Assignment 3 - Pierluigi Marchioro 881929

Table of Contents

Introduction

Perform classification of the MNIST database (or a sufficiently small subset of it) using:

- mixture of Gaussians with diagonal covariance (Gaussian Naive Bayes with latent class label);
- mean shift;
- normalized cut.

The unsupervised classification must be performed at varying levels of dimensionality reduction through PCA (say going from 2 to 200) In order to asses the effect of the dimensionality in accuracy and learning time.

Provide the code and the extracted clusters as the number of clusters k varies from 5 to 15, for the mixture of Gaussians and normalized-cut, while for mean shift vary the kernel width. For each value of *k* (or kernel width) provide the value of the Rand index:

$$R = 2(a+b)/(n(n-1))$$

where

- *n* is the number of images in the dataset.
- *a* is the number of pairs of images that represent the same digit and that are clustered together.
- *b* is the number of pairs of images that represent different digits and that are placed in different clusters.

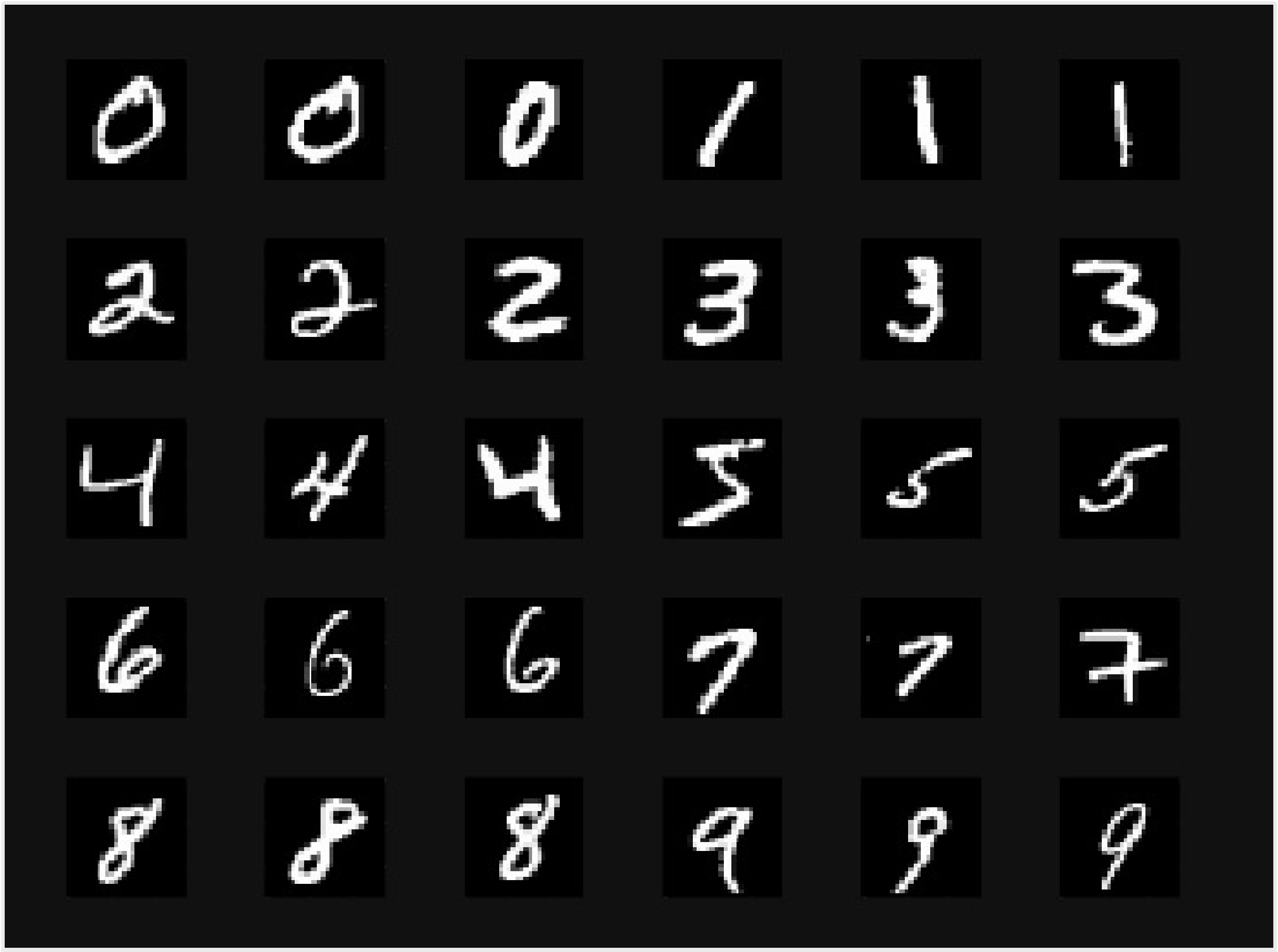
Explain the differences between the three models.

Tip: the means of the Gaussian models can be visualized as a greyscale images after PCA reconstruction to inspect the learned model

Dataset and Libraries

As aforementioned, the dataset used for the assignment was MNIST database. The dataset is a collection of 70k 28×28 hand-written digit images, each represented as a vector $x \in \mathbb{R}^{784} : x_i \in [0, 255]$, and equally as much labels, consisting of integers in

the range $[0, 9]$.
Below follows a sample of what the data looks like:


Sample images from the dataset - 3 samples per digit/label

The main libraries used to complete this assignment were the following:

- `sklearn` , `pandas`, `numpy` for data retrieval, feature engineering, training and predicting with the clustering models and, finally, scoring
- `seaborn`, `plotly` for plotting

Summary of the Next Sections

The report first provides the theoretical foundations of the machine learning techniques used for the project, notably:

- **PCA** for dimensionality reduction
- some preliminary sections on Latent Variable Models (LVMs), Expectation-Maximization (EM) Algorithms, Eigenvector-based Clustering, Graph-based

Clustering, and Hierarchical-Clustering, all of which will be useful to understand the workings of the unsupervised learning models used for this assignment

- **Gaussian Mixtures Clustering**
- **Normalized-Cut Clustering (Spectral Clustering)**
- **Mean Shift Clustering**
- **Rand Index** for model evaluation

Later on, a discussion on implementation details as well as the experimental results is provided.

Theoretical Foundations

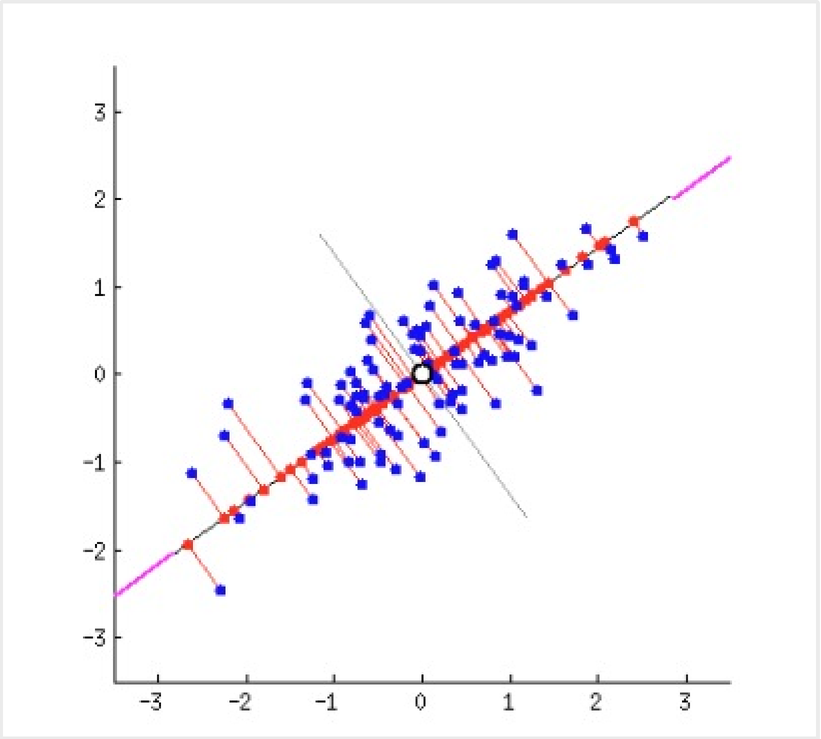
PCA - What is Principal Component Analysis?

Principal component analysis, or PCA, is a dimensionality reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set.

Reducing the number of variables of a data set naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity. Because smaller data sets are easier to explore and visualize and make analyzing data points much easier and faster for machine learning algorithms without extraneous variables to process.

Covariance Matrix and Principal Components

Preserving the maximum amount of information in PCA is synonymous with finding the independent components that explain the highest amount of variance in the data. For example, given the below scatterplot of the data, the component that explains the largest variance in the data is the one corresponding to the grey-colored line that traverses the points.



PCA - Example

The second principal components would be a vector perpendicular to the first, highlighting the fact that each component explains an independent portion of variance in the dataset.

But how are such components mathematically found? The first k principal components are the k **eigenvectors of the covariance matrix of the standardized dataset** (**standardization is important** here because we do not want the peculiarities of a single feature to have abnormal influence on the components) associated with the **highest k eigenvalues**. Such eigenvectors, in fact, represent the orthogonal directions of maximum variance of the dataset, and the percentage of the variance that they explain is given by the ration between their associated eigenvalue and the sum of all the eigenvalues.

Dimensionality-Reducing Transformation

As described in the previous section, ordering the eigenvectors by their respective eigenvalue allows us to find the principal components in order of significance (i.e. explained variance). Therefore, to reduce the dimensionality of the dataset we initially select the first k eigenvectors and use them to create the so called *feature vector*.
Such a vector F has the following form:

$$F = \begin{pmatrix} p_1^T & \dots & p_n^T \end{pmatrix}$$

where p_i is a principal component (column vector).
Finally, in order to obtain the transformed, lower-dimensionality dataset D, we multiply the transpose of the standardized dataset S by the transpose of the feature vector F:

$$D = S^T \cdot F$$

- compute the feature vector, which is made of the first $p \leq n$ components that I want to keep
- multiply the feature vector with the standardized original set

Intuition

Point Projection definition

Suppose we have point x. We want to project it onto some direction expressed by the unit vector u. The projection x' of point x is given by

$$x' = (x^T u)u$$

where the inner product $(x^T u)$ gives the of the point along the line described by u.

Preserved Information Definition

We want to maximize the *preserved information* of x'. But how can we define *preserved information*? We define it as the square of the magnitude of the projection, i.e.

$$(x^T u)^2$$

Intuitively, this quantity if maximum if x is parallel to u, and is 0 if x is perpendicular to u.

Intuitive Explanation

PCA asks the following question: *how can we arrange all the points in a line and preserve as much information as possible?*
To answer it, what we want to do is maximize the variance (preserved information) of the projected data onto some unit vector u, which represents *the line*, i.e. our new dimension. The variance of the projected data is

$$\frac{1}{n}\sum_{i=1}^{n}(u^T x_i - u^T \bar{x})^2 = u^T S u$$

where

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$
$$\text{and}$$
$$S = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^T (x_i - \bar{x})$$

Crucially, by assuming that our dataset of points x_i has a mean of 0 (which can be obtained by subtracting the mean $\bar{x}$ from the dataset beforehand), then the above equations take the following form:

$$\frac{1}{n}\sum_{i=1}^{n}(u^T x_i)^2 = u^T S u$$
where
$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$
$$S = \frac{1}{n}\sum_{i=1}^{n} x_i^T x_i$$

Thus, we can see that all we want to do is find the unit vector u that maximizes the variance (preserved information) of the dataset, i.e.

$$\max_u u^T S u$$
$$u^T u = 1$$

Such a quantity is found to be the eigenvector with the largest eigenvalue, after solving the above optimization problem with the Lagrange Multipliers method (check Formally - Optimization Problem).

Below, another intuitive interpretation about why u is the leading eigenvector is given, quoted from this StackExchange answer

Quote

Take A as the original matrix, x is the eigenvector and λ is the corresponding eigenvalue, then we have $Ax = \lambda x$. There is an intuitive way to think about it. What we are actually comparing is the projection of A onto the eigenvector x. The larger the projection is, the more variance that vector will represent. Based on this idea, the projection of A onto x is $A\frac{x}{\|x\|}$, which equals to $\lambda\frac{x}{\|x\|}$. Since $\frac{x}{\|x\|}$ is the unit vector of the eigenvector, we only need to check λ. Therefore, the largest λ indicates that the eigenvector retains the largest variance.

Latent Variable Models (LVMs)

Formally, a latent variable model (LVM) p is a probability distribution over two sets of variables x, z:

$$p(x, z; \theta)$$

where:

- x are variables observed at learning time in a dataset $\mathcal{D}$
- z are variables never observed, i.e. **latent variables**
- θ are the parameters of the distribution

Intuition and Basic Terminology

Latent variable models aim to model the probability distribution with latent variables.

Intuition

Latent variables are a transformation of the data points into a **continuous lower-dimensional space**.

Intuitively, the latent variables will describe or “explain” the data in a simpler way.

In a stricter mathematical form, data points x that follow a probability distribution $p(x)$, are mapped into latent variables z that follow a distribution $p(z)$.
Given that idea, we can now define five basic terms:

- The **prior distribution** $p(z)$ that models the behaviour of the latent variables
- The **likelihood** $p(x \mid z)$that defines how to map latent variables to the data points
- The **joint distribution** $p(x, z) = p(x \mid z)p(z)$, which is the multiplication of the likelihood and the prior and essentially describes our model.
- The **marginal distribution** $p(x)$ is the distribution of the original data and it is the ultimate goal of the model. The marginal distribution tells us how possible it is to generate a data point.

- The **posterior distribution** $p(z \mid x)$ which describes the latent variables that can be produced by a specific data point

Why Are They Useful?

The most important reason for studying LVMs is that they enable us to leverage our prior knowledge when defining a model.

Example

We have a dataset of articles such that we do not know to which topic (*topic* is the unobserved, **latent** variable) each article belongs to, but we do know that there are a total of k. We can leverage this information by setting up a model that accounts for these k latent variables (i.e. k probability distribution, independent and identical), such as a Gaussian Mixtures model with k clusters.

LVMs can also be viewed as increasing the expressive power of our model. In the case of GMMs, the distribution that we can model using a mixture of Gaussian components is much more expressive than what we could have modeled using a single component.

Learning Latent Variable Models

The core point to address about learning LVMs is that we have to resort to numerical, approximate (i.e. sensitive to local optima) optimization methods, because closed form expressions to learn the parameters (usually, at least) can't be obtained and the objective probability function is not convex.
The main algorithm used for this purpose is the *Expectation-Maximization* algorithm.

Marginal Likelihood Training

This section gives an overview of why an EM algorithm is needed.

How do we train an LVM? Our goal is still to fit the marginal distribution $p(x)$ over the visible variables x to that observed in our dataset $\mathcal{D}$. We should therefore maximize the *marginal* (w.r.t. the problem instances x) *log-likelihood* of the data

$$\log p(\mathcal{D}) = \sum_{x \in \mathcal{D}} \log p(x) = \sum_{x \in \mathcal{D}} \log \left(\sum_z p(x \mid z) p(z) \right)$$

This optimization objective is considerably more difficult than regular log-likelihood, even for directed graphical models. For one, we can see that the summation inside the log makes it impossible to decompose $p(x)$ into a sum of log-factors. Hence, even if the model is directed, we can no longer derive a simple closed form expression for the parameters.

Looking closer at the distribution of a data point x, we also see that it is actually a mixture

$$p(x) = \sum_z p(x \mid z) p(z)$$

of distributions $p(x \mid z)$ with weights $p(z)$. Whereas a single **exponential** family distribution $p(x)$ **has a concave log-likelihood**, the **log of a weighted mixture of such distributions is no longer concave or convex.**.

This non-convexity requires the development of specialized algorithms, such as the *EM Algorithm* explained in the following section.

Expectation-Maximization Algorithm

Info

As aforementioned, the main reason as to why we resort to such an algorithm for LVM learning is that the function to optimize is not convex and does not have a closed form expression.

The *Expectation-Maximization (EM)* algorithm is a hugely important and widely used algorithm for learning directed latent-variable graphical models $p(x, z; \theta)$ with parameters θ and latent z.

The EM algorithm relies on two simple observations:

1. If the latent variable z were fully observed, then the log-likelihood of the probability density/mass function $p(x, z)$ could be optimized directly with a closed

form solution (to understand why it isn't closed form, refer to Marginal Likelihood Training)

2. Knowing the parameters, we can often efficiently compute the posterior $p(z \mid x; \theta)$ (this is an assumption; it is not true for some models)

EM follows a simple iterative two-step strategy: given an estimate θ_t of the parameters, compute $p(z \mid x)$ and use it to *"hallucinate"* values for z.

Quote from the Reference

We haven't exactly defined what we mean by "hallucinating" the data. The full definition is a bit technical, but its instantiation is very intuitive in most models like GMMs.

By "hallucinating" the data, we mean computing the expected log-likelihood

$$E_{z \sim p(z|x)}[\log p(x, z; \theta)]$$

Intuition

1. $E_{z \sim p(z|x)}$ is the expected value of the latent variable z, which is distributed like (according to) $p(z \mid x)$
2. Why do we use/say expected value? Because in practice we are working with the observed conditional probabilities of each value of the latent variable z, i.e. the probabilities $p(z_k \mid x; \theta_t)$, and from those probabilities/likelihoods we want to extract the likelihood of z "as a whole"

The above expectation is what gives the EM-algorithm half of its name. If z is not too high-dimensional (e.g. in Gaussian Mixture Models it is a one-dimensional categorical variable), then we can compute this expectation.

We can formally define the EM algorithm as follows.

1. Let $\mathcal{D}$ be our dataset
2. Starting at an initial θ_0, repeat until convergence for $t = 1, 2, \ldots$:
    1. *E-Step*: For each $x \in \mathcal{D}$, compute the posterior $p(z \mid x; \theta_t)$

2. *M-Step*: Compute new parameters θ_{t+1} as:

$$\theta_{t+1} = \arg\max_{\theta} \sum_{x \in \mathcal{D}} E_{z \sim p(z|x)} [\log p(x, z; \theta)]$$

Intuition

At the expectation step, we calculate the likelihoods that will be used to optimize the parameters at the maximization step. This process is repeated until convergence, meaning that the parameters reached a local optimum.

Characteristics: Pros and Cons

The EM-algorithm, it follows that the latter has the following characteristics:

- The marginal likelihood increases after each EM cycle and it is upper-bounded by its true global maximum, meaning that EM must eventually converge
- Since we are optimizing a non-convex objective, we have no guarantee to find the global optimum. In fact, EM in practice converges almost always to a local optimum, which, moreover, heavily depends on the choice of initialization.
    - Different initial parameters θ_0 can lead to very different solutions, meaning that it is generally a good idea to use multiple restarts of the algorithm and choose the best one in the end.

Eigenvector-based Clustering - A General Framework

Eigenvectors: Max-Variance Direction

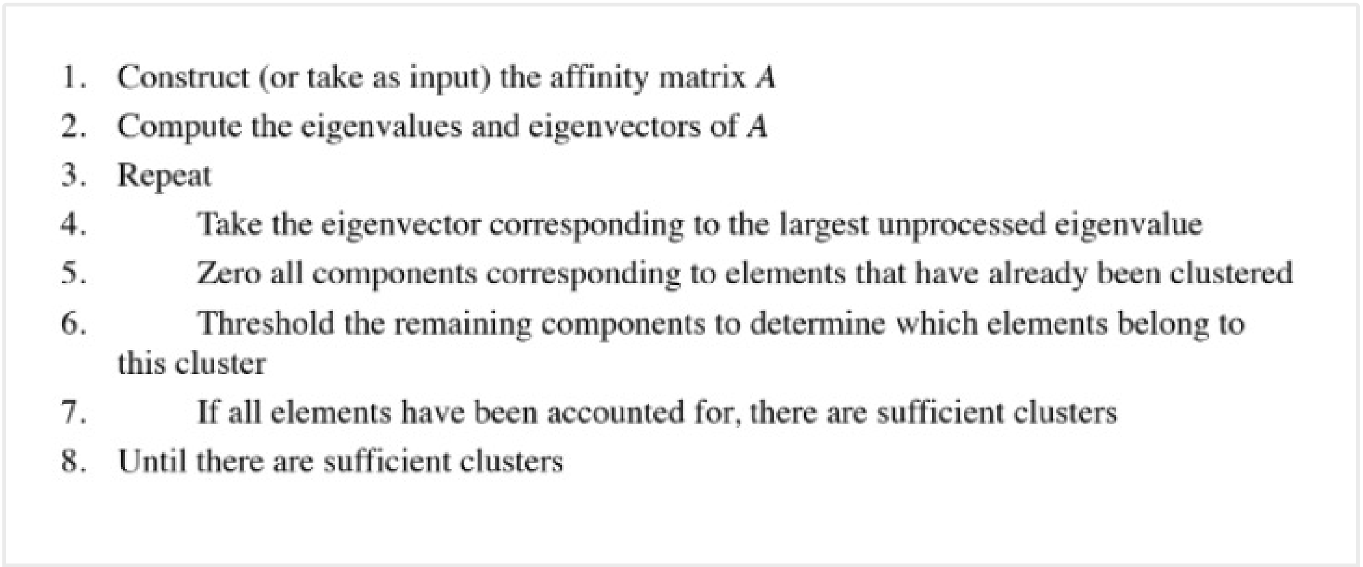
The eigenvectors represent the directions in the data that explain the majority of the variation, while the eigenvalues represent the magnitude of the variation along each eigenvector.

Main Idea

The main idea is that if we could somehow generate a desired number of eigenvectors from the original dataset (whose eigenvalues are maximum, highlighting that they are the top variance direction), we could combine them in some way with said dataset to generate additional information that helps us to determine clusters.

In particular, we could threshold each data point based on such new information, meaning that we check if a metric applied to each (transformed) data point is above some cluster membership metric.

Pseudocode



```
1. Construct (or take as input) the affinity matrix $A$
2. Compute the eigenvalues and eigenvectors of $A$
3. Repeat
4.         Take the eigenvector corresponding to the largest unprocessed eigenvalue
5.         Zero all components corresponding to elements that have already been clustered
6.         Threshold the remaining components to determine which elements belong to
   this cluster
7.         If all elements have been accounted for, there are sufficient clusters
8. Until there are sufficient clusters
```

Eigenpair Clustering - General framework pseudocode

Note

> 6. Threshold the remaining components [...]

This means to apply some form of affinity metric to each remaining element (possibly transformed with the use of the eigenpairs), to check if it belongs to a new cluster

Formally - Optimization Problem

The problem of eigenvector-based clustering fundamentally consists in solving the below optimization problem:

$$\max_x \sum_i^n \sum_j^n w_{ij} x_i x_j = x^T W x,$$
$$x \in \mathbb{R}^n : x_k \in \{0,1\}, \forall k = 1, \ldots, n$$

where:

- x_i measures the participation of the node i in some cluster Z ($x_i = 1$ if node $i \in Z$, $x_i = 0$ otherwise), meaning that the **vector x contains the participation information of all the points in the dataset to cluster Z.**
- W is the affinity (similarity) matrix built from the dataset, with w_{ij} being the affinity (similarity, weight of the edge) between nodes i and *j*.

Intuition

By maximizing the above sum with respect to x, we are trying to find the "partecipation mask" that maximizes the sum of the similarities (weights w_{ij}). In fact, we have that the product

$$w_{ij} x_i x_j \neq 0 \iff x_i = 1 \wedge x_j = 1$$

meaning that we want to mark points as part of cluster Z if they can contribute significantly to the sum of similarities, i.e. to the cluster cohesiveness.

We have to impose the constraint $x^T x = 1$, that is, that x is a unit vector, to ensure that we are looking for the direction of maximum change in the data without being influenced by the magnitude of the direction. Without this constraint, we could end up with a vector that is arbitrarily large and not related to the underlying data.

The mathematical connection between this optimization problem and eigenpairs becomes obvious once the Lagrange method is applied to solve it:

1. Given the optimization problem

$$\begin{aligned} &\max_x x^T W x \\ &\text{s.t. } x^T x = 1 \end{aligned}$$

2. Transform it to a minimization problem

$$\begin{aligned} &\min_x -x^T W x \\ &\text{s.t. } x^T x = 1 \end{aligned}$$

3. Compute the Lagrangian

$$L(x, \lambda) = -x^T W x + \lambda(x^T x - 1)$$

4. Compute the partial derivative with respect to x to find its optimal value:

$$\frac{\partial L(x, \lambda)}{\partial x} = -2Wx + 2\lambda x$$

5. Set it to 0 and solve it
   $$
   \begin{align}
   -2Wx + 2\lambda x &= 0 \
   2Wx &= 2\lambda x \
   Wx &= \lambda x \
   \end{align}

$$Which is exactly the definition of eigenpairs.$$

Alternative Form: Rayleigh Quotient

A variant of this optimization problem uses the Rayleigh Quotient as objective function:

$$R(W, x) = \frac{x^T W x}{x^T x}$$

If x is eigenvector, $R(W, x)$ is its eigenvalue

Note that, if x is an eigenvector of W, $R(W, x)$ can be written as

$$\frac{x^T \lambda x}{x^T x} = \lambda \frac{x^T x}{x^T x} = \lambda$$

This is because of the theorem behind Rayleigh's Quotient, which will not be discussed here.

Plugging this objective function into the previously described optimization problem yields the same results, that is, optimizing w.r.t. x yields the largest eigenpair.

Intuition

Note, in fact, that under the constraint that $x^T x = 1$, the denominator of the Rayleigh Quotient, can be omitted, meaning that the objective function becomes the same.

What do Eigenvectors Actually Represent?

In the formal discussion of the underlying optimization problem, it is mentioned that x, the (eigen)vector that we are trying to find represents some kind of membership score to some cluster for each element of the dataset. In other words, each x_i is a value, ideally discrete but typically continuous (a real number), that represents "how much instance i is part of the cluster". This is why we are able to identify the instances that belong to the same cluster: all we need to do is define some kind of threshold t such that if $x_i \geq t$, then it belongs to the cluster, else it doesn't. Usually, t is 0 or the median of the values in the eigenvector.

Graph-based Clustering

Let a graph $G = (V, E, w)$ be a weighted graph representing a dataset $\mathcal{D}$, where:

- V is the set of nodes, each representing a point $\in \mathcal{D}$
- E is the set of weighted edges with weights $\in w$, each weight representing some sort of affinity/similarity between the two incident nodes

This clustering methodology is based on finding the *best cut*, that is, the graph cut that removes the least valuable edges in terms of similarity, consequently splitting the graph into connected components with maximum cohesiveness. Finding such a cut is defined as the *Minimum Cut Problem*

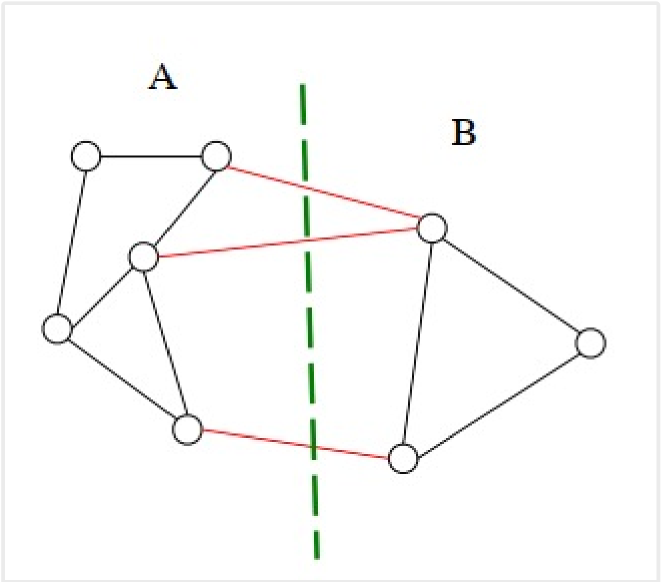
Minimum Cut Problem

Given a graph $G = (V, E, w)$ and a cut (A, B), with $B = V \setminus A$, define:

$$cut(A, B) = \sum_{i \in A} \sum_{j \in B} w_{ij}$$

Intuition

$cut(A, B)$ is the sum of the weights of the edges that connect the components A, B, that is, the sum of the weights of the edges traversed by the cut.

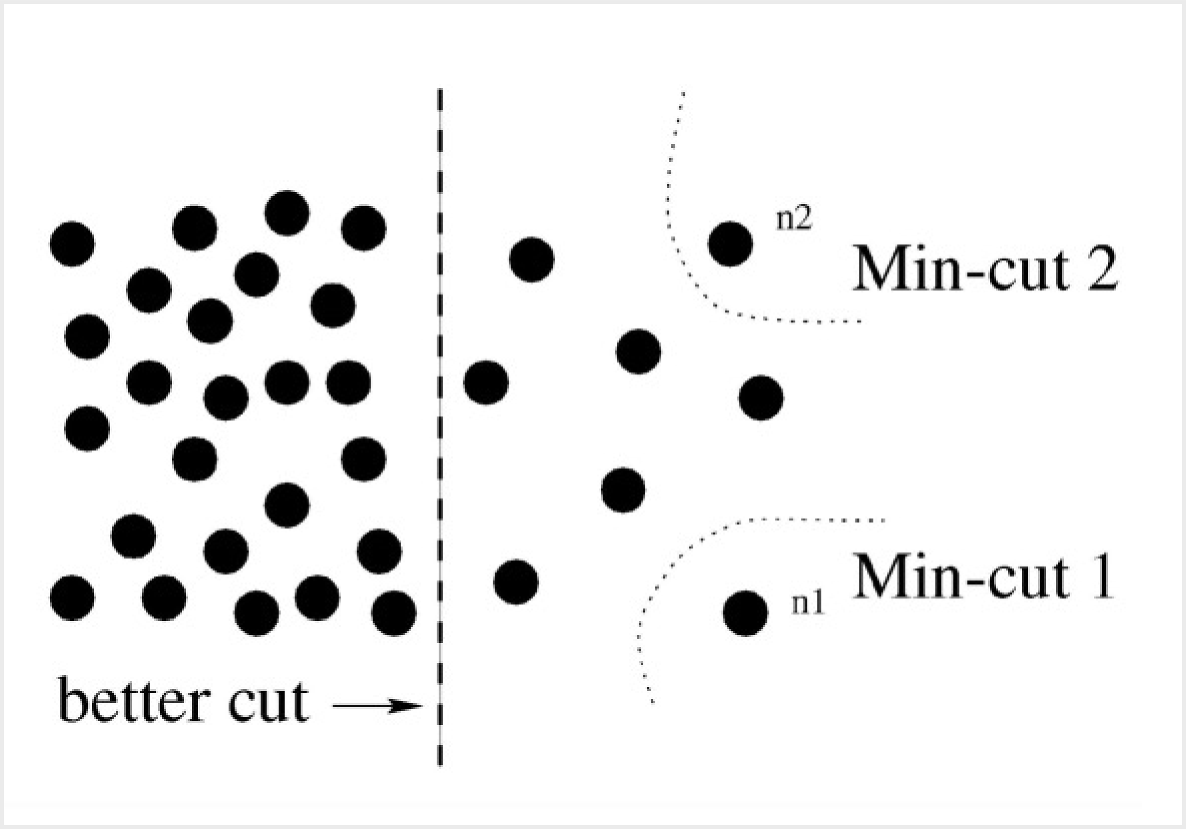


Example of graph cut

Pros and Cons

Advantages: Such a problem is solvable in polynomial time, although with local optima

Shortcomings: Optimizing such a cut means to highly favor unbalanced clusters, usually by splitting isolated nodes from the rest. Such a problem is solved by using the notion of *Normalized cut* (see following sections, Normalized-Cut Clustering)

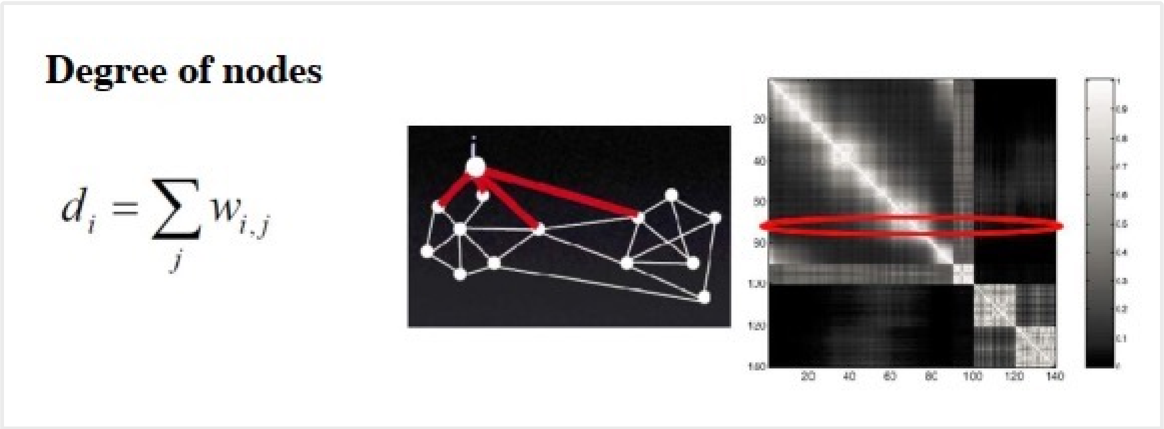


Note that the min cuts split the isolated nodes from the rest of the dataset, intuitively because such nodes are connected with low weights due to their low similarity with the rest of the data (they are isolated)

Preliminary Terminology

Some additional basic graph terminology before diving into the following sections.

Degree of Nodes



Degree of nodes

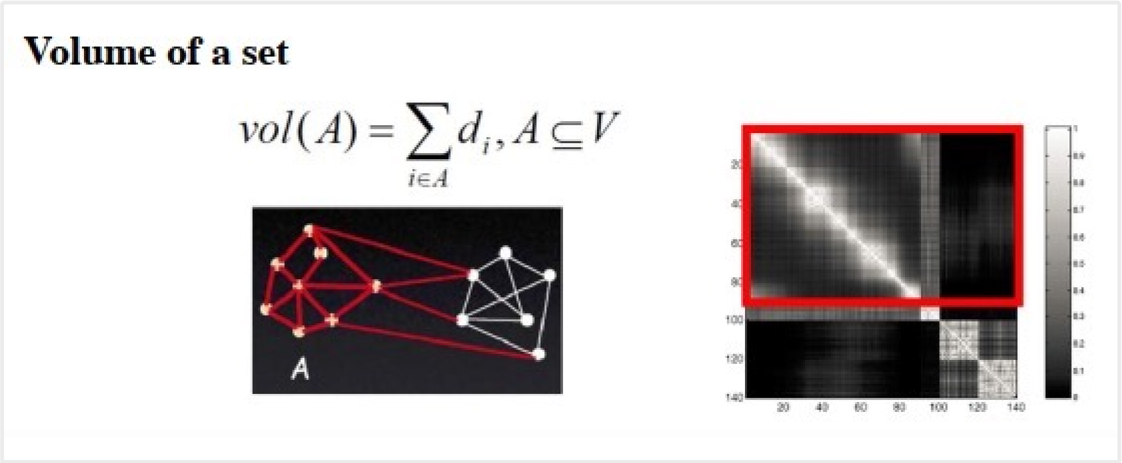
$$d_i = \sum_j w_{i,j}$$

Node Degree

Intuition

The degree d_i of a node i is the sum of the weights of its incident edges. Such weights/edges are highlighted in red in the image above, respectively in the graph and similarity matrix (note that values in the same row are circled) representations.

Volume of a set



Volume of a set

$$vol(A) = \sum_{i \in A} d_i, A \subseteq V$$

Set Volume

Intuition

The volume of a set of nodes A is the sum of the degrees of each node $\in A$. Such weights are highlighted in red in the image above, respectively in the graph and similarity matrix (note that all the rows belonging to a certain set of nodes are circled) representations.

Graph's Laplacian Matrix

A graph's Laplacian Matrix L is a compact representation of a graph w.r.t. the relationships between nodes (similarities) and the connection properties (degrees, volumes). It is defined as follows:

$$L = D - W$$

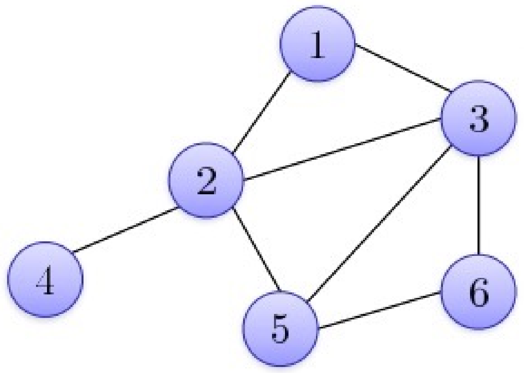
where:

- W is the affinity/similarity/adjacency matrix; it is a symmetric matrix, the diagonal being filled with 0s and the non-diagonal elements being the weights of the edges (which are just $0, 1, -1$ in case of non-weighted, directed/undirected graphs)
- D is the degree (see Degree of Nodes) matrix; it is a diagonal matrix (with non-zero values, the degrees, only in the diagonal)

Spectral Properties

The properties of the Laplacian matrix are referred to as *spectral properties*.

1. **Symmetric**: Since both W and D are symmetric matrices, L is also symmetric
2. **Rows/Columns sum up to 0**: This is because the diagonal element of row/column i is the degree of the node i (sum of the weights), while non-diagonal elements are the negative weights (because $L = D - W$) of edges incident to i
3. **L is PSD**: L is *Positive Semi-Definite* (proof below, see Proof that L is Positive Semi-Definite), meaning that its eigenvalues are all ≥ 0.
4. **0 is an eigenvalue**: 0 is always an eigenvalue of L
5. **$\lambda_1 = 0$ is the lowest eigenvalue**: since L is PSD, then it is also the lowest eigenvalue (proof below, see Proof that 0 is the Smallest Eigenvalue of L), meaning that $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$
6. **The multiplicity of $\lambda_1 = 0$ is the number of connected components**



$$L = \begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 \\ -1 & -1 & 4 & 0 & -1 & -1 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & -1 & 0 & 3 & -1 \\ 0 & 0 & -1 & 0 & -1 & 2 \end{pmatrix}$$

Assume the weights of edges are 1

Example of Laplacian Matrix - Note the symmetry and that rows/cols sum up to 0

Proof that L is Positive Semi-Definite

For all vectors f in $\mathbf{R}^n$, we have:

$$f^\top Lf = \frac{1}{2}\sum_{ij=1}^{n} w_{ij}(f_i - f_j)^2$$

Indeed:

Recall that L = D - W

$$\begin{aligned} f^\top Lf &= f^\top Df - f^\top Wf \\ &= \sum_i d_i f_i^2 - \sum_{i,j} f_i f_j w_{ij} \end{aligned}$$

Note that, w.r.t. f_i and f_j, this is a notable binomial square: (a-b)^2 = a^2 -2ab + b^2

$$\begin{aligned} &= \frac{1}{2}\left(\sum_i\Big(\sum_j w_{ij}\Big)f_i^2 - 2\sum_{ij} f_i f_j w_{ij} + \sum_j\Big(\sum_i w_{ij}\Big)f_j^2\right) \\ &= \boxed{\frac{1}{2}\sum_{ij} w_{ij}(f_i - f_j)^2} \end{aligned}$$

This quantity is always >= 0, thereby proving that L is positive semi-definite

Proof that Laplacian Matrices are PSD

Proof that 0 is the Smallest Eigenvalue of L

There is a nice theorem that states the following:

☰ Theorem

Given a matrix $A \in \mathbb{R}^{n \times n}$, if each row (or column) of A sums up to k, then k is an eigenvalue of A, with a corresponding eigenvector $v = (1, \ldots, 1) \in \mathbb{R}^n$

The proof is quite simple:

$$\begin{aligned} &\text{Let } v = (1, \ldots, 1) \in \mathbb{R}^n \\ &\forall i = 1, \ldots, n : \sum_j^n a_{ij} = 1 \cdot k \implies Av = kv \end{aligned}$$

the last line being exactly the definition of eigenpairs.

Since L is such that every row/column sums up to 0, then 0 is an eigenvalue of L. In particular, it is the lowest, because L is Positive Semi-Definite.

Implications of 0 Being an Eigenvalue

Because of the reason why $\lambda_1 = 0$ is an eigenvalue (see Proof that 0 is the Smallest Eigenvalue of L), we have that the associated eigenvector is $v = (1, \ldots, 1) \in \mathbb{R}^n$. By taking into consideration what eigenvectors actually represent w.r.t. clustering, the interpretation that we can give to v is that all nodes are part of the same component, being the whole graph.

Spectral Gap and Cheeger Constant

- The spectral gap is defined as the difference between the moduli of the two largest eigenvalues of a matrix.
- The Cheeger constant is a measure of "bottleneckedness" of a graph: a graph is bottlenecked if there are few edges that connect two components. The constant $h(G)$ is $> 0 \iff$ the graph is connected.
  - The more $h(G) \to 0^+$, the more the graph is "bottlenecked" (albeit still connected)

The relationship between the spectral gap and the Cheeger constant is described by the *Cheeger Inequalities*, which roughly tell us that the bigger the spectral gap, the more robust is the connectedness of the graph.

Normalizations

For some applications the need to normalize the Laplacian L might rise.

Row Normalization: rows of L sum up to 1

$$L_{rw} = D^{-1}L = D^{-1}(D - W) = I - D^{-1}W$$

Symmetric Normalization: both rows and columns sum up to 1

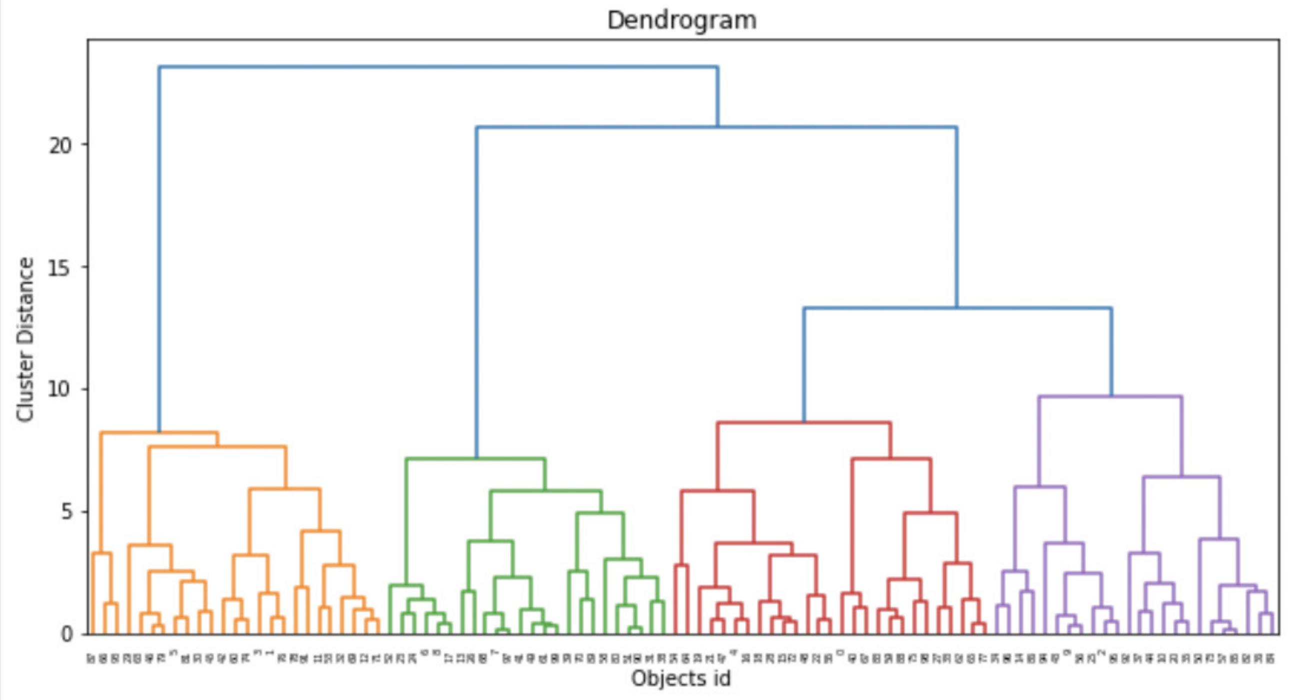
$$D^{-\frac{1}{2}}LD^{-\frac{1}{2}} = I - D^{-\frac{1}{2}}WD^{-\frac{1}{2}}$$

Hierarchical-Clustering

In data mining and statistics, hierarchical clustering (also called hierarchical cluster analysis or HCA) is a method of cluster analysis that seeks to build a hierarchy of clusters. Strategies for hierarchical clustering generally fall into two categories:

- **Agglomerative**: This is a "bottom-up" approach, where each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.
- **Divisive**: This is a "top-down" approach, where all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy.

In general, the merges and splits are determined in a greedy manner. The results of hierarchical clustering are usually presented in a dendrogram.



Example dendogram

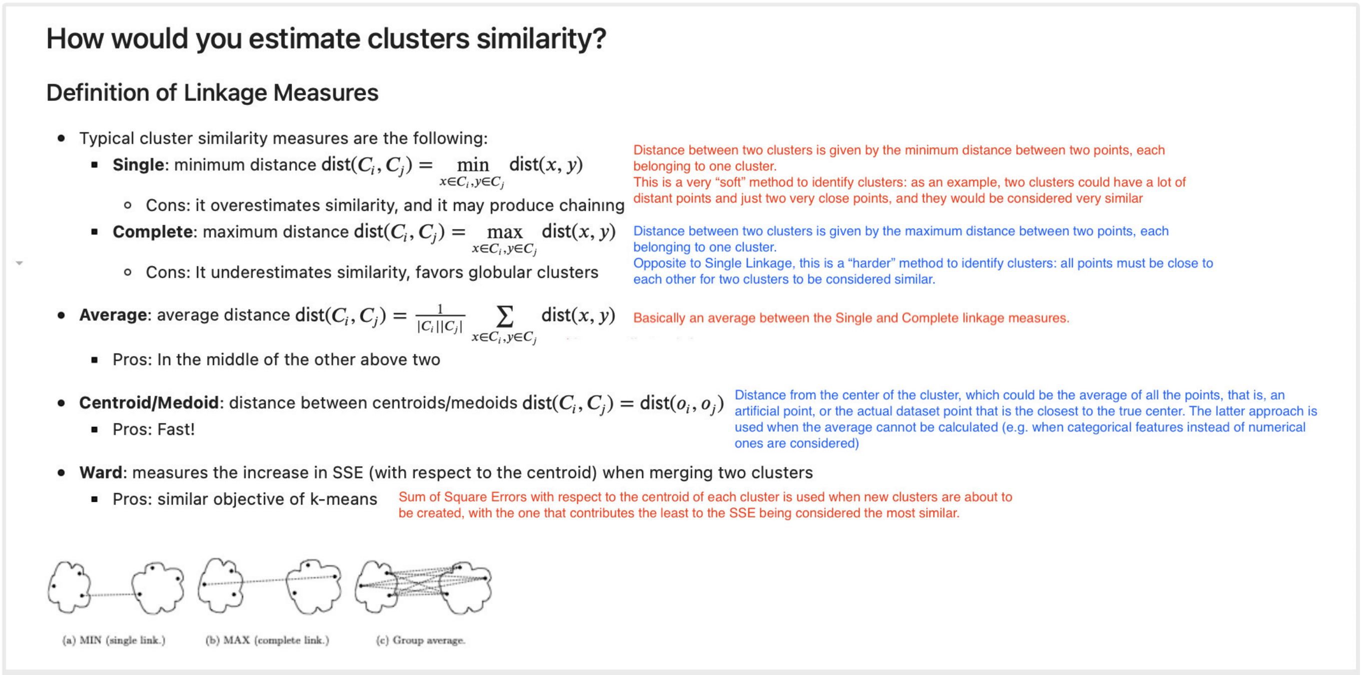
Hierarchical clustering has the distinct advantage that any valid measure of distance can be used. In fact, the observations themselves are not required: all that is used is a matrix of distances/similarities. On the other hand, except for the special case of single-linkage distance, none of the algorithms (except exhaustive search in $O(2^n)$) can be guaranteed to find the optimum solution.

Complexity

Agglomerative: The standard algorithm for hierarchical agglomerative clustering (HAC) has a time complexity of $O(n^3)$ and requires $\Omega(n^2)$ memory, which makes it too slow for even medium data sets. However, for some special cases, optimal efficient agglomerative methods (of complexity $O(n^2)$) are known: SLINK for single-linkage and CLINK.

Divisive: Divisive clustering with an exhaustive search is $O(2^n)$, but it is common to use faster heuristics to choose splits, such as k-means.

Linkage Measures



How would you estimate clusters similarity?

Definition of Linkage Measures

- Typical cluster similarity measures are the following:
  - **Single**: minimum distance $\text{dist}(C_i, C_j) = \min_{x \in C_i, y \in C_j} \text{dist}(x, y)$
    - Cons: it overestimates similarity, and it may produce chaining
  - **Complete**: maximum distance $\text{dist}(C_i, C_j) = \max_{x \in C_i, y \in C_j} \text{dist}(x, y)$
    - Cons: It underestimates similarity, favors globular clusters
- **Average**: average distance $\text{dist}(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{x \in C_i, y \in C_j} \text{dist}(x, y)$
  - Pros: In the middle of the other above two
- **Centroid/Medoid**: distance between centroids/medoids $\text{dist}(C_i, C_j) = \text{dist}(o_i, o_j)$
  - Pros: Fast!
- **Ward**: measures the increase in SSE (with respect to the centroid) when merging two clusters
  - Pros: similar objective of k-means

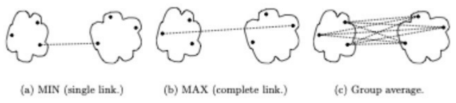
Distance between two clusters is given by the minimum distance between two points, each belonging to one cluster.
This is a very "soft" method to identify clusters: as an example, two clusters could have a lot of distant points and just two very close points, and they would be considered very similar

Distance between two clusters is given by the maximum distance between two points, each belonging to one cluster.
Opposite to Single Linkage, this is a "harder" method to identify clusters: all points must be close to each other for two clusters to be considered similar.

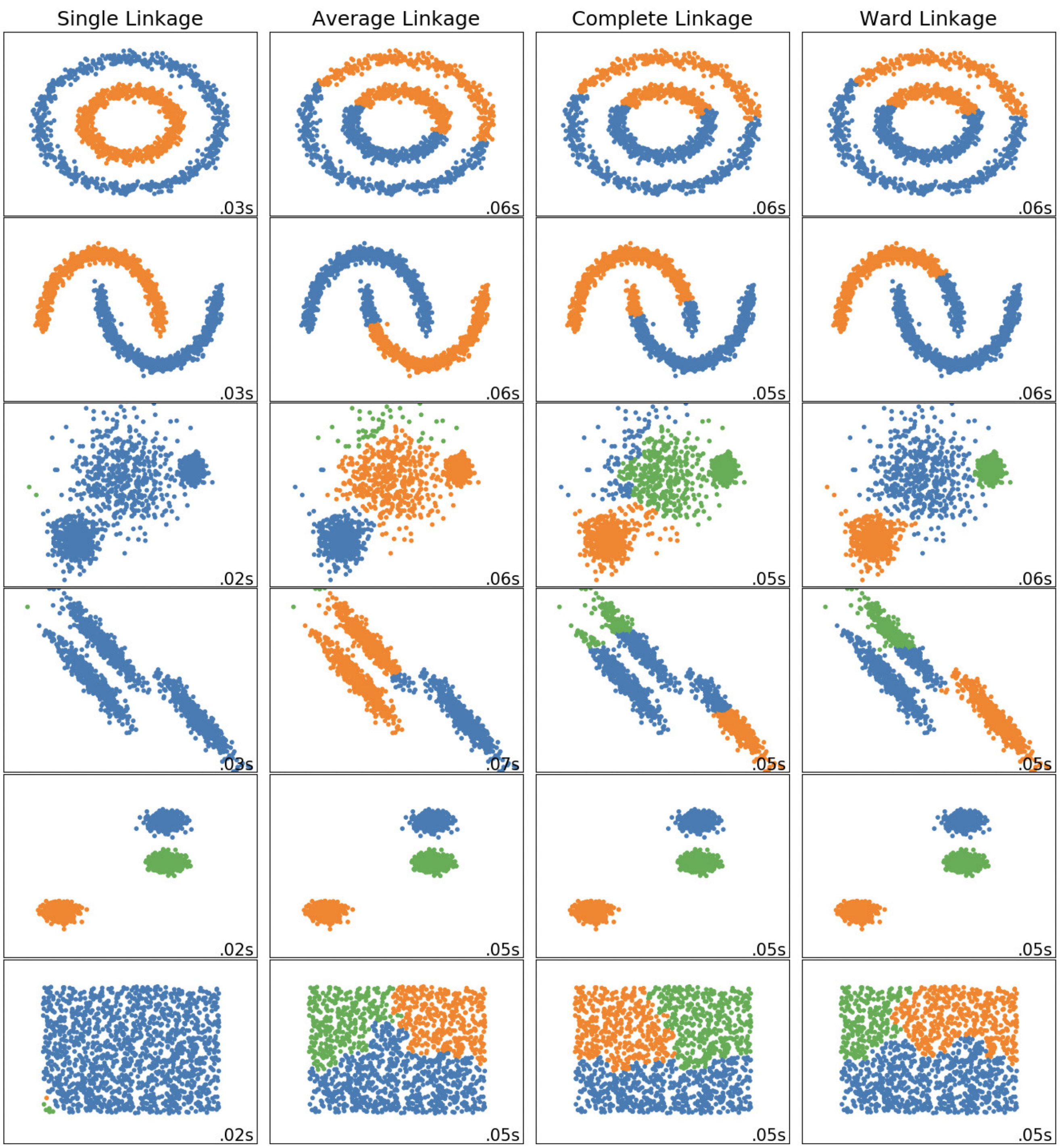
Basically an average between the Single and Complete linkage measures.

Distance from the center of the cluster, which could be the average of all the points, that is, an artificial point, or the actual dataset point that is the closest to the true center. The latter approach is used when the average cannot be calculated (e.g. when categorical features instead of numerical ones are considered)

Sum of Square Errors with respect to the centroid of each cluster is used when new clusters are about to be created, with the one that contributes the least to the SSE being considered the most similar.



(Some) Hierarchical Clustering linkage measures



Linkage measures visually - Effects on clusters

Gaussian Mixtures Clustering

The Gaussian Mixture Model (GMM) is a probabilistic model used for clustering and density estimation. It assumes that the data is generated from a mixture of several Gaussian distributions, each representing a distinct cluster. GMM assigns probabilities to data points, allowing them to belong to multiple clusters simultaneously, hence why such models are said to use a *soft clustering approach*.
The model can be thought of as a generalized form of the K-Means algorithm that tries to fit a gaussian distribution on a set of points belonging to one of the K clusters, with K being a hyper-parameter of the model, meaning that clusters can be oval-shaped instead of strictly circular.
Such a model is widely used in machine learning and pattern recognition applications.

Formally

Gaussian Mixture Models (GMMs) are part of the family of so called *Latent Variable Models (LVMs)*.

In a GMM, each data point is a tuple (x_i, z_i), with $x_i \in \mathbb{R}^d$ and $z_i \in \{1, 2, \ldots, K\}$ (z_i is discrete). The joint probability function p is a model

$$p(x, z) = p(x \mid z)p(z)$$

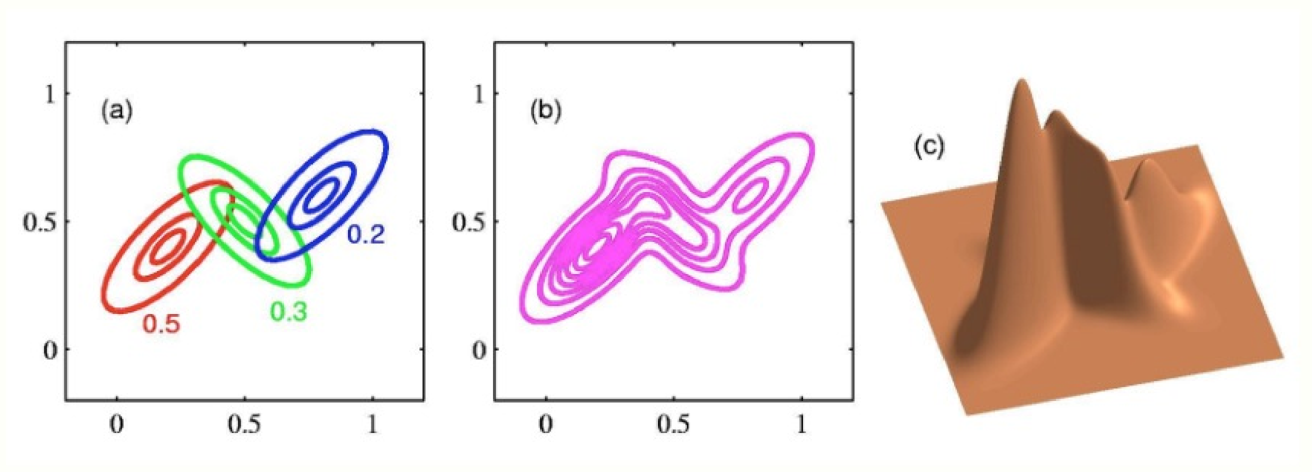
where $p(z = k) = \pi_k$ for some vector of class probabilities $\pi \in \Delta_{K-1}$ (Δ_{K-1} is the $(K-1)$-dimensional standard simplex) and

$$p(x \mid z = k) = \mathcal{N}(x; \mu_k, \Sigma_k)$$

is a multivariate Gaussian with mean and variance μ_k, Σ_k.

This model assumes that our observed data is comprised of K clusters with proportions $\pi = \pi_1, dots, \pi_K$ and that each clusters is a Gaussian. We can see that $p(x)$ is a mixture by explicitly writing out this probability:

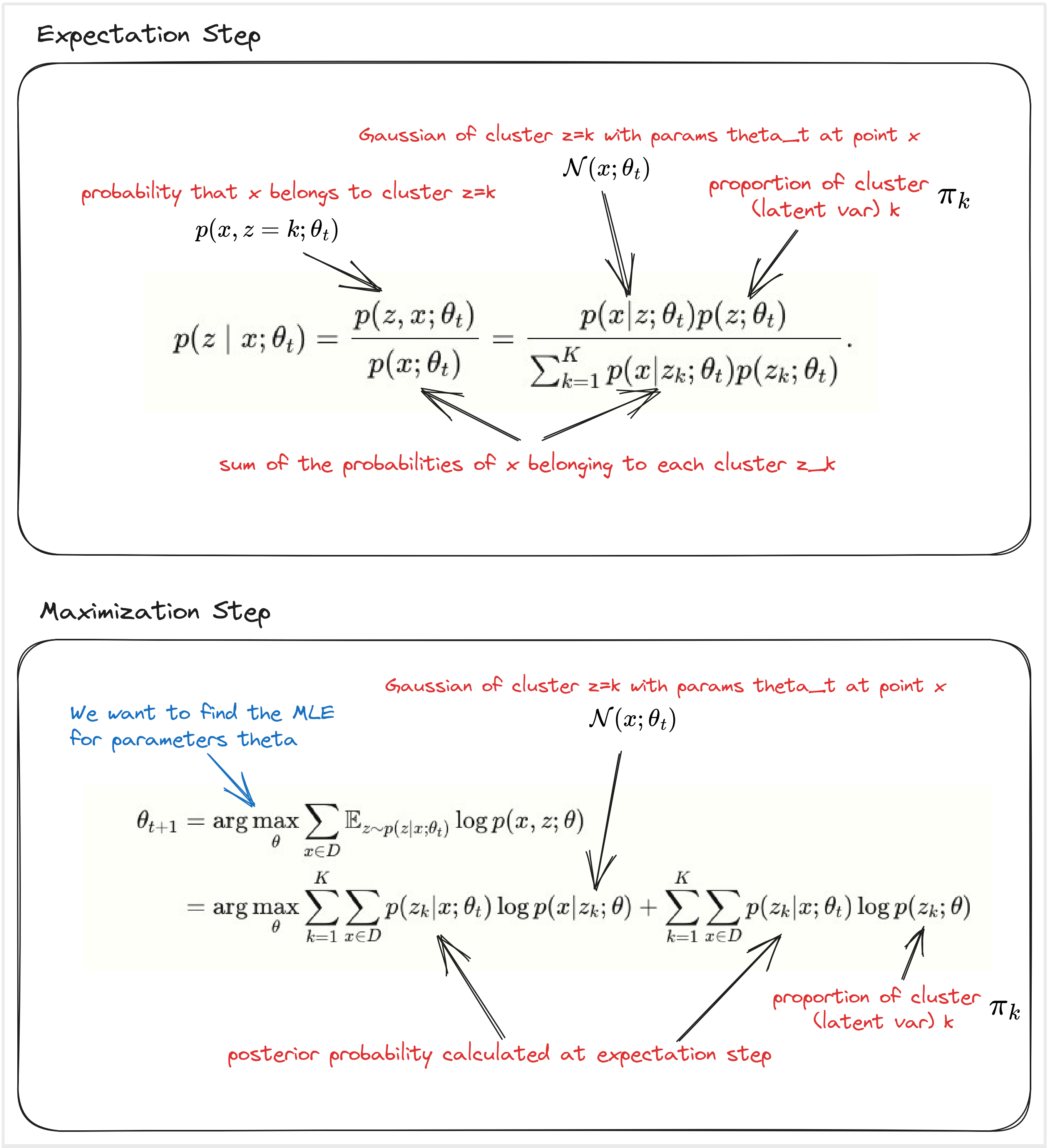
$$p(x) = \sum_{k=1}^{K} p(x \mid z = k)p(z = k) = \sum_{k=1}^{K} \mathcal{N}(x; \mu_k, \Sigma_k)\pi_k$$



GMM clusters - Example

Expectation-Maximization Steps for GMMs

The Expectation-Maximization algorithm applied to the Gaussian Mixture model consists in the calculations (at iteration t) highlighted in the below image:


Expectation Step
Gaussian of cluster z=k with params theta_t at point x
$\mathcal{N}(x;\theta_t)$
probability that x belongs to cluster z=k
$p(x, z=k; \theta_t)$
proportion of cluster (latent var) k
π_k
$p(z \mid x; \theta_t) = \frac{p(z, x; \theta_t)}{p(x; \theta_t)} = \frac{p(x|z; \theta_t) p(z; \theta_t)}{\sum_{k=1}^{K} p(x|z_k; \theta_t) p(z_k; \theta_t)}.$
sum of the probabilities of x belonging to each cluster z_k
Maximization Step
We want to find the MLE for parameters theta
Gaussian of cluster z=k with params theta_t at point x
$\mathcal{N}(x;\theta_t)$
$\theta_{t+1} = \arg\max_{\theta} \sum_{x \in D} \mathbb{E}_{z \sim p(z|x;\theta_t)} \log p(x, z; \theta)$
$= \arg\max_{\theta} \sum_{k=1}^{K} \sum_{x \in D} p(z_k|x; \theta_t) \log p(x|z_k; \theta) + \sum_{k=1}^{K} \sum_{x \in D} p(z_k|x; \theta_t) \log p(z_k; \theta)$
proportion of cluster (latent var) k
π_k
posterior probability calculated at expectation step

EM steps for GMMs

Normalized-Cut Clustering

Introduction

Normalized-cut is a divisive hierarchical algorithm that operates on dataset represented as a graph. The points inside the feature space are represented as nodes, the distance between each point is represented as a weighted edge. The closer two points are between each other, the higher the value of the related edge. The normalized-cut criterion is based on the idea of minimizing the dissimilarity between data points within a cluster, and maximizing the dissimilarity between data points across different clusters. The dissimilarity is calculated as the sum of the weights of the edges that connect the data points, where the weights represent the similarity or distance between them. The normalized-cut criterion divides this sum by the total weight of the edges that connect the data points to all other data points, to account for the size and density of the clusters.

Various approaches can be adopted to solve this problem, all based on solving an underlying eigenvalue problem, better explained in the following sections, that allows to detect clusters in feasible time.

Main Objective

The objective of normalized cut is to split the graph while minimizing the value of $NCut$. This is performed with the goal of having the two resulting clusters with high cohesiveness (high intra-similarity) and high separation (low inter-similarity).

A normalized cut $NCut(A, B)$ is defined as follows:

$$\begin{aligned} NCut(A,B) &= \left(\frac{1}{Vol(A)} + \frac{1}{Vol(B)}\right) cut(A,B) = \\ &= \frac{cut(A,B)}{Vol(A)} + \frac{cut(A,B)}{Vol(B)} \end{aligned}$$

Intuition

The idea of dividing by the sum of the volumes of the two components A, B is useful to **overcome the problem of isolated nodes**: let's say that B is an isolated node (component), then the ratio $\frac{cut(A,B)}{Vol(B)}$ will be very high, because $Vol(B)$ will be small, meaning that such a cut is probably unfavorable.

Formally - Optimization Problem

We can represent the normalized cut $NCut(A, B)$ as an $n = \mid V \mid$-dimensional vector defined as:

$$x = (x_i) : x_i = 1 \text{ if } i \in A \wedge x_i = -1 \text{ if } i \in B$$

We can then exploit the Rayleigh Quotient to describe the minimization of $Ncut(x)$ (with x being the vector defined above) as

$$\min_x NCut(x) = \min_y \frac{y^T(D-W)y}{y^T Dy}$$
$$\text{s.t. } y^T D1 = \sum_i y_i d_i = 0$$

where:

- 1 represent a $\mid V \mid$-dimensional vector of 1s
- $y_i \in \{1, -b\}$, for some b, meaning that y (as is x) is a discrete-valued vector.

NP-Complete: This minimization problem is however NP-Hard, meaning that we have to adopt another strategy to solve it.

NCut as (Relaxed) Eigen-System

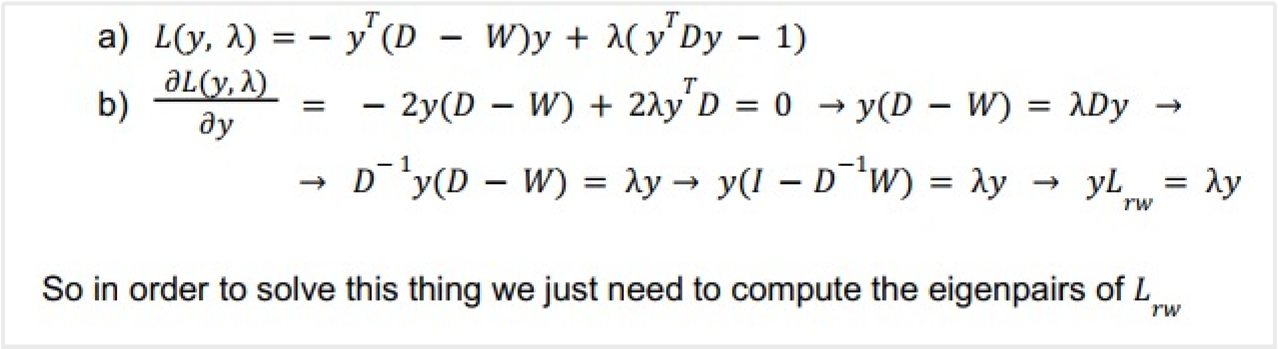
If we relax the constraint of the previous optimization problem and allow y to be a real-valued (as opposed to being discrete with just two possible values) vector, then it can be shown that the problem can be equivalently rewritten as

$$\min_y y^T(D-W)y$$
$$\text{s.t. } y^T Dy = 1$$

Constraint Relaxation

Constraint relaxation means to make the constraints of an optimization problem shallower, consequently widening the solution space with the gain of making the problem more tractable. The hope is that the optima of the relaxed problem coincide with the optima of the original one; this is generally true only for some optima of the relaxed problem.

This amounts to solving the eigenvalue problem $(D - W)y = \lambda Dy$ (note that $L = (D - W)$) after applying the Lagrange multipliers method.



a) $L(y, \lambda) = -y^T(D - W)y + \lambda(y^T D y - 1)$

b) $\frac{\partial L(y,\lambda)}{\partial y} = -2y(D - W) + 2\lambda y^T D = 0 \rightarrow y(D - W) = \lambda D y \rightarrow$

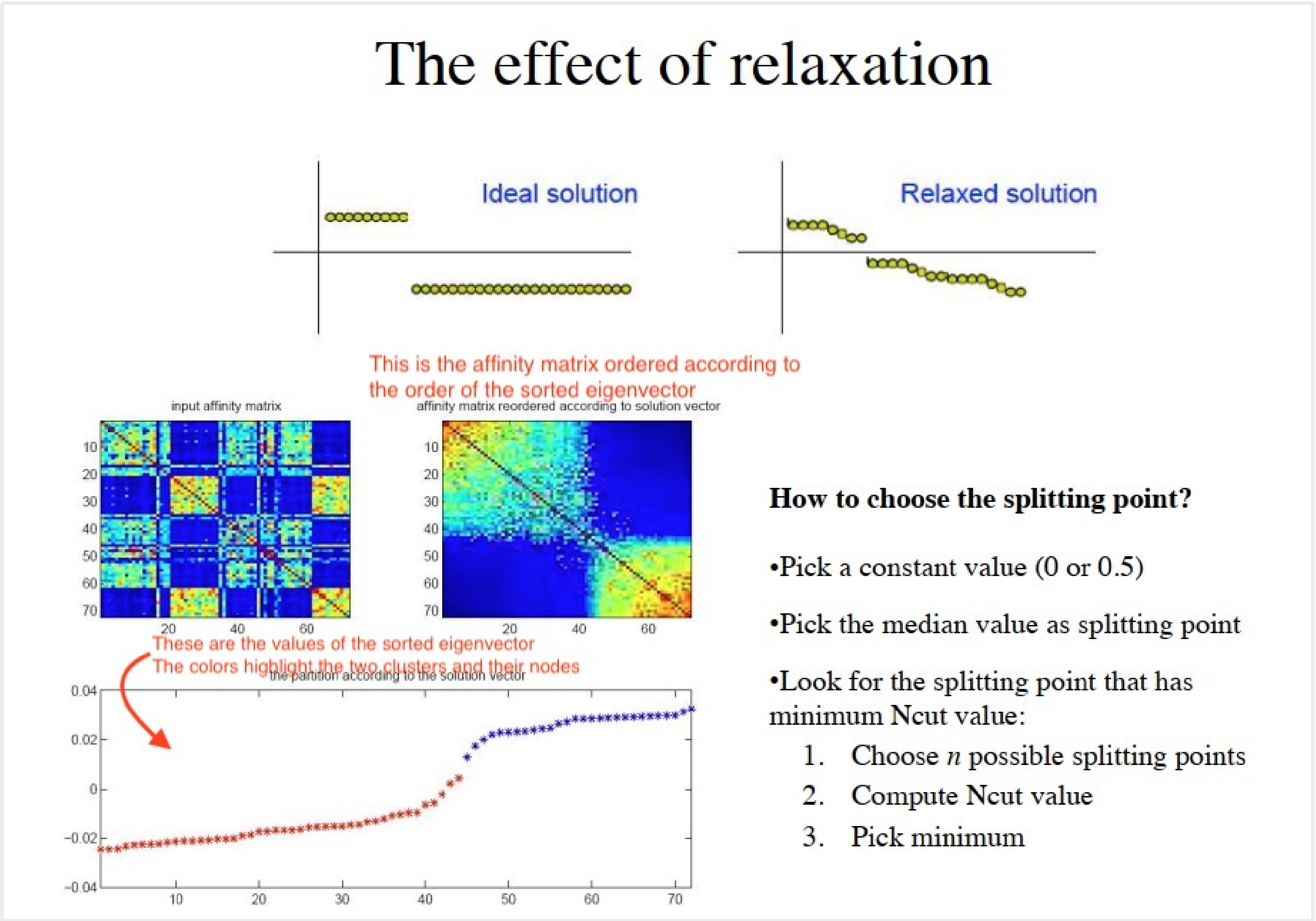
$\rightarrow D^{-1}y(D - W) = \lambda y \rightarrow y(I - D^{-1}W) = \lambda y \rightarrow yL_{rw} = \lambda y$

So in order to solve this thing we just need to compute the eigenpairs of L_{rw}

Application of Lagrange Multipliers method to the NCut Eigen-system

(Recursive) 2-Way NCut

Recall that the optimization problem is an *eigenvalue-minimization* problem, meaning that we are interested in the smallest eigenpair (the eigenvector that minimizes the NCut) to partition the graph in two parts. In particular, since we know that the smallest eigenpair (the one associated to eigenvalue $\lambda_1 = 0$) represents the trivial partition $A = V, B = \{\}$ (see Implications of 0 Being an Eigenvalue), we are interested in the ***second smallest eigenvalue*** (and associated eigenvector) to bi-partition the graph.
We can then recursively bi-partition components until some threshold is reached, for example $NCut$ exceeding a certain value (meaning that the min NCut is very close to the volume of the components involved, hence that such components are probably small)


Ideal solution
Relaxed solution
This is the affinity matrix ordered according to the order of the sorted eigenvector
input affinity matrix
affinity matrix reordered according to solution vector
These are the values of the sorted eigenvector
The colors highlight the two clusters and their nodes
the partition according to the solution vector

The effect of relaxation

How to choose the splitting point?

•Pick a constant value (0 or 0.5)

•Pick the median value as splitting point

•Look for the splitting point that has minimum Ncut value:

1. Choose *n* possible splitting points
2. Compute Ncut value
3. Pick minimum

How to 2-way NCut and the effects of relaxing the optimization problem

Pros and Cons

The main advantage of this approach is that we do not need to know the number of clusters in advance: the algorithm just keeps applying cuts until the resulting cluster configuration is deemed satisfying (some threshold is reached). It comes, however, with the disadvantage of being quite expensive, because eigenpairs need to be calculated for each partition (eigenpair decomposition is $\approx O(n^{2.4})$)

Smallest k eigenvectors and k-means

Another approach for normalized cut is that of building a matrix $U = [u_1, \dots, u_k] \in \mathbb{R}^{n \times k}$, where $u_1, \dots, u_n$ are the smallest k eigenvectors (**excluding** the eigenvector whose eigenvalue is 0) of the (unnormalized) Laplacian L.
U can be rewritten as

$$U = \begin{pmatrix} u_{11} & \dots & u_{1k} \\ \vdots & \ddots & \vdots \\ u_{n1} & \dots & u_{nk} \end{pmatrix} = \begin{pmatrix} y_1^T \\ \vdots \\ y_n^T \end{pmatrix}$$

where $y_i = U_i$ is the i-th row of the matrix U and can be thought of as a new datapoint in a reduced k-dimensional space.
Finally, such points are then clustered via k-means.

Pros and Cons

The main advantage of this approach is that we just have to compute eigenpairs once, making it less expensive, coming with the added cost of having an hyper-parameter k that defines the number of clusters, as in k-means.

But How Does Partitioning Actually Work?

There are two different ways to split a graph based on some eigenvector x:

- define a threshold k (usually 0, 0.5 or the median value in x) and assign node i to either of the two cluster by checking the truth of $x_i > k$ (if true one cluster, if false the other)
- choose n candidate splitting points in the x and check the one whose $NCut$ is minimum

Spectral Clustering

In this report sklearn's *SpectralClustering* class is used to cluster data via the implementation of Normalized-Cut. More specifically, spectral clustering is based on the second approach of normalized cut (see Smallest k eigenvectors and k-means), with two main differences:

1. The eigenvectors used as basis get normalized
2. The Laplacian matrix used is not in the form

   $$L_{rw} = D^{-1}L = D^{-1}(D - W) = I - D^{-1}W$$

   but in this form:

   $$L_{sym} = D^{-\frac{1}{2}}LD^{-\frac{1}{2}} = I - D^{-\frac{1}{2}}WD^{-\frac{1}{2}}$$

"assets/spectral-clustering-pseudocode.jpg" is not created yet. Click to create.

Mean-Shift Clustering

Consider a set of points in two-dimensional space. Assume a circular window centered at C and having kernel k_h of radius h. *Mean-shift is a hill climbing algorithm* which involves shifting this kernel iteratively to a higher density region until convergence.
Every shift is defined by a mean shift vector $m(x)$. The mean shift vector always points toward the direction of the maximum increase in the density. At every iteration the kernel k_h is shifted to the centroid or the mean of the points within it (i.e. the points whose distance from the center is $\leq h$).
The method of calculating this mean depends on the choice of the kernel. If a Gaussian kernel is chosen instead of a flat kernel, then every point will first be assigned a weight which will decay exponentially as the distance from the kernel's center increases (as according to the Gaussian kernel's profile).
At convergence, there will be no direction at which a shift can accommodate more points inside the kernel.

Overview

Mean shift is a procedure for locating the maxima (the modes) of a density function given discrete data sampled from that function. This is an iterative method, and we start with an initial estimate x. Let a kernel function $K(x_i - x)$ be given. This function determines the weight of nearby points for re-estimation of the mean.
Typically, a Gaussian kernel on the distance to the current estimate is used, $K(x_i - x) = e^{-c\|x_i - x\|^2}$, where c is the scaling factor (bandwidth) of the kernel, usually a function of the standard deviation σ.
The weighted mean of the density in the window determined by

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

where $N(x)$ is the neighborhood of x, a set of points for which $K(x_i - x) \neq 0$.

Important

Let x be the estimated centroid of the cluster. The difference $m(x) - x$ is called *mean shift*. At each iteration, the mean-shift algorithm sets $x \leftarrow m(x)$, and

repeats the estimation until $m(x)$ converges.
(Gaussian) Mean-Shift is an Expectation-Maximization Algorithm, meaning that it is iterative and only locally optimal in nature, also being relatively sensitive to the initialization state.

Kernel Smoother

This method relies on the application of a Gaussian kernel smoother to calculate the mean point $m(x)$ of the neighborhood of bandwidth h.

Implementation and Optimizations

The EM steps of the algorithm are ideally executed for each point in the dataset. Although being highly parallelizable, it is a lot of computational work, meaning that some optimization could be used: typically, once some point converges to a density peak, all the other points within a radius k from it are assigned to the same cluster and their associated computation is stopped.

Unsupervised Learning Extrinsic Evaluation

The methods of evaluation of unsupervised machine learning models are mainly two:

- **Intrinsic/Internal validation**: does not have any kind of ground truth, meaning that it relies on the concept of intra-class and inter-class dissimilarity to measure the goodness of predictions (clusters).
- **Extrinsic/External validation**: relies on some kind of ground truth, for example the correct labelling for the dataset.

For the purposes of this assignment, we will focus on the latter method, *extrinsic evaluation*, particularly on the *Rand Index*.

Definition

Let $C(o_i)$ be the cluster id provided by the clustering algorithm for object o_i
Assume we have $L(o_i)$ as ground truth for object o_i. i.e. its true label.

We can build the following contingency table (*confusion matrix*):

	Different Cluster	**Same Cluster**
Same Label	#True Positives	#False Negatives
Different Label	#False Positives	#True Negatives

Where:

- True Positives $TP = \mid \{o_i, o_j : C(o_i) = C(o_j) \wedge L(o_i) = L(o_j)\} \mid$
- False Positives $FP = \mid \{o_i, o_j : C(o_i) = C(o_j) \wedge L(o_i) \neq L(o_j)\} \mid$
- True Negatives $TN = \mid \{o_i, o_j : C(o_i) \neq C(o_j) \wedge L(o_i) \neq L(o_j)\} \mid$
- False Negatives $FN = \mid \{o_i, o_j : C(o_i) \neq C(o_j) \wedge L(o_i) = L(o_j)\} \mid$

Intuition

Intuitively, TP, TN are the number of agreements between the predictions and the ground truth, while FP, FN are the number of disagreements.

Rand Index (Rand Statistic)

The *Rand Index* is defined as follows:

$$RI = \frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{\binom{n}{2}}$$

Such a statistic is a number $\in [0, 1]$ that represents the ratio of correct decisions over the total number of decisions made. The algorithm performed best when $RI = 1$.

Implementation Details

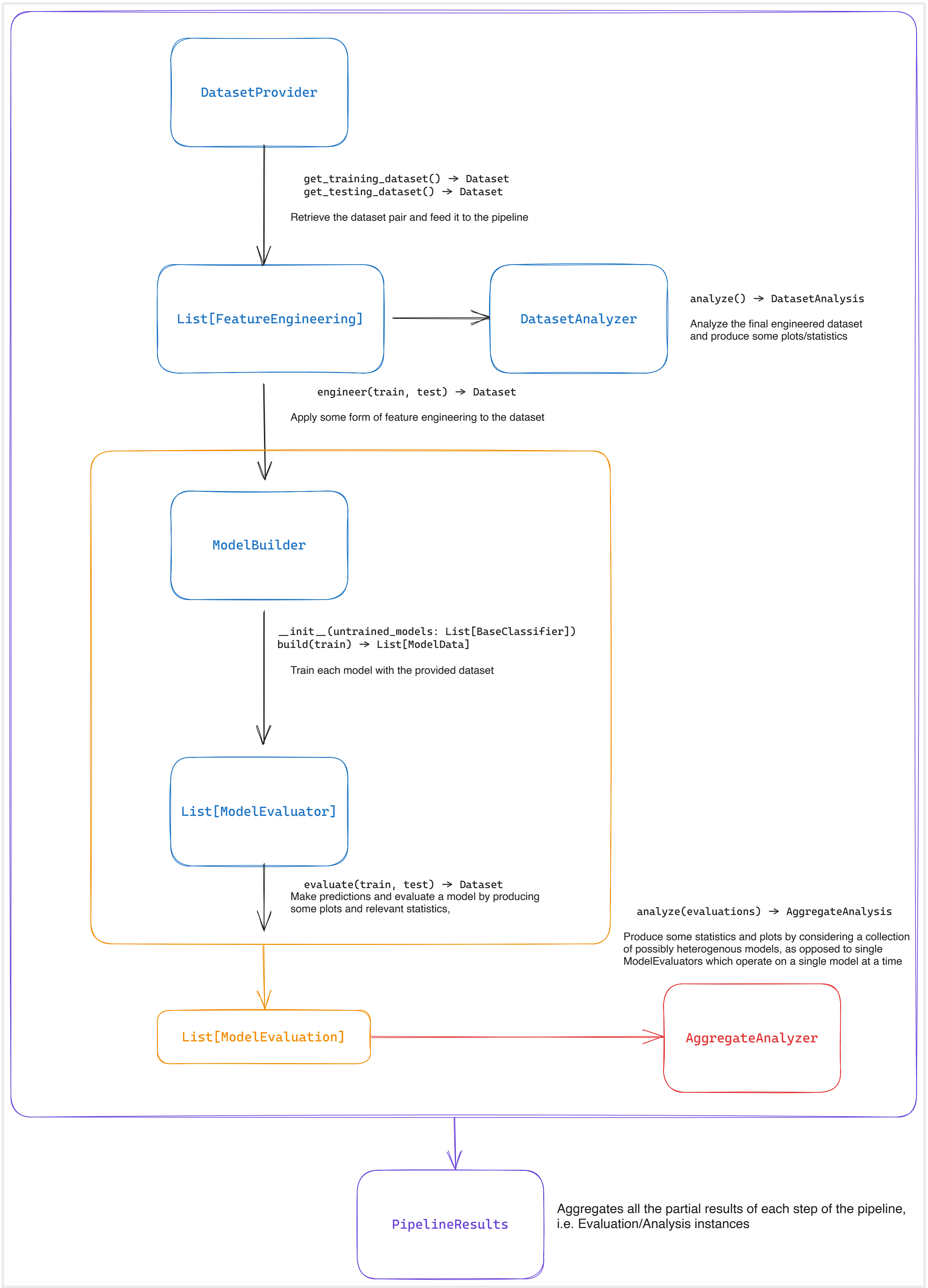
The project consists of two main *macro* modules:

- `notebooks`, which contains the only notebook used to carry out this assignment's analysis, `main.ipynb`
- `src`, containing the data structure and utility implementations used to implement the aforementioned analysis

The module `src` itself has three main sub-modules:

- `classifiers`, containing the data structures that wrap sklearn's models
- `visualization`, containing visualization functions and related utils
- `pipeline`, containing the components used to process the dataset, build and evaluate the models

As regards to the latter module, `pipeline`, the processing pipeline that it represents is described at a high level by the image below, which cites all of the main abstractions defined by the module.


DatasetProvider
get_training_dataset() → Dataset
get_testing_dataset() → Dataset
Retrieve the dataset pair and feed it to the pipeline
List[FeatureEngineering]
DatasetAnalyzer
analyze() → DatasetAnalysis
Analyze the final engineered dataset and produce some plots/statistics
engineer(train, test) → Dataset
Apply some form of feature engineering to the dataset
ModelBuilder
__init__(untrained_models: List[BaseClassifier])
build(train) → List[ModelData]
Train each model with the provided dataset
List[ModelEvaluator]
evaluate(train, test) → Dataset
Make predictions and evaluate a model by producing some plots and relevant statistics,
analyze(evaluations) → AggregateAnalysis
Produce some statistics and plots by considering a collection of possibly heterogenous models, as opposed to single ModelEvaluators which operate on a single model at a time
List[ModelEvaluation]
AggregateAnalyzer
PipelineResults
Aggregates all the partial results of each step of the pipeline, i.e. Evaluation/Analysis instances

Pipeline Architecture - A High Level Overview

Finally, the notebook `main.ipynb` leverages the pipeline as well as all the various analyzer implementations to process the data, train the models and produce some interesting statistics and plots.

Experimental Results

Setup

Dataset

The tests were performed on a stratified 10% sample of the original dataset, which equals to roughly 7k samples, 25% of which were used as testing data.
A smaller portion of the dataset was used in order to make the computation feasible and provide a relatively quick feedback cycle.

Other than the PCA required by the assignment, the only feature engineering techniques applied to the dataset were:

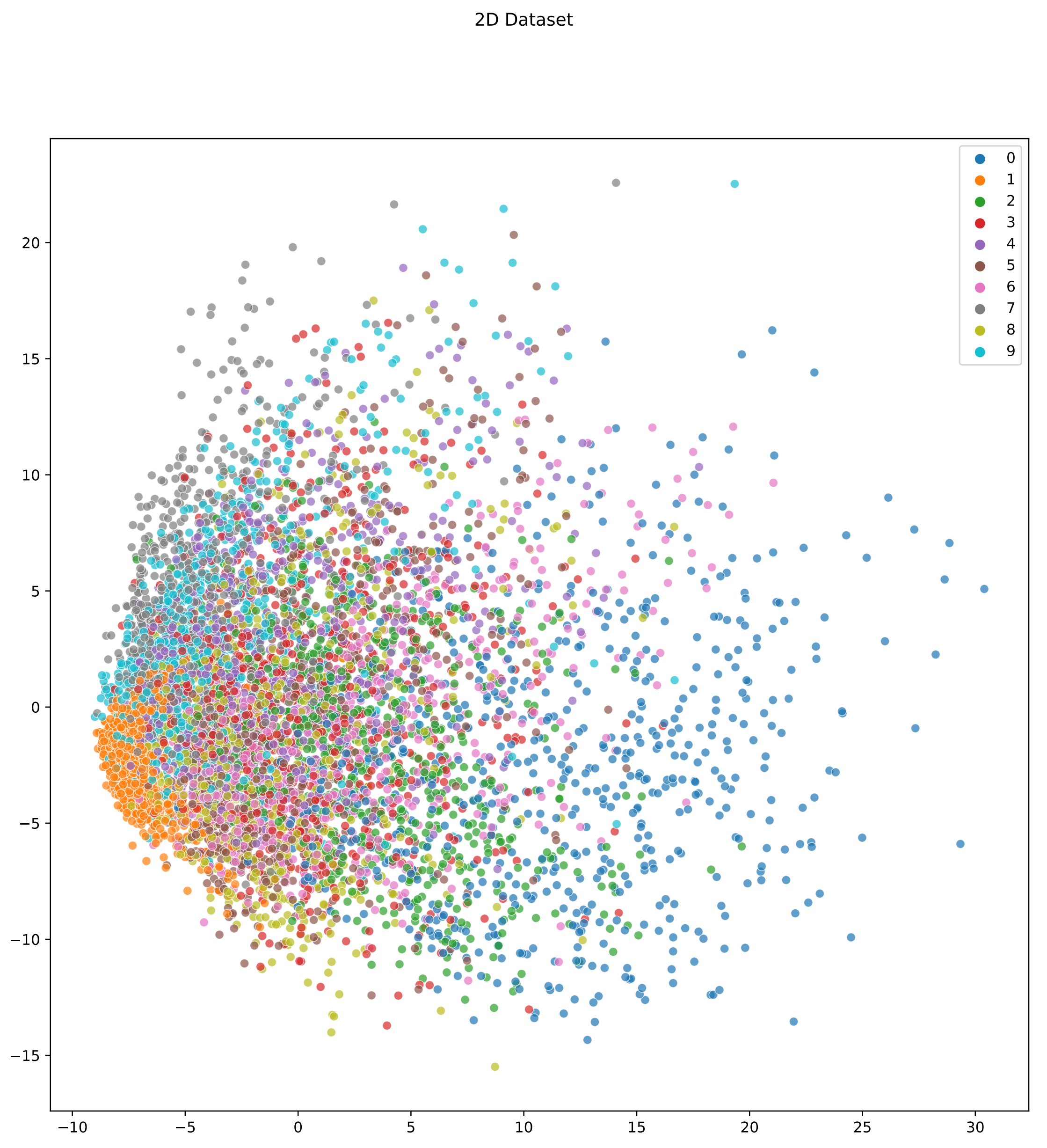
- The scaling of the data into the $[0, 1]$ interval, achieved by dividing each sample by 255
- The standardization of the data, very useful to make PCA more effective because it eliminates difference in magnitude and variance between features.

Hyperparameters

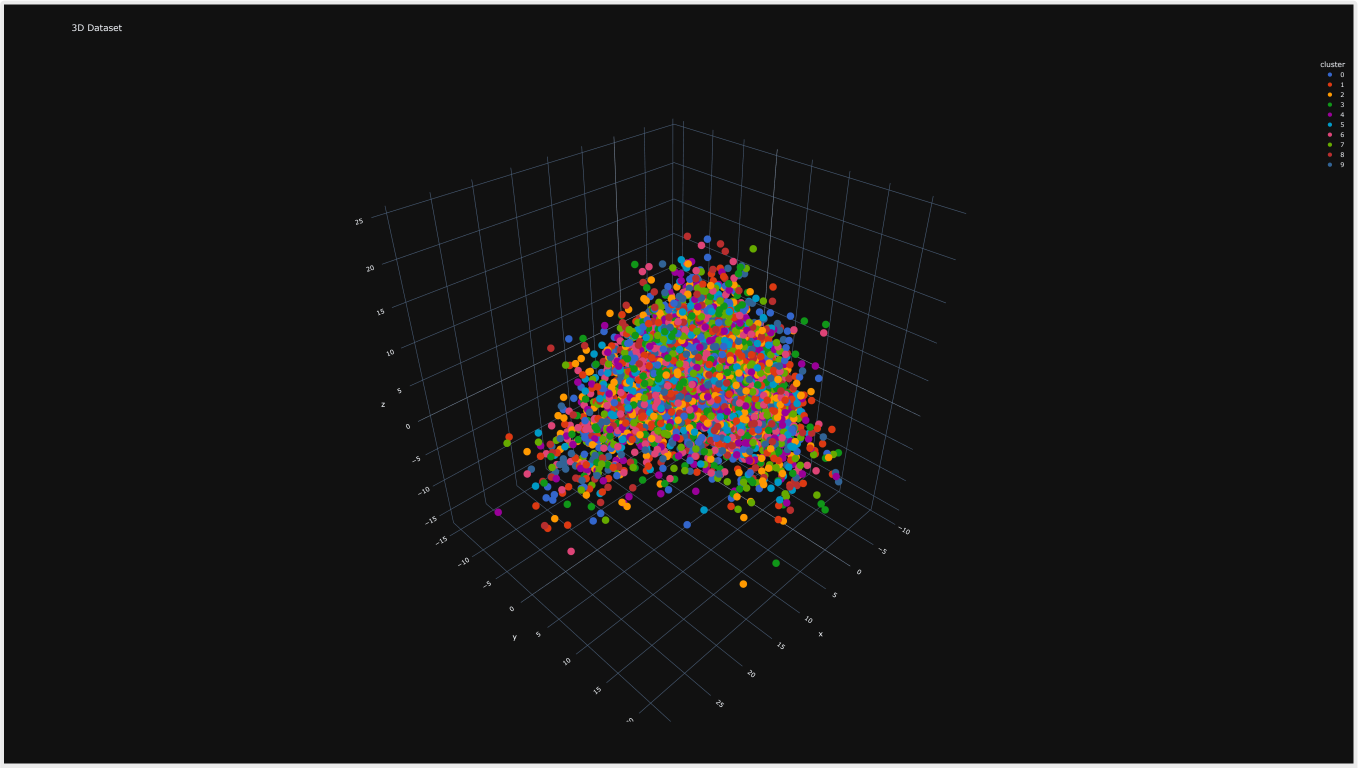
The set of hyperparameters used to complete the assignment are the following:

- GMM and NCut's `n_components`: $[5, 7, 10, 13, 15]$
- MeanShift flat kernel's `bandwidth`: $[1, 5, 15, 30, 50]$
- PCA dimensions: $[2, 10, 25, 50, 100, 150, 200]$

Dataset Exploration

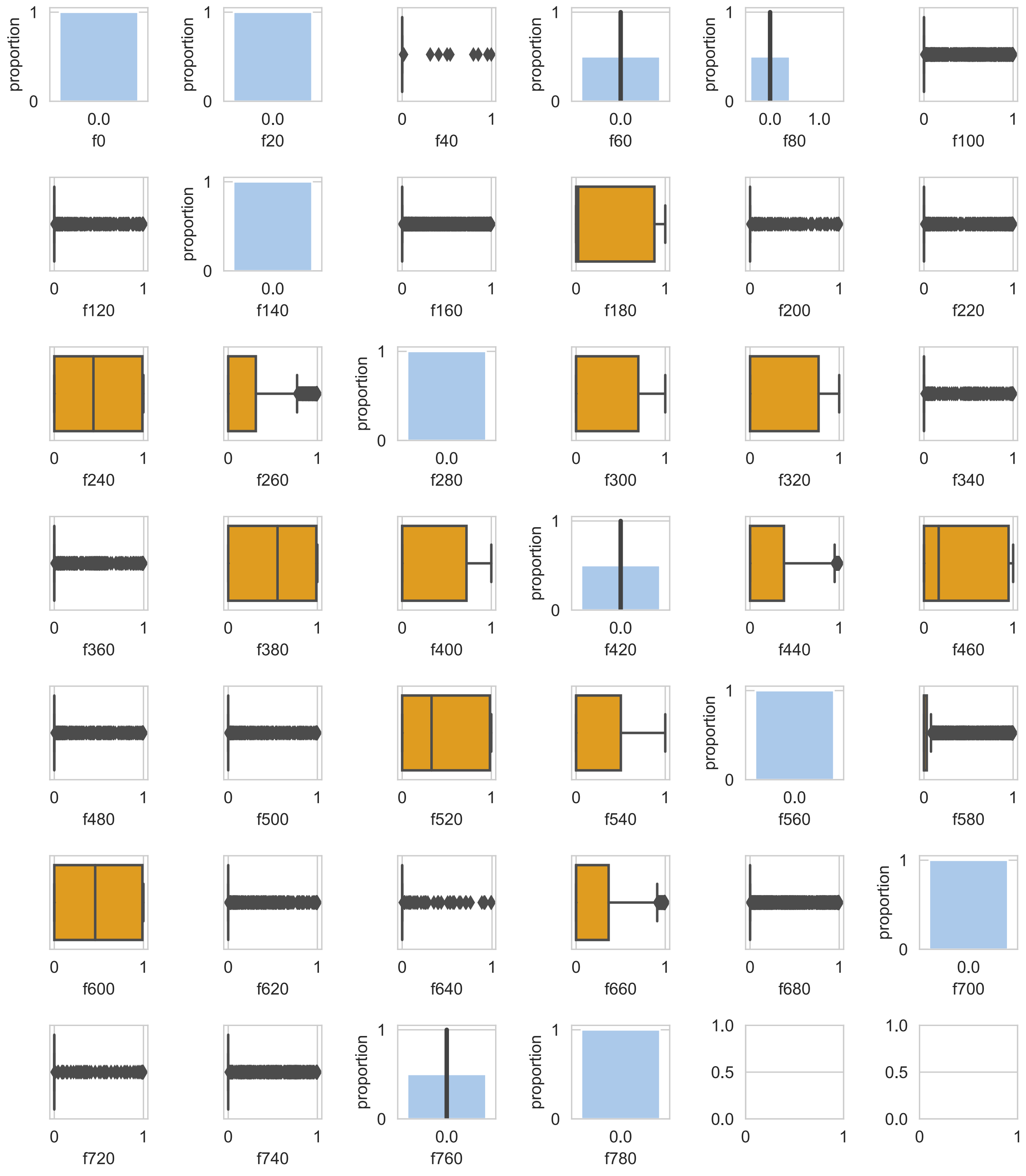


2D Representation of the dataset after Standardization and PCA

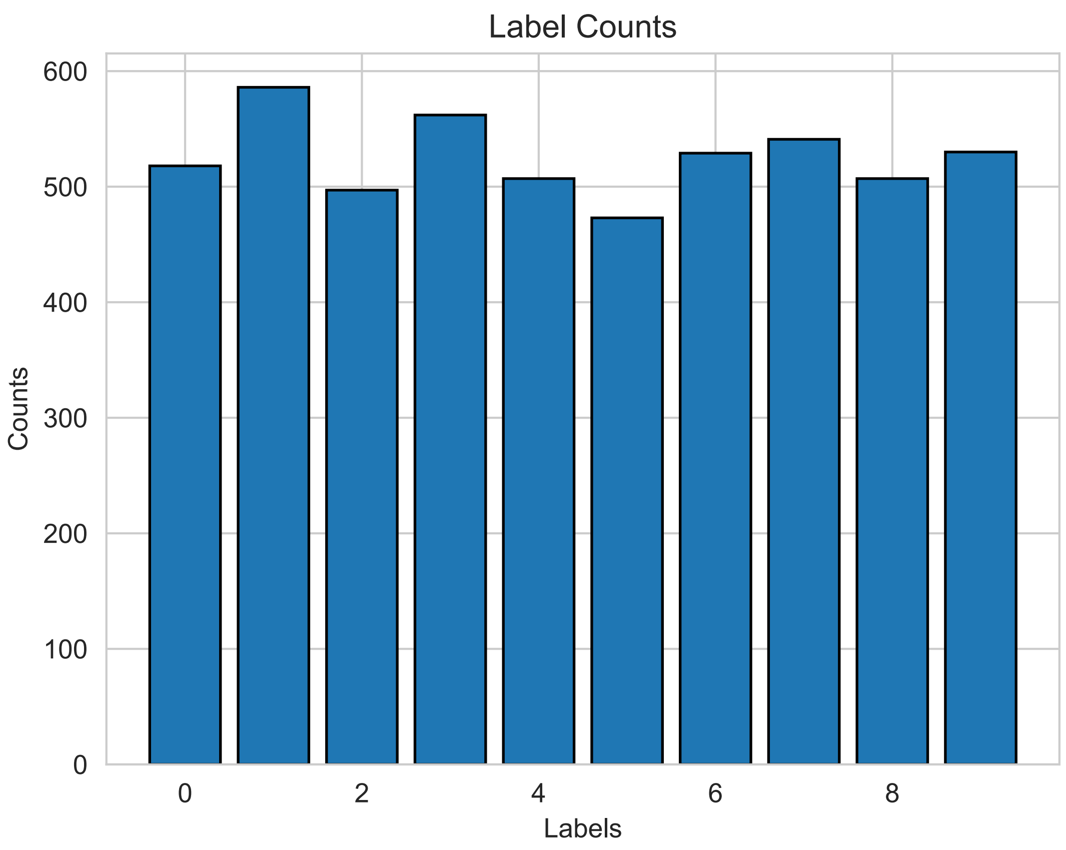


3D Representation of the dataset after Standardization and PCA

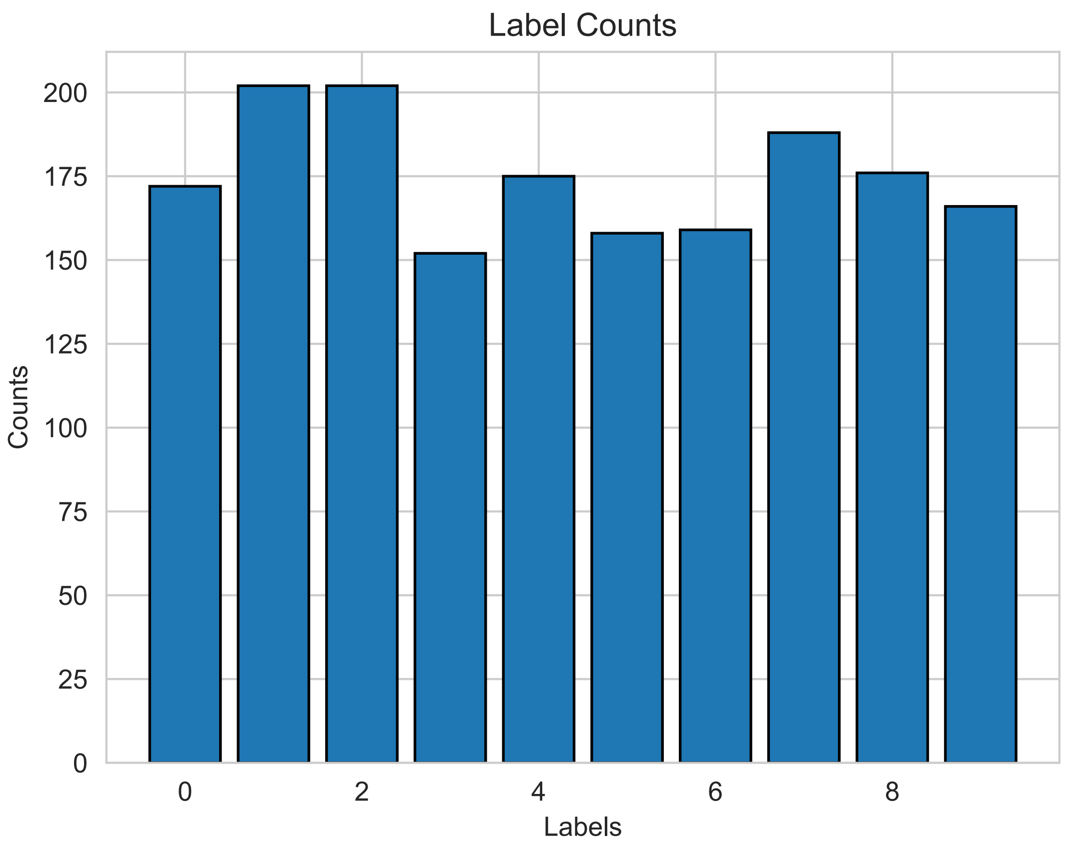
mnist_784_train (MNIST Digit Dataset)



Distribution of a sample of features from the original (but scaled) dataset



Training Dataset Label Counts



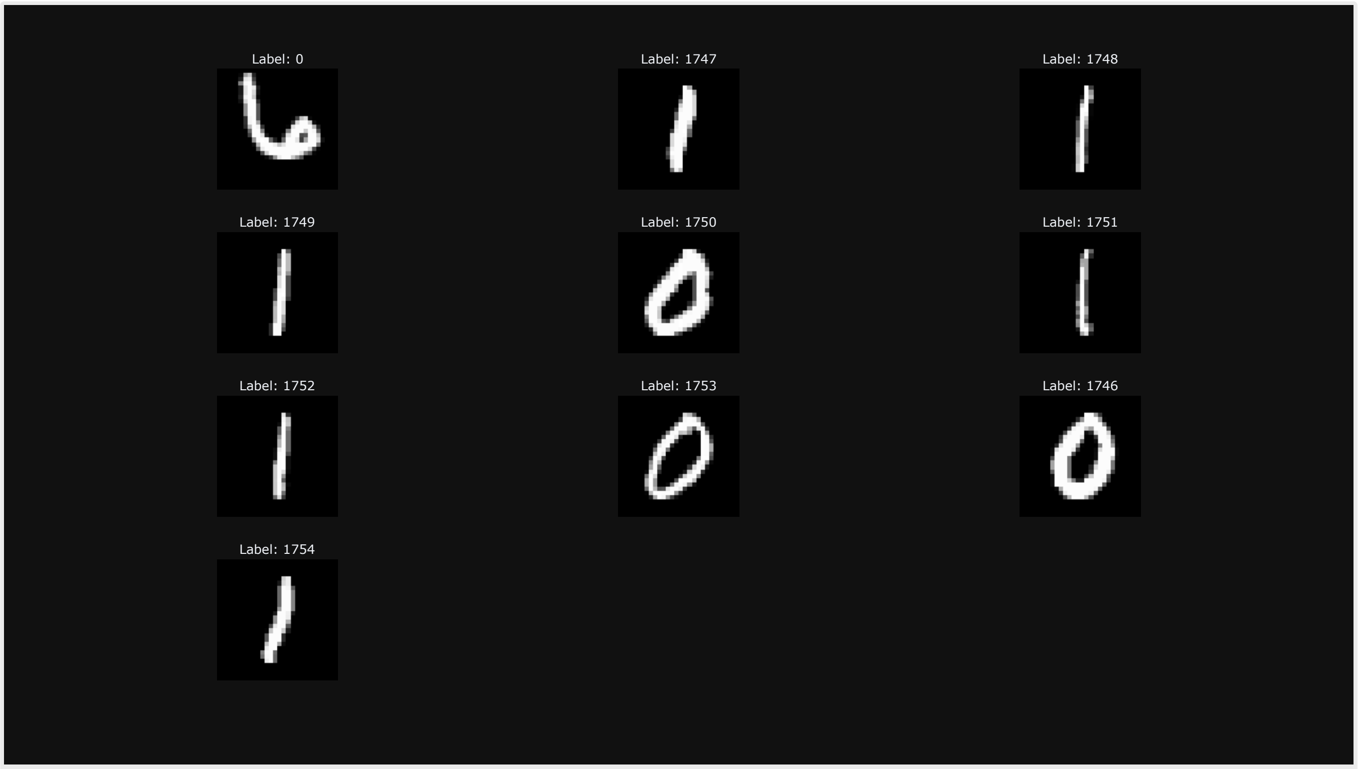
Testing Dataset Label Counts

Mean Shift and GMM Internals

The following plots represent the mean internal representation of a cluster for the MeanShift and GMM models. Such plots were achieved by respectively looking at the cluster centroids and at the means of each mixture, which, in the full $\mathbb{R}^{784}$ dataset, can be interpreted as the mean image for the cluster.
The title of each subplot is the label assigned by the model to the cluster mean.

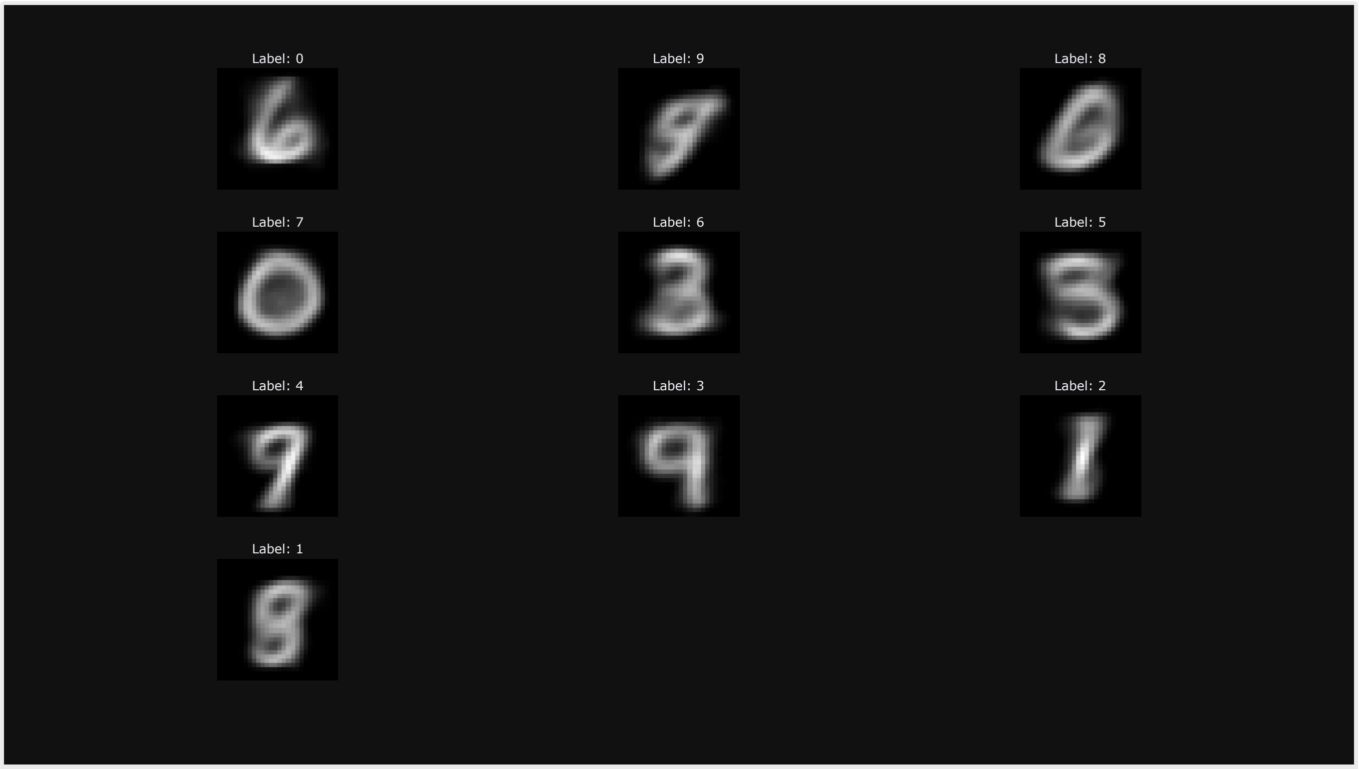
One thing to note in both plots is that labels assigned by the model are integers that do not correspond to the actual digits represented by the image and, consequently, neither to their true labels.

Mean Shift



Mean Shift (bandwidth equal to 1) Cluster Centroid Representation (Top 10 clusters by cardinality)

GMM



GMM (with 10 mixtures) Cluster Mean Representation

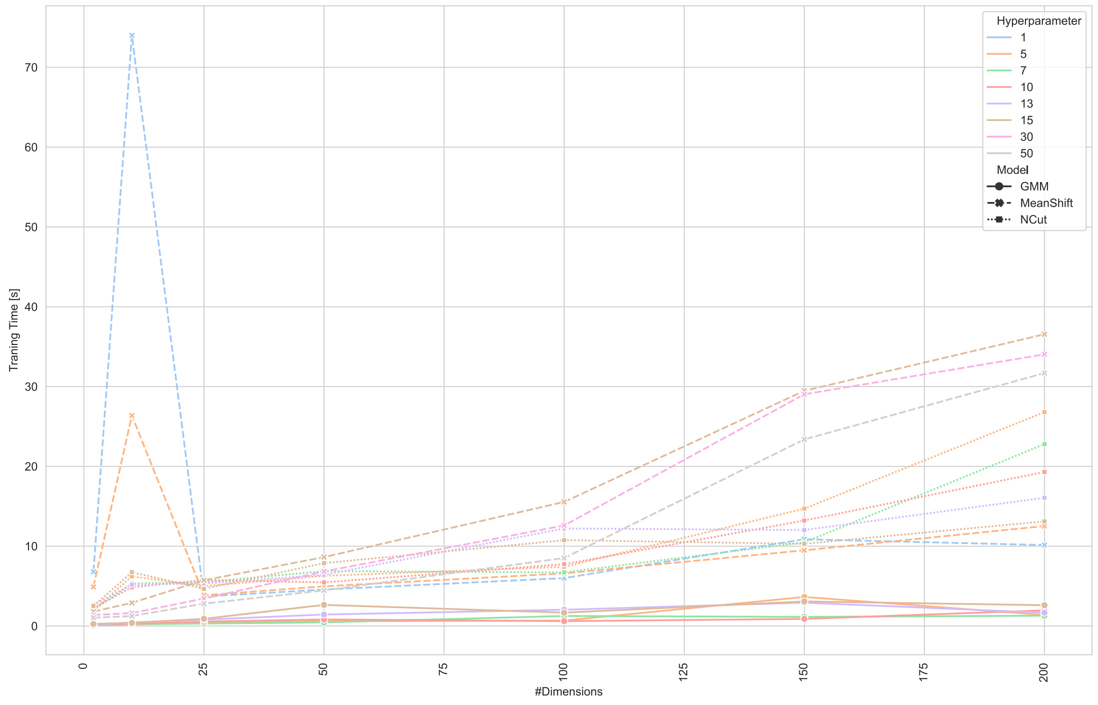
Training Times Comparison

Below are the plots for the training times of each pair $(model, hyperparameter)$ over the number of PCA dimensions.
It can be noted that Normalized Cut and Mean Shift training times are vastly superior to those of the GMM model. This is mostly because:

- in the case of Normalized Cut, the eigenpair calculation needed to solve the previously described optimization problem, which is $\sim O(n^3)$ in time complexity w.r.t. to the cardinality of the dataset
- in the case of Mean Shift, according to sklearn's MeanShift notes section the time complexity tends towards $O(Tn^2)$, where "T is the number of points and n is the number of samples", because the model is implemented by using a flat kernel and a Ball Tree to look up members of each kernel

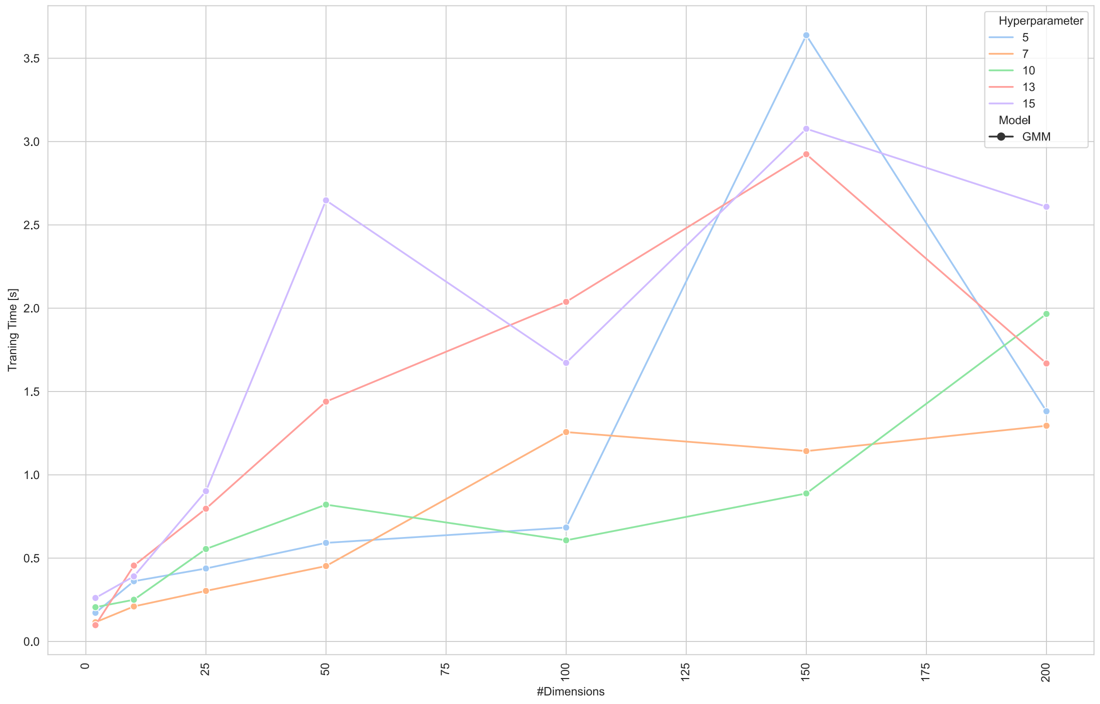
GMM, instead, scales linearly with the number of samples, a difference which is clearly highlighted by the chart.

All Models



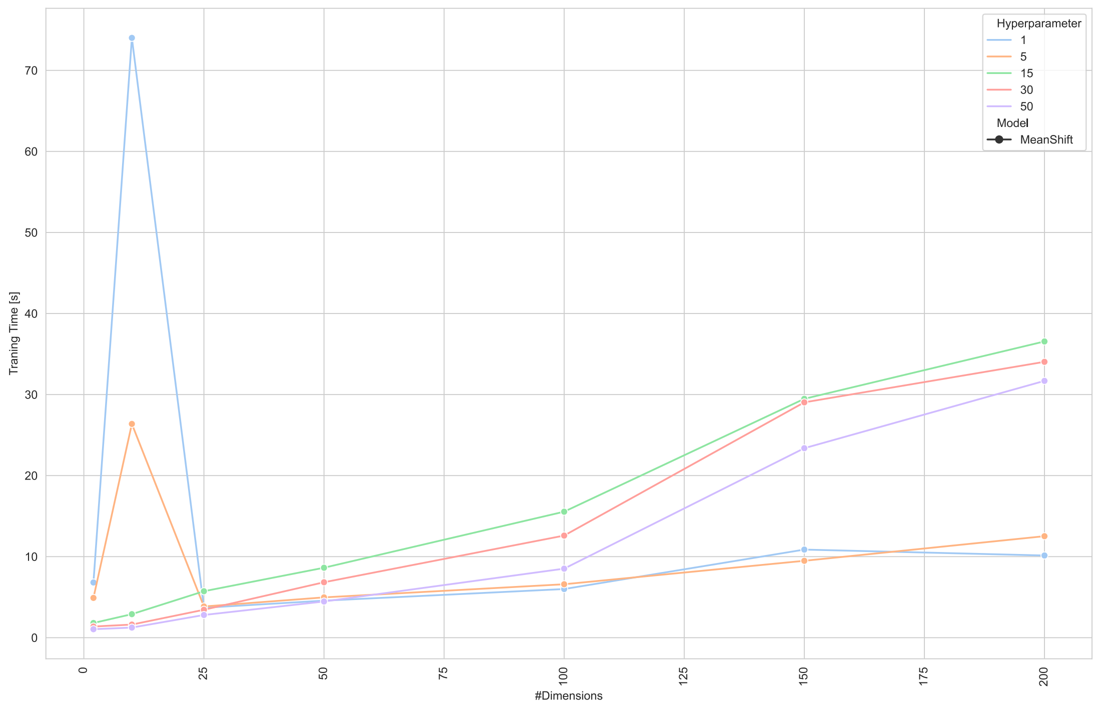
All Models - Traning Times

GMM



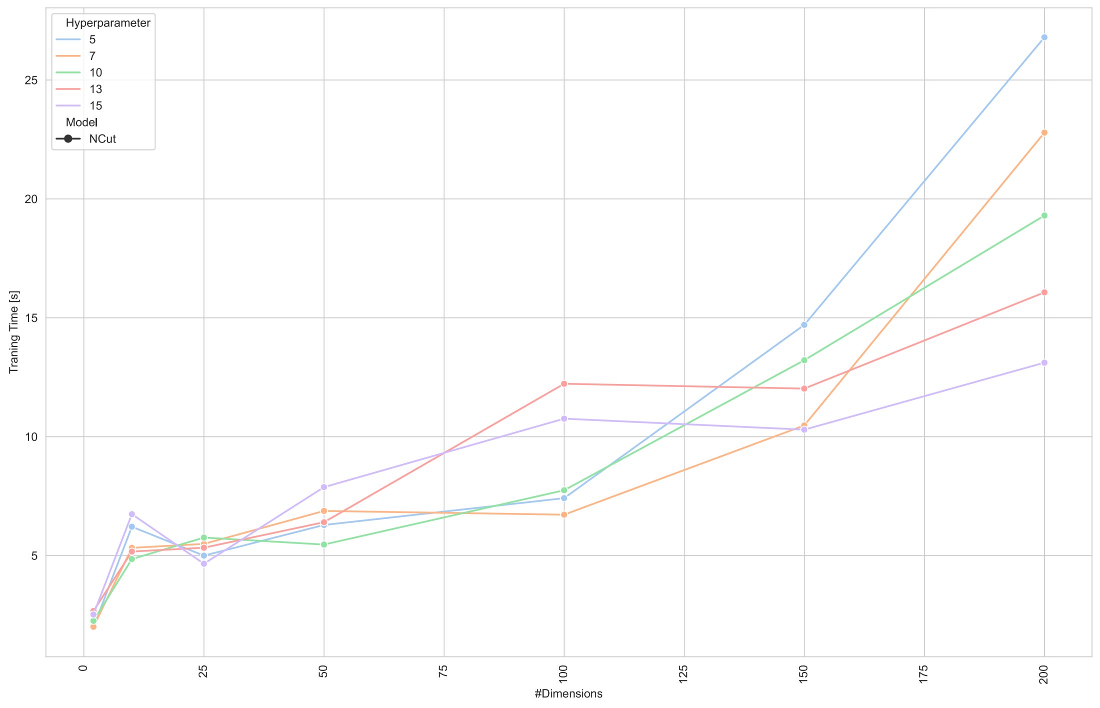
GMM Training Times - A positive correlation between training time and number of features can be observed; indeed, GMM time complexity is cubic w.r.t. the number of features in the data

Mean Shift



Mean Shift Training Times - The spike in some of the lowest dimensional datasets might be due to difficulties in reaching convergence, because of the too narrow bandwidths

Normalized Cut

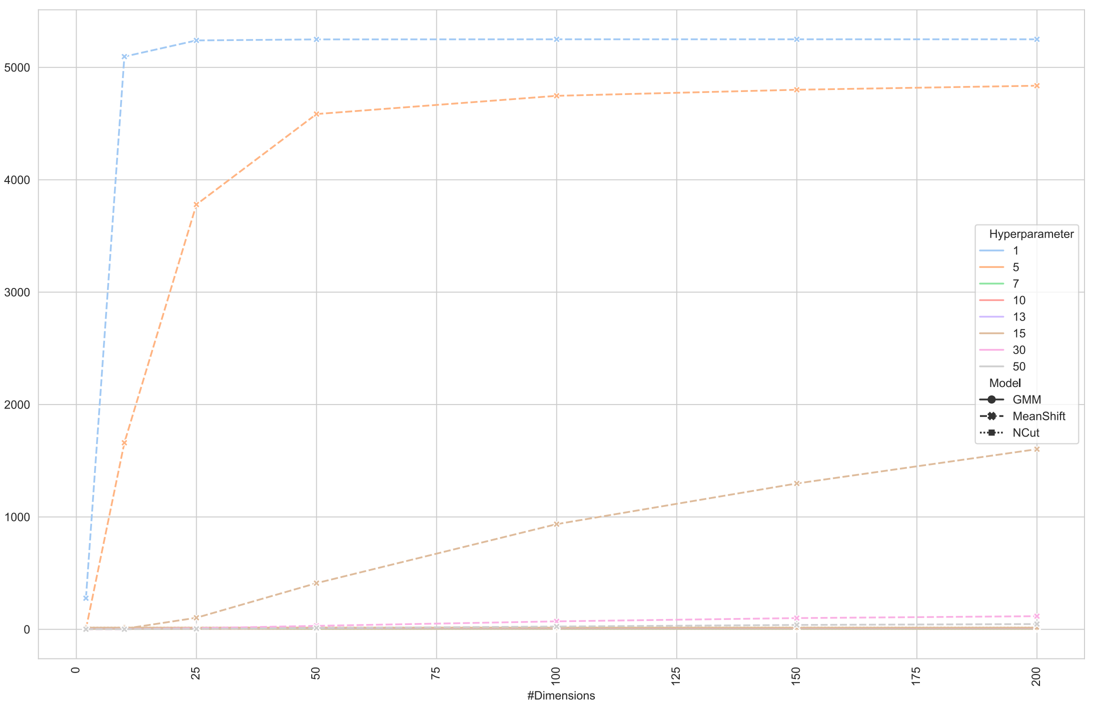


Normalized Cut Training Times

Number of Clusters Comparisons

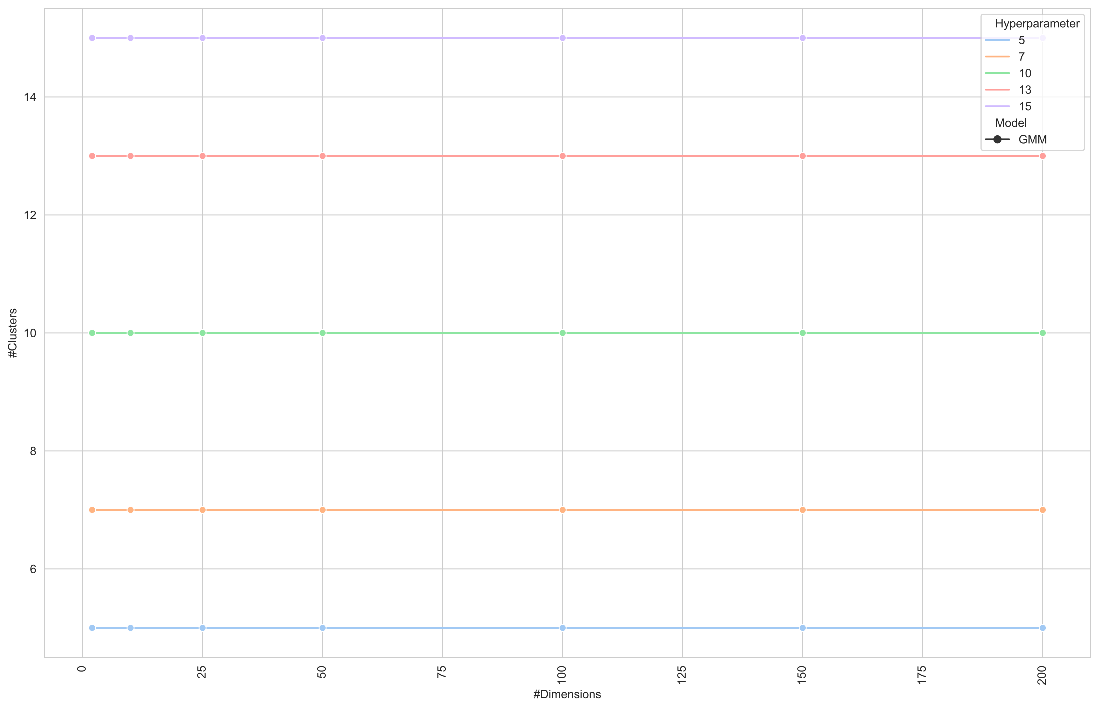
As expected, the number of clusters identified by the GMM and NCut models is constant w.r.t. the hyperparameter (which was, in fact, the number of clusters).
As regards to Mean Shift, the charts highlight very well the relationship between bandwidth and number of clusters: the narrower the bandwidth, the more clusters identified, up to the extreme case of having 1 cluster per training sample (which was probably achieved by the lowest setting of bandwidth).

All Models



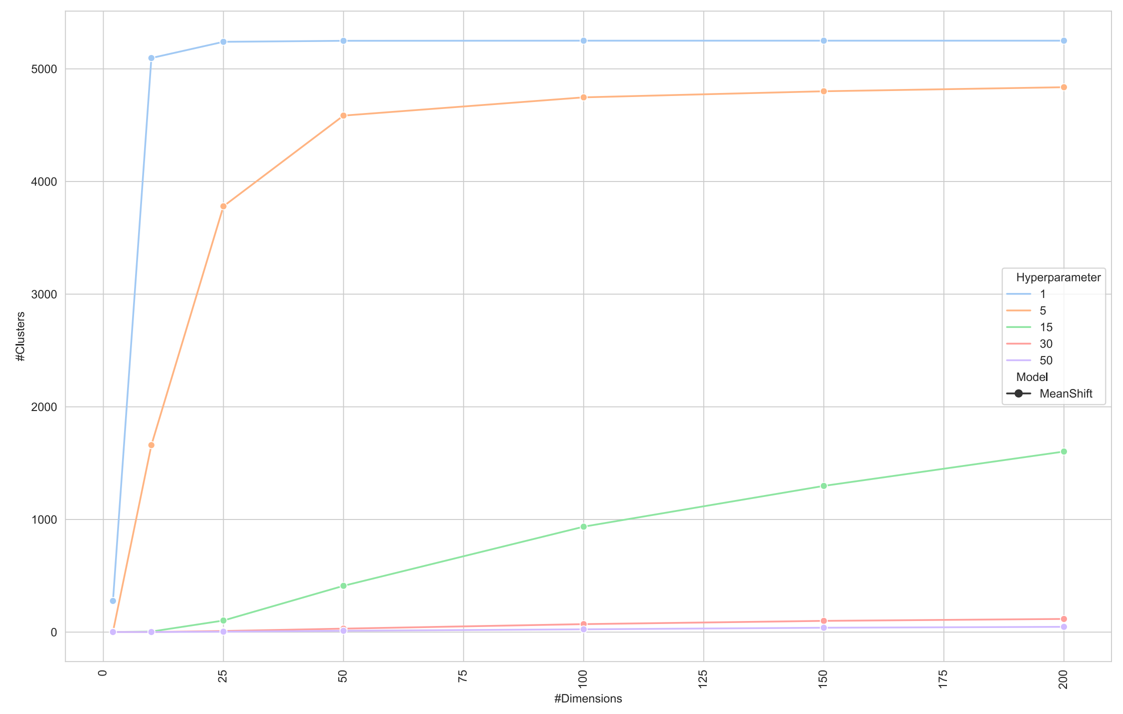
All Models - Number of Clusters

GMM



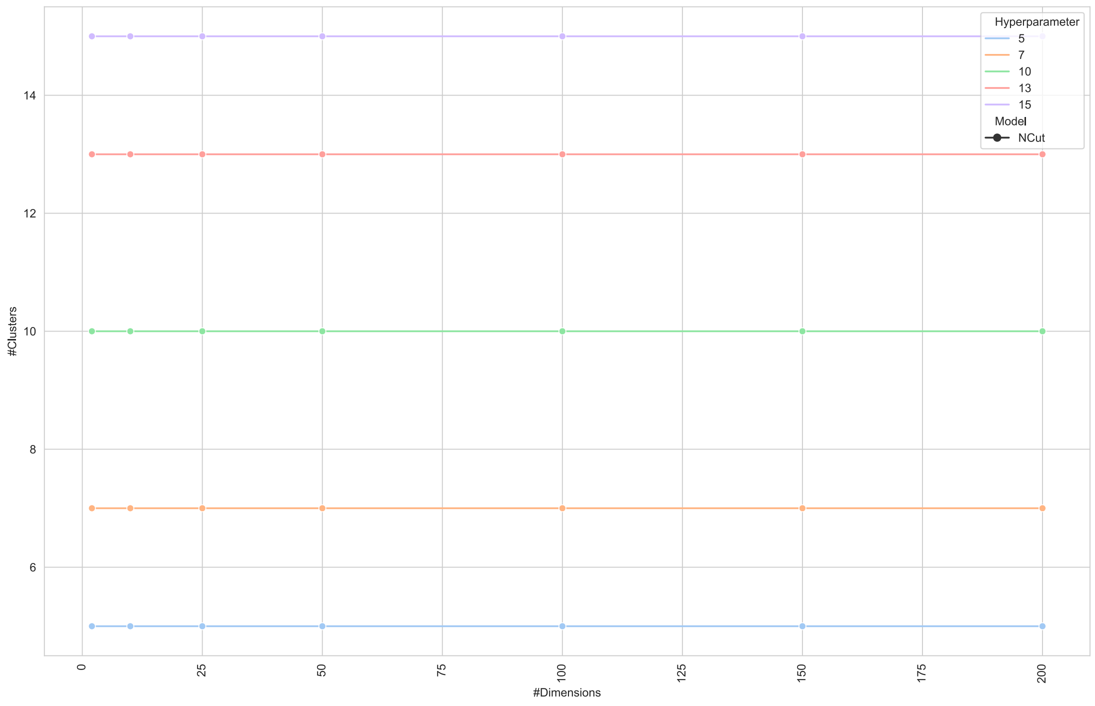
GMM Number of Clusters

Mean Shift



Mean Shift Number of Clusters

Normalized Cut



Normalized Cut Number of Clusters

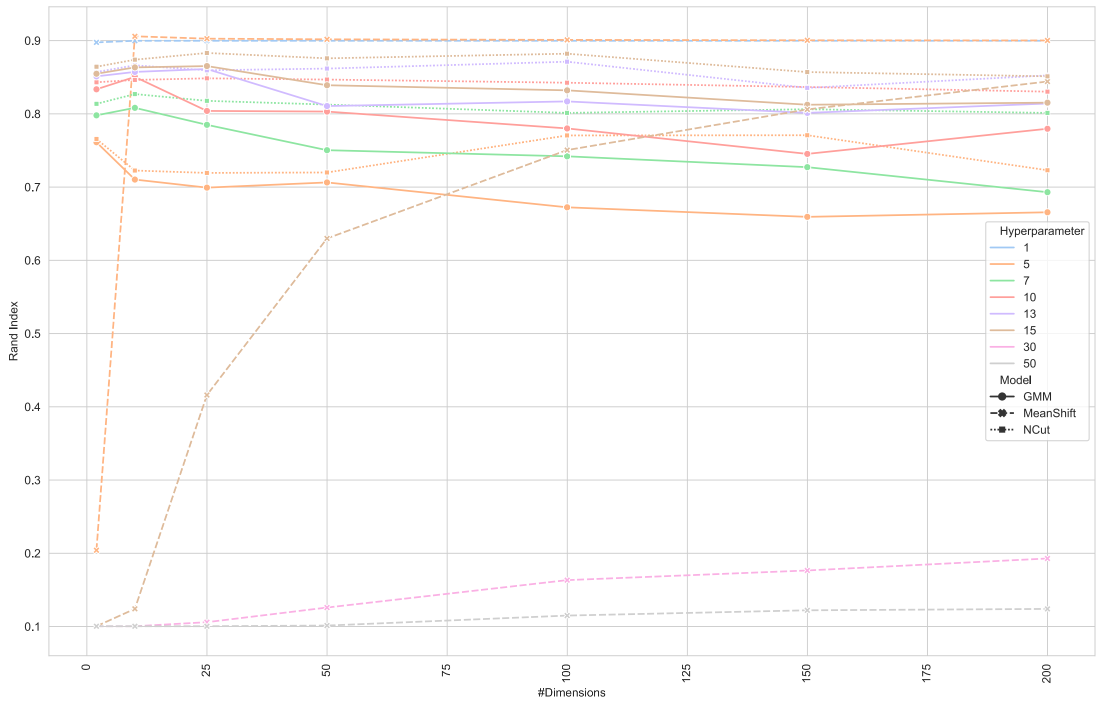
Rand Index Comparisons

The plots of the cluster means for the best and worst performing models, which are shown in the following section, should give more context to this chart.

What is clear, however, is that a higher number of predicted generally results in a better Rand Index score. There are two main intuitions behind this:

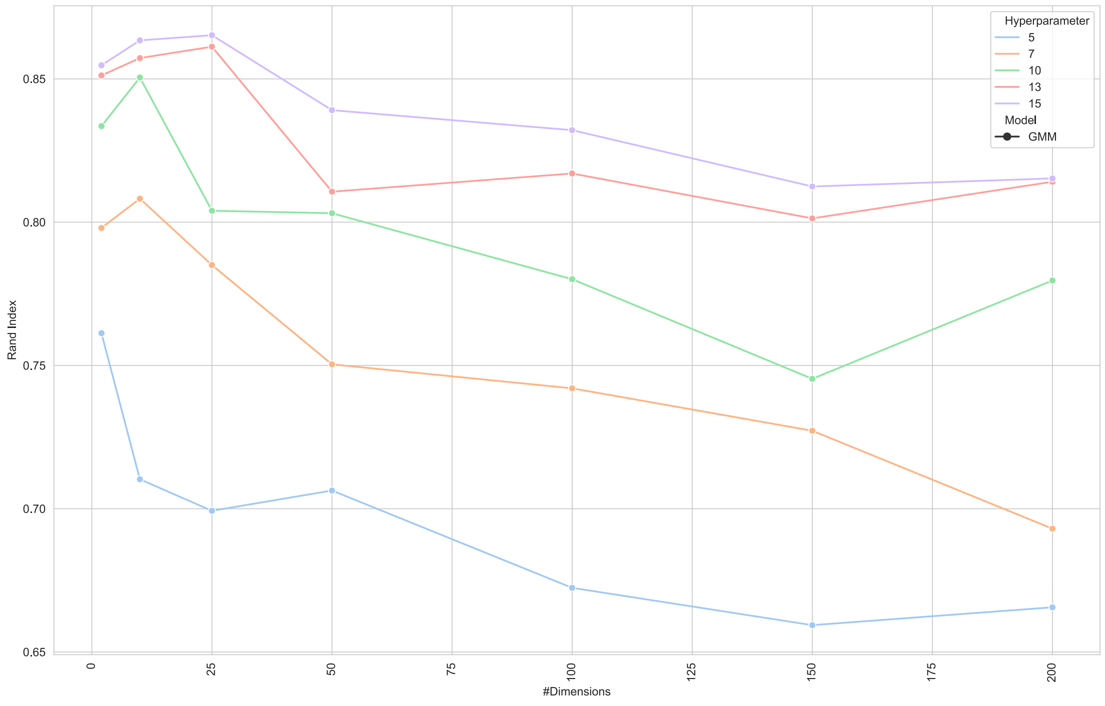
1. The true labels and the predicted labels do not have the same meaning
2. With a sufficiently high number of clusters and points, it is easier for the model to correctly classify *true negatives*, which also tend to have a higher weight in the Rand Index computation, pushing the numerator and hence the final score up.

All Models



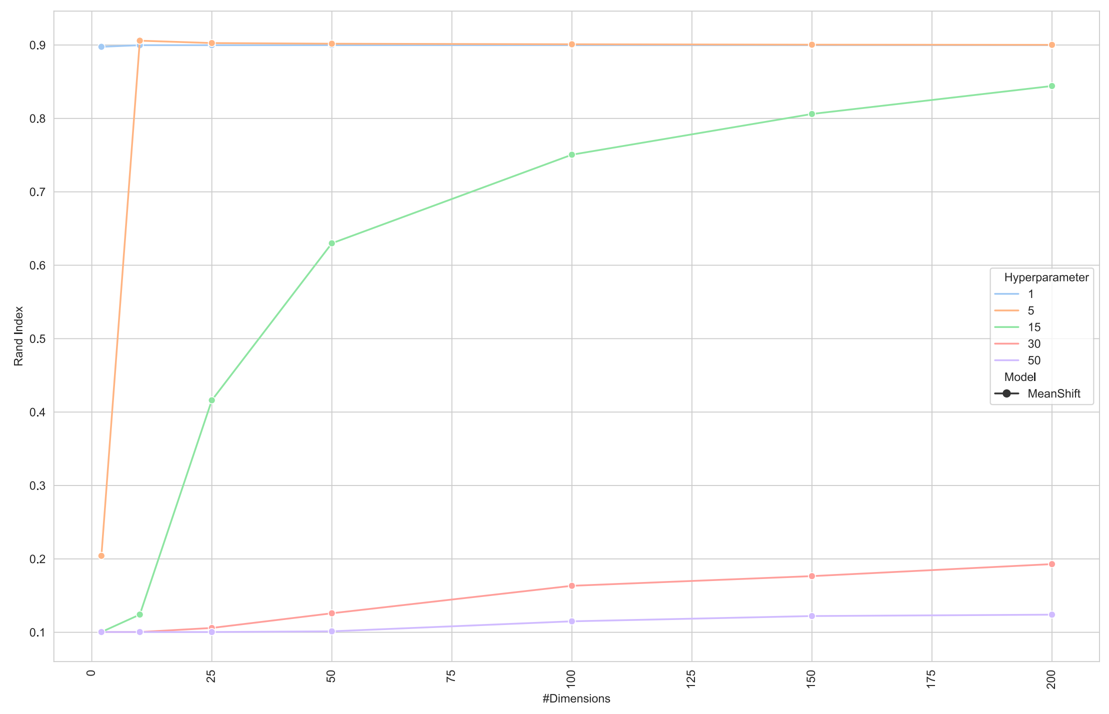
All Models - Rand Index Scores

GMM



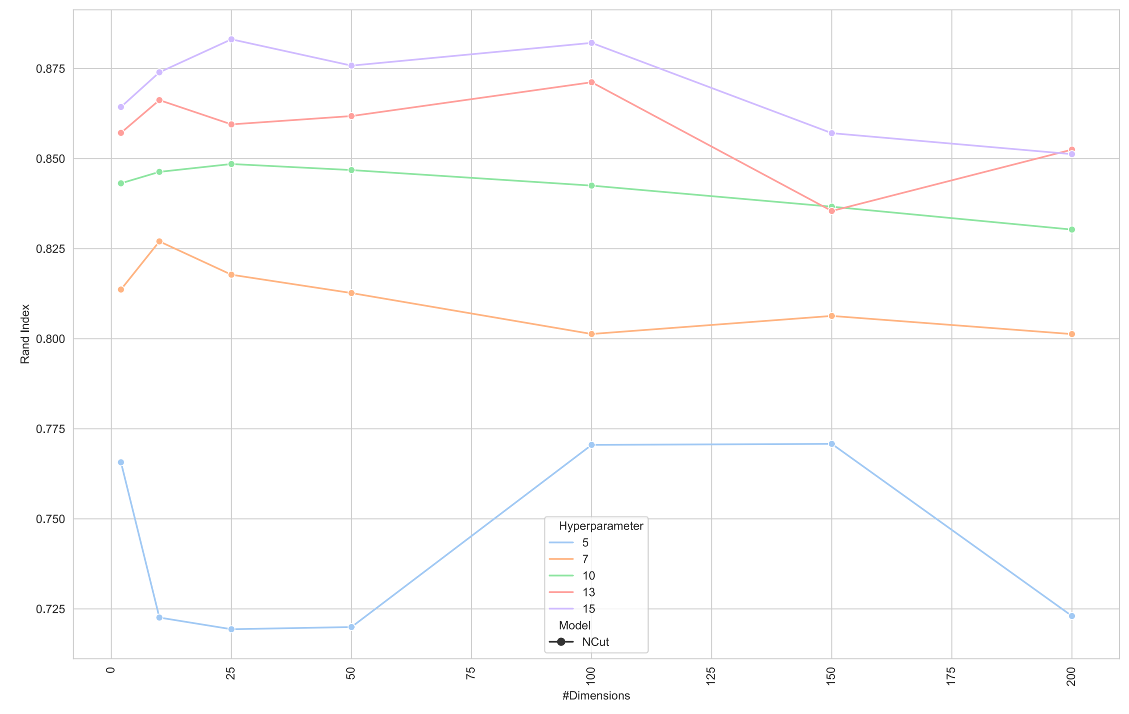
GMM Rand Index Scores

Mean Shift



Mean Shift Rand Index Scores

Normalized Cut



Normalized Cut Rand Index Scores

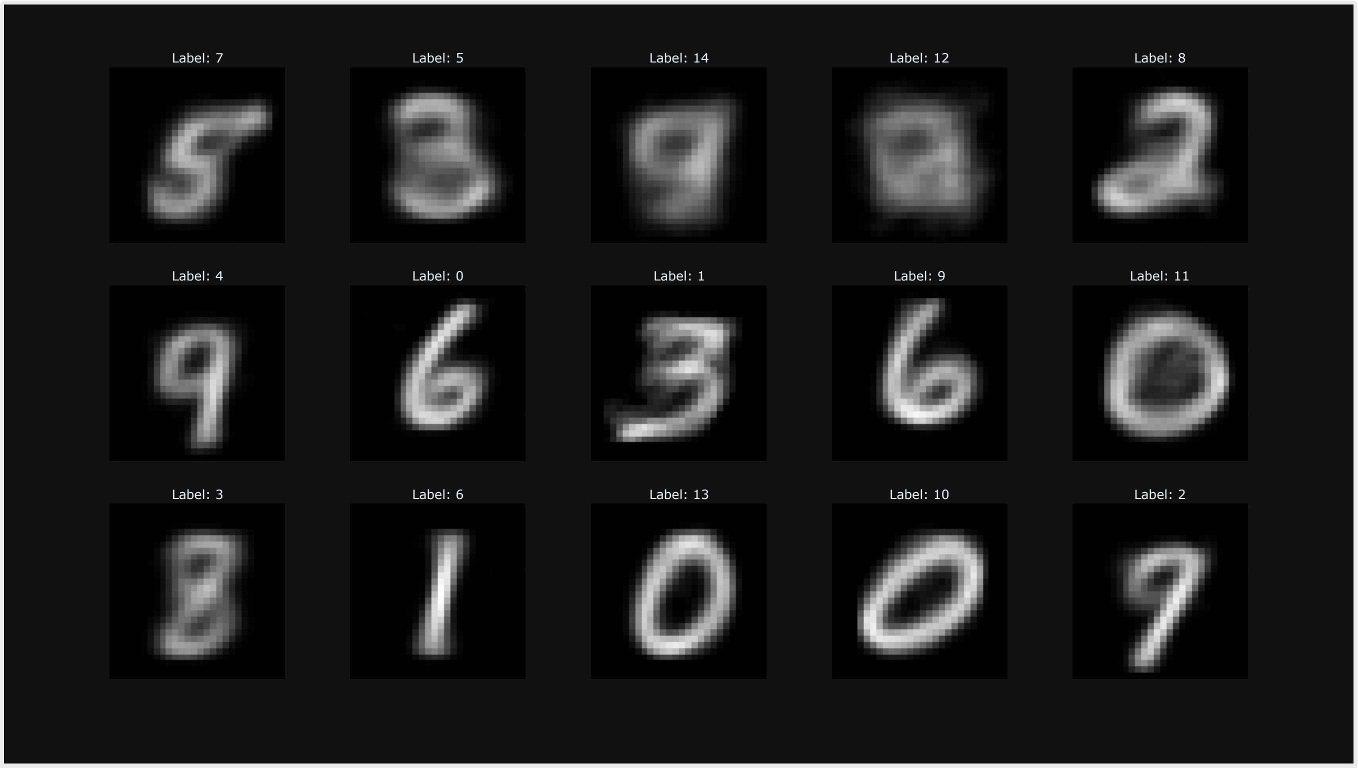
Best and Worst Models Mean Predictions

Note

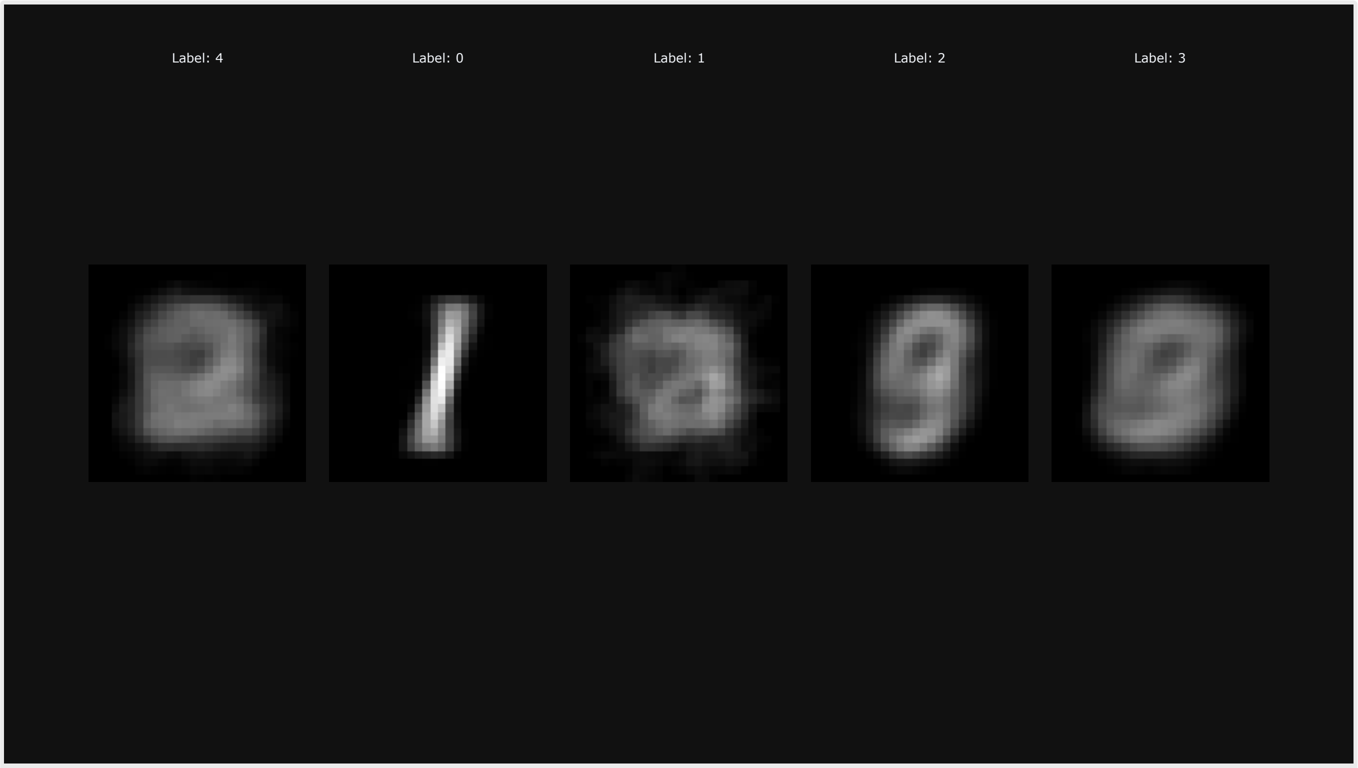
In the following plots, only the mean predictions for the top k most numerous clusters are shown. This is particulary important in the case of Mean Shift models, where the number of clusters was sometimes in the thousands.

Additionally, the text on top of each image is the label associated to that particular cluster by the model.

GMM

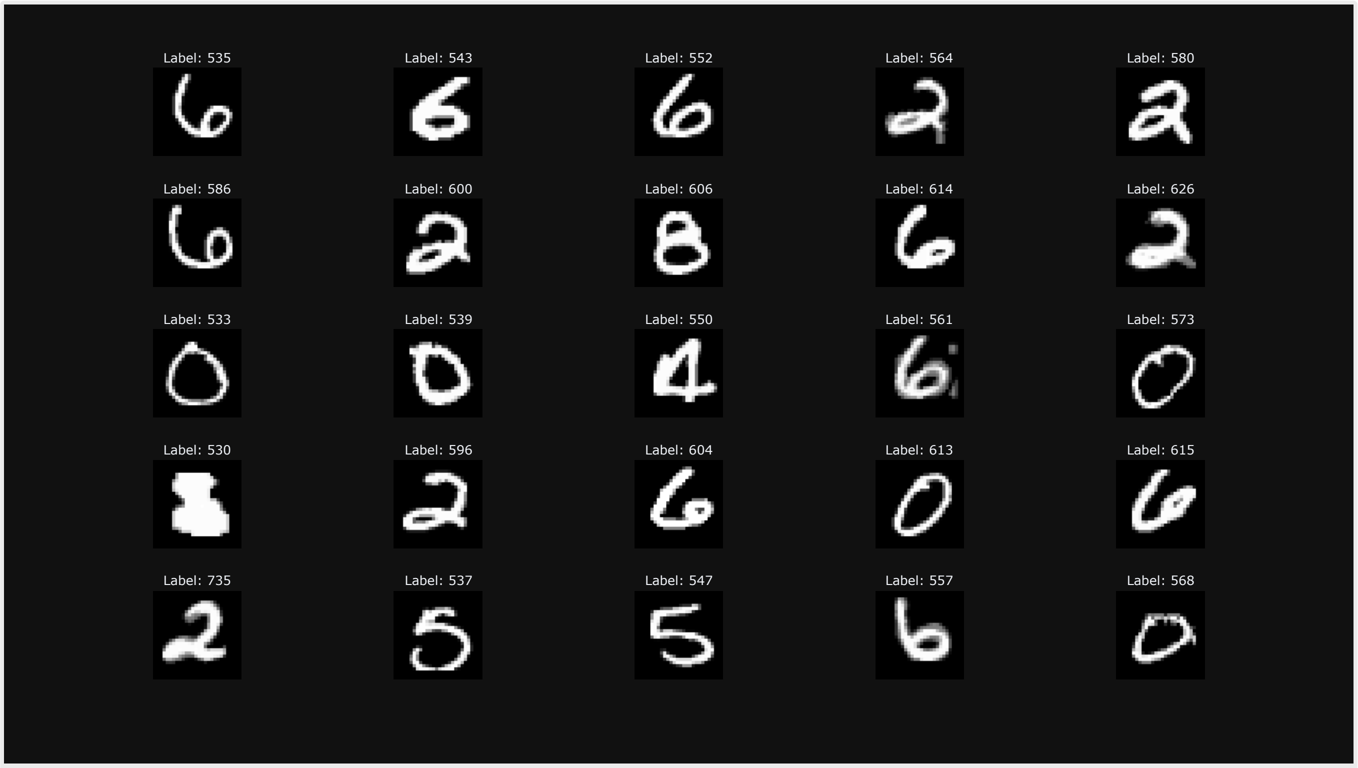


GMM - Best Performing Model



GMM - Worst Performing Model

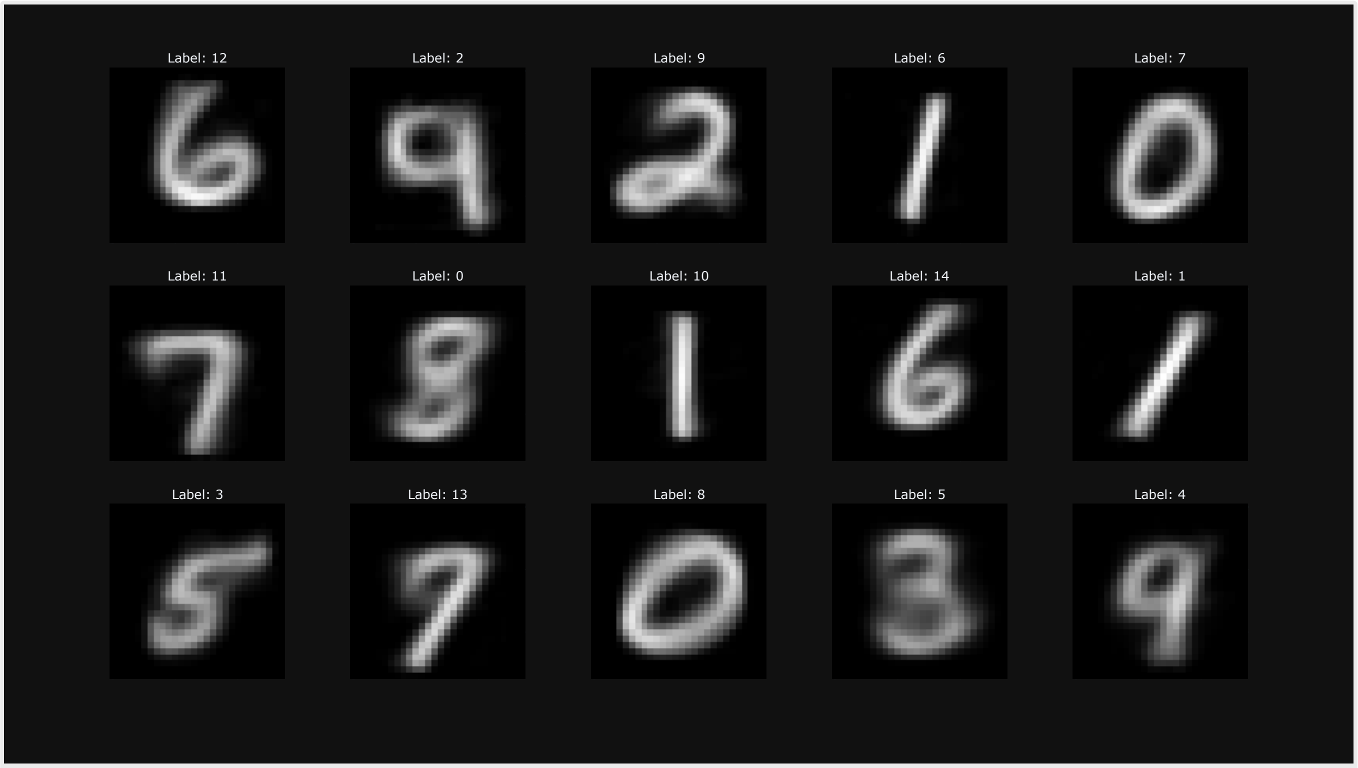
Mean Shift



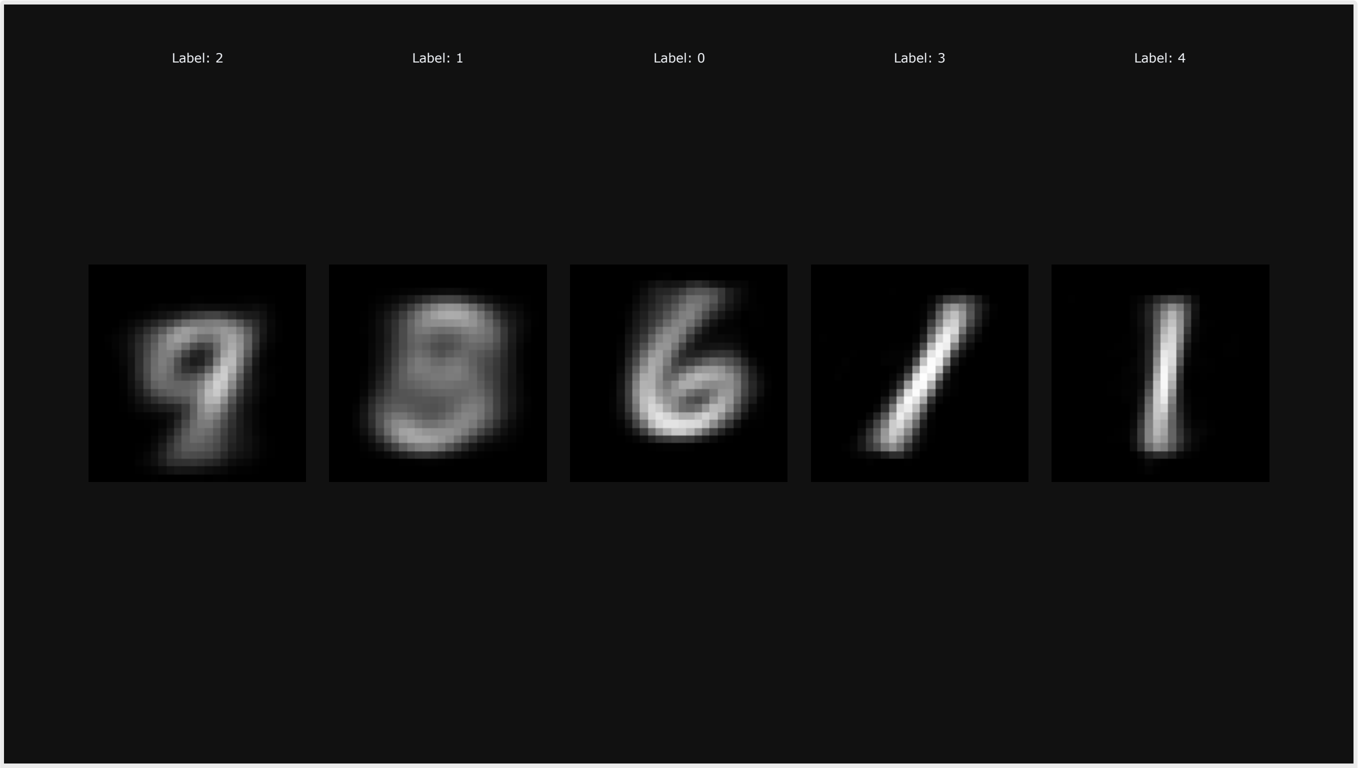
Mean Shift - Best Performing Model

Mean Shift - Worst Performing Model

Normalized Cut



Normalized Cut - Best Performing Model

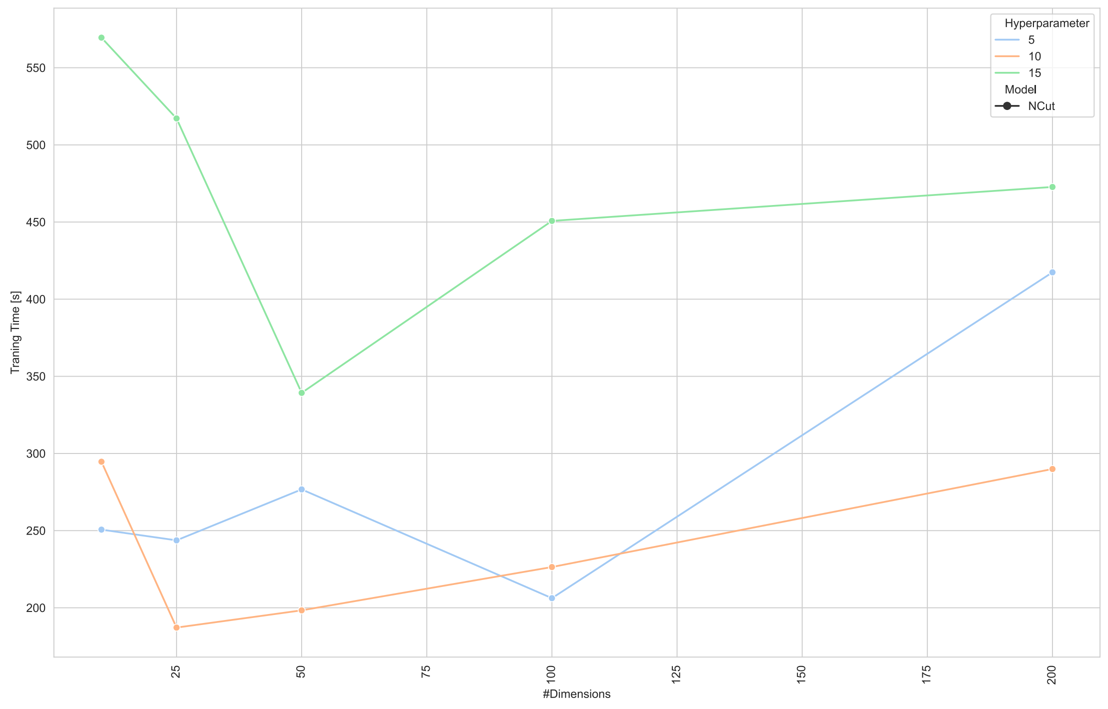


Normalized Cut - Worst Performing Model

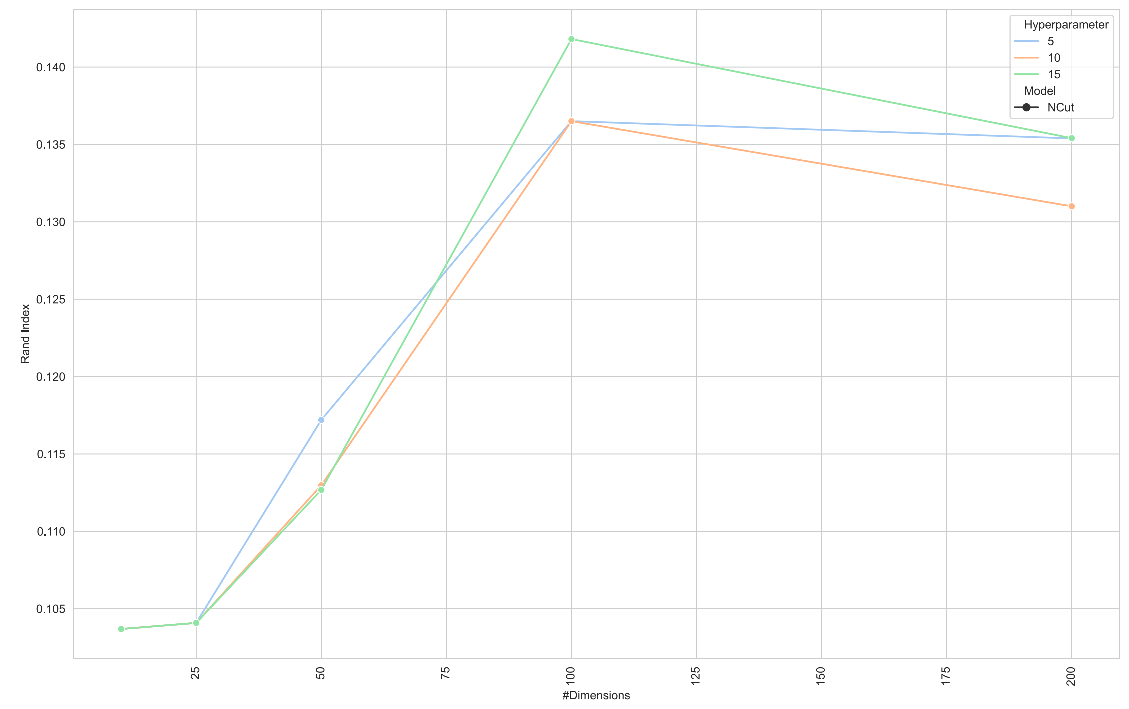
Parenthesis on a Wrong Usage of NCut

Below are some charts about one previous experimental results for Normalized Cut clustering, achieved on a much smaller dataset (1.5% of the full MNIST dataset) and with different hyperparameters.

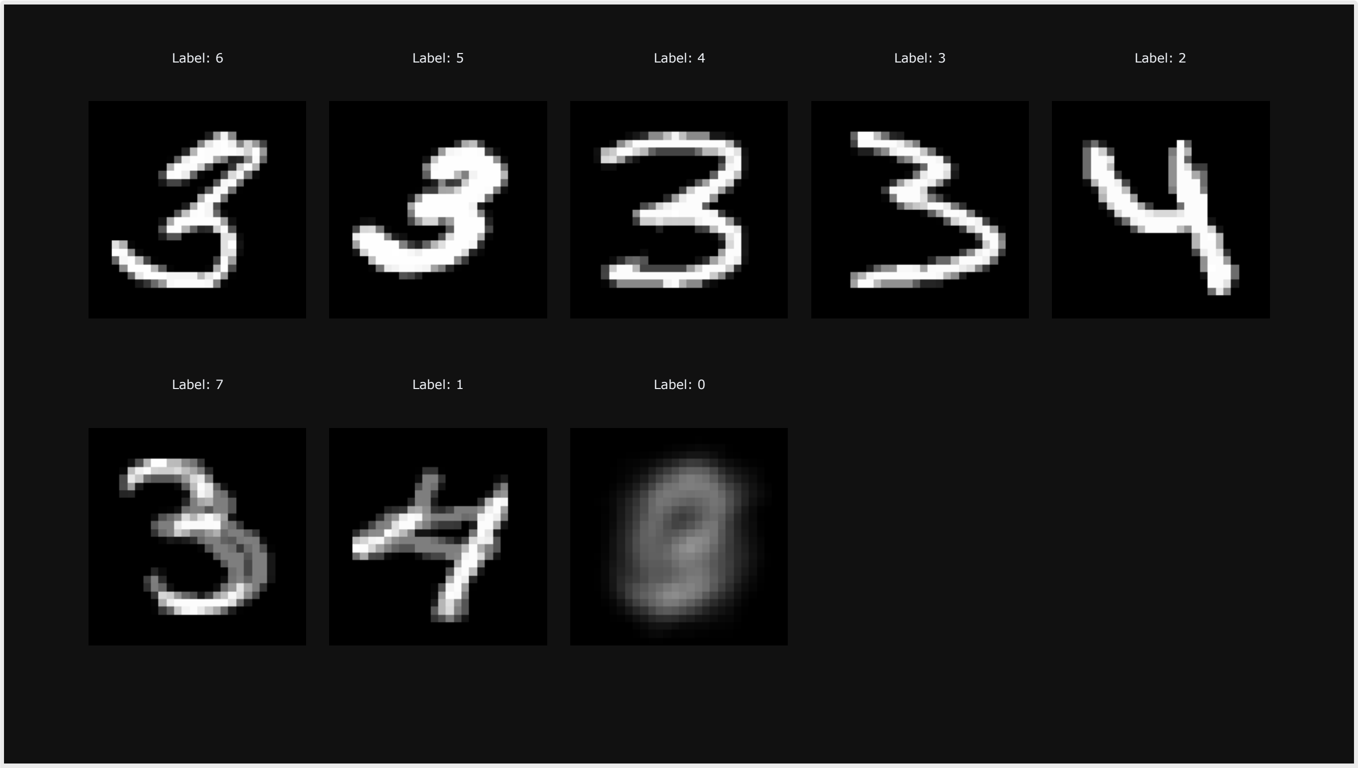
I decided to include these charts in the report because the difference with respect to my final results is very impressing. What changed from these experimental settings was the way that the affinity matrix was calculated: in the below charts, NCut used a RBF kernel with $\gamma = 1.0$, while in the final results, sklearn's k-neighbors graph approach was used, which is based on the minkowski distance.
The former approach, likely because of a too narrow kernel width, resulted in an affinity matrix with a good amount similarity values very close 0, consequently meaning that the corresponding graph was disconnected. The normalized cut method is based on the concept of finding the minimum (normalized) cut, meaning that in a disconnected graph, where the min cut is the *"zero cut"* between two disconnected components, leads to unexpected results and possibly convergence issues.
The latter approach, instead, guarantees that at least k (which defaults to 10 in this case) nearest neighbors are connected to each sample, meaning that it is much more unlikely to have a disconnected graph. It also improved stability and training times by two orders of magnitude in some cases, likely due to the affinity matrix being more numerically stable



NCut Training Times



NCut Rand Index



NCut Best Performing Model Mean Clusters - This might be evidence that the component with 3s and 4s was disconnected from the rest of the data

Label: 1
Label: 2

NCut Worst Performing Model Mean Clusters

Conclusions

Dimensionality reduction affected the training times of most models, with, as expected, models training on more features taking more time.
Additionally, despite some models actually performing well, the scoring methodology adopted for this assignment turned out to be not particularly meaningful, due to the fact that true and predicted labels were "not aligned" and the tendency of Rand Index to give scores close to 1 with a high number of clusters and/or data samples.