ISYE 6740 – HW #1

Jing Ma

2025-01-16

Problem 1:

1.1

Generally, supervised learning uses labeled classes whereas unsupervised learning uses algorithms to find patterns or categories instead of using predefined labels.

For supervised learning, the advantages are as follows:
1. Accuracy can be more easily measured based on the number of correctly categorized instances by the learning algorithm;
2. Supervised machine learning algorithms tend to be more straightforward and are simple to implement;
3. Most of real-world modeling can be performed using supervised learning algorithms.

The limitations of supervised learning are:
1. Supervised learning cannot handle certain complicated tasks like unsupervised learning can;
2. Not suitable for unstructured data ([1]).

For unsupervised learning, the advantages are:
1. Reveal patterns when the exact labels or categories are unknown;
2. Identify outliers or deviations from the normal data range ([1]);
3. Does not require manual data labeling ([1]).

The limitations of unsupervised learning are:
1. Results from unsupervised learning algorithms, such as random forest, can be difficult to explain to stakeholders;
2. The performance of an unsupervised learning model can be tricky to measure.

1.2

Since x contains two categorical variables, location (e.g., "Dallas") and property type (e.g., "house"), we can first convert these categories using one-hot encoding. For instance, we may categorize "Atlanta" as 1 and "Dallas" as 0. Similarly, we may categorize "house" as 1. Then we use Hamming distance to count the number of different binary values between the two sample points. Further, since the observations also contain real values, we can use combine Hamming distance with Euclidean distance to measure similarity between two data points as follows. m represents the feature index, namely, the location and building type categorical variables, whereas x_3^i refers to the third feature (i.e., housing price).

$$d(x^i, x^j) = \sum_{m=1}^{2} d_H(x_m^i, x_m^j) + \|x_3^i - x_3^j\|^2$$

1.3

$$\pi(i) = \arg\min_{j=1,\ldots,k} \|x^i - c^j\|^2 = \|x^i\|^2 - 2(c^j)^T x^i + \|c^j\|^2$$

since $\|x^i\|^2$ does not change based on the centroid, we can treat it as constant and eliminate from the formula. Further, after we rearrange and factor out 2 from the $-2x^i c^j$ term, we get to the following:

$$\pi(i) = \arg\min_{j=1,\dots,k} \frac{1}{2}\|c^j\|^2 - \left(c^j\right)^T x^i = \arg\min_{j=1,\dots,k} \left(c^j\right)^T \left(\frac{1}{2}c^j - x^i\right)$$

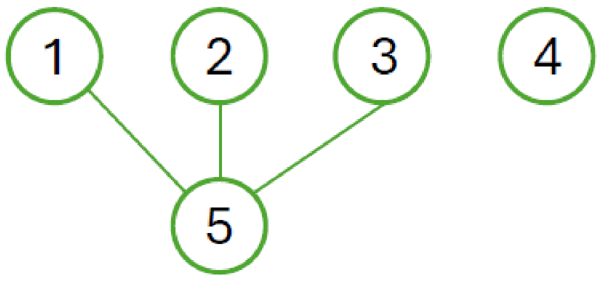
1.4

K-means is a NP-hard problem due to multiple local minimums in addition to the global minimum. This means that different initiations of centroids and lead to different local minimums, and hence different results. Due to the non-convexity nature of the problem, we use heuristic method to find local solutions, which is the k-means algorithm. This is essentially an optimization problem that tries to minimize the sum of the squared distance between the data points and the assigned centroids.

1.5

K-means will stop after finite number of iterations because the finite number of data points determined that there are only limited ways for iterations. Further, the kmeans objective function only monotonically decrease until it does not change significantly anymore. This is because centroids only get updated if they improve/reduce the sum of the squared distance between data points and their corresponding centroids. At certain point, when the objective function cannot make any further improvements, the algorithm would stop.

1.6



Based on the above graph, we can construct the adjacency matrix, A, below:

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

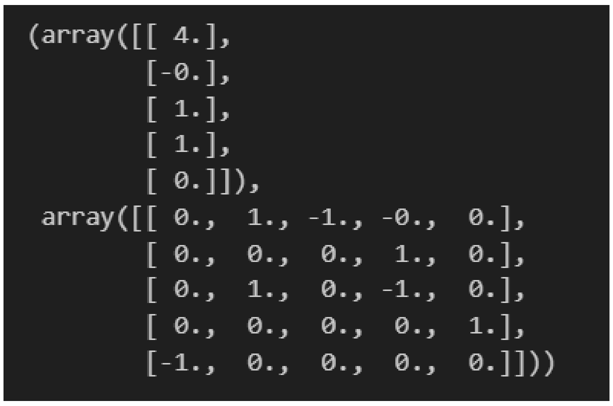
The degree matrix, D, is as follows:

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{pmatrix}$$

Based on the formula that we learned from lecture, we can construct the Laplacian matrix, $L = D - A$:

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & 0 & 3 \end{pmatrix}$$

After we input Laplacian matrix in our code file and use np.linalg.eig() function to solve for eigenvalues and eigenvectors, we obtain the outputs below:



```
(array([[ 4.],
        [-0.],
        [ 1.],
        [ 1.],
        [ 0.]]),
 array([[ 0.,  1., -1., -0.,  0.],
        [ 0.,  0.,  0.,  1.,  0.],
        [ 0.,  1.,  0., -1.,  0.],
        [ 0.,  0.,  0.,  0.,  1.],
        [-1.,  0.,  0.,  0.,  0.]]))
```

As we can see, there are two eigenvectors associated with eigenvalue of 0. This makes sense since in our graph, we have two communities where nodes 1, 2, 3, and 5 form one community and node 4 itself forms another community.

These eigenvectors are:

$$v_1 = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$v_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

If we combine these two eigenvectors, we create a matrix below:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Each row is a feature vector, and we can see that there are two communities since row 1 and row 3 contain 1 in the first eigenvector and row 4 contains 1 in the second eigenvector, showing that there are two disconnected clusters.

Problem 2:

2.1

Given the distortion function:

$$J = \frac{1}{mk}\sum_{i=1}^{m}\sum_{j=1}^{k} r^{ij}\ \|x^i - \mu^j\|^2 \tag{1}$$

where,

$$r^{ij} = \begin{cases} 1 \text{ if } x^i \in \mu^j \text{ cluster} \\ 0 \text{ otherwise} \end{cases}$$

We can first expand the distortion function as follows:

$$\begin{aligned} J &= \frac{1}{mk}\sum_{i=1}^{m}\sum_{j=1}^{k} r^{ij}\left[\left(x^i\right)^2 - 2x^i\mu^j + \left(\mu^j\right)^2\right] \\ &= \frac{1}{mk}\sum_{i=1}^{m}\sum_{j=1}^{k}\left[r^{ij}\left(x^i\right)^2 - 2r^{ij}x^i\mu^j + r^{ij}\left(\mu^j\right)^2\right] \end{aligned}$$

If we take the partial derivative of J with respect to μ^j, with r^{ij} fixed:

$$\frac{\partial J}{\partial \mu^j} = \frac{2r^{ij}}{mk}\sum_{i=1}^{m}\sum_{j=1}^{k}\left[r^{ij}\mu^j - r^{ij}x^i\right]$$

Since we want to find the minimum of the distortion function J, it means that we want to set the partial derivative to 0. Further, we treat m and k as constants and can eliminate them from the function. Hence, the above function can be rewritten as below:

$$\sum_{i=1}^{m}\sum_{j=1}^{k}\left[r^{ij}\mu^j - r^{ij}x^i\right] = 0$$

If we factor out μ^j and put it on the left side of the equation, we get the following:

$$\mu^j = \frac{\sum_i r^{ij}x^i}{\sum_i r^{ij}}$$

2.2

Follow the same logic as part 1, if we want to find the assignment variables r^{ij} that minimize the distortion function with the centroid μ^j fixed, we obtain the following function by taking the partial derivative of function J with respect to r^{ij} based on Equation 1:

$$\frac{\partial J}{\partial r^{ij}} = \frac{1}{mk}\sum_{i=1}^{m}\sum_{j=1}^{k}\|x^i - \mu^j\|^2$$

Since we treat m and k as constants, we can eliminate them from the equation without changing the outcome. Also, for each x^i, only one $r^{ij} = 1$ if it minimizes the squared distance between x^i and μ^j. So we can simplify the equation further as the following:

$$r^{ij} = \begin{cases} 1 \text{ if } J = \arg\min\|x^i - \mu^j\|^2 \\ 0 \text{ otherwise} \end{cases}$$

2.3

Next, we will derive μ^j and r^{ij} from the distortion function J based on Mahalanobis distance. We are given:

$$J = \frac{1}{mk} \sum_{i=1}^{m} \sum_{j=1}^{k} r^{ij} (x^i - \mu^j)^T \Sigma (x^i - \mu^j)$$

According to Christopher M. Bishop ([2]) and the problem, Σ is symmetric, positive definite matrix. By utilizing this property, per Matrix Cookbook ([3]) equation #86, we can obtain the following derivative with respect to μ^j:

$$\frac{\partial J}{\partial \mu^j} = -2 \sum_{i=1}^{m} r^{ij} \Sigma (x^i - \mu^j)$$

Since we are finding the minimum, again similar to part 1, we set this partial derivative to 0 and obtain:

$$-2 \sum_{i=1}^{m} r^{ij} \Sigma (x^i - \mu^j) = 0$$

We treat Σ as constant and eliminate both -2 and Σ from the equation:

$$\sum_{i=1}^{m} r^{ij} (x^i - \mu^j) = 0$$

After re-arranging the formula above, we get the following:

$$\mu^j = \frac{\sum_{i=1}^{m} r^{ij} x^i}{\sum_{i=1}^{m} r^{ij}}$$

To find r^{ij} that minimizes function J, we note that it's the same process as part 2 as when we take the partial derivative with respect to r^{ij}, the $(x^i - \mu^j)^T \Sigma (x^i - \mu^j)$ term remains.

Therefore, alike part 2, r^{ij} has values below:

$$r^{ij} = \begin{cases} 1 \text{ if } J = \arg\min \, (x^i - \mu^j)^T \Sigma (x^i - \mu^j) \\ 0 \text{ otherwise} \end{cases}$$

Problem 3:

3.1

To implement k-means clustering algorithm, I referenced the demo code, kmeans_digit, for the vectorized computation of $L2$ norm.

The following are the implementation steps:

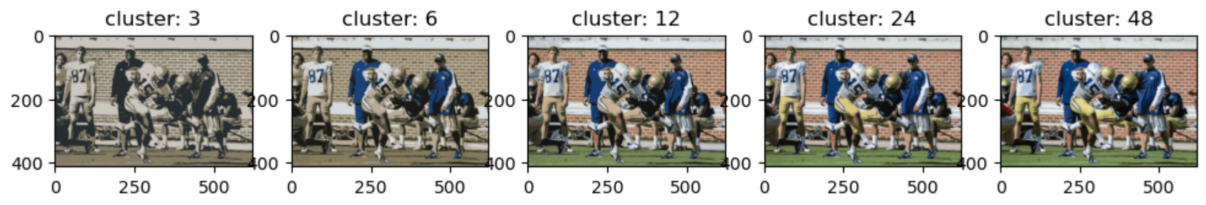
- Reshape imported images from 3D to 2D where rows represent pixels and columns are the three color channels (red, green, blue);
- Instantiate random centroids based on the k;
- Set epsilon = 10^{-2} as the stopping criterion. This means that each time when centroids are updated, we want to compare the total change to the threshold. If the change is small enough, we can satisfy ourselves that convergence is achieved;
- As noted in the instruction, the issue with too large of k values is that sometimes certain clusters could be empty. This could mean that no pixels are close enough to these clusters and therefore

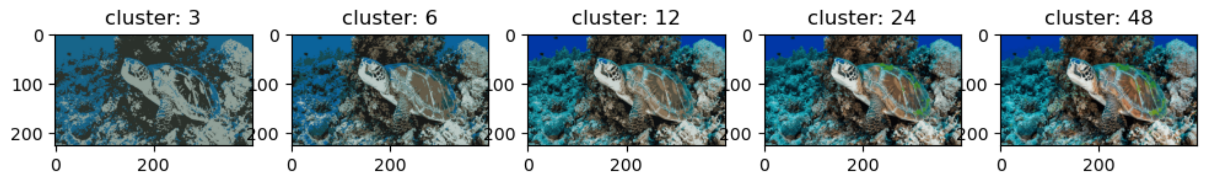
are not assigned to them. To handle this scenario, each time when we update clusters, we check whether there are any empty clusters. If so, we want to decrement k, and continue running the algorithm until convergence;

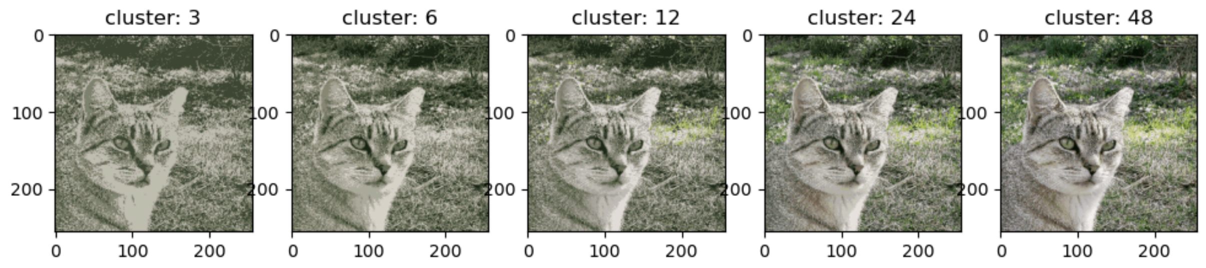
- Once the algorithm has converged, we retrieve the compressed data based on which cluster that each pixel is assigned to.

To find the best seeds, I created seeds ranging from 0 to 40 (code is commented out in the programming file as running it takes ~30 minutes to find the best seed within $[0, 40]$ interval) and totaled cost for $k = [3, 6, 12, 24, 48]$. The seed that yields the lowest total cost is considered to be the best seed. As a result, the seed that generated the least total cost for all 5 k clusters is 14. Therefore, I used np.random.seed(14) for running the kmeans clustering algorithm for all three images.

For the compressed football, sea turtle, and cat images are shown below:







Further, the following is the summary of the time in seconds that it takes to converge (change $< \varepsilon$ (10^{-2})):

	k	Convergence in sec *	# of Iterations
Football	3	0.44	21
	6	0.84	29
	12	6.32	144
	24	9.18	133
	48	29.38	241
Turtle	3	0.12	15
	6	1.08	99
	12	1.48	91
	24	7.35	279
	47	17.97	389

	k	Convergence in sec *	# of Iterations
Cat	3	0.06	12
	6	0.13	20
	9	0.12	13
	21	0.16	9
	44	0.26	8

* Note that runtime may vary by +/- 50 milliseconds

Note that although we used $k = [3, 6, 12, 24, 48]$ to compress all three images, the actual k values differ as there could be empty clusters for the reasons mentioned above. When facing this situation, we automatically decrement the number of clusters until all clusters have been assigned pixels. Therefore, in the output summary above, not all k values equal to what we set out testing with. Take cat image for instance, the largest k value that was returned is 44 instead of 48.

3.2

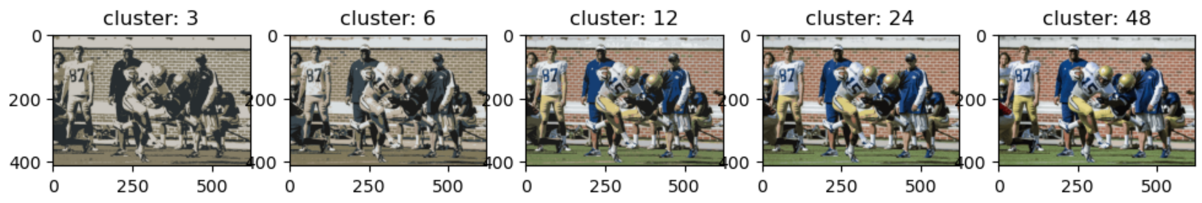
In this next question, we use $L1$ norm (Manhattan distance) instead of $L2$.

The implementation steps remain largely the same except that the equation used for computing Manhattan distance is below and coded as such:

$$d(x^i, \mu^j) = \frac{1}{m}\sum_{i=1}^{m} |x^i - \mu^j|$$

where m represent the number of data points.

Since in this question, we only test on the football image, below are the results:



Similar to part 1, convergence in seconds and number of iterations are summarized below:

	k	Convergence in sec *	# of Iterations
Football	3	0.53	13
	6	4.70	61
	12	7.39	48
	24	17.15	61
	48	46.78	82

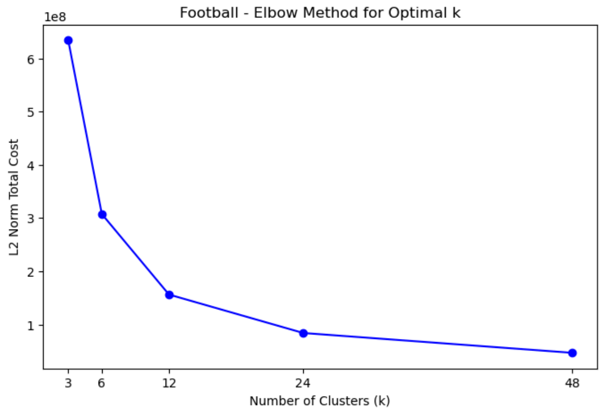
* Note that runtime may vary by +/- 50 milliseconds

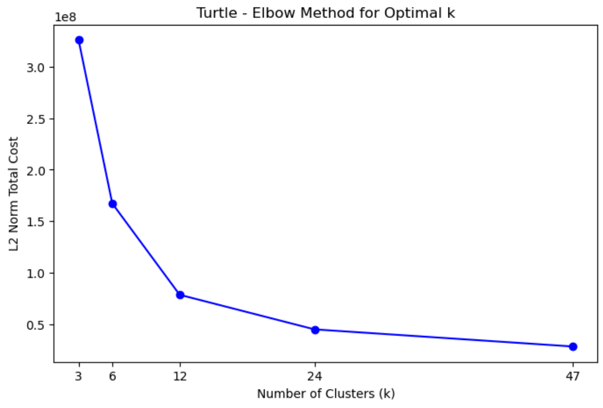
When comparing the results produced using $L1$ norm versus $L2$ norm, we can see that $L1$ norm overall takes less iterations but slightly more time in seconds to converge. For instance, most noticeably, when $k = 48$, $L2$ converges after 241 iterations whereas $L1$ takes only 82 iterations. These differences might be a result of the simplicity in the $L1$ norm computation, as we simply take the absolute value between the data points and each centroid. As for why $L2$ norm has more complex computation but converges faster, my suspicion is that $L2$ norm uses vectorized operations, which is more optimized than $L1$ norm.

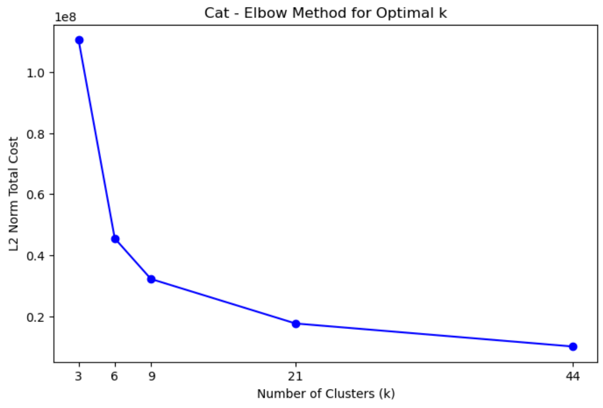
3.3

To find the best k, I plotted "elbow" graphs to help identify that for each image, at which k value there exists the steepest decrease.

Below are the graphs for football, turtle, and cat images, respectively:







As we can observe from above, the steepest decrease in the total distance for the football image happens when $k = 24$. This is the same for turtle image. For cat image, the steepest decrease happens when $k = 21$. We can also observe via visual inspection of the compressed images above where when $k = 24$, the image is already very close to when $k = 48$, meaning that the quality loss between these two cluster values appear to be minimum.

Problem 4:

4.1

In the political blog question, we implement spectral clustering algorithm. My implementation method is based on the provided demo code, test_football, and the steps are outlined below:

- Import nodes and edges files. I imported nodes IDs and true labels separately, though coming from the same nodes txt file;
- Create adjacency matrix, A. We first create an array of $1'$s as values and insert these values into a sparse matrix at positions (i, j), which are nodes IDs reduced by 1 to match Python's 0-based indexing. We then add A to its transpose to make an undirected network. Lastly, we convert A to a dense matrix;
- Pre-processing. Since not all nodes have edges, we need to remove them from A by first summing all values along the rows. Any values containing $0'$s indicate no edges. We then use boolean mask to index rows and columns of A to obtain a more compact adjacency matrix where all nodes have connections;
- Create degree matrix. Since we are using the following equation to compute matrix B (denoted as L in the code file), the way that we compute the degree matrix, D, is by first summing all the edges (indicated by 1) and taking the inverse square root of the values:

$$B = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$$

- Create matrix, B (denoted as L in the code file), using the equation above;
- Find k largest eigenvalues and associated eigenvectors. Generally, we want to find eigenvectors associated with the smallest k eigenvalues because of the multiplicity of the eigenvalue 0, which corresponds to the number of "communities" in a network ([4]). But using the above equation, we are interested in the largest eigenvalues. This is because when we maximize matrix, B, we are effectively minimizing the Laplacian matrix, L ([4]);
- Run KMeans function to fit the eigenvectors and retrieve the cluster labels.

After obtaining the cluster labels, we need to find the majority of the labels within a cluster and assign it to the entire group within the cluster. We also need to compute the mismatch rate for each group within the cluster and the weighted average mismatch rate for the entire cluster as a whole.

On the next page, it shows mismatch rate by group within the same cluster. Note that each column represent a different k value from the specified k list in the instruction. Namely, the following columns from left to right represent $k = 2, 5, 10, 30, 50$, respectively. Since each k value contains k groups within that cluster, this means that there are 50 rows (due to 50 groups in cluster $k = 50$), with each row representing a group within the cluster and its associated mismatch rate. If the group number is out of scope for a given cluster, its associated value is empty (blank value). For readability, values in the tables below have been rounded up to 2 decimal points.

	k = 2	k = 5	k = 10	k = 30	k = 50
group 0	0.48	0.02	0.02	0.01	0
group 1	0	0.03	0.02	0.02	0
group 2		0.45	0.19	0.02	0.02
group 3		0.2	0.46	0.07	0.13
group 4		0.4	0.02	0.03	0.35
group 5			0.47	0.13	0.25
group 6			0.03	0.06	0.05

	k = 2	k = 5	k = 10	k = 30	k = 50
group 7			0	0.09	0.03
group 8			0.03	0.06	0.22
group 9			0.02	0.02	0.09
group 10				0.09	0.02
group 11				0.38	0.07
group 12				0	0
group 13				0.11	0
group 14				0	0.07
group 15				0.2	0.14
group 16				0.09	0.03
group 17				0.05	0.06
group 18				0.17	0
group 19				0.03	0.2
group 20				0	0
group 21				0.22	0
group 22				0.04	0.41
group 23				0	0.02
group 24				0	0.11
group 25				0.15	0
group 26				0.02	0.03
group 27				0.02	0.07
group 28				0.2	0
group 29				0	0
group 30					0.26
group 31					0.03
group 32					0
group 33					0
group 34					0
group 35					0
group 36					0.25
group 37					0
group 38					0.21
group 39					0
group 40					0.03
group 41					0.36
group 42					0.06
group 43					0.07
group 44					0
group 45					0.08

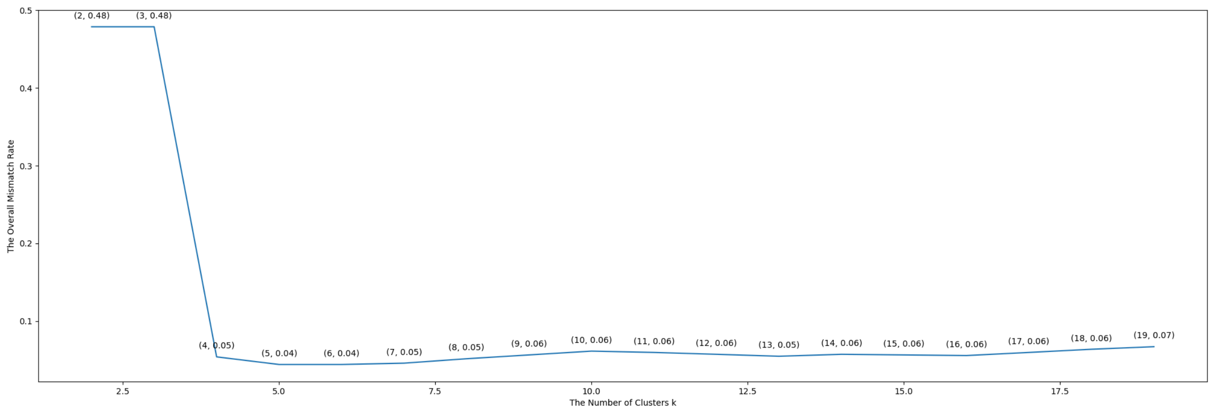
	k = 2	k = 5	k = 10	k = 30	k = 50
group 46					0.12
group 47					0
group 48					0.05
group 49					0

For reference, below is the weighted average mismatch rate for each cluster k. Since we tested 5 k values, there are 5 overall mismatch rates accordingly:

	k = 2	k = 5	k = 10	k = 30	k = 50
Overall Mismatch Rate	0.48	0.05	0.06	0.06	0.07

4.2

I tested a range of k values in $[2, 20]$ interval. Below is the line graph of the corresponding mismatch rate:



Based on the result, I noted that when there are 5 clusters, the minimum overall mismatch rate is 0.0441.

This seems to suggest that although there are only two classes per the true labels, there are minor communities that are weakly connected together, which is why the algorithm performed the best when there are 5 clusters.

Problem 5:

5.1

In this MNIST dataset problem, I used KMeans() function in sklearn package, which by default uses Euclidean distance as a metric for clustering.

Below are my implementation steps for computing purity score for each cluster:

- First load in training and test sets, and perform normalization by dividing pixel values by 255.0;
- Run KMeans function and fit the training set;
- For each cluster, we find the indexed values in the true labels (ytrain) and count the occurrences of labels of all the data points grouped in the same cluster. We pick the label that is the most frequent in the cluster and count how many data points in that cluster have the same label;
- Compute purity score by dividing the count found in the last step by the size of that cluster.

To summarize, the following shows the purity score by cluster (rounded to 2 decimal points):

	0	1	2	3	4	5	6	7	8	9
Purity Score	0.53	0.62	0.86	0.43	0.36	0.90	0.90	0.53	0.79	0.53

We can observe from the table that the algorithm did a much better job at grouping values like 2, 5, 6, and 8. In contrast, the purity scores seem quite low for the other digits. This could be because certain strokes in writing digits like 0 and 1 appear very similar. Likewise, sometimes the writing of 7 or 9 could be easily mistaken due to the high similarity in pixel data.

5.2

In this part of the question, we used Hamming distance instead of Euclidean distance based on the dissimilarity of all pairs of features between each data point and the centroids. All pixel values are converted into binary depending on whether the value is over 128.

Below are the implementation steps:

- Initialize 10 centroids randomly;
- Compute Hamming distance matrix using pairwise_distances() function;
- Assign data points to centroids based on the smallest dissimilarity;
- Update centroids using the majority label for each feature based on the assigned data points;
- Check convergence through the number of Hamming distance changes between the previous centroids and the new centroids. If the average Hamming distance is 0, consider convergence has been achieved

The remaining steps of computing purity score are the same as part 1.

Below are the purity scores for each digit class (rounded to 2 decimal points):

	0	1	2	3	4	5	6	7	8	9
Purity Score	0.53	0.79	0.56	0.33	0.42	0.72	0.29	0.49	0.19	0.34

As a comparison to results from part 1, we can observe that overall, Hamming distance does not perform as well as Euclidean distance as overall the purity scores are quite low for all classes. Therefore, Eulidean distance provides more accurate cluster results than Hamming distance.

References

1. https://eyer.ai/blog/pros-and-cons-of-supervised-vs-unsupervised-algorithms-for-scalable-anomaly-detection/
2. Pattern Recognition and Machine Learning.
3. Matrix Cookbook.
4. Topic 3 Spectral Clustering lecture slides.