The Gaussian kernel is of the form $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|_2^2)$. The parameter γ stands for the inverse of the variance of the Gaussian function used. Unfortunately, calculating the matrix containing the pairwise distances between samples when a Gaussian kernel is used takes too long.

We run tests to find the optimal value of γ based on the achieved accuracy. The results were averaged over 5 runs on 10000 samples each. There was no significant difference found in performance for $\gamma \leq 1/30$. This means that from this point on, the variance of the corresponding Gaussian is large enough to 'cover' all nearby samples and the algorithm performance is only slightly affected.

Table 1: Accuracy results for different γ values.

γ value	1/5	1/10	1/30	1/50	1/100	1/500	10^{-3}	10^{-4}
Accuracy	0.436	0.409	0.538	0.552	0.565	0.547	0.557	0.561

2.1.3 Results

The results below are averaged over 5 runs on datasets of 10000 samples. Another approach that does not save the entire matrix could be preferred in order to use the entire dataset for Gaussian kernels as well.

Kernel type	Parameter	Accuracy	Time [s]
Polynomial	2	50.64%	50
Polynomial	3	38.93%	67
Polynomial	4	31.78%	63
Gaussian	1/100	60.47%	258

We see that a polynomial kernel of order $2, 3, 4$ leads to a reduction in accuracy compared to the standard K-means. On the other hand, the Gaussian kernel seems to perform equally well with the standard K-means (the difference is too small to claim that a Gaussian kernel works better, more tests should be performed to evaluate that). Unfortunately, the Gaussian kernel takes 4x longer for this dataset size.

2.2 Kernels inspired from image processing

The above approach works for any type of kernel, also those defined by infinite dimensional mappings ϕ. Another approach we took was to a we implement a function ϕ which consists of several image processing step and then use the standard K-means on the processed images. These steps are: a) blurring, b) equalization and c) rotation correction.

Blurring: In order to account for small inaccuracies/noise (or speckles, etc.) in the written digits, we use a small Gaussian blurring function. The Gaussian function in 2D is defined as $G(x,y) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2-y^2}{2\sigma^2}\right)$, where x and y define the distance from the center pixel.

A so-called (discrete) Gaussian convolution matrix K_G of size 3×3 is equal to $(K_G)_{x,y} = cG(x,y)$, where c is a constant such that the elements of K_G sum to 1. For 2D, K_G is given below.

$$K_G^{2D} = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{1}$$

The library OpenCV has a function called `GaussianBlur` to apply this kernel as filtering (convolution) operation. Given that `sigmaX = 0` is parsed to this function, `GaussianBlur` computes the variance σ^2 derived from the size of K_G.

Equalization: Some digits are written very light and some heavy (bold). In order to make these differences smaller, (histogram) equalization is applied. The mapping h we considered for contrast adjustment is as follows:

$$h(v) = \text{round}\left(\frac{\text{cdf}(v) - \text{cdf}_{\min}}{(M\times N) - \text{cdf}_{\min}} \times (L-1)\right)$$

where $\text{cdf}_{\min}$ is the minimum non-zero value of the cumulative distribution function (in this case 1), $M\times N$ gives the image's number of pixels and L is the number of grey levels used. $\text{cdf}(v)$ denotes the cumulative distribution function of the pixel value v.

Rotation Correction: Our image rotation correction function comprises a rotation detection step, giving the rotation angle, which is then used to rotate the image.

Rotation Detection: In order to make an approximation of the rotation of the written digit, firstly, we compute the average index of the non-zero pixels per row. Then we fit a line through these points, using a linear least squares (LLS).

Let the linear approximation be $f_{a,b}(x) = a + bx$. Let the residual for vector i be $r_i(a,b) = y_i - f_{a,b}(x_i), i = 1,...,|X|$. LLS minimizes the sum $S(a,b) = \sum_{i=1}^{|X|} r_i^2(a,b)$. The optimal values for a and b can be found by solving $\frac{\partial S}{\partial a} = \frac{\partial S}{\partial b} = 0$. Given the slope b, the rotation angle is atan(b) [rad]. Hence the correction rotation angle is -atan(b) [rad].

Rotation Correction: In order to rotate the image over the given angle, we compute a rotation matrix, which, for a 2×2 image is $R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$. Given the input image vector x, the rotated image is $x' = R_\theta x$. This can be generalized for larger input and output vectors. The resulting vector x' is the original image x, rotated over angle θ.

Why is rotating the image useful? Suppose we have two written ones, where the one is exactly vertically aligned and the other has a slope of 30 degrees. Then the Euclidean distance between the corresponding image vectors might be large and as a result, the ones may belong to different clusters after applying the K-means clustering algorithm. Using the rotation correction function, we try to prevent this.

An example of the output of the image processing steps is shown in Figure 3.

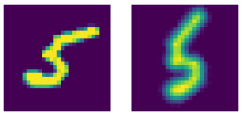

Figure 3: Example input and output of the image processing steps.

2.2.1 Best Blurring Radius and results

In order to test which value to take as blurring radius R, we do the image processing for $R \in [1, 3, 5, 7]$ and then run the K-means clustering algorithm 3 times. The resulting average accuracy is logged and shown in Figure 4. The radius is picked and used in the next image processing steps. Blurring may help to diminish noise or small irregularities in the hand writing. If r becomes too large, then the digit may become too vague to distinguish, which leads to less accurate results. Therefore, the optimum lies in between, as can be seen in the Figure 4. As expected, the processing time increases for an increasing size of the Gaussian 2D filter. This is directly related to the larger matrix size of K_G, which gives more numbers to multiply.

The accuracy achieved using the best $R = 3$ and the K-means clustering algorithm subsequently has increased to 73%, whereas it was approximately 60% for the standard K-means.

For an example of the centers of the clusters after applying K-Means on the processed images, see Figure 5. Here, we clearly see that there is exactly one cluster for ones as expected, due to the rotation correction. This is not the case for the digit 0, which appears in a thinner and a wider version.

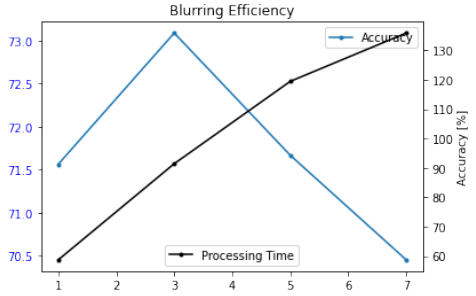

Figure 4: The effect of the blurring radius R on the accuracy of the K-Means algorithm.

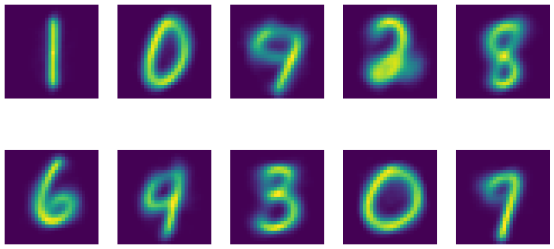

Figure 5: An example of the centroids after applying K-Means on the processed images.

3 Exercise 2a — Graph Laplacian (*2a.py*)

The unnormalized graph Laplacian matrix is defined as $L = D - W$, where D is a diagonal matrix with $D_{i,i} = \sum_{j=1}^{N} w_{ij}$. The matrix W can be the adjacency matrix of the graph or the similarity matrix, which is a generalization of the adjacency matrix such that $W_{ij} = K(x_i, x_j)$, where $K(\cdot, \cdot)$ is a kernel function of choice.

Algorithm 24 was implemented using different kernel functions to construct W. The last step of the algorithm is the application of the standard K-means algorithm on H, the matrix whose columns are the eigenvectors corresponding to the K smaller eigenvalues of L. Algorithm 25 uses the normalized Laplacian matrix of the graph. The normalized Laplacian matrix is defined as $L = D^{-1/2} L D^{-1/2}$. This matrix can be used to improve the performance of the spectral clustering algorithm since the algorithm output can be affected by a few nodes with the largest degree in the graph. Below, the results for all version of the spectral clustering algorithms are discussed.

3.1 Graph Laplacian using the similarity matrix

The results for various kernel functions are given below. It is clear that normalizing the Laplacian matrix brings great improvement in the accuracy of the algorithm. None of the kernel functions perform well in the unnormalized case. Moreover, we see that the polynomial kernels perform better in combination with spectral clustering compared to when the kernelized K-means algorithm was used. It seems that all polynomial kernels up to order 4 perform on par with the Gaussian kernel-based spectral clustering and similarly to the standard K-means algorithm.

Spectral clustering version	Kernel type	Parameter	Accuracy	Time [s]
Unnormalized	Polynomial	1	22.59%	123
Unnormalized	Polynomial	2	20.97%	133
Unnormalized	Polynomial	3	18.87%	136
Unnormalized	Polynomial	4	16.51%	107
Unnormalized	Polynomial	5	15.65%	109
Unnormalized	Gaussian	1/50	15.78%	350
Normalized	Polynomial	1	57.00%	123
Normalized	Polynomial	2	58.70%	119
Normalized	Polynomial	3	59.00%	130
Normalized	Polynomial	4	57.15%	148
Normalized	Polynomial	5	51.80%	149
Normalized	Gaussian	1/50	58.56%	343

3.2 Graph Laplacian using the adjacency matrix

Formulating the adjacency matrix in this dataset that is not given in the form of a graph requires thresholding the similarity matrix. After running the algorithm multiple times, the Gaussian kernel was found to perform best with thresholds in the interval $[0.07, 0.2]$, therefore a value 0.1 was chosen. Following the same logic, for the polynomial kernel with parameter 3, the value 9000 was chosen for the threshold. For values above 10000 it was found that some nodes became disconnected from the graph and the performance dropped dramatically. The exact results can be found below.

Viewing the digit images as nodes of a graph characterized by an adjacency matrix does not work well in this case, due to the variability of the digits in the dataset. Additionally, an additional step to tune the threshold is required.

Spectral clustering version	Kernel type	Parameter	Accuracy	Time [s]
Normalized	Polynomial	3	37.03%	154
Normalized	Gaussian	1/50	46.76%	436

4 Exercise 2b — Own Solver to Compute the Eigenvectors (*2b.py*)

In order to find the K smallest eigenvalues of L, it is useful to exploit the matrix structure. Note that L is Hermitian, therefore, we can use the Lanczos algorithm or QR-algorithm. Lanczos method is prone to numerical instability and therefore, we chose the QR algorithm.

The QR algorithm to calculate the eigenvalues is based on a series of QR factorizations on matrices that have the same eigenvalues (Algorithm 7). Since the similarity matrix in this problem is symmetric, the eigenvectors can be recovered with the same algorithm. The QR factorizations are computed using the Gram-Schmidt orthonormalization algorithm (Algorithm 4).

The algorithm was written using two stopping criteria: a tolerance on $\|L_{k+1}\|$ (the norm of the lower triangular part of the matrix A_{k+1}), and a maximum of 50 iterations. Given the large values in the matrix, the algorithm always exited at 50 iterations which were enough to recover almost the exact same eigenvectors and eigenvalues as the built-in function `numpy.linalg.eigh`. The disadvantage was that the QR factorization step is very costly and all eigenvalues are updated at the same time. The algorithm run in about 350 seconds for 1000 samples, which is rather slow compared to the built-in function. The bottleneck is the QR factorization step. Alternatively, the inverse power method could be used to recover only the necessary smallest eigenvalues and the corresponding eigenvectors.

Since the recovered eigenvalues and eigenvectors are approximately the same as the ones recovered by the built-in solver, the results on accuracy of spectral clustering are not included here again.

5 Exercise 3 — Compressed Images

As a of 4 people, we attempted exercise 3 as well. The first step for this exercise was to inspect the singular values of the images. For this, the singular value decomposition (SVD) was applied to each 2D image separately, considering it as an array. Since the images contain a large number of 'black' pixels, we expect image compression to work well. Given a threshold $k = 0.1$ We found that there was no sample in the dataset that had more than 20 significant singular values. Therefore, in the investigation below we limited L, the number of singular values considered, between 1 and 20.

The results for different clustering algorithms are given below. For each L values, clustering was performed 5x per algorithm and the following accuracy and time were averaged.

5.1 K-means

We compressed the images using SVD and tuned the degree of compression L to optimize the accuracy and elapsed time. Figure 6 shows the accuracy and time of standard K-means as a function of L. The best accuracy of approximately 63% was obtained for $L = 3$.

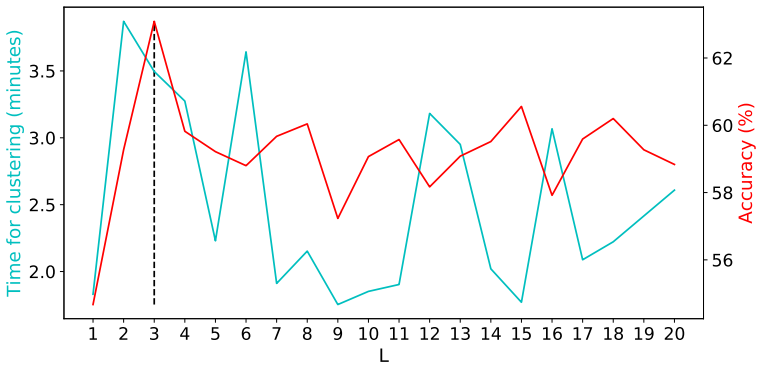

Figure 6: Accuracy and elapsed time of K-means for different L values.

In Figure 7, we show the initial centroids of the dataset compressed using SVD with $L = 3$ and the centroids of the clusters after performing K-means clustering is shown in Figure 8. The effect of image compression is clear on the initial centroids. When it comes to the output centroids, they are pretty different from each other, which shows that these few singular values actually capture the essence of each digit.

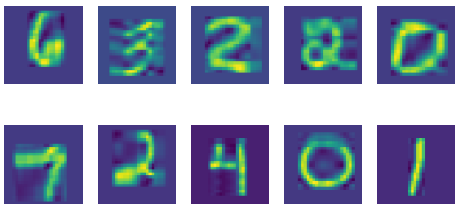

Figure 7: Initial centroids of SVD compressed images with L = 3

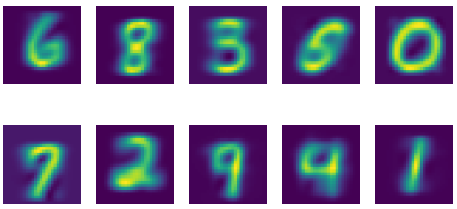

Figure 8: Cluster centroids found using K-means on the compressed images with $L = 3$.

5.2 Kernelized K-means

The same procedure was applied for the kernelized K-means algorithm using a polynomial kernel with parameter $r = 2$. Figure 9 shows that the optimal accuracy of approximately 53 % was obtained for $L = 3$. This is slightly better than the accuracy reported for the same algorithm without compression in Section .

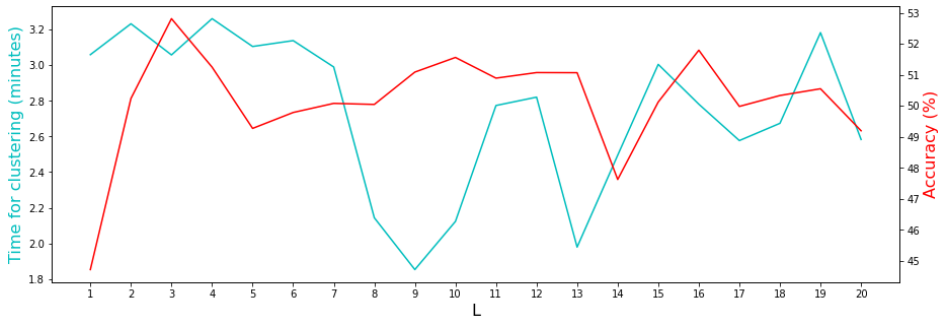

Figure 9: Accuracy and elapsed time of kernelized K-means (polynomial kernel with $r = 2$) for different L values.

5.3 Spectral clustering

The same procedure was applied for the normalized spectral clustering algorithm that uses the similarity matrix. The kernel function was polynomial with parameter $r = 3$. Figure 10 shows that the optimal accuracy of approximately 61 % was obtained for $L = 2$. This is slightly better than the accuracy reported for the same algorithm without compression in Section 3.1.

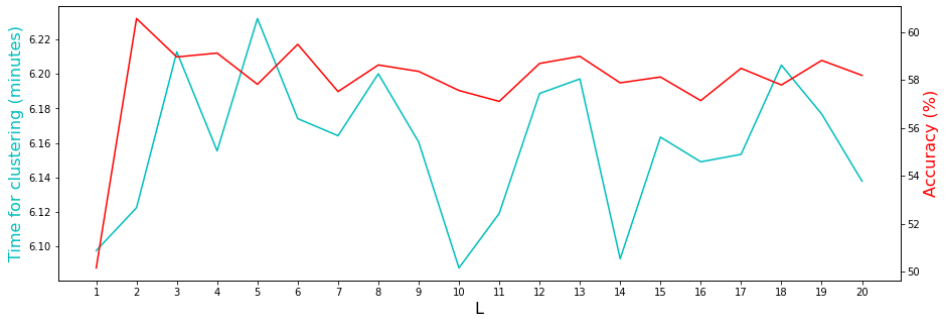

Figure 10: Accuracy and elapsed time of spectral clustering for different L values.

6 Conclusions and future Work

In all the above analysis, we have collapsed all images into features vectors, assuming that each pixel is one feature. Given the differences between handwritten instances corresponding to the same digits, as explained in 2.2, this does not seem to be the best approach. All clustering algorithms seem to give an accuracy of approximately 60%, when the correct parameters are chosen. Standard K-means was found to improve by applying rotation correction beforehand to at approximately 70%. That is why it would be interesting to develop algorithms that use different types of features and retain the images in 2D form.

Higher accuracies might be obtained by using DBSCAN (Density-Based Spatial Clustering of Applications with Noise) or hierarchical clustering like Ward agglomerative clustering.

Regarding the running time of the algorithms, this can be improved especially for the algorithms based on Gaussian kernels.

Finally, the results for compressed images show that this is a dataset that admits large compression that retains only a few singular values. Interestingly, the clustering algorithms seem to perform slightly better on highly compressed images compared to the original images; this indicates that these few singular values contain the most distinguishing characteristics of the handwritten digits.