Ca' Foscari
University
of Venice

Foundations of Artificial Intelligence

Assignment 3: Clustering

Professor
Andrea Torsello

Student
Run Jie Xia
Matriculation Number 879779

Academic Year
2022-23

Contents

1 Introduction

This document reports on the theory and software implementation process used to achieve the goal of the assignment described below.

1.1 Assignment

Perform classification of the MNIST database[1] (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 assess 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.

1.2 MNIST database

The MNIST database[1], short for "Modified National Institute of Standards and Technology database," is a widely used resource for training various image processing systems. It was created by remixing samples from NIST's original datasets, which is how it got its name. The original black and white images were normalized to fit within a 28x28 pixel bounding box and anti-aliased, introducing grayscale levels. This dataset consists of 70,000 28x28 pixel grayscale images of handwritten digits.

In Python, you can easily fetch and load this dataset using the following code:

```
from sklearn.datasets import fetch_openml
X,y = fetch_openml('mnist_784', version=1, return_X_y=True)
y = y.astype(int)
X = X/255.
```

This code retrieves the dataset, resulting in 784-dimensional feature vectors (28*28) with pixel values ranging from 0 (white) to 1 (black).

2 Theoretical Background

The chapter provides the theoretical basis of each major concept for the experimental phase of the assignment, notably:

- **PCA** for dimensionality reduction;
- Sections covering essential concepts such as Latent Variable Models (LVMs), Expectation-Maximization (EM) Algorithms, Clustering methods (Eigenvector-based, Graph-based, and Hierarchical-Clustering), which are fundamental for understanding the unsupervised learning models used in this assignment;
- **Rand Index** for model evaluation;
- **Mean Shift** Clustering;
- **Normalized-Cut** Clustering (Spectral Clustering);
- **Gaussian Mixtures** Clustering.

Following that, an examination of implementation details and the presentation of experimental results will be presented.

2.1 Dimensionality reduction

Dimensionality reduction is a technique used when dealing with high dimensional datasets, where the number of features/dimensions becomes unmanageable. It serves the purpose or reducing the dataset's size to a more manageable scale while preserving its intrinsic characteristics. Excessive dimensions can pose significant challenges, including the risk of overfitting in machine learning models and making it difficult in visualizing data.

By employing methods like Principal Component Analysis (PCA), dimensionality reduction identifies and extracts the most informative features, effectively simplifying the dataset. This process enhances the efficiency of machine learning algorithms, enabling more accurate analyses, model training, and data interpretation.

2.2 Principal Component Analysis (PCA)

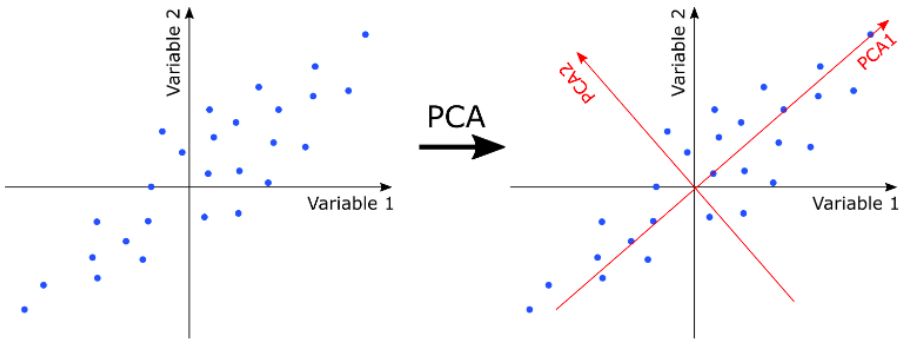
Principal Component Analysis (PCA) is a powerful dimensionality reduction method widely used in data analysis and machine learning to reduce the dimensionality of large datasets. Its core objective is to transform large, high-dimensional datasets into more compact representations while retaining critical information.

By doing so, PCA strikes a balance between simplifying the data for enhanced exploration and visualization and minimizing extraneous variables that can hinder machine learning algorithms. This technique employs a linear transformation to create a new data representation, yielding a

set of principal components. These components capture the most significant sources of variance within the dataset, effectively reducing redundancies and facilitating data compression. While dimensionality reduction inherently involves a trade-off between reduced dimensionality and some loss of accuracy, PCA allows to navigate and analyze complex datasets more efficiently and effectively.

2.2.1 Principal Components and Covariance Matrix

Preserving maximum information in PCA means identifying the key components that explain the most variance in the data. Imagine looking at a scatterplot of your data: the component that explains the largest variance corresponds to a line running through the data points like the line names PCA1 (you can see this in the image below).



The second principal component PCA2 is a line perpendicular to the first, showing that it explains a different source of variation in the dataset.

Now, let's discuss how we find these components mathematically. We start by standardizing the dataset, ensuring that no single feature has an abnormal influence. Then, we calculate the first k principal components by finding the k eigenvectors of the covariance matrix. These eigenvectors represent orthogonal directions of maximum variance in the data. To understand how much variance each component explains, we look at the associated eigenvalues and their ratio to the sum of all eigenvalues. This ratio tells us the proportion of variance captured by each component.

2.2.2 Intuition

Mathematically, these principal components are determined by computing the eigenvectors of the covariance matrix of the standardized dataset. Standardization is crucial to ensure that no single feature excessively influences the components. The first k principal components, where k is less than the original dimensionality, form a feature vector represented as:

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

where each p_i is a principal component (column vector). To transform the original dataset into a lower-dimensional representation D, we multiply the transpose of the standardized dataset S by the transpose of the feature vector F:

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

The intuition behind PCA lies in maximizing the variance (preserved information) of projected data along specific unit vectors. Imagine wanting to arrange data points along a line to preserve the most information possible. This involves finding unit vectors that maximize

the variance of the projected data, effectively simplifying it. The variance of projected data is represented as:

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

In this formula:

- u represents the unit vector.
- x_i are data points.
- $\bar{x}$ is the mean of the data.
- S is the covariance matrix of the data.

where:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i,$$

and

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

Ultimately, we aim to find the unit vector u that maximizes the variance (preserved information) of the dataset:

$$\max_u u^T S u \quad \text{subject to} \quad u^T u = 1$$

This quantity corresponds to the eigenvector with the largest eigenvalue, determined through an optimization process. By selecting a subset of principal components, we effectively reduce the dataset's dimensionality, albeit with some information loss. This reduction is crucial for simplifying data analysis while maintaining meaningful insights.

2.3 Unsupervised Machine Learning

Unsupervised machine learning encompasses a class of algorithms designed to explore the inherent structure within unlabeled data. Unlike supervised learning, where models are trained on labeled examples, unsupervised learning techniques seek to discover patterns, groupings, and relationships within the data without prior knowledge of class labels or target variables.

2.3.1 Clustering

One of the primary tasks in unsupervised learning is clustering. Clustering involves grouping similar data points together based on their inherent characteristics. It helps reveal the underlying structure of a dataset, making it a valuable tool for various applications, including image segmentation, customer segmentation, and anomaly detection.

Types of Clustering

There are several types of clustering techniques, each with its unique approach to grouping data points:

- Exclusive Clustering: Assigns each data point to exactly one cluster.
- Overlapping Clustering: Allows data points to belong to multiple clusters to varying degrees.
- Hierarchical Clustering: Forms clusters in a hierarchical manner, either starting from individual data points (bottom-up) or from a single cluster containing all data points (top-down).

2.3.2 Distance Metrics

Distance metrics play a crucial role in clustering algorithms. They quantify the dissimilarity or similarity between data points. Common distance metrics include:

- Euclidean Distance: Measures the straight-line distance between two points in a multidimensional space.
- Manhattan Distance: Measures the distance between two points along the axis-aligned paths.
- Cosine Similarity: Computes the cosine of the angle between two vectors, indicating their similarity.

2.4 Latent Variable Models (LVMs)

Latent Variable Models are probabilistic models that incorporate unobserved (latent) variables to explain observed data. They are widely used in various fields, including dimensionality reduction and clustering.

2.4.1 Definition

A Latent Variable Model is defined by a joint probability distribution over two sets of variables: observed variables (x) and latent variables (z), parameterized by θ:

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

Where:

- x represents observed variables in a dataset $\mathcal{D}$.
- z represents latent variables that are not directly observed.
- θ denotes the parameters of the distribution.

Basic Terminology

Latent variable models aim to simplify the representation of data by mapping observed variables to latent variables. Key terms include:

- Prior Distribution ($p(z)$): Models the behavior of latent variables.
- Likelihood ($p(x \mid z)$): Defines how latent variables map to observed data.
- Joint Distribution ($p(x, z)$): Describes the overall model and how data is generated.
- Marginal Distribution ($p(x)$): Represents the distribution of the original data.
- Posterior Distribution ($p(z \mid x)$): Describes latent variables given observed data.

2.4.2 Learning LVMs

Learning LVMs is an iterative process aimed at optimizing model parameters to maximize the likelihood of observed data. The Expectation-Maximization (EM) algorithm is a commonly used approach for this purpose.

Marginal Likelihood Training

The EM algorithm aims to maximize the marginal log-likelihood of the data:

$$\log p(\mathcal{D}) = \sum_{x \in \mathcal{D}} \log p(x)$$

This objective is challenging because it involves a sum over latent variables and doesn't have a closed-form solution.

2.4.3 Expectation-Maximization (EM) Algorithm

The Expectation-Maximization (EM) algorithm is helpful for learning LVMs. It operates through the following iterative steps:

1. E-Step (Expectation): Compute the posterior distribution of latent variables. In this step, we estimate the values of the latent variables based on the observed data.
2. M-Step (Maximization): Update model parameters to maximize the expected log-likelihood. We adjust the model's parameters to fit the data based on the estimated latent variables.

Advantages:

- Convergence: The EM algorithm guarantees convergence, meaning it will eventually find a local optimum.
- Flexibility: It can be applied to a wide range of latent variable models, making it a versatile tool.
- Utilizes Uncertainty: By estimating latent variables in the E-step, EM captures uncertainty in the model.

Disadvantages:

- Local Optima: EM may converge to a local optimum rather than the global optimum.
- Initialization Sensitivity: Results can depend on the initial parameter values, requiring multiple restarts.
- Computationally Intensive: EM can be computationally demanding, especially for complex models or large datasets.

EM's significance arises from the non-convex nature of the optimization problem and the lack of closed-form solutions. While it may not guarantee a global optimum, it efficiently finds local optima.

2.4.4 Why Are Latent Variable Models Useful?

Latent Variable Models serve several critical purposes:

- Expressiveness: They increase the expressive power of models, enabling them to capture complex data distributions more effectively.
- Incorporating Prior Knowledge: LVMs allow the incorporation of prior knowledge or assumptions about data generation, enhancing model performance.
- Dimensionality Reduction: LVMs often serve as dimensionality reduction techniques, reducing data complexity while retaining meaningful information.
- Clustering: They can be used for clustering, such as Gaussian Mixture Models, where latent variables represent cluster assignments.

2.5 Eigenvector-based Clustering

Eigenvector-based clustering is a technique that leverages the eigenvectors of certain matrices to uncover clustering patterns within data.

2.5.1 Max-Variance Direction

Eigenvectors represent directions in data that explain the majority of the variation, while eigenvalues indicate the magnitude of the variation along those eigenvectors.

Eigenvectors play a crucial role in this clustering method. They represent the directions in the data that explain the majority of the variation, while the eigenvalues quantify the magnitude of variation along each eigenvector. The main idea behind eigenvector-based clustering is to generate eigenvectors from the original dataset, with each eigenvector highlighting a direction of maximum variance. These eigenvectors are then combined with the dataset to generate additional information used for clustering.

In practical terms, this involves thresholding each data point based on the transformed data, effectively checking if a metric applied to each transformed data point is above a certain cluster membership threshold.

2.5.2 Optimization Problem

Eigenvector-based clustering involves solving an optimization problem to find the "participation mask" that maximizes the sum of similarities between data points.

The optimization problem at the core of eigenvector-based clustering is formalized as follows:

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

Here, x_i measures the participation of node i in a cluster, with $x_i = 1$ indicating inclusion and $x_i = 0$ indicating exclusion. The matrix W represents the affinity or similarity between data points.

The objective is to find the "participation mask" that maximizes the sum of similarities, effectively identifying data points belonging to the same cluster. The optimization problem includes the constraint that x is a unit vector, ensuring that the magnitude of the direction is considered.

Alternative Form: Rayleigh Quotient

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

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

If x is an eigenvector of W, $R(W, x)$ simplifies to the corresponding eigenvalue λ. This connection highlights that optimizing $R(W, x)$ is equivalent to finding the largest eigenpair.

2.5.3 Characteristics: Pros and Cons

Eigenvector-based clustering has certain characteristics, including the convergence to local optima and the dependence on initialization.

Eigenvector-based clustering, while powerful, has its strengths and limitations:

Advantages:

- Leverages the most informative directions in the data.
- Can handle complex data structures.
- Well-suited for spectral clustering.

Disadvantages:

- Sensitive to initialization: Different initializations can lead to different solutions.
- May converge to local optima: The optimization problem may find suboptimal solutions.
- Requires thresholding: Determining cluster membership typically involves setting a threshold on the transformed data.

2.5.4 Spectral Properties

The Laplacian matrix of a graph plays a key role in eigenvector-based clustering. It has several spectral properties, including being symmetric and positive semi-definite.

The Laplacian matrix (L) of a graph is central to eigenvector-based clustering. It possesses several spectral properties:

1. Symmetry: The Laplacian matrix L is symmetric, making it suitable for spectral analysis.
2. Positive Semi-Definite (PSD): L is positive semi-definite, meaning its eigenvalues are all non-negative. This property is crucial for ensuring that the optimization problem has well-defined solutions.
3. Eigenvalues: 0 is always an eigenvalue of L, and it is the smallest eigenvalue. The multiplicity of 0 corresponds to the number of connected components in the graph.
4. Spectral Gap: The spectral gap refers to the difference between the modules of the two largest eigenvalues of a matrix. It plays a role in determining the connectedness of the graph.
5. Cheeger Constant: The Cheeger constant is a measure of the "bottleneckedness" of a graph. A larger spectral gap indicates greater robustness in terms of graph connectivity.

Different normalizations of the Laplacian matrix, such as row and symmetric normalizations, are often used in graph-based clustering to adapt the matrix to specific applications.

2.6 Graph-based Clustering

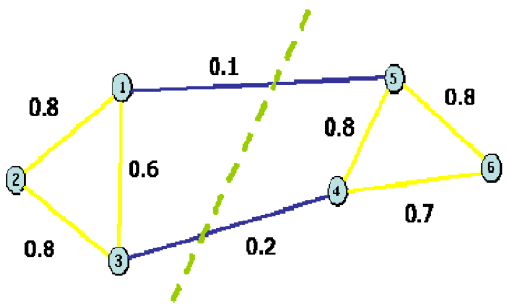
Graph-based clustering methods rely on the representation of data as a weighted graph to discover clusters.

2.6.1 Minimum Cut Problem

The Minimum Cut Problem involves finding a partition of a graph into two sets that minimize the sum of weights of the edges cut.

In the context of graph-based clustering, the Minimum Cut Problem is central. It is defined as finding a partition of a graph into two sets, typically denoted as A and B, where $B = V \setminus A$. The objective is to minimize the sum of weights of the edges cut. This cut is quantified using the "cut function" $cut(A, B)$, which represents the sum of the weights of edges severed when dividing the graph. Mathematically:

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



This problem has both advantages and shortcomings:

Advantages:

- Solvable in polynomial time.
- Provides a way to find natural divisions in the data.

Disadvantages:

- Tends to favor unbalanced clusters: In some cases, it may split isolated nodes from the rest of the clusters, which can lead to unbalanced solutions.
- Addressed by Normalized Cut: To mitigate the problem of unbalanced clusters, normalized cut measures are often used.

2.6.2 Preliminary Terminology

Basic graph terminology, such as node degree, set volume, and the Laplacian matrix, is essential for understanding graph-based clustering.

Before delving further into graph-based clustering, it's important to understand some fundamental graph terminology:

Degree of Nodes:

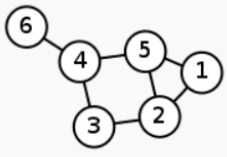
The degree d_i of a node i is the sum of the weights of its incident edges. For example, if we have a graph where node i is connected to nodes j and k with edge weights of 2 and 3, respectively, then $d_i = 2 + 3 = 5$. In essence, it quantifies how connected a node is within the graph.

Volume of a Set:

The volume of a set of nodes A is the sum of the degrees of each node in that set. For instance, if we have a set A containing nodes i, j, and k with degrees 2, 3, and 4, respectively, then the volume of A is $2 + 3 + 4 = 9$. This measure is crucial for certain clustering algorithms, especially when considering spectral properties.

Graph's Laplacian Matrix:

A graph's Laplacian Matrix (L) encapsulates key information about the graph's structure. It is defined as $L = D - W$, where W is the affinity or similarity matrix, and D is the degree matrix. The Laplacian matrix has several spectral properties, including symmetry, positive semi-definiteness, and eigenvalues associated with graph connectivity.

Labelled graph	Degree matrix	Adjacency matrix	Laplacian matrix
	$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$

2.6.3 Spectral Gap and Cheeger Constant

The spectral gap and Cheeger constant are measures used to analyze the connectivity and bottleneck of a graph.

The analysis of graph connectivity in the context of graph-based clustering involves two important measures:

Spectral Gap:

The spectral gap is defined as the difference between the moduli of the two largest eigenvalues of a matrix. In the context of graph analysis, a larger spectral gap indicates greater connectedness in the graph, making it easier to identify meaningful clusters. Practically, it helps determine the quality of clusters discovered through graph-based methods.

Cheeger Constant:

The Cheeger constant is a measure of the bottleneck of a graph. It quantifies how easily a graph can be split into two disjoint sets while minimizing the number of edges between them. A higher Cheeger constant indicates a graph that is less bottlenecked and more interconnected. This constant is particularly useful when you want to find a balance between well-connected clusters.

The relationship between the spectral gap and the Cheeger constant is described by the Cheeger Inequalities, which help assess the robustness of a graph's connectedness.

2.6.4 Normalizations

Row and symmetric normalizations of the Laplacian matrix are often used in graph-based clustering.

To adapt the Laplacian matrix (L) to specific applications in graph-based clustering, different normalizations are used:

Row Normalization (L_{rw}):

Row normalization scales the rows of the Laplacian matrix such that they sum up to 1. This transformation, expressed as $L_{rw} = D^{-1}L = D^{-1}(D-W) = I-D^{-1}W$, can be particularly useful when dealing with graphs where node degrees vary widely. It helps balance the contributions of nodes to clustering.

Symmetric Normalization:

Symmetric normalization ensures that both rows and columns of the Laplacian matrix sum up to 1. By using $D^{-\frac{1}{2}}LD^{-\frac{1}{2}} = I - D^{-\frac{1}{2}}WD^{-\frac{1}{2}}$, symmetric normalization maintains a balanced representation of nodes and edges in the graph. It is commonly employed in spectral clustering to create more robust cluster representations.

These normalizations are chosen based on the specific characteristics of the data and the desired clustering outcomes.

2.7 Hierarchical Clustering

Hierarchical clustering is an approach to cluster analysis that builds a hierarchy of clusters through either agglomerative or divisive methods.

2.7.1 Agglomerative vs. Divisive

Hierarchical clustering can be performed using either agglomerative or divisive strategies, each with its own approach to merging or splitting clusters.

Hierarchical clustering offers flexibility through two primary strategies:

Agglomerative Clustering:

In agglomerative clustering, often referred to as the "bottom-up" approach, each observation starts in its own cluster, and clusters are merged as one moves up the hierarchy. This strategy iteratively combines smaller clusters into larger ones, ultimately forming a hierarchy.

Divisive Clustering:

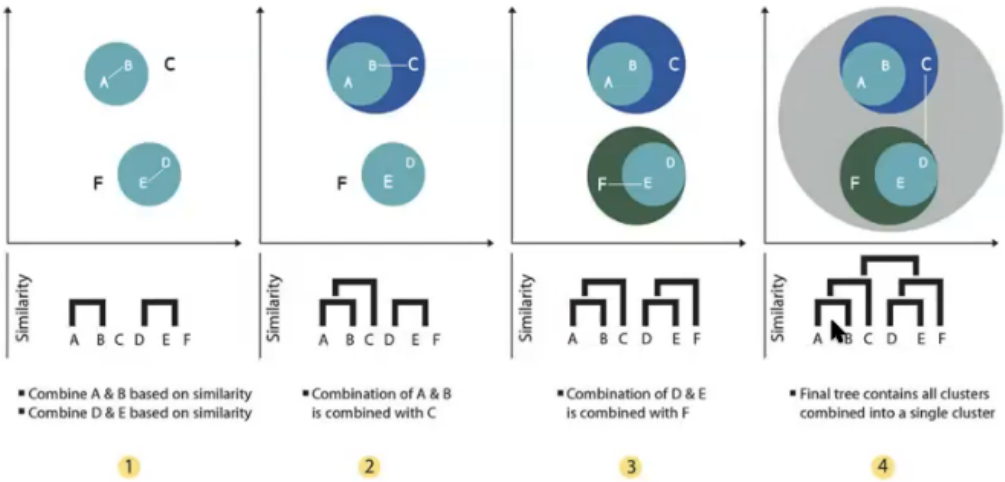
In divisive clustering, the "top-down" approach is employed. Initially, all observations belong to a single cluster, and the algorithm recursively splits this cluster into smaller ones as one moves down the hierarchy. Divisive clustering focuses on breaking down large clusters into finer-grained subclusters. It is a choice made when you want to start with a single, all-encompassing cluster and progressively divide it into smaller, more specific clusters.

2.7.2 Dendrogram

A dendrogram is a tree-like diagram that visually represents the hierarchy of clusters produced by hierarchical clustering.

A key visualization in hierarchical clustering is the dendrogram. It serves as a tree-like diagram that visually captures the hierarchy of clusters generated by the clustering algorithm. In a dendrogram, each node represents a cluster, and the height of the branches or links connecting nodes reflects the dissimilarity or distance between clusters. endrograms provide an intuitive way to explore the nested structure of clusters at different levels of granularity.

When interpreting a dendrogram, note that the height of the branches indicates the dissimilarity or distance between clusters. Clusters with shorter branch heights are more similar, while those with longer branch heights are less similar.

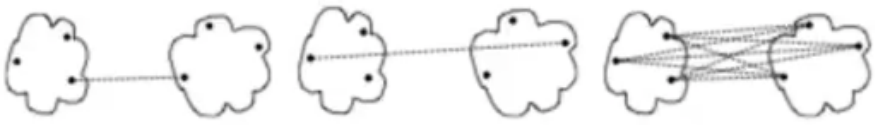


2.7.3 Linkage Methods

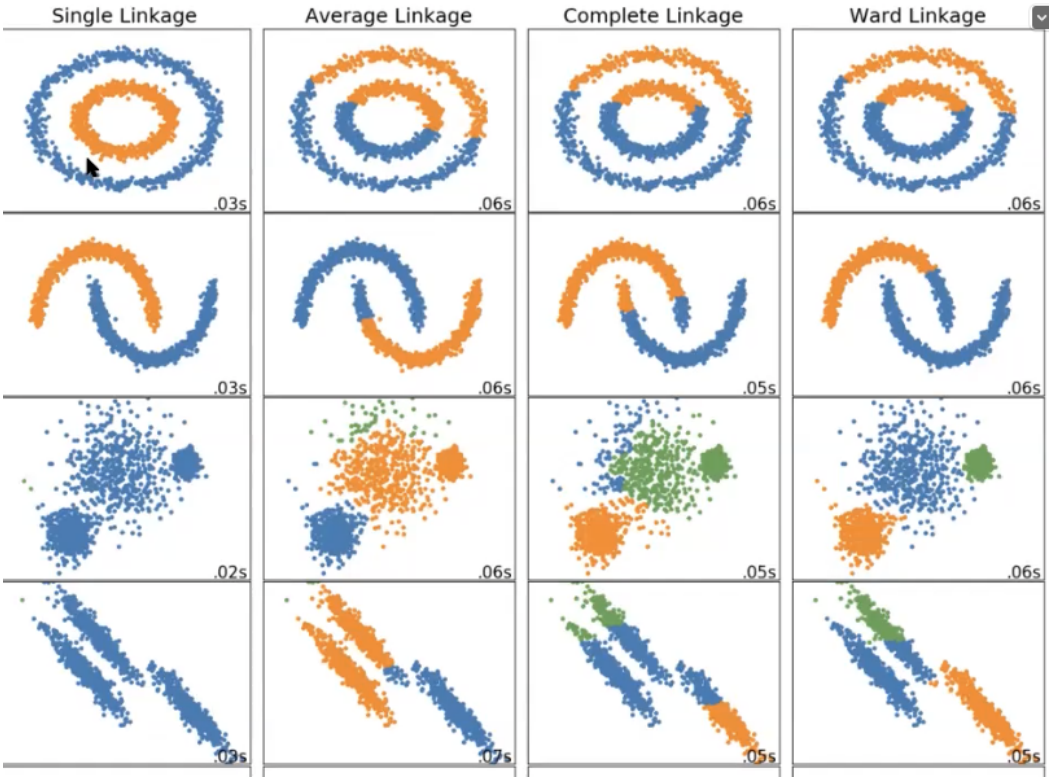
Different linkage methods, such as single linkage, complete linkage, and average linkage, determine how the proximity between clusters is measured during the agglomerative process.

The choice of linkage method in hierarchical clustering significantly influences the clustering results. Here are some common linkage methods:

- Single Linkage (Nearest Neighbor Linkage): This method measures the proximity between two clusters by considering the closest (most similar) data points between them. It tends to create elongated clusters.
- Complete Linkage (Farthest Neighbor Linkage): Complete linkage, on the other hand, assesses cluster proximity based on the farthest (least similar) data points between clusters. It tends to produce compact, spherical clusters.
- Average Linkage: Average linkage calculates the average similarity between all pairs of data points, one from each cluster. It aims to strike a balance between single and complete linkage, often resulting in well-balanced clusters.
- Ward's Linkage: Ward's linkage minimizes the variance within clusters when merging. It is known for creating balanced and compact clusters.



(a) MIN (single link.) (b) MAX (complete link.) (c) Group average.



Choosing the most appropriate linkage method depends on the nature of the data and the desired cluster structures. The choice should be guided by the specific characteristics of your dataset and the objectives of your analysis.

3 Project Setup

This chapter describes the methods used for the experimental evaluation of the models.

3.1 Unsupervised Learning Model Evaluation

In unsupervised machine learning, evaluating models can be a challenging task due to the absence of ground truth labels. However, there are two main approaches to assess the performance of unsupervised models: **Intrinsic/Internal Validation** and **Extrinsic/External Validation**.

3.1.1 Intrinsic/Internal Validation

Intrinsic validation does not rely on any external ground truth but assesses the quality of clusters generated by a model based on the concept of intra-class and inter-class dissimilarity. The primary goal is to measure how well clusters are separated and shaped within the data. A model is considered effective in intrinsic validation if it can create well-separated clusters with high cohesion.

3.1.2 Extrinsic/External Validation

Extrinsic validation, on the other hand, leverages some form of external ground truth, such as correct labels for the dataset, to evaluate the quality of clusters. It aims to measure the similarity between the model's output and the true clusters present in the data. One common method of extrinsic evaluation involves computing a **confusion matrix** to compare the model's clusters with the ground truth clusters.

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

3.1.2.1 Confusion Matrix

A confusion matrix is a 2x2 matrix containing four values:

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

Where:

- True Positives (TP): The number of pairs of elements that are in the same cluster both according to the model and the ground truth.
- False Positives (FP): The number of pairs of elements that are in the same cluster according to the model but in different clusters according to the ground truth.

- True Negatives (TN): The number of pairs of elements that are in different clusters both according to the model and the ground truth.
- False Negatives (FN): The number of pairs of elements that are in different clusters according to the model but in the same cluster according to the ground truth.

3.1.2.2 Rand Index (Rand Statistic)

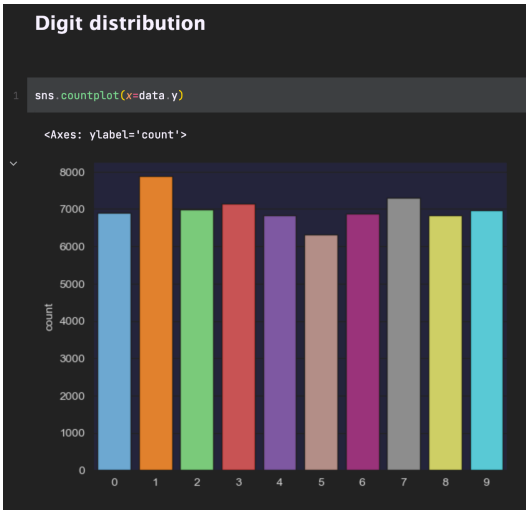
The Rand Index (RI) is a popular metric for measuring the similarity between two methods of clustering, especially in extrinsic evaluation. It quantifies the ratio of correct decisions over the total number of decisions made. The formula for Rand Index is as follows:

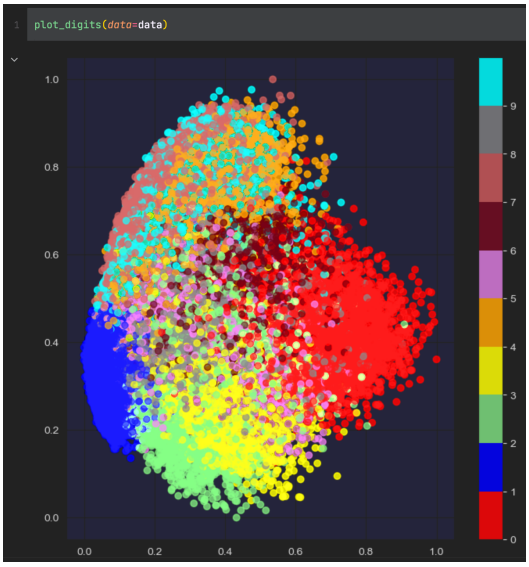
$$RI = \frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{\binom{n}{2}} = \frac{TP + TN}{\frac{n(n-1)}{2}} = \frac{2(TP + TN)}{n(n-1)}$$

Here, n represents the number of samples to group, making the denominator equal to all possible pairs of elements. The Rand Index ranges between 0 and 1, where 0 indicates that the two clustering methods are not compatible, while 1 signifies that the two methods have clustered the data in an identical manner.

3.2 Data

We used the MNIST database. This is composed of 70000 28x28 pixel gray-scale images of handwritten digits divided into 60000 training set and 10000 test set.





3.2.1 Limitations

In order to reduce the workload of process, the dataset used during the tuning phase was 20% of the whole dataset . The applied splitting method was given by the usage of sklearn function `train_test_split`, which via some flags split the dataset in training and testing sets.

Reducing dataset size

```python
def reduce_data(data: Dataset, percentage: float = 1.) -> Dataset:
    """
    Return a randomly reduced-percentage dataset
    return: new dataset
    """

    if not 0. <= percentage <= 1.:
        raise Exception(f"Percentage {percentage} not in range [0, 1] ")

    _,x,_,y = train_test_split(data.x, data.y, test_size=percentage, random_state=RANDOM_SEED)

    return Dataset(x, y)
```

```python
reduced_data = reduce_data(data=data, percentage=0.20)
reduced_data
```

```
[Length: 14000; Features: 784]
```

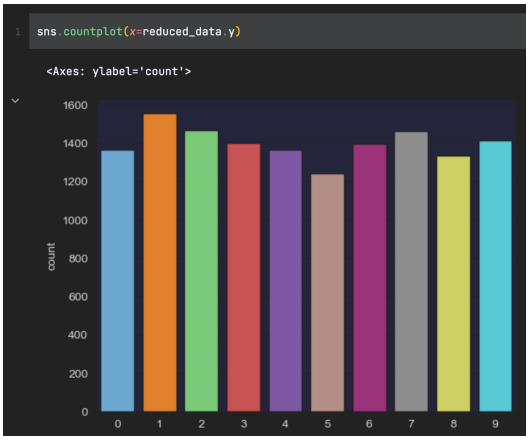
The set flags were:

- `test_size`: which identify the number or the percentage (if $0 <= x <= 1$) of the desired test set size;
- `random_state`: represent an int which is used to compute the shuffle methods before the slicing.

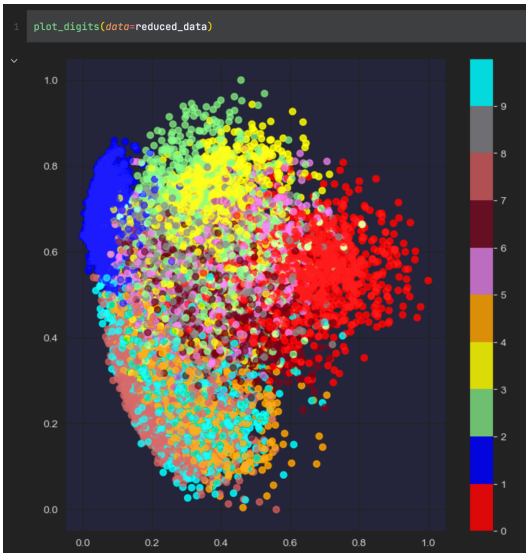
For all the models the above flags were set to 0.2 for `test_size` and 28 for `random_state`.

```python
sns.countplot(x=reduced_data.y)
```

```
<Axes: ylabel='count'>
```




```python
plot_digits(data=reduced_data)
```



3.3 Library usage

Throughout all the assignment, the following libraries have been used:

- numpy
- pandas

- seaborn
- scikit-learn
- matplotlib
- json
- pickle

3.4 Specifications of the Machine

The tests were performed on a MacBook Pro 2017 with the following characteristics:

- Processor: 2,3 GHz Dual-Core Intel Core i5
- RAM: 8 GB
- SSD: 256 GB
- macOS Monterey 12.0.1

3.5 Dataset Representation

The *Dataset* class plays a central role in managing and representing our data. This class serves as a container for both feature data and corresponding labels, encapsulating essential functionalities for data manipulation and analysis.

3.5.1 Class Structure

The *Dataset* class is designed to encapsulate the core components of a dataset:

- **Feature Data**: Represented as a Pandas DataFrame, the feature data contains the observations or samples of our dataset, where each row corresponds to a data point, and each column represents a specific feature.
- **Labels**: The class stores labels as a NumPy array, which provides the ground truth information associated with each data point. In unsupervised learning, where the ground truth is typically unavailable, the labels may not always be used directly for model training but can be leveraged for evaluation.

```python
class Dataset:
    """
    Represents a dataset with features and corresponding labels as a tuple of:
     - feature data as a Pandas DataFrame
     - labels as a NumPy array
    The class allows you to create, manipulate, and save datasets.
    """
    def __init__(self, x: pd.DataFrame, y: np.ndarray):
        """
        :param x: feature data
        :param y: labels
        """
        # data and labels must have the same length
        if len(x) != len(y):
            raise Exception(f"X has length {len(x)}, while y has {len(y)}")

        self._x: pd.DataFrame = x
        self._y: np.ndarray = np.array(y)
```

The constructor of the *Dataset* class accepts these two components as input parameters. It ensures that the data and labels have the same length, maintaining data integrity throughout the class's operations.

3.5.2 Functionalities

The *Dataset* class offers several critical functionalities to support data preprocessing and model evaluation in our project:

Feature Normalization

The *normalize* method is responsible for normalizing the feature data. Normalization is essential to ensure that all features have similar scales, preventing any single feature from dominating the learning process. This method employs Min-Max scaling, transforming feature values to a common range of [0, 1]. The result is a new *Dataset* object with normalized features, facilitating more robust and stable model training.

```python
    def normalize(self) -> 'Dataset':
        """
        Normalizes the features of a dataset using Min-Max scaling.
        Scales the features to the range [0, 1] to ensure that all features have similar scales.
        :return: a new Dataset object with normalized features using Min-Max scaling.
        """
        new_x = pd.DataFrame(MinMaxScaler().fit_transform(self.x), columns=self.features)
        return Dataset(
            x=new_x,
            y=self.y
        )
```

Dimensionality Reduction

Dimensionality reduction is a critical step in unsupervised learning to reduce the complexity of the dataset. The `reduction_PCA` method applies Principal Component Analysis (PCA) to

reduce the dimensionality of the feature data. PCA identifies the most significant patterns in the data and retains the specified number of principal components. This reduced-dimension *Dataset* object is valuable when dealing with high-dimensional datasets.

```
def reduction_PCA(self, n_comps: int) -> 'Dataset':
    """
    Reduces the dimensionality of a dataset to the given number of principal components
    using Principal Component Analysis (PCA).
    :param n_comps: number of principal components to retain.
    it must be a positive integer less than the number of features in the original dataset.
    :return: a new dataset with reduced dimensionality.
    """
    if n_comps < 0:
        raise Exception(
            "Number of components must a be positive integer")

    n_features = len(self.x.columns)
    if n_comps >= n_features:
        raise Exception(
            f"Number of components must be less than the number of features in the original dataset {n_features}")

    # return new dataset with reduced dimensionality
    return Dataset(
        x=pd.DataFrame(PCA(n_components=n_comps).fit_transform(self.x)),
        y=self.y
    )
```

Data Saving

To facilitate data management, the *Dataset* class provides a *save* method. This method stores both feature data and labels in separate CSV files within the dataset directory. This feature is particularly useful when data is modified or preprocessed, ensuring that the changes are persistently saved for future use.

```
# SAVE
def save(self, x_name: str = 'data_X', y_name: str = 'data_y'):
    """
    Stores the dataset in dataset directory
    :param x_name: name of feature file
    :param y_name: name of labels file
    """
    if not os.path.exists(get_dataset_dir()):
        os.mkdir(get_dataset_dir())

    x_out = os.path.join(get_dataset_dir(), f"{x_name}.csv")
    y_out = os.path.join(get_dataset_dir(), f"{y_name}.csv")

    print(f"Saving {x_out}")
    self.x.to_csv(x_out, index=False)

    print(f"Saving {y_out}")
    pd.DataFrame(self.y).to_csv(y_out, index=False)
```

Core Usage

The *Dataset* class serves as the primary data structure in our project for unsupervised model fitting. When data is loaded from the local disk, it is encapsulated within a *Dataset* object. This encapsulation allows us to smoothly apply various data preprocessing techniques, such as normalization and dimensionality reduction, while also providing easy access to features, labels, and the original data.

3.6 Method Evaluation

In this section, we discuss the evaluation methodology employed in our project and how the results are stored for further analysis and visualization. Our project focuses on evaluating different clustering models (MeanShift, SpectralClustering, and GaussianMixture) across various configurations.

3.6.1 Evaluation Framework

To systematically evaluate the performance of clustering models, we developed a comprehensive evaluation framework encapsulated within the *ClusteringModel* class. This abstract class facilitates the evaluation of clustering models by considering combinations of two essential factors:

- PCA Dimensions: We evaluate the models with different numbers of Principal Components (PCA dimensions). This allows us to explore the impact of dimensionality reduction on clustering performance.
- Model-Specific Hyperparameters: Each clustering model has its own set of hyperparameters. We systematically evaluate the models by varying these hyperparameters. For example, for MeanShift, we consider different bandwidth values, while for SpectralClustering, we vary the number of clusters, and for GaussianMixture, we experiment with different numbers of components.

```python
class ClusteringModel(ABC):
    """
    Abstract class for evaluating various clustering models (MeanShift, SpectralClustering, GaussianMixture).
    This class generalizes the evaluation process across different models by considering combinations of:
        - PCA dimensions
        - model specific hyperparameters
        (MeanShift: bandwidth, SpectralClustering: n_clusters, GaussianMixture: n_components)
    The class also provides methods for retrieving the best models and plotting evaluation results.
    """
    model_type: Optional[ModelType] = None
    model_name: str = 'model'
    hyperparameter_name: str = 'hyperparameter'
```

```python
    def __init__(self, data: Dataset, n_components: List[int], hyperparam_vals: List[int | float]):
        """
        Initialize the ClusteringModel instance.
        :param data: dataset for evaluation
        :param n_components: list of PCA dimensions to evaluate
        :param hyperparam_vals: list of model's hyperparameter values
        """
        self.data: Dataset = data
        self._n_components: List[int] = n_components
        self._hyperparam_values: List[int | float] = hyperparam_vals

        self._results: Dict[int, Dict[float, Dict[str, float]]] = dict()
        self._results_bestmodels: Dict[int, Dict[str, ModelType | float]] = dict()
        self._best_model: Dict[str, ModelType | float] = dict()
        self._evaluated: bool = False
```

The evaluation process involves fitting the models to the data, calculating the Random Index Score, determining the number of clusters formed, and measuring execution time. The best-performing models are identified based on their Random Index Scores, and the results are stored for subsequent analysis.

```python
def evaluate(self):
    """
    Evaluate the given clustering model over all combination of PCA dimensions and hyperparameters.
    All results will be stored in self._results providing:
        - number of clusters
        - random index score
        - evaluation time
    The models with the best score for each PCA dimensions will be stored in self._results_bestmodels.
    The best model overall will be stored in self._best_model.
    """
    # dictionaries keyed by PCA dimensions
    components_results = {}
    components_bestmodels = {}

    # initialize with the lowest rand scores possible
    best_score_glb = -1
```

```python
    for n in tqdm_notebook(self._n_components, desc=''):
        tqdm.write(f'Processing PCA dimension: {n}')
        data_pca = self.data.reduction_PCA(n_comps=n).normalize()  # applying PCA reduction to dataset
        best_score_lcl = -1
        hyperparameters = {}  # dictionary keyed by hyperparameter value

        for k in tqdm_notebook(self._hyperparam_values, desc='', leave=False):
            tqdm.write(f'PCA dimension: {n} - {self.hyperparameter_name} value: {k}')
            model = self.model_type.set_params(**{self.hyperparameter_name: k})  # changing model's parameters
            t1 = time.perf_counter()
            labels = model.fit_predict(data_pca.x)
            t2 = time.perf_counter()
            elapsed = t2 - t1

            score = rand_score(data_pca.y, labels)
            results = {
                'score': score,
                'n_clusters': len(set(labels)),
                'time': elapsed
            }
```

```
            # for local best model (over hyperparameter values)
            if score > best_score_lcl:
                best_score_lcl = score
                components_bestmodels[n] = {
                    f'{self.hyperparameter_name}': k,
                    'score': score,
                    'n_clusters': len(set(labels)),
                    'time': elapsed
                }

            # for overall best model
            if score > best_score_glb:
                best_score_glb = score
                self._best_model = {
                    'model': copy.deepcopy(model),
                    'n_components': n,
                    f'{self.hyperparameter_name}': k,
                    'score': score,
                    'n_clusters': len(set(labels)),
                    'time': elapsed
                }

            hyperparameters[k] = results
        tqdm.write("")
        components_results[n] = hyperparameters
    self._results = components_results
    self._results_bestmodels = components_bestmodels
    self._evaluated = True

    # Save the results to files
    self._save_result()
```

3.6.2 Result Storage

To ensure reproducibility and facilitate further analysis, we employ a structured approach to save the evaluation results:

- JSON Files: The evaluation results, including Random Index Scores, number of clusters, and execution times, are saved in JSON format. These files are named based on the clustering model and the hyperparameter under investigation. This structure allows easy retrieval of results for specific configurations.
- Pickled Model: The best-performing model for each configuration is serialized and stored using Python's *pickle* module. This enables us to load and utilize these models for further investigation or deployment.

```python
def _save_result(self):
    """
    Save the evaluation results and best models to JSON and pickle files.
    """
    if not os.path.exists(get_results_dir()):
        os.mkdir(get_results_dir())

    results_name = os.path.join(
        get_results_dir(), f"{self.model_name}_{self.hyperparameter_name}_result.json")
    results_bestmodel_name = os.path.join(
        get_results_dir(), f"{self.model_name}_{self.hyperparameter_name}_result_bestmodels.json")
    bestmodel_name = os.path.join(
        get_results_dir(), f"{self.model_name}_{self.hyperparameter_name}_bestmodel.pkl")

    print(f"Saving {results_name}")
    with open(results_name, 'w') as file:
        json.dump(self.results, file)

    print(f"Saving {results_bestmodel_name}")
    with open(results_bestmodel_name, 'w') as fl:
        json.dump(self.results_bestmodels, fl)

    print(f"Saving {bestmodel_name}")
    with open(bestmodel_name, 'wb') as f:
        pickle.dump(self.best_model, f)
```

```python
def load_results(self):
    """
    Load the evaluation results and best models from files.
    """
    results_name = os.path.join(
        get_results_dir(), f"{self.model_name}_{self.hyperparameter_name}_result.json")
    results_bestmodel_name = os.path.join(
        get_results_dir(), f"{self.model_name}_{self.hyperparameter_name}_result_bestmodels.json")
    bestmodel_name = os.path.join(
        get_results_dir(), f"{self.model_name}_{self.hyperparameter_name}_bestmodel.pkl")

    print(f"Loading {results_name}")
    with open(results_name, 'r') as file:
        results = json.load(file)

    print(f"Loading {results_bestmodel_name}")
    with open(results_bestmodel_name, 'r') as fl:
        result_bestmodels = json.load(fl)

    print(f"Loading {bestmodel_name}")
    with open(bestmodel_name, 'rb') as f:
        best_model = pickle.load(f)

    self._results = results
    self._results_bestmodels = result_bestmodels
    self._best_model = best_model
    self._evaluated = True
```

3.7 Plot functions

In this section, we introduce and describe the various plot functions used in our project to visualize and interpret the clustering results. These plots provide valuable insights into the performance of the clustering models and the structure of the data.

3.7.1 Cluster Frequencies

Function:

`plot_cluster_frequencies(data, model_name, best_model_info, save=False)`

- Purpose: This function visualizes the distribution of data points across clusters. It generates a histogram to display how data points are distributed among the clusters created by the chosen clustering model. Additionally, it identifies and displays clusters that may contain data points from multiple classes.
- Usage: This plot is particularly useful for understanding the distribution of data in the clusters and identifying potential issues with the clustering algorithm.

```python
def plot_cluster_frequencies(data: Dataset, model_name: str, best_model_info: dict, save: bool = False):
    """
    Plot the distribution of cluster frequencies in a dataset.
    It creates a histogram to display how data points are distributed across each cluster.
    Display only max_clusters number of clusters.
    :param data: dataset
    :param model_name: name of the clustering model
    :param best_model_info: dictionary containing information about the best clustering model
    :param save: Whether to save the plots as images.
    :return:
    """
    # Get cluster labels
    labels = _get_labels(data=data, model_name=model_name, best_model_info=best_model_info)

    # Determine the number of unique clusters
    n_clusters = len(set(labels))
    max_clusters = 20

    # Sort the labels in descending order
    sorted_labels = sorted(labels, reverse=True)

    cmap = plt.cm.get_cmap('tab20')
    plt.figure(figsize=(16, 6))
```

```python
    # Handle cases: number of clusters more or less than max_clusters
    if n_clusters <= max_clusters:
        for i in range(n_clusters):
            cls_labels = [label for label in labels if label == i]
            color = i % cmap.N
            plt.hist(cls_labels, bins=[i-0.5 for i in range(n_clusters + 1)], color=cmap(color), label=f'Cluster {i}')
    else:
        # Plot the highest max_clusters clusters
        for i in range(max_clusters - 1):
            cls_labels = [label for label in sorted_labels if label == i]
            color = i % cmap.N
            plt.hist(cls_labels, bins=[i - 0.5 for i in range(max_clusters)], color=cmap(color), label=f'Cluster {i}')

        # Plot the remaining cluster
        other_clusters = [label for label in sorted_labels if label >= max_clusters - 1]
        color = (max_clusters - 1) % cmap.N
        plt.hist(other_clusters, bins=[max_clusters-1-0.5, max_clusters-0.5], color=cmap(color), label='Others')

    plt.xlabel('Cluster')
    plt.ylabel('Frequency')
    plt.title(f'Cluster Frequencies: {n_clusters} clusters')
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.tight_layout()

    # Save the image
    if save:
        if not os.path.exists(get_images_dir()):
            os.mkdir(get_images_dir())
        file_name = os.path.join(get_images_dir(), f"{model_name}_cluster_frequencies.png")
        plt.savefig(file_name)

    plt.show()
```

3.7.2 Cluster Composition Analysis

Function:

`plot_cluster_composition(data, model_name, best_model_info, save=False)`

- Purpose: This function calculates and displays cluster composition analysis. It shows the probabilities of clusters being assigned to each actual class label. Additionally, it identifies clusters that are proficient at recognizing specific classes.
- Usage: Cluster composition analysis provides insights into how well the clustering model separates data points based on their actual class labels.

```python
def plot_cluster_composition(data: Dataset, model_name: str, best_model_info: dict, save: bool = False):
    """
    Plot the composition of clusters in terms of actual digit labels.
    Calculates and displays cluster composition analysis, including probabilities of clusters assigned to each digit.
    Note: the percentage for a cluster to be considered correctly able to recognise a digit is set to 50%
    :param data: dataset
    :param model_name: name of the clustering model
    :param best_model_info: dictionary containing information about the best clustering model
    :param save: Whether to save the plots as images.
    :return:
    """
    # Get cluster labels and actual labels
    best_labels = _get_labels(data=data, model_name=model_name, best_model_info=best_model_info)
    actual_labels = data.y

    # Calculate the confusion matrix
    confusion = confusion_matrix(best_labels, actual_labels)
    # Normalize the confusion matrix to get probabilities
    cluster_probs = confusion / confusion.sum(axis=1, keepdims=True)

    # Display cluster-to-digit probabilities in a DataFrame if MeanShift model
    # Otherwise, plot a heatmap
    if model_name == "MeanShift":
        cluster_df = pd.DataFrame(cluster_probs[:, :10])
        cluster_df = cluster_df.applymap(lambda x: f'{x:.3f}')
        cluster_df.columns = [f'Digit {i}' for i in range(0, 10)]
        cluster_df.index.name = "Cluster"
        display(cluster_df)
    else:
        plt.figure(figsize=(8, 10))
        sns.heatmap(cluster_probs[:, :10], annot=True, fmt=".3f", cmap='inferno', square=True)
        plt.xlabel('Actual Digit')
        plt.ylabel('Cluster')
        plt.title('Cluster Composition Analysis (Probabilities)')
```

```python
        # Save the image
        if save:
            if not os.path.exists(get_images_dir()):
                os.mkdir(get_images_dir())
            file_path = os.path.join(get_images_dir(), f"{model_name}_cluster_composition.png")
            plt.savefig(file_path)

        plt.show()

    # Calculate percentage of clusters focused on each digit (0 to 9)
    digit_focus = (cluster_probs[:, :10] >= 0.5).mean(axis=0) * 100  # set to 50%
    underperforming_pct = 100 - digit_focus.sum()

    print("Percentage of clusters focused on each digit:")
    for digit, pct in enumerate(digit_focus):
        print(f"For digit {digit}: {pct:.3f}%")

    print(f"Underperforming Clusters (distributed across multiple digits): {underperforming_pct:.3f}%")
```

3.7.3 Reconstructed Images

Function:

`plot_reconstructed_images(data, model_name, best_model_info)`

- Purpose: This function visualizes reconstructed images for each cluster using the original data points within each cluster. It helps assess the quality of clustering by displaying representative images for each cluster.
- Usage: This plot aids in understanding the characteristics of the data within each cluster and the effectiveness of the clustering algorithm.

```python
def plot_reconstructed_images(data: Dataset, model_name: str, best_model_info: dict):
    """
    Plot some reconstructed images for each cluster using the original data points in each cluster.
    :param data: dataset
    :param model_name: name of the clustering model
    :param best_model_info: dictionary containing information about the best clustering model
    :return:
    """
    # Get cluster labels
    best_labels = _get_labels(data=data, model_name=model_name, best_model_info=best_model_info)

    # Fit and reduce dataset dimensionality using PCA with the specified number of components on the original data
    pca = PCA(n_components=best_model_info['n_components'])
    pca.fit(data.x)
    data_pca = pca.transform(data.x)

    unique_clusters = np.unique(best_labels)  # Get unique clusters
    max_clusters = 20  # Set the maximum number of clusters to visualize
```

```python
    # Loop over clusters and visualize images for each cluster
    for idx, cluster_id in enumerate(unique_clusters):
        if idx >= max_clusters:
            print(f"Cluster visualization limit reached. Remaining clusters won't be displayed.")
            break

        # Find the indices of data points belonging to the current cluster
        cluster_indices = np.where(best_labels == cluster_id)[0]
        cluster_data = data_pca[cluster_indices]  # Extract original data points

        # Select a random subset of data points to display
        n_images_to_display = min(3, len(cluster_data))  # Limit to the number of available images
        random_indexes = random.sample(range(len(cluster_data)), n_images_to_display)
        random_data = cluster_data[random_indexes]

        # Perform PCA inverse transform on the random subset to get original data points
        random_original_data_points = pca.inverse_transform(random_data)

        # Visualize the reconstructed images
        fig, axes = plt.subplots(1, n_images_to_display, figsize=(10, 2))

        for i, ax in enumerate(axes):
            ax.imshow(random_original_data_points[i].reshape(28, 28), cmap='plasma')
            ax.axis('off')

        plt.suptitle(f"Cluster {cluster_id}", fontsize=16)
        plt.show()
```

3.7.4 Cluster Means

Function:

`plot_model_means(data, model_name, best_model_info, save=False)`

- Purpose: This function plots the mean of each cluster in the dataset using PCA. It provides insight into the average representation of data points within each cluster in the reduced feature space.
- Usage: Visualizing cluster means can help in interpreting the structure and characteristics of each cluster, aiding in model evaluation.

```python
def plot_model_means(data: Dataset, model_name: str, best_model_info: dict, save: bool = False):
    """
    Plot the mean of each cluster in the dataset using PCA.
    :param data: dataset
    :param model_name: name of the clustering model
    :param best_model_info: dictionary containing information about the best clustering model
    :param save: Whether to save the plots as images.
    :return:
    """
    # Get cluster labels using the best model
    best_labels = _get_labels(data=data, model_name=model_name, best_model_info=best_model_info)

    # Fit and reduce dataset dimensionality using PCA with the specified number of components on the original data
    pca = PCA(n_components=best_model_info['n_components'])
    pca.fit(data.x)
    data_pca = pca.transform(data.x)

    unique_clusters = np.unique(best_labels)  # Get unique clusters
    num_clusters_to_plot = min(20, len(unique_clusters))  # Limit to 20 or the number of clusters
    num_images_per_row = 3  # Number of cluster means to display in each row of subplots

    # Calculate the number of rows for and create subplot layout
    rows, cols = (num_clusters_to_plot + (num_images_per_row - 1)) // num_images_per_row, num_images_per_row
    fig, axes = plt.subplots(rows, cols, figsize=(10, 3 * rows))
```

```python
    for idx, cluster_id in enumerate(unique_clusters[:num_clusters_to_plot]):
        row = idx // num_images_per_row
        col = idx % num_images_per_row

        # Find the indices of data points belonging to the current cluster
        cluster_indices = np.where(best_labels == cluster_id)[0]
        cluster_data_pca = data_pca[cluster_indices]  # Extract transformed data points

        # Calculate the mean of the cluster in the PCA space
        cluster_mean = np.mean(cluster_data_pca, axis=0)

        # Inverse transform the cluster mean to the original data space
        reconstructed_mean = pca.inverse_transform(cluster_mean)

        ax = axes[row, col] if rows > 1 else axes[col]
        ax.imshow(reconstructed_mean.reshape(28, 28), cmap='viridis')
        ax.set_title(f"Cluster {cluster_id} Mean")
        ax.axis('off')

    # Hide any remaining empty subplots
    for idx in range(len(unique_clusters), rows * num_images_per_row):
        row = idx // num_images_per_row
        col = idx % num_images_per_row
        axes[row, col].axis('off')

    if len(unique_clusters) > num_clusters_to_plot:
        print("Note: Only showing the first 20 clusters. Rest are not displayed.")

    plt.tight_layout()
```

```
    # Save the image
    if save:
        if not os.path.exists(get_images_dir()):
            os.mkdir(get_images_dir())
        file_path = os.path.join(get_images_dir(), f"{model_name}_cluster_means.png")
        plt.savefig(file_path)

    plt.show()
```

4 Mean Shift

4.1 Theoretical Background

Mean-Shift clustering is a powerful non-parametric algorithm used for discovering clusters within a dataset without making any assumptions about the data's underlying distribution. It operates by iteratively shifting data points towards the mode of the estimated probability density function (PDF) derived from the data. This mode represents the densest region of the data, which, in turn, corresponds to the center of a cluster. The Mean-Shift algorithm initiates with an initial set of data points and iteratively updates their positions by shifting them toward the mean of the data points within a certain distance, known as the bandwidth, from each point. This iterative process continues until convergence, at which point each data point is assigned to its corresponding cluster based on its final position. Mean-Shift clustering excels in identifying clusters with irregular shapes and varying densities.

4.1.1 Mean-Shift Algorithm Overview

Consider a set of points in a two-dimensional space. Imagine a circular window centered at C with a kernel k_h of radius h. The Mean-Shift algorithm can be described as a hill-climbing technique, where this kernel is shifted iteratively towards higher-density regions until convergence is achieved. At each iteration, the kernel k_h is shifted towards the centroid or mean of the points falling within it, i.e., the points whose distance from the center is less than or equal to h.

The direction and magnitude of each shift are determined by a mean shift vector $m(x)$, which always points towards the direction of maximum increase in density. When using a Gaussian kernel instead of a flat one, each point is assigned a weight that decays exponentially with its distance from the kernel's center, following the Gaussian kernel's profile.

At convergence, there exists no direction in which a further shift can accommodate more points inside the kernel.

4.1.2 Mathematical Definition

Mean-Shift is a procedure for finding the maxima (modes) of a density function given discrete data sampled from that function. This iterative process begins with an initial estimate x. A kernel function $K(x_i - x)$ is employed to determine the weights of nearby points for re-estimating the mean. Typically, a Gaussian kernel based on the distance to the current estimate is used, defined as $K(x_i - x) = e^{-c\|x_i - x\|^2}$, where c is the scaling factor (bandwidth) of the kernel, often dependent on the standard deviation σ.

The weighted mean of the density within the window defined by $N(x)$ is calculated as follows:

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

Here, $N(x)$ represents the neighborhood of x, comprising the set of points for which $K(x_i - x) \neq 0$.

The difference $m(x) - x$ is termed the mean shift, and at each iteration, the Mean-Shift algorithm updates x as $x \leftarrow m(x)$, repeating this estimation until $m(x)$ converges.

Mean-Shift is categorized as an Expectation-Maximization Algorithm, meaning it is iterative and locally optimal, with sensitivity to initialization states.

4.1.2.1 Kernel Smoother

The Mean-Shift method relies on the application of a Gaussian kernel smoother to calculate the mean point $m(x)$ within the neighborhood of bandwidth h.

4.1.3 Algorithm Convergence

Mean-Shift is an exclusive machine learning algorithm where, at each iteration, a window shaped by the kernel type is moved to the centroid of the data. It follows a hill-climbing approach, progressively approaching a local maximum. The algorithm uses a window, and at each step, it analyzes the inner part of this window to find the center of mass.

The algorithm converges when the gradient of the kernel density reaches zero, indicating a local maximum. The gradient is computed as follows:

$$\nabla f(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{x_i - x}{h}\right)(x_i - x)$$

The mean shift vector is then computed as:

$$m(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{x_i - x}{h}\right) x_i$$

At each step, after calculating the mean shift vector, a new window is generated as:

$$x_{t+1} = x_t + m(x_t)$$

The new window is generated by adding the old window to the return value of the mean shift function given the old window.

To identify all the necessary clusters, this process can be parallelized by generating multiple windows, each with a random center within the feature space.

The size of the window (bandwidth) depends on the type of kernel applied by the algorithm. Typically, a smaller bandwidth results in more data points being considered peaks, while a larger kernel size leads to fewer clusters.

4.2 Implementation

4.2.1 MeanShift Clustering Model Evaluation

In this section, we will discuss the evaluation of MeanShift clustering models using a combination of Principal Component Analysis (PCA) dimensions and hyperparameter bandwidth values. The goal is to understand how different configurations of PCA dimensions and bandwidth values affect the performance of the MeanShift clustering algorithm.

4.2.1.1 Model Configuration

To perform this evaluation, we define a Python class named `MeanShiftEvaluation`. This class is responsible for configuring and evaluating MeanShift clustering models using various settings. Here is an overview of the key components:

- **Data**: We use a dataset, denoted as `data`, as the input for evaluation.
- **PCA Dimensions**: We consider a range of PCA dimensions, specified as `n_components`. These dimensions are used to reduce the dimensionality of the input data.
- **Bandwidth Values**: A list of hyperparameter values, denoted as `hyperparam_vals`, is provided. These values are used to control the bandwidth parameter of the MeanShift algorithm.

4.2.1.2 Class Definition

The `MeanShiftEvaluation` class is defined as follows:

```python
class MeanShiftEvaluation(ClusteringModel):
    """
    Class for evaluating MeanShift clustering models using combination of
    PCA dimensions and hyperparameter bandwidth values.
    """
    model = "MeanShift"
    hyperparameter_name = "bandwidth"

    def __init__(self, data: Dataset, n_components: List[int], hyperparam_vals: List[int | float]):
        """
        Initialize a MeanShift evaluation instance using SpectralClustering.
        :param data: The dataset for evaluation.
        :param n_components: List of PCA dimensions to evaluate.
        :param hyperparam_vals: List of hyperparameter values to evaluate.
        """
        super().__init__(data, n_components, hyperparam_vals)
        self.hyperparameter = self.hyperparameter_name
        self.model_name = self.model
        self.model_type = MeanShift(n_jobs=N_JOBS)
```

In this class definition:

- We specify the model name as "MeanShift" and the hyperparameter name as "bandwidth."
- The constructor `__init__` initializes the evaluation instance with the dataset, PCA dimensions, and hyperparameter values.
- We set the model type to use the MeanShift clustering algorithm with parallel processing (`n_jobs`).

The goal of this evaluation is to assess how the combination of PCA dimensionality reduction and varying bandwidth values impacts the performance of the MeanShift clustering model. The results will provide insights into the suitability of MeanShift for different datasets and settings.

4.3 Results

4.3.1 Random Index Score vs PCA Dimension

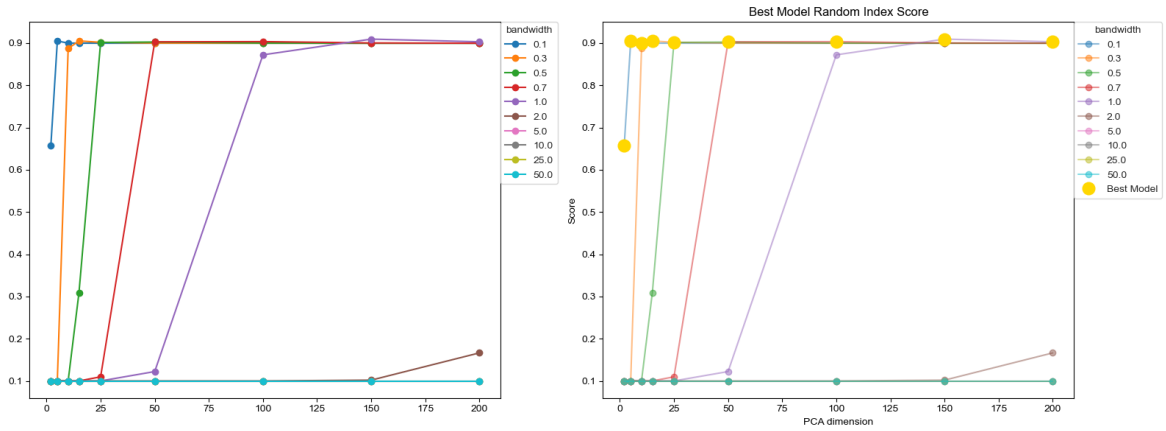
In this subsection, we present the visualization of the "Random Index Score" as a function of PCA dimension. The Random Index Score is a metric used to evaluate the quality of clustering results, with higher scores indicating better clustering performance.

Plot Description

The plot consists of two side-by-side line graphs. The first graph displays the Random Index Score for various PCA dimensions, while the second graph highlights the best model results by marking them in gold. Each point on the graphs corresponds to a specific PCA dimension, and the Random Index Score is represented on the y-axis.

Interpretation

This visualization allows us to observe how the Random Index Score changes as we vary the dimensionality of PCA. The highlighted points in gold indicate the PCA dimensions where the clustering models perform optimally in terms of the Random Index Score. By comparing the two graphs, we can identify the PCA dimensions that yield the best clustering results.



As observed from the positions of the best models across various PCA dimensions, it becomes evident that the models with a hyperparameter value within the range of 0.1 to 1.0 for `bandwidth` exhibit notably higher random index scores.

4.3.2 Number of Clusters vs PCA Dimension

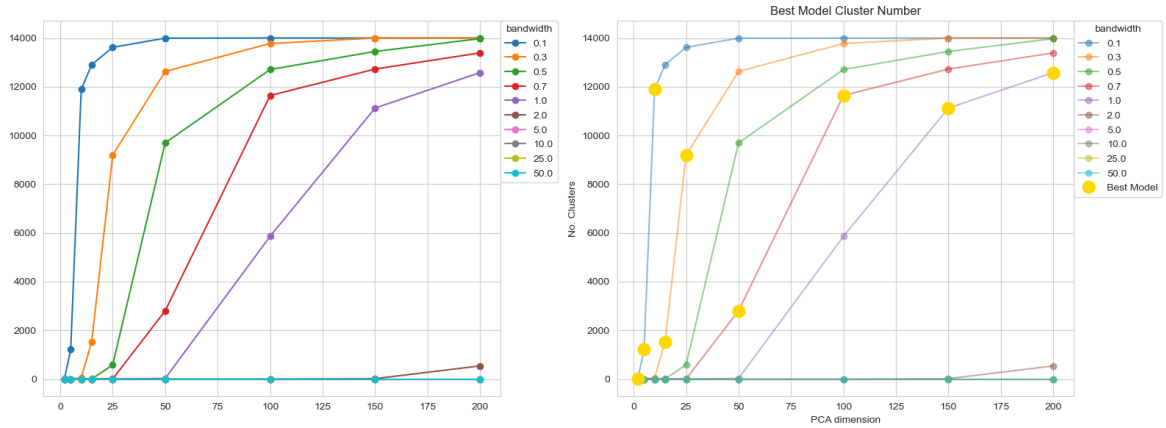
In this subsection, we present the visualization of the "Number of Clusters" as a function of PCA dimension. The number of clusters is a crucial parameter in clustering models, as it determines the granularity of the clustering.

Plot Description

Similar to the previous plot, this visualization consists of two side-by-side line graphs. The first graph displays the number of clusters for various PCA dimensions, while the second graph highlights the best model results by marking them in gold. Each point on the graphs corresponds to a specific PCA dimension, and the number of clusters is represented on the y-axis.

Interpretation

By examining this plot, we can gain insights into how the choice of PCA dimension affects the determination of the number of clusters. The gold-highlighted points signify the PCA dimensions that lead to the optimal number of clusters according to our evaluation metric. This information helps in selecting the appropriate PCA dimension for achieving the desired level of granularity in clustering.



4.3.3 Execution Time vs PCA Dimension

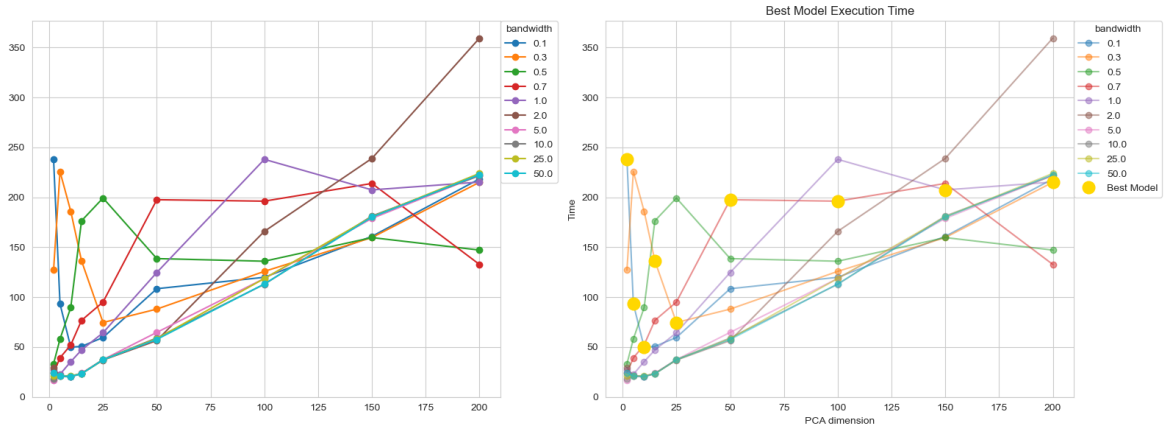
In this subsection, we present the visualization of "Execution Time" as a function of PCA dimension. Execution time is an important practical consideration, especially when dealing with large datasets and complex clustering algorithms.

Plot Description

The visualization consists of two side-by-side line graphs. The first graph displays the execution time for various PCA dimensions, while the second graph highlights the best model results by marking them in gold. Each point on the graphs corresponds to a specific PCA dimension, and the execution time is represented on the y-axis.

Interpretation

This plot provides insights into the computational efficiency of the clustering models across different PCA dimensions. The gold-highlighted points indicate the PCA dimensions where the models achieve the best trade-off between clustering performance and execution time. Analyzing these results helps in selecting an optimal PCA dimension that balances clustering quality with computational efficiency.



4.3.4 Best Model

The best model found for Meanshift is the following:

```
'model': MeanShift(bandwidth=1.0, n_jobs=-1),
'n_components': 150,
'bandwidth': 1.0,
'score': 0.9094231016501179,
'n_clusters': 11125,
'time': 207.4217698350003
```

4.3.5 Cluster Frequencies

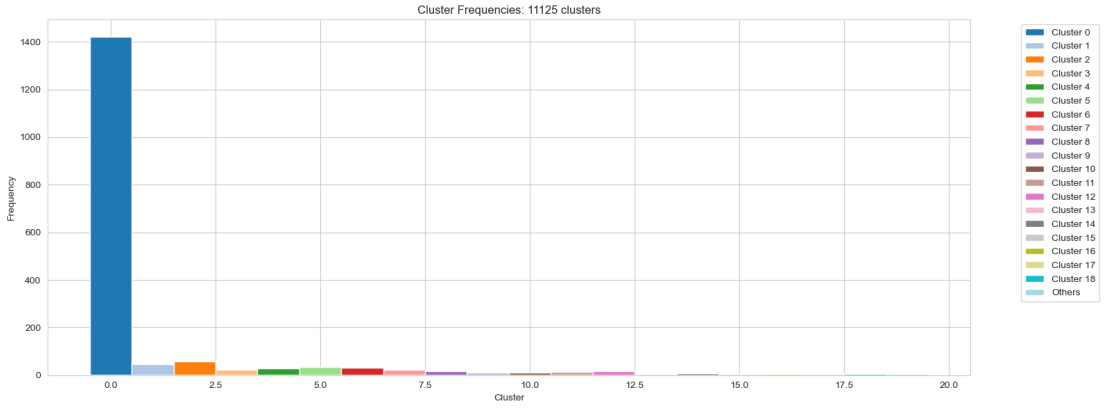
This section discusses the visualization of cluster frequencies in the dataset using the `plot_cluster_freque` function. The primary goal of this plot is to provide insights into how data points are distributed across different clusters produced by the clustering model.

Plot Description

The plot generated by this function displays a histogram that represents the distribution of data points across clusters. Each bar in the histogram corresponds to a specific cluster, and the height of the bar indicates the frequency (number of data points) in that cluster. To prevent clutter, only a limited number of clusters are displayed, with the option to show the top clusters or group the remaining clusters as "Others."

Interpretation

This visualization allows us to understand the composition of clusters in the dataset. By observing the distribution of data points across clusters, we can identify the clusters with the highest and lowest frequencies. It provides an overview of the effectiveness of the clustering model in segmenting the data into meaningful groups.



In this visualization, we observe a notable pattern in the MeanShift clustering results. Despite the generation of a vast number of clusters (11125 in total), one cluster stands out prominently with a high frequency of 1400 data points. This dominant cluster captures a significant portion of the dataset, while the majority of the remaining clusters exhibit much lower frequencies, each containing fewer than 100 data points. This distribution highlights an imbalance in cluster sizes, indicating that the clustering algorithm has identified a substantial and distinct group

within the data. Analyzing the composition of this dominant cluster and exploring the unique characteristics of the smaller clusters can offer valuable insights into the dataset's structure and diversity.

4.3.6 Cluster Composition Analysis

In this section, we explore the `plot_cluster_composition` function, which analyzes the composition of clusters in terms of actual digit labels. This analysis helps assess the model's ability to correctly group similar data points.

Plot Description

The function generates a cluster composition analysis, including probabilities of clusters assigned to each digit. It visualizes this information using a heatmap. Each row in the heatmap represents a cluster, while each column corresponds to an actual digit label (0 to 9). The values in the heatmap cells represent the probability that a data point in a cluster corresponds to a specific digit.

Interpretation

The heatmap provides valuable insights into how well the clustering model organizes data points based on their true digit labels. A high probability value indicates that the cluster is effective at recognizing and grouping data points of a particular digit. This analysis helps assess the model's accuracy in capturing digit-specific patterns within clusters.

[3]:
```
plot_cluster_composition(data=data, model_name=mean_shift_evaluation.model_name, best_model_info=best, save=True)
```

Cluster	Digit 0	Digit 1	Digit 2	Digit 3	Digit 4	Digit 5	Digit 6	Digit 7	Digit 8	Digit 9
0	0.000	0.983	0.000	0.000	0.003	0.000	0.000	0.009	0.000	0.005
1	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000
2	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.915	0.000	0.085
3	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
4	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000
...	...	...	...	...	...	...	...	...	...	...
11120	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000
11121	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000
11122	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
11123	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000
11124	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

11125 rows × 10 columns

```
Percentage of clusters focused on each digit:
For digit 0: 11.164%
For digit 1: 0.539%
For digit 2: 13.160%
For digit 3: 12.467%
For digit 4: 11.622%
For digit 5: 11.074%
For digit 6: 10.634%
For digit 7: 8.306%
For digit 8: 11.928%
For digit 9: 9.429%
Underperforming Clusters (distributed across multiple digits): -0.324%
```

Since the Means Shift model has a lot of clusters, it was decided to provide the composition in percentages for an easier view.

4.3.7 Cluster Means

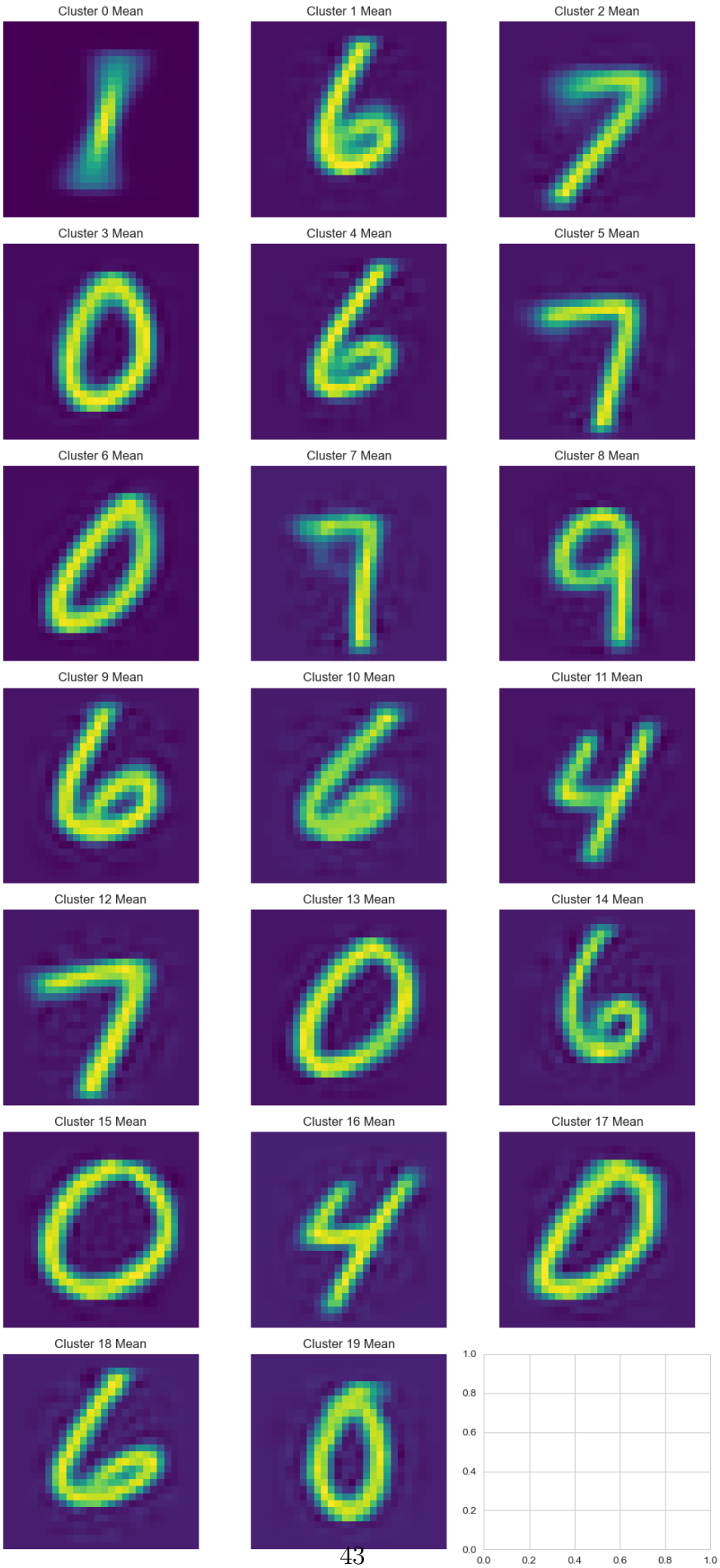
In this section, we introduce the `plot_model_means` function, which visualizes the mean of each cluster in the dataset using PCA. This plot helps us explore the centroids or representative points of each cluster.

Plot Description

The function calculates the mean of each cluster in the PCA space and then performs an inverse transform to display the reconstructed images of these cluster means. The result is a grid of images, where each row represents a cluster, and each column shows the mean image of a cluster.

Interpretation

The cluster means plot allows us to examine the average characteristics of each cluster. By visualizing the mean images, we can gain insights into the typical features and patterns captured by the clustering model for each group of data points. This analysis helps us understand the cluster representations and their significance.


Cluster 0 Mean
Cluster 1 Mean
Cluster 2 Mean
Cluster 3 Mean
Cluster 4 Mean
Cluster 5 Mean
Cluster 6 Mean
Cluster 7 Mean
Cluster 8 Mean
Cluster 9 Mean
Cluster 10 Mean
Cluster 11 Mean
Cluster 12 Mean
Cluster 13 Mean
Cluster 14 Mean
Cluster 15 Mean
Cluster 16 Mean
Cluster 17 Mean
Cluster 18 Mean
Cluster 19 Mean
0.0
0.2
0.4
0.6
0.8
1.0

As mentioned before, we have many clusters (11125) where one specific cluster stands out as the dominant group. As a consequence, the mean image of this cluster appears somewhat blurred, reflecting the aggregation of diverse data points. In contrast, the means of the remaining clusters appear notably sharper, representing their more homogeneous composition. Due to the high count of clusters, only the first 20 were shown.

5 Normalized Cut

5.1 Theoretical Background

5.1.1 Introduction

Normalized-Cut is a divisive hierarchical algorithm that operates on datasets represented as graphs. In this method, the points within the feature space are treated as nodes, and the distances between each pair of points are represented as weighted edges. The algorithm aims to partition the graph into clusters by minimizing the normalized cut value. This criterion seeks to minimize dissimilarity within clusters while maximizing dissimilarity between clusters, taking into account the sizes and densities of the clusters.

5.1.2 Main Objective

The primary goal of Normalized-Cut is to divide the graph into clusters while minimizing the normalized cut value, denoted as $NCut(A, B)$. This value is defined as:

$$NCut(A, B) = \left(\frac{1}{Vol(A)} + \frac{1}{Vol(B)}\right) \cdot cut(A, B) = \frac{cut(A, B)}{Vol(A)} + \frac{cut(A, B)}{Vol(B)}$$

Here, $cut(A, B)$ represents the sum of edge weights connecting data points between clusters A and B, and $Vol(A)$ and $Vol(B)$ denote the total weights of edges connecting data points within clusters A and B, respectively.

Constraint Relaxation

The division of clusters using $NCut$ helps overcome the issue of isolated nodes. When a cluster, such as B, becomes an isolated node, the ratio $\frac{cut(A,B)}{Vol(B)}$ becomes very high because $Vol(B)$ is small. This indicates an unfavorable cut.

5.1.3 Formally - Optimization Problem

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

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

The minimization of $Ncut(x)$ can be expressed using the Rayleigh Quotient as:

$$\min_x NCut(x) = \min_y \frac{y^T(D - W)y}{y^T Dy}$$

subject to the constraint:

$$y^T D1 = \sum_i y_i d_i = 0$$

Here, 1 represents an $|V|$-dimensional vector of ones, and $y_i \in \{1, -b\}$ for some b. This constraint relaxation helps make the problem more tractable.

NP-Completeness

The minimization problem is NP-Hard, necessitating alternative strategies for solving it.

5.1.4 Normalized-Cut as (Relaxed) Eigen-System

By relaxing the constraint and allowing y to be a real-valued vector, the optimization problem can be rewritten as:

$$\min_y y^T(D - W)y \quad \text{subject to} \quad y^T Dy = 1$$

This formulation leads to solving the eigenvalue problem $(D - W)y = \lambda Dy$ after applying the Lagrange multipliers method.

5.1.5 Recursive 2-Way Normalized-Cut

To achieve graph partitioning, we focus on the second smallest eigenvalue and associated eigenvector to perform a binary partition. The process can be recursively applied, generating a binary tree of partitions. The algorithm stops when the normalized cut value falls below a specified threshold.

Pros and Cons

The advantage of this approach is that it doesn't require prior knowledge of the number of clusters. However, it is computationally expensive since eigenpairs need to be calculated for each partition, with a complexity of approximately $O(n^{2.4})$.

5.1.6 Smallest k Eigenvectors and k-Means

An alternative approach involves constructing a matrix $U = [u_1, \ldots, u_k] \in \mathbb{R}^{n \times k}$ containing the smallest k eigenvectors of the Laplacian matrix, excluding the eigenvector associated with eigenvalue 0. These eigenvectors are used as columns in the matrix U. The data points can be represented as rows in this matrix, and clustering can be performed using k-means.

Pros and Cons

This approach requires computing eigenpairs only once, reducing computational cost. However, it necessitates specifying the number of clusters, k, as in k-means.

Partitioning Strategies

Graph partitioning can be achieved by either setting a threshold k or by evaluating n candidate splitting points in the eigenvector x and choosing the one with the minimum normalized cut value.

Implications of 0 Being an Eigenvalue

The presence of eigenvalue 0 signifies a trivial partition where $A = V$ and $B = \emptyset$. Thus, the focus is on the second smallest eigenvalue for meaningful partitioning.

5.1.7 Spectral Clustering

Spectral clustering is a powerful technique used for clustering data based on the spectral properties of a similarity or affinity matrix. It is particularly effective when dealing with complex datasets that don't have well-separated clusters in the original feature space.

5.1.7.1 Algorithm Overview

The key idea behind spectral clustering is to transform the data into a new space represented by the eigenvectors of a chosen affinity matrix. The standard steps of spectral clustering include:

1. Constructing an affinity matrix, typically based on pairwise distances between data points.
2. Computing the Laplacian matrix, which is used to capture the relationships between data points.
3. Calculating the eigenvectors and eigenvalues of the Laplacian matrix.
4. Selecting the top k eigenvectors (excluding the one corresponding to eigenvalue 0) to form a new data representation matrix.
5. Applying traditional clustering algorithms like k-means to this new representation to obtain the final clusters.

5.1.7.2 Advantages and Use Cases

Spectral clustering has several advantages:

- It can discover clusters of various shapes and sizes, making it suitable for complex data.
- It does not rely on specific assumptions about cluster shapes or densities.
- It allows for dimensionality reduction by selecting a smaller number of eigenvectors.

Spectral clustering is commonly used in image segmentation, community detection in social networks, and various other fields where traditional clustering algorithms may struggle.

5.1.7.3 Limitations

Despite its strengths, spectral clustering has some limitations:

- Choosing the appropriate affinity matrix and number of clusters (parameter k) can be challenging.
- It may not perform well on very large datasets due to the computation of eigenvectors.
- The spectral embedding step can be sensitive to noise in the data.

Normalized-Cut clustering leverages graph representation and spectral properties to partition datasets into meaningful clusters. While it offers advantages such as not requiring the number of clusters to be predefined, it comes with the drawback of computational cost and sensitivity to threshold values.

5.2 Implementation

5.2.1 Normalized Cut Clustering Model Evaluation

In this section, we will discuss the evaluation of Normalized Cut clustering models using a combination of Principal Component Analysis (PCA) dimensions and hyperparameter values for the number of clusters (`n_clusters`). The objective is to analyze the performance of the Normalized Cut algorithm under different settings and understand how the choice of PCA dimensions and cluster count affects clustering results.

5.2.1.1 Model Configuration

To conduct this evaluation, we utilize a Python class named `NormalizedCutEvaluation`. This class is responsible for configuring and assessing Normalized Cut clustering models with various configurations. Here are the key components of the model configuration:

- **Data**: We employ a dataset, referred to as `data`, as the input for evaluation.
- **PCA Dimensions**: A list of PCA dimensions, denoted as `n_components`, is considered for dimensionality reduction of the input data.
- **Cluster Count**: We evaluate different hyperparameter values for the number of clusters (`n_clusters`). These values are specified as `hyperparam_vals`.

Tuning

```
ncut_evaluation = NormalizedCutEvaluation(
    data=data,
    n_components=[2, 5, 10, 15, 25, 50, 100, 150, 200],
    hyperparam_vals=[x for x in range(5, 16)]
)
```

```
ncut_evaluation.load_results()
```

```
Loading /Users/a/GitHub/clustering/src/results/NormalizedCut_n_clusters_result.json
Loading /Users/a/GitHub/clustering/src/results/NormalizedCut_n_clusters_result_bestmodels.json
Loading /Users/a/GitHub/clustering/src/results/NormalizedCut_n_clusters_bestmodel.pkl
```

```
%%time
# ncut_evaluation.evaluate()
```

5.2.1.2 Class Definition

The `NormalizedCutEvaluation` class is defined as follows:

```
class NormalizedCutEvaluation(ClusteringModel):
    """
    Class for evaluating NormalizedCut clustering models using combination of
    PCA dimensions and hyperparameter n_clusters values.
    """
    model = "NormalizedCut"
    hyperparameter_name = "n_clusters"

    def __init__(self, data: Dataset, n_components: List[int], hyperparam_vals: List[int | float]):
        """
        Initialize a NormalizedCut evaluation instance using SpectralClustering.
        :param data: The dataset for evaluation.
        :param n_components: List of PCA dimensions to evaluate.
        :param hyperparam_vals: List of hyperparameter values to evaluate.
        """
        super().__init__(data, n_components, hyperparam_vals)
        self.hyperparameter = self.hyperparameter_name
        self.model_name = self.model
        self.model_type = SpectralClustering(n_jobs=N_JOBS, random_state=RANDOM_SEED)
```

In this class definition:

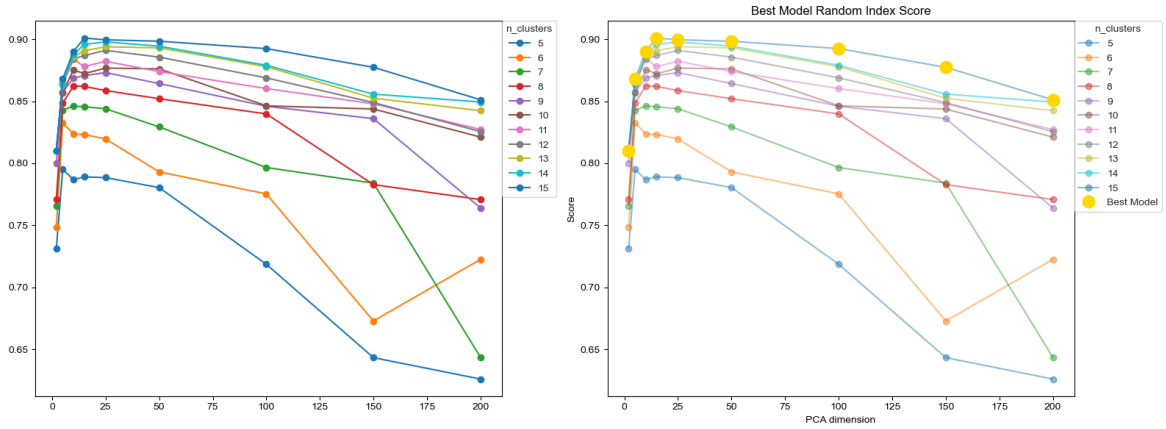
- We specify the model name as "NormalizedCut," and the hyperparameter name as `n_clusters`.
- The constructor `__init__` initializes the evaluation instance with the dataset, PCA dimensions, and hyperparameter values.

- We set the model type to use the SpectralClustering algorithm with parallel processing (`n_jobs`) and a specified random seed (`random_state`).

The purpose of this evaluation is to investigate how the combination of PCA dimensionality reduction and varying cluster counts affects the performance of the Normalized Cut clustering model. The results will provide insights into the suitability of Normalized Cut for different datasets and configurations.

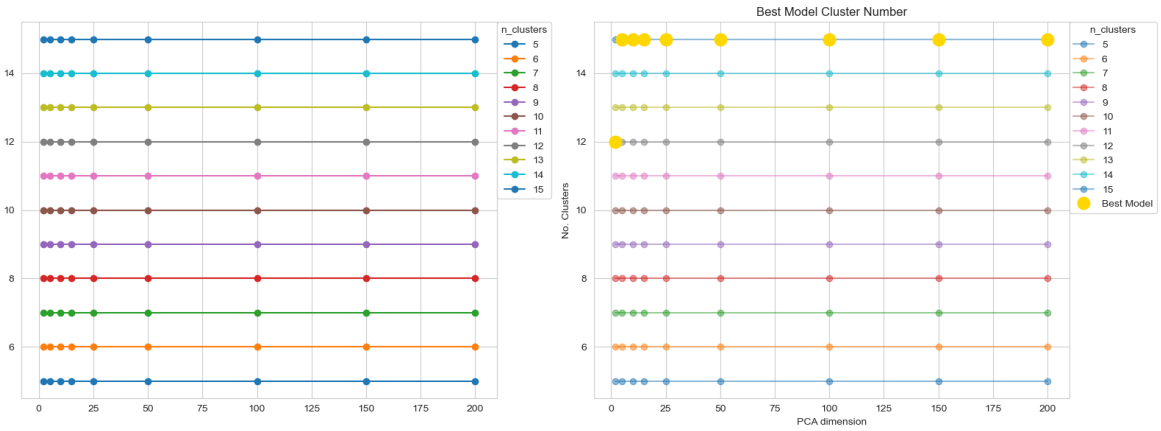
5.3 Results

5.3.1 Random Index Score vs PCA Dimension

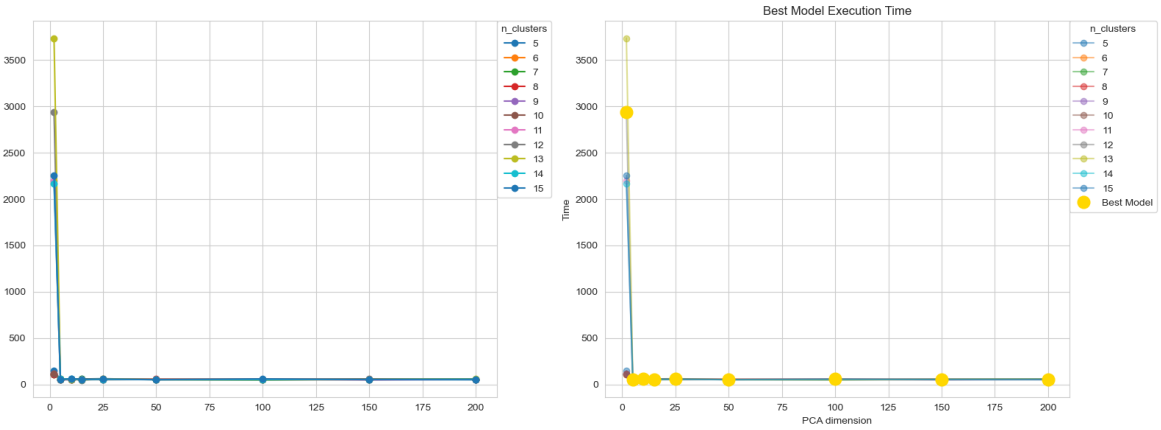


clustering models consistently achieve their highest Random Index Scores when the PCA dimension is set to 15. The Random Index Score serves as an essential metric for evaluating clustering quality, with higher scores indicating superior results. The repeated prominence of PCA dimension 15 suggests that this particular configuration excels in segmenting the dataset into meaningful clusters. A plausible explanation for this phenomenon lies in the intricacies of the dataset itself. It is conceivable that certain digits within the dataset are written in ways that challenge conventional clustering algorithms, leading to a more specialized segmentation requiring additional clusters for accurate representation.

5.3.2 Number of Clusters vs PCA Dimension



5.3.3 Execution Time vs PCA Dimension

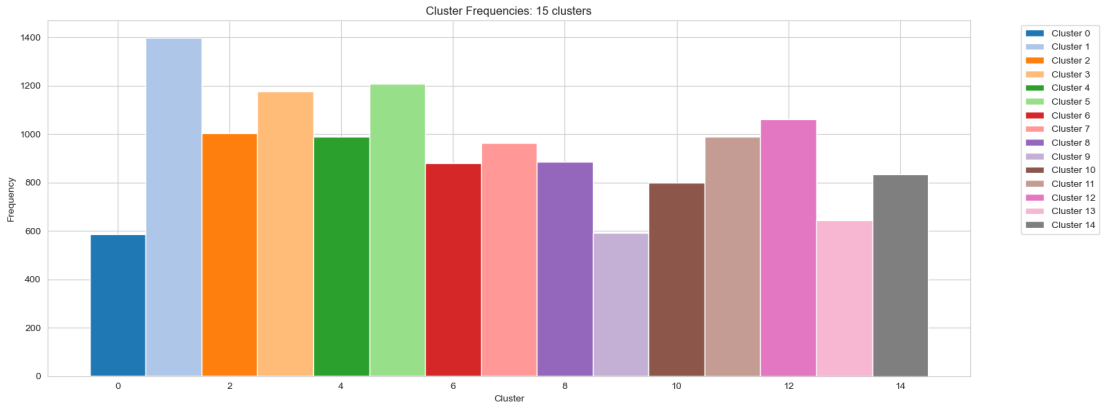


5.3.4 Best Model

The best model found for NormalizedCut is the following:

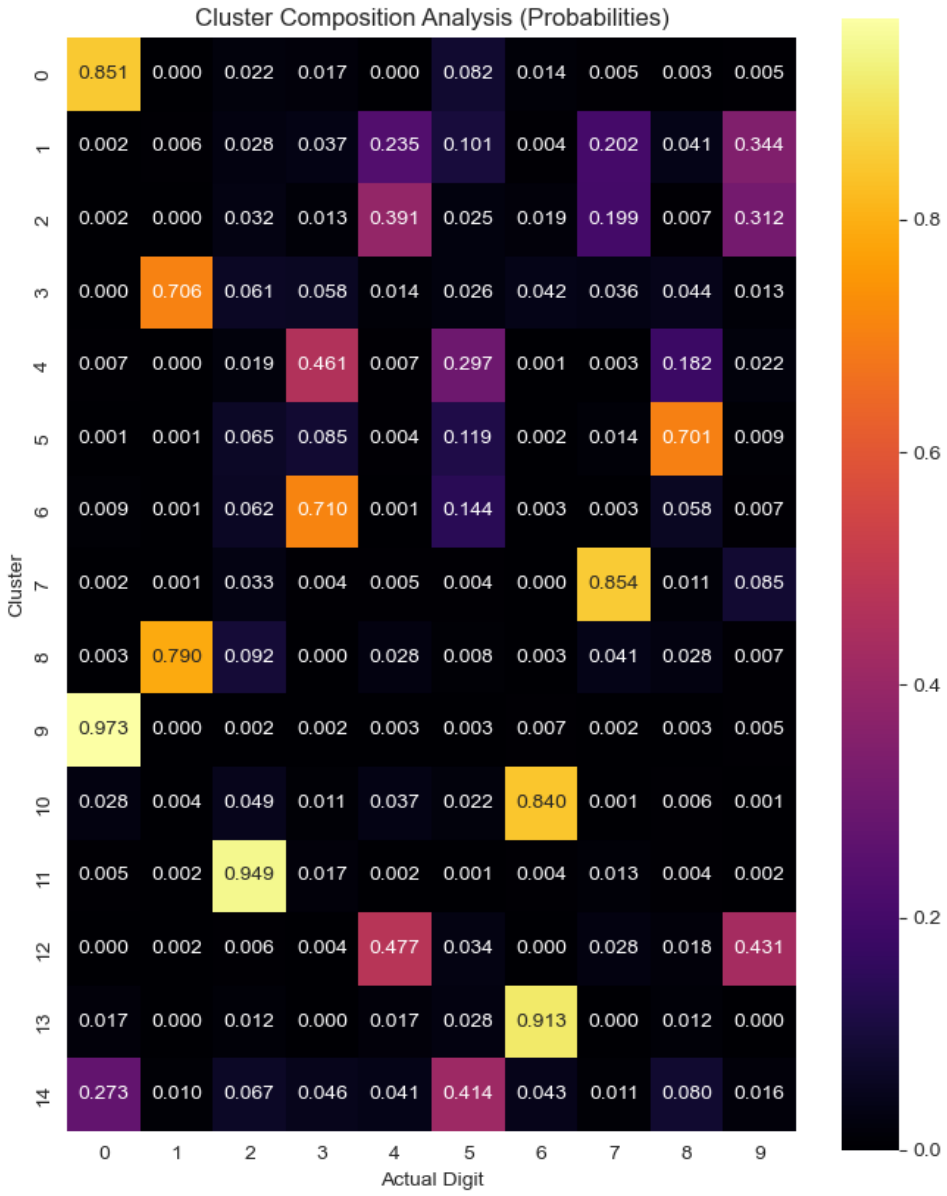
```
'model': SpectralClustering(n_clusters=15, n_jobs=-1, random_state=28),
'n_components': 15,
'n_clusters': 15,
'score': 0.9010313695876236,
'time': 54.89419081300002
```

5.3.5 Cluster Frequencies



In contrast to the MeanShift clustering results, the analysis of cluster frequencies using the NormalizedCut method reveals a distinct pattern. Here, the distribution of data points among the clusters displays a relatively uniform and high average frequency across all clusters. This observation implies that the NormalizedCut algorithm tends to create clusters of relatively similar sizes, resulting in a more balanced distribution of data points. The absence of prominently dominant or underrepresented clusters suggests that the NormalizedCut algorithm may be effective at partitioning the dataset into clusters of comparable sizes, making it particularly suitable for scenarios where achieving a balanced distribution of data points among clusters is a priority

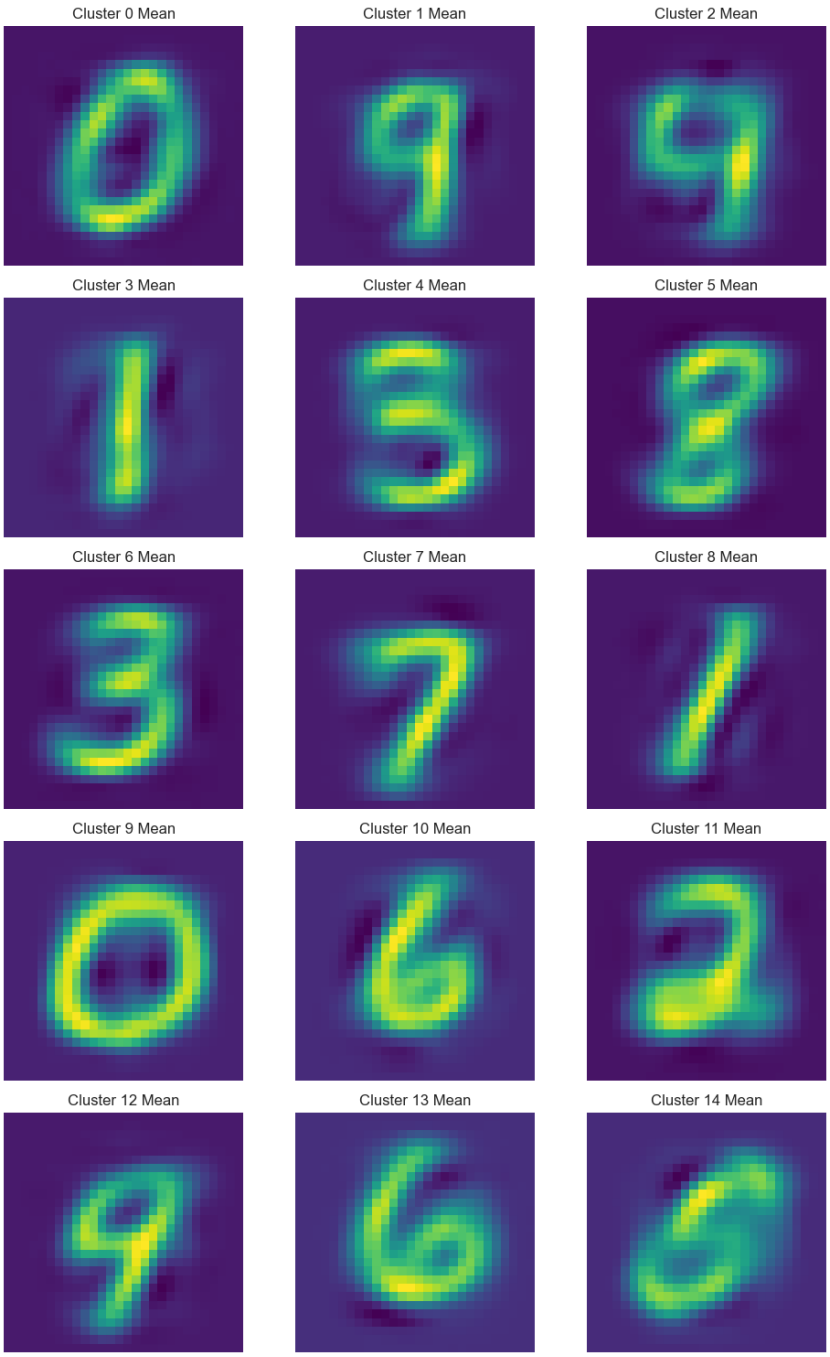
5.3.6 Cluster Composition Analysis



Notably, each of the 15 clusters appears to exhibit a distinctive recognition pattern for a specific digit, with high probability values associated with most digits. However, the presence of five clusters displaying probabilities close to 40% for two distinct digits suggests an interesting observation. This phenomenon may be attributed to the inherent similarity in the handwriting styles of these digit pairs, such as 3 and 5, 4 and 9, or 7. The clustering model seems to

encounter challenges when distinguishing between these visually similar digits, resulting in the co-occurrence of these pairs within certain clusters.

5.3.7 Cluster Means



Each cluster appears to exhibit a distinct specialization in recognizing a particular digit. The relatively sharp and recognizable mean images for each cluster signify that the model has effectively grouped data points that share similar digit patterns. However, the subtle blurring observed in some cluster means suggests a layer of versatility. It implies that these clusters might

not only be adept at recognizing their primary assigned digit but could also possess the capability to identify closely related digits that share visual similarities.

6 Gaussian Mixture

6.1 Theoretical Background

6.1.1 Gaussian Mixture Clustering

Gaussian Mixture Models (GMMs) are a versatile probabilistic modeling technique used for clustering and density estimation. They are particularly effective when dealing with complex datasets that don't adhere to simple geometric shapes. GMMs assume that the data is generated from a mixture of several Gaussian distributions, each representing a distinct cluster. This model allows data points to belong to multiple clusters simultaneously, making it a "soft clustering" approach.

6.1.1.1 Formal Representation

In a GMM, each data point is represented as a tuple (x_i, z_i), where $x_i \in \mathbb{R}^d$ is the observed data, and $z_i \in \{1, 2, \ldots, K\}$ is a discrete latent variable representing the cluster assignment. The joint probability function p can be defined as:

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

Here: - $p(z = k)$ represents the prior probability of data point x belonging to cluster k, typically denoted as π_k. - $p(x \mid z = k)$ represents the conditional probability of data point x given that it belongs to cluster k. This is modeled as a multivariate Gaussian distribution with mean μ_k and covariance matrix Σ_k.

The overall probability density function $p(x)$ is obtained by summing over all clusters:

$$p(x) = \sum_{k=1}^{K} p(x \mid z = k) \cdot p(z = k)$$

6.1.1.2 Expectation-Maximization Algorithm for GMMs

The key to learning GMMs from data lies in the Expectation-Maximization (EM) algorithm, which iteratively estimates the parameters. The EM steps for GMMs include:

1. Initialization: Initialize the model with random means, covariances, and mixing coefficients.
2. Expectation (E-step): Given the current parameters, compute the likelihood that each data point belongs to each cluster. This step involves calculating the responsibilities of each cluster for each data point.
3. Maximization (M-step): Update the model parameters (means, covariances, and mixing coefficients) using the computed responsibilities.

4. Repeat the E-step and M-step until convergence is reached or a maximum number of iterations is reached.

The EM algorithm aims to find the optimal Gaussian distributions that best represent the data.

6.1.1.3 Advantages and Use Cases

Gaussian Mixture Models offer several advantages:

- They can model complex data distributions with multiple clusters and varying shapes.
- GMMs provide a probabilistic framework for clustering, allowing for uncertainty in cluster assignments.
- They are widely used in applications such as image segmentation, speech recognition, and anomaly detection.

6.1.1.4 Limitations

However, GMMs have some limitations:

- The choice of the number of clusters (K) is often subjective and needs to be specified in advance.
- GMMs may not perform well on high-dimensional data due to the curse of dimensionality.
- Convergence to a global optimum in the EM algorithm is not guaranteed, so different initializations can lead to different solutions.

In practice, GMMs provide a flexible tool for modeling data with complex structures, and their effectiveness depends on the specific problem and dataset.

6.2 Implementation

6.2.1 Gaussian Mixture Clustering Model Evaluation

In this section, we delve into the evaluation of Gaussian Mixture clustering models. The evaluation is conducted by considering a combination of Principal Component Analysis (PCA) dimensions and hyperparameter values for the number of components (`n_components`). The primary objective is to assess the performance of Gaussian Mixture models under different configurations and explore the impact of varying PCA dimensions and cluster counts on clustering results.

6.2.1.1 Model Configuration

To carry out this evaluation, we employ a Python class named `MixtureGaussianEvaluation`. This class serves the purpose of configuring and evaluating Gaussian Mixture clustering models with different settings. Key components of the model configuration include:

- **Data**: We utilize a dataset, denoted as `data`, as the input for evaluation.
- **PCA Dimensions**: A list of PCA dimensions, represented as `n_components`, is considered for dimensionality reduction of the input data.

- **Cluster Count**: We evaluate various hyperparameter values for the number of components (`n_components`). These values are specified as `hyperparam_vals`.

Tuning

```
mg_evaluation = MixtureGaussianEvaluation(
    data=data,
    n_components=[2, 5, 10, 15, 25, 50, 100, 150, 200],
    hyperparam_vals=[x for x in range(5, 16)]
)
```

```
mg_evaluation.load_results()
```

```
Loading /Users/a/GitHub/clustering/src/results/GaussianMixture_n_components_result.json
Loading /Users/a/GitHub/clustering/src/results/GaussianMixture_n_components_result_bestmodels.json
Loading /Users/a/GitHub/clustering/src/results/GaussianMixture_n_components_bestmodel.pkl
```

```
%%time
# mg_evaluation.evaluate()
```

6.2.1.2 Class Definition

The `MixtureGaussianEvaluation` class is defined as follows:

```python
class MixtureGaussianEvaluation(ClusteringModel):
    """
    Class for evaluating GaussianMixture clustering models using combination of
    PCA dimensions and hyperparameter n_components values.
    """
    model = "GaussianMixture"
    hyperparameter_name = "n_components"  # refers to number of clusters, not PCA dimension

    def __init__(self, data: Dataset, n_components: List[int], hyperparam_vals: List[int | float]):
        """
        Initialize a GaussianMixture evaluation instance.
        :param data: The dataset for evaluation.
        :param n_components: List of PCA dimensions to evaluate.
        :param hyperparam_vals: List of hyperparameter values to evaluate.
        """
        super().__init__(data, n_components, hyperparam_vals)
        self.hyperparameter = self.hyperparameter_name
        self.model_name = self.model
        self.model_type = GaussianMixture(max_iter=1000, random_state=RANDOM_SEED)
```

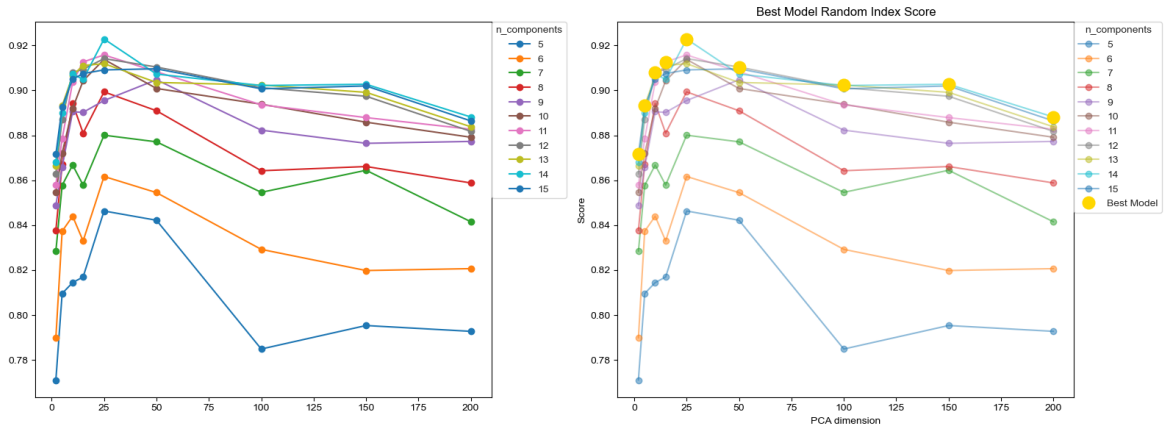
Within this class definition:

- We specify the model name as "GaussianMixture" and set the hyperparameter name as `n_components` to denote the number of components.
- The constructor `__init__` initializes the evaluation instance with the dataset, PCA dimensions, and hyperparameter values.
- The model type is set to use Gaussian Mixture with a maximum number of iterations (`max_iter`) and a specified random seed (`random_state`).

The goal of this evaluation is to investigate how the Gaussian Mixture clustering model performs under varying conditions, including different PCA dimensions and cluster counts. The results will provide insights into the suitability of Gaussian Mixture models for different datasets and configurations.

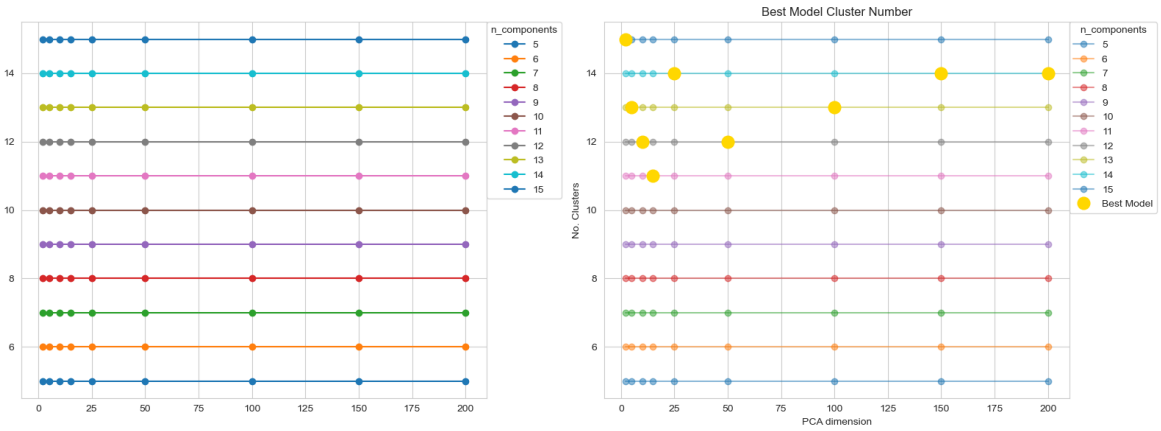
6.3 Results

6.3.1 Random Index Score vs PCA Dimension

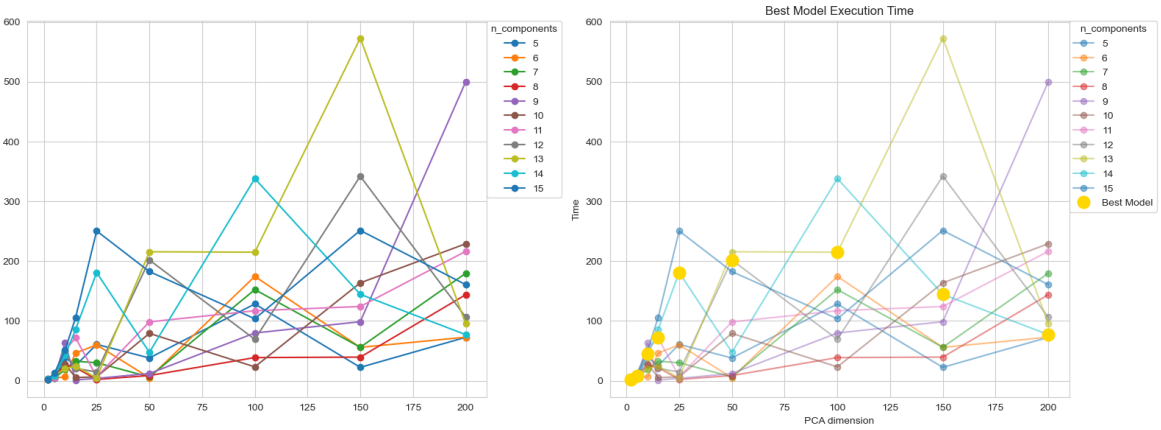


We can observe that clustering models tend to perform optimally with more than 10 PCA dimensions. The highlighted points, indicating the best-performing models, consistently emerge in higher-dimensional PCA spaces, suggesting improved clustering performance. This pattern hints at a possibility that the dataset may contain subtle variations in digit handwriting styles, resulting in clusters that specialize in distinguishing these nuances. The dominance of higher-dimensional PCA configurations in achieving better Random Index Scores implies their efficacy in capturing and discriminating these subtle variations.

6.3.2 Number of Clusters vs PCA Dimension



6.3.3 Execution Time vs PCA Dimension

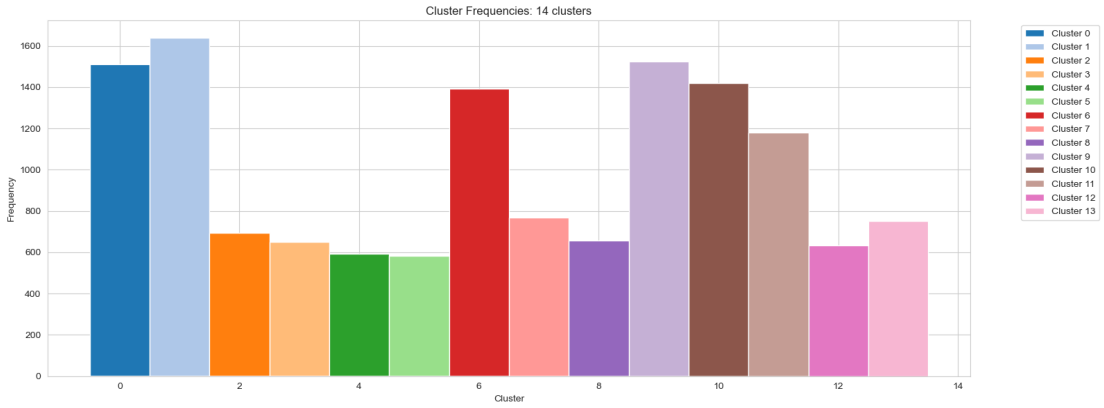


6.3.4 Best Model

The best model found for GaussianMixture is the following:

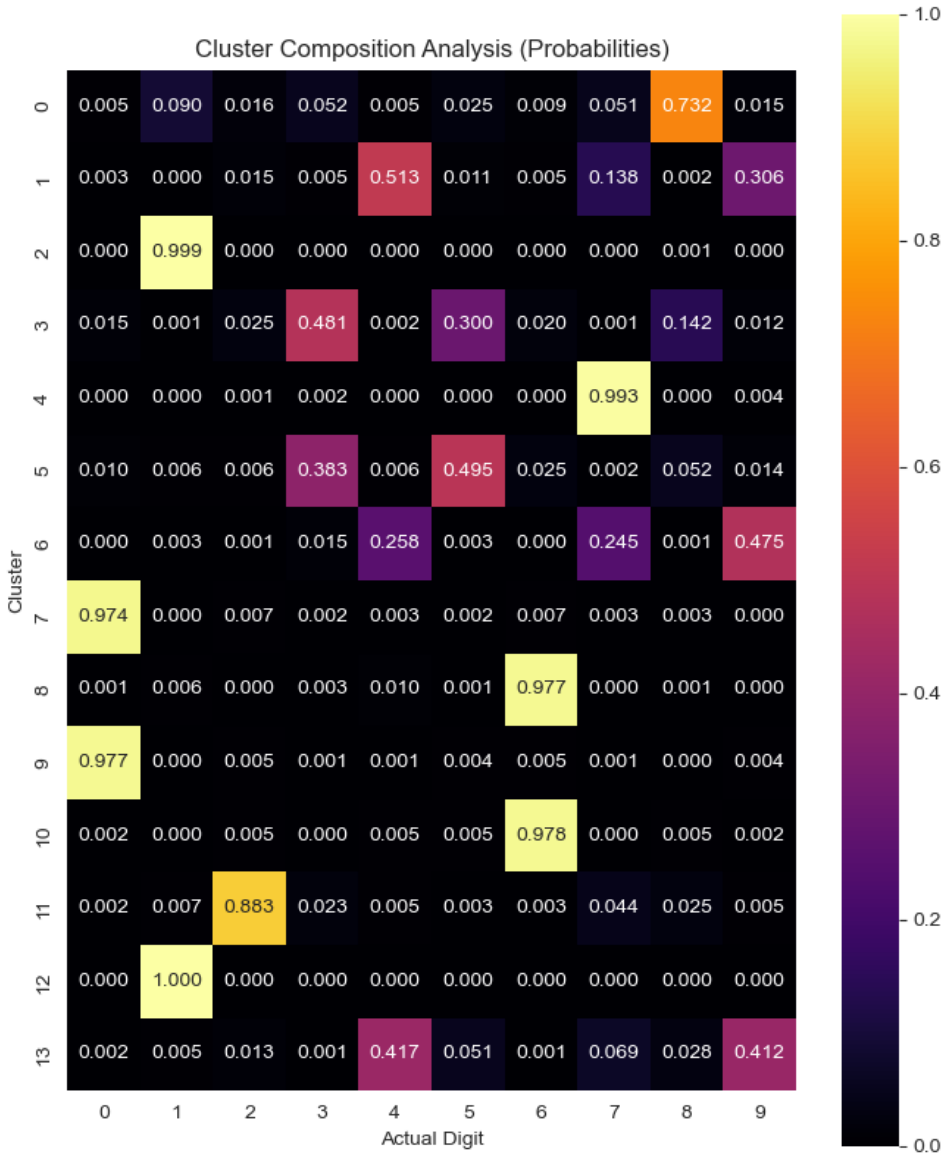
```
'model': GaussianMixture(max_iter=200, n_components=14, random_state=28),
'n_components': 14,
'score': 0.9228860122661823,
'n_clusters': 14,
'time': 180.9962048740001
```

6.3.5 Cluster Frequencies



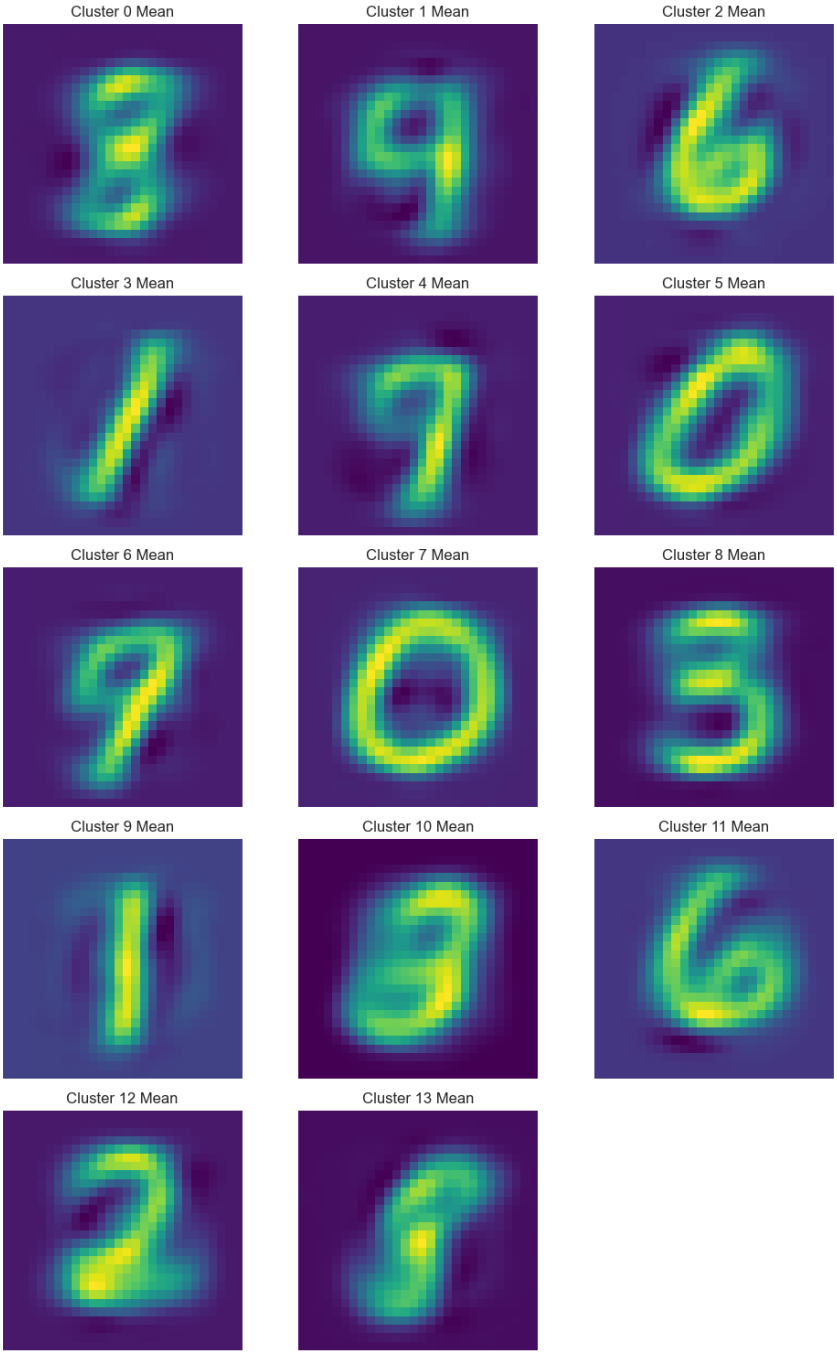
Upon examining the cluster frequencies plot pertaining to Gaussian Mixture models, a conspicuous trend emerges: 6 clusters exhibit considerably higher effectiveness compared to the remaining 8 clusters. The intriguing question is: what attributes distinguish these clusters from the rest? It is plausible that these prominent 6 clusters possess a specialized ability to capture distinct writing styles or unique digit formations, thus excelling in recognizing specific handwriting nuances. Conversely, the less emphasized 8 clusters might encapsulate more general or conventional writing patterns.

6.3.6 Cluster Composition Analysis



Approximately seven clusters exhibit remarkable recognition rates of over 90% for specific digits, each seemingly specialized in identifying a particular handwritten number. Conversely, the remaining clusters struggle to achieve similar accuracy, often recognizing two different digits with percentages below 40%.

6.3.7 Cluster Means



Notably, a subset of clusters appears to grapple with achieving a distinct representation of digits, evident in their slightly blurred mean images. These clusters struggle to encapsulate the defining characteristics of specific handwritten numbers, resulting in a certain degree of ambiguity. In contrast, other clusters exhibit sharper and more recognizable mean images, showcasing their proficiency in capturing the unique features of individual digits. This phenomenon aligns with our observations in the cluster composition, where some clusters excel in recognizing particular digits, while others encompass a broader range of styles.

7 Conclusion

In conclusion, the three clustering models, namely Meanshift, NormalizedCut (Spectral Clustering), and Gaussian Mixture, exhibit distinct characteristics and performances based on the provided information:

1. **Meanshift**:
    - Best Model Characteristics: The best Meanshift model achieved a high Random Index Score of approximately 0.909, indicating strong clustering performance.
    - Cluster Count: This model generates a notably large number of clusters, approximately 11125.
    - Cluster Composition: Despite its high score, the clusters' composition shows a more uniform distribution across all digits, with no single cluster specializing in any particular digit.
    - Recognition: While achieving a high score, it faces challenges in uniquely recognizing digits, with most clusters exhibiting similar recognition probabilities for each digit.
2. **NormalizedCut (Spectral Clustering)**:
    - Best Model Characteristics: The top-performing NormalizedCut model boasts a Random Index Score of around 0.901, indicating strong clustering quality.
    - Cluster Count: This model generates a moderate number of clusters, 15, which aligns well with the expected 10-digit classes.
    - Cluster Composition: The clusters are relatively specialized, with distinct recognition probabilities for different digits.
    - Recognition: This model excels in recognizing specific digits, with clusters focusing on digits 0, 1, 2, 3, 6, 7, and 8, each achieving recognition rates above 70%.
3. **Gaussian Mixture**:
    - Best Model Characteristics: The top-performing Gaussian Mixture model scores impressively with a Random Index Score of approximately 0.923, indicating excellent clustering quality.
    - Cluster Count: Similar to NormalizedCut, this model generates a number of clusters, 14, aligning well with the expected digit classes.
    - Cluster Composition: It demonstrates specialization in recognizing digits, with clusters 0, 1, 2, 6, 7, and 8 achieving recognition rates above 70%.
    - Recognition: The Gaussian Mixture model exhibits strong digit recognition capabilities, with these specialized clusters providing clear and accurate representations.

In summary, each model exhibits unique characteristics. Meanshift, while achieving a high overall score, generates a large number of clusters, potentially leading to challenges in interpretation. NormalizedCut and Gaussian Mixture, on the other hand, strike a balance between clustering quality and interpretability. They generate a moderate number of clusters and excel in recognizing specific digits. The choice among these models depends on the trade-off between clustering quality, interpretability, and the specific requirements of the task at hand.

Bibliography

[1] Christopher J.C. Burges Yann LeCun, Corinna Cortes. THE MNIST DATABASE of handwritten digits.

List of Figures

List of Tables