Course IV - Unsupervised ML

4.1 Introduction to Unsupervised Learning

Unsupervised learning is useful for finding structures within datasets and partitioning them into smaller groups. Common use cases include clustering, where data is grouped into clusters based on similarities, and dimensionality reduction, which reduces the number of features while retaining information.

The curse of dimensionality refers to the challenges posed by datasets with a high number of features. Too many features can lead to spurious correlations, increased noise, and higher computational complexity. Distance-based algorithms like k-nearest neighbors are particularly affected by high dimensionality.

Dimensionality reduction techniques like PCA can help mitigate the curse of dimensionality by reducing the number of features while preserving information. Unsupervised learning models provide alternative methods for dimensionality reduction, allowing for more efficient and interpretable data analysis.

Unsupervised learning begins with an unlabeled dataset. Models are trained on the dataset to identify patterns or structures. Once trained, the models can be used to analyze new, unlabeled data and make predictions or groupings based on the learned patterns.

In real-world scenarios, clustering is commonly used for various purposes, including classification, anomaly detection, customer segmentation, and improving supervised learning models:

1. **Classification**:
    - Clustering algorithms can be applied to unlabeled data to identify heterogeneous groupings within the dataset, even without explicit class labels.
    - For example, clustering can help identify unusual patterns in emails to detect spam or group similar product reviews.
2. **Anomaly Detection**:
    - Clustering can be used for anomaly detection by identifying clusters that deviate significantly from the majority of the data.
    - For instance, in credit card transactions, a small cluster with unusual transaction patterns may indicate potential fraudulent activity.
3. **Customer Segmentation**:
    - By analyzing customer data, clustering can identify different segments of customers based on various characteristics such as recency, frequency, and demographics.
    - This segmentation helps businesses tailor marketing strategies and services to different customer groups, such as single customers, new parents, or empty nesters.
4. **Improving Supervised Learning**:
    - Clustering can aid in improving supervised learning models by segmenting the data into homogeneous groups.
    - Models can be trained separately for each cluster, potentially leading to better performance compared to training on the entire dataset.
    - While not guaranteed to work in all cases, this approach can enhance classification accuracy by considering the specific characteristics of each cluster.

In addition to clustering, another type of unsupervised learning discussed is dimensionality reduction, which is commonly used for high-resolution images. Techniques like Principal Component Analysis (PCA) help compress images into a more compact representation while retaining essential information. This reduction in dimensionality can significantly speed up image processing tasks, making them more computationally efficient.

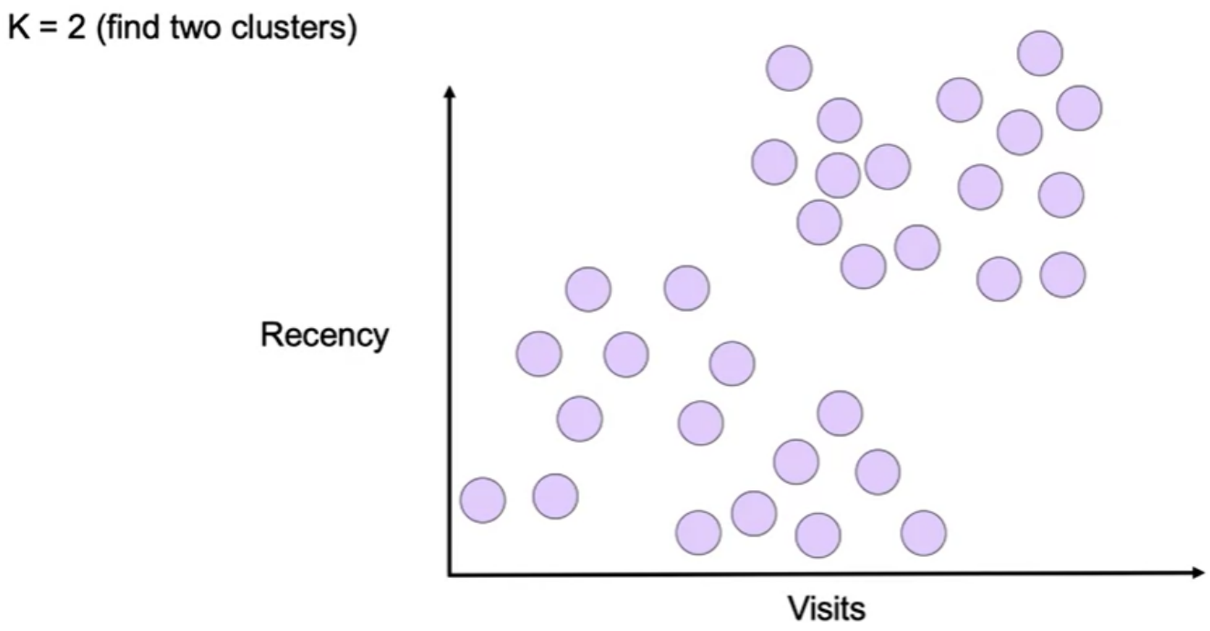
In the simple example provided, customers of a site are segmented based on a single feature: the number of visits. The goal is to use clustering to partition these customers into groups.

When segmenting customers into two clusters, a logical decision point would be to draw the line between the groups where there is a clear distinction between low-visiting customers and high-visiting customers. Visually, this decision point may appear as a threshold separating the data into two distinct clusters.

If the objective requires segmenting customers into three or five clusters, the data would be divided accordingly, with each cluster representing a different level of customer engagement based on the number of visits.

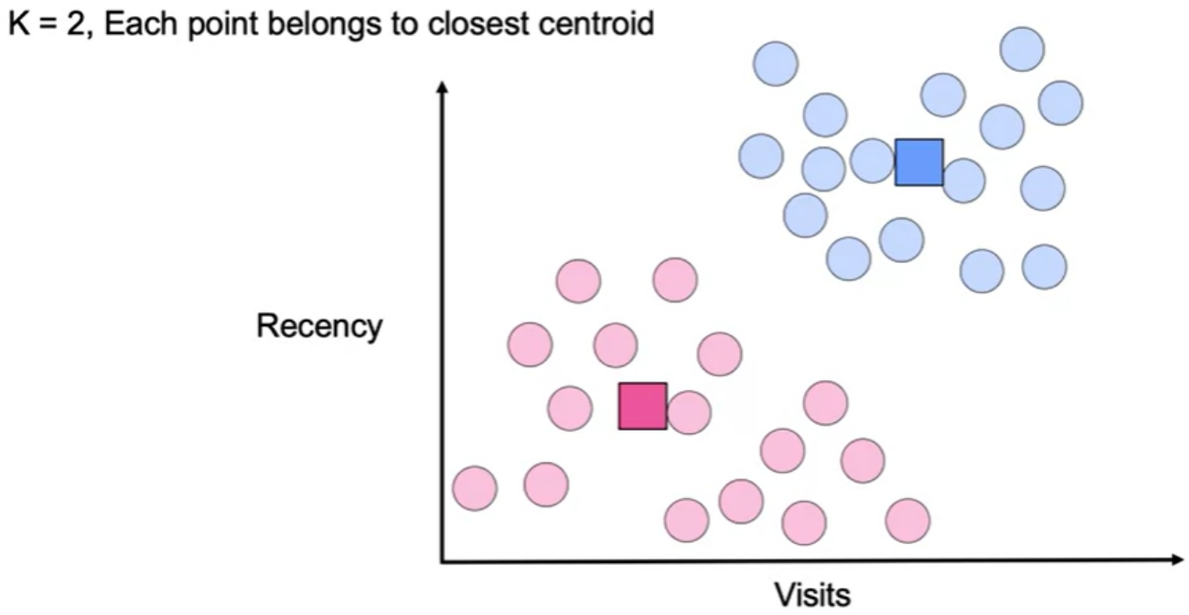
4.2 K Means Clustering

In the introduction to the K-Means algorithm for clustering, we use a dataset with two features: the number of visits to a site and the recency of the customer's visit. Visually, it's apparent that there are two clusters in the data.



Here's how the K-Means algorithm works algorithmically:

1. **Initialization**: We start by prescribing the number of clusters (in this case, two) and randomly selecting two points to act as the centroids of our clusters.
2. **Assigning Data Points to Clusters**: Each data point is then assigned to the nearest centroid based on Euclidean distance.
3. **Adjusting Centroids**: The centroids are then adjusted to the mean of the data points assigned to each cluster.
4. **Iteration**: Steps 2 and 3 are repeated iteratively until no data point is assigned to a different cluster, indicating convergence.



However, the K-Means algorithm is sensitive to the choice of initial centroids, and different initial configurations may yield different results. Therefore, we will discuss strategies for choosing the right model and handling the sensitivity of K-Means to initial points.

In order to ensure better optimization of the K-Means algorithm and avoid getting stuck at local optima, one approach is to initialize the centroids in a smarter way. Here's how it can be done using the K-means++ algorithm:

1. **Random Initialization**: Start by randomly selecting the first centroid.
2. **Weighted Selection**: For subsequent centroids, prioritize selecting points that are far away from existing centroids. This is achieved by assigning a probability to each point based on the squared distance from the nearest centroid. Points farther away are given higher probabilities.
3. **Repeat**: Repeat the selection process for each centroid, ensuring that each new centroid is far away from existing centroids.
4. **K-means++ Algorithm**: This algorithm, with its smarter initialization, helps avoid local optima and is often the default implementation of K-means in libraries like ScikitLearn.

By initializing centroids in this way, K-means++ helps improve the likelihood of converging to a globally optimal solution rather than a locally optimal one.

Choosing the right number of clusters, denoted as k, is a crucial aspect of clustering analysis. Sometimes, the number of clusters is predetermined based on specific requirements or objectives. However, there are situations where the optimal number of clusters is unclear, and we need a method to determine it empirically.

One metric commonly used for this purpose is inertia, which provides a measure of the total sum of squared distances of each point to its cluster centroid. Inertia penalizes spread-out clusters and rewards tighter clusters, making it useful for evaluating cluster cohesion. However, inertia is sensitive to the number of points in the clusters, meaning it will increase as more members are added to each cluster, regardless of their proximity to the centroids.

$$\text{Inertia} = \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2$$

- k is the number of clusters.
- C_i is the set of data points assigned to cluster i.
- μ_i is the centroid of cluster i.

- $||x - \mu_i||^2$ represents the squared Euclidean distance between data point x and centroid $\mu_i

Another metric is distortion, which calculates the average of the squared distances from each point to its cluster centroid. Like inertia, distortion also rewards tighter clusters, but adding more points to a cluster may not necessarily increase distortion, as closer points help decrease the average distance.

$$\text{Distortion} = \frac{1}{n} \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2$$

- n is the total number of data points.

When deciding between inertia and distortion, consider whether the similarity of points in the cluster or having clusters with similar numbers of points is more important for your analysis. Inertia is suitable when the number of points is a concern, while distortion is preferable when the similarity of points is more critical.

To find the clustering with the best inertia or distortion, the K-means algorithm is initiated multiple times with different initial configurations. Then, the resulting inertia or distortion is computed for each initialization, and the configuration leading to the lowest value is selected as the optimal solution.

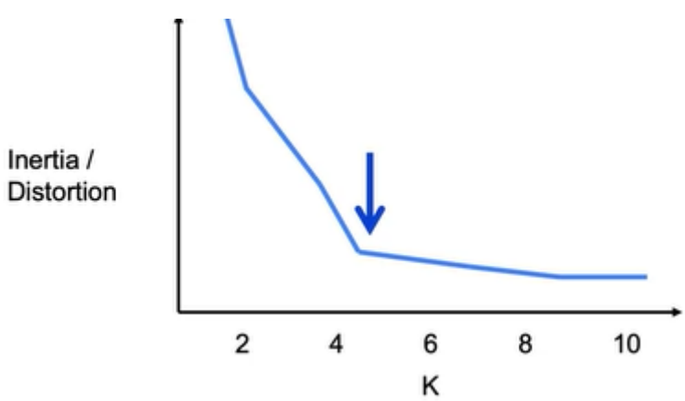
For example, if we have three different K-means algorithms converged with $k = 3$, each producing a different inertia value, we would choose the configuration with the lowest inertia value among them.

Understanding Inertia or Distortion:

- Inertia measures the total sum of squared distances of each point to its cluster centroid.
- Distortion calculates the average of the squared distances from each point to its cluster centroid.
- Both metrics decrease as the number of clusters increases.
- They penalize spread-out clusters and reward tighter clusters.

Elbow Method:

- The elbow method helps in selecting the optimal number of clusters.
- Plot the number of clusters (K) against inertia or distortion.
- Initially, inertia/distortion decreases rapidly as K increases.
- Identify the point where the rate of decrease slows down significantly, forming an "elbow" shape in the plot.
- This "elbow point" indicates a natural point where the number of clusters makes sense and serves as a logical choice for K.



Implementing K-Means in Python:

- Import the `KMeans` class from `sklearn.cluster`.
- Initialize an instance of `KMeans` with the desired number of clusters (K).
- Use the K-Means++ initialization method for better initialization.
- Fit the K-Means model to the data using the `.fit()` method.
- Predict clusters for new data points or existing data using the `.predict()` method.
- Optionally, use batch mode for faster processing, especially with large datasets.

Elbow Method Implementation:

- Initialize an empty list to store inertia values.
- Iterate over a range of possible cluster numbers (e.g., from 1 to 10).
- For each K, fit a K-Means model to the data and compute its inertia.
- Plot the number of clusters against the corresponding inertia values.
- Identify the "elbow point" where the inertia starts to decrease at a slower rate.

4.3 Mixture of Gaussians

Use cases of GMM:

- **Recommender systems** that make recommendations to users based on preferences (such as Netflix viewing patterns) of similar users (such as neighbors).
- **Anomaly detection** that identifies rare items, events or observations which deviate significantly from the majority of the data and do not conform to a well defined notion of normal behavior.
- **Customer segmentation** that aims at separating customers into multiple clusters, and devise targeted marketing strategy based on each cluster’s characteristics.

When is GMM better than K-Means?

Imagine you are a Data Scientist who builds a recommender for selling cars using K-Means clustering and you have two clusters. Everybody in cluster A is recommended to buy car A which costs **100k** with a **25k** profit margin and everyone in cluster B is recommended to buy car B which costs **50k** with a **10k** profit margin.

Let’s say you want to get as many people in cluster A as possible, why not use an algorithm that informs you of exactly how likely somebody would be interested in purchasing car A, instead of one that only tells you a hard yes or no (This is what K-Means does!).

With GMM, not only will you be getting the predicted cluster labels, the algorithm will also give you the probability of a data point belonging to a cluster. How amazing is that!

Whoever is selling those cars should definitely work on a better plan for a customer with a 90% chance of purchasing than for someone with a 75% chance of purchasing, even though they might show up in the same cluster.

Put simply, Gaussian Mixture Models (GMM) is a clustering algorithm that:

- Fits Gaussian distributions to your data
- The data scientist (you) needs to determine the number of gaussian distributions (`k`)

Hard vs Soft Clustering:

- **Hard clustering** algorithms cluster each data point in exactly one cluster.

- **Soft clustering** algorithms can cluster data in partially one cluster and partially others.
- GMM is a soft clustering algorithm

A Gaussian mixture is a weighted combination of k Gaussians, where each is identified by the following parameters:

1. a mean vector μ_i
2. a covariance matrix Σ_i
3. a component weight π_i that indicates the contribution of the i^{th} Gaussian

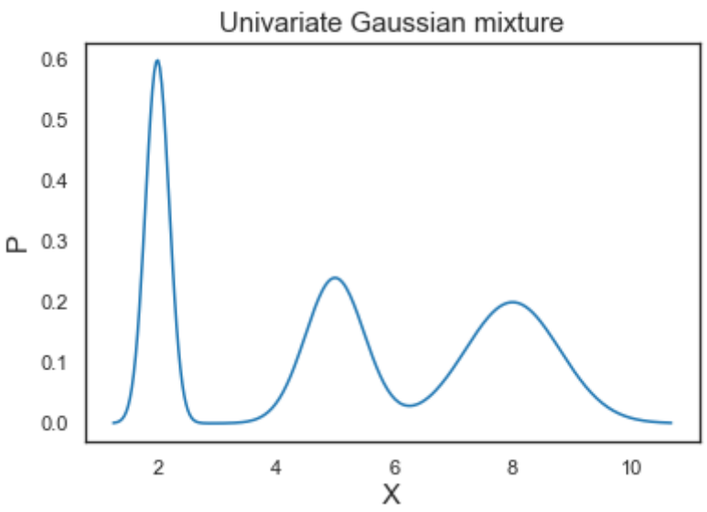
When put altogether, the pdf of the mixture model is formulated as:

$$p(x) = \sum_{i=1}^{K} \pi_i \mathcal{N}(x|\mu_i, \Sigma_i), \sum_{i=1}^{K} \pi_i = 1$$

Example 1: 1-Dimensional Gaussian Mixture:

Let's look at a mixture of 3 univariate Gaussians with

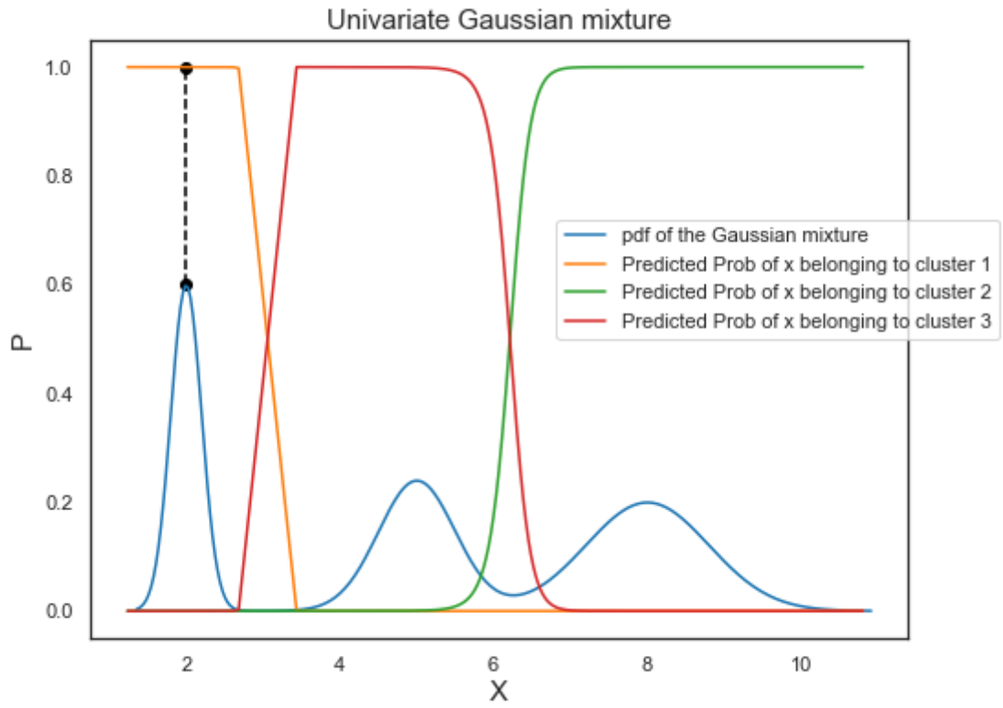
- means equal to **2, 5, 8** respectively
- std equal to **0.2, 0.5, 0.8** respectively
- component weight equal to **0.3, 0.3, 0.4** respectively



Thus, the means determine the centers of the mixed Gaussians; the standard deviations determine the width and shape of the mixed Gaussians; the weights determine the contributions of the Gaussians to the mixture.

Let's fit a GMM with `n_components=3` to our simulated data and plot the prior probabilities. The **GaussianMixture** class from **Scikit-learn** allows us to estimate the parameters of a Gaussian mixture distribution.

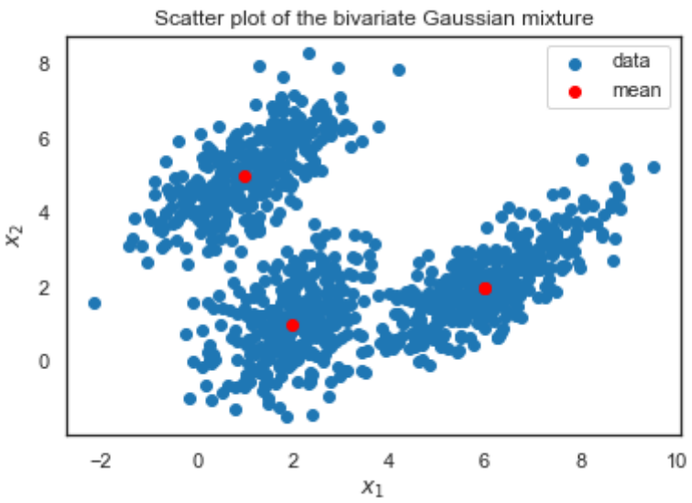
GaussianMixture.predict*proba* evaluates the components' density for each sample or for sample x_n the probability $p(i|x_n)$



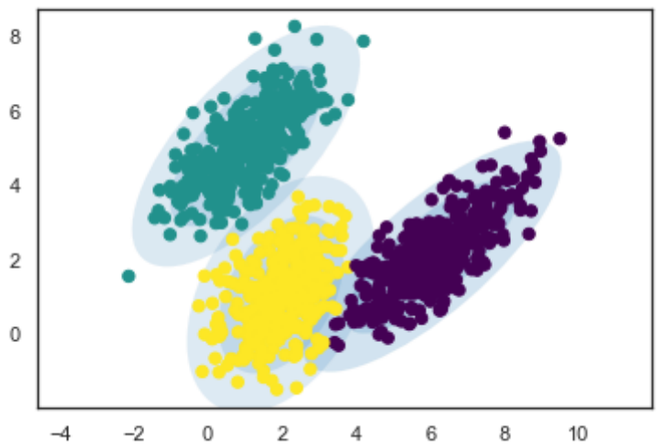
To interpret the predicted probabilities, let's take a look at the point colored in black as an example. On the Gaussian mixture pdf, the point is at the the peak of the first bell-shaped curve. Its corresponding probability of belonging to cluster 1 is equal to 1, which demonstrates that the probability of the center of a Gaussian distribution belonging to its own cluster is 100%.

Example 2: 2-Dimensional Gaussian Mixture:

In this example, you have a simulated 2-dimensional data that looks like this:



Like before, to work with GMM, we can use the **GaussianMixture** function from **sklearn.mixture**. We fit a GMM with **n_components = 3** to the simulated dataset, and plot the clustering result as follows:



The fitted clusters indeed match the individual Gaussians we simulated, and the ellipses drawn based on the estimated parameter values (means, covariances, weights) contain the clusters.

The default value of **covariance_type** in GMM is `full`, which allows each component (Gaussian) to have its own covariance matrix.

Since our dataset was simulated using three different covariance matrices, using the default covariance_type value would work the best. However, note that sometimes you can't use **covariance_type = full**, because you won't be able to invert it and that will give you an error.

Example 3: Image Segmentation

Image segmentation is the process of segmenting an image into multiple important regions. We can use a GMM to segment an image into **K** regions `(n_components = K)` according to significant colors.

Each pixel would be a data point with three features (r, g, b) (Or 1 feature if greyscale).

For instance, if we are working with a 256 × 256 image, you would have 65536 pixels in total and your data X would have a shape of 65536 × 3.

Let's look at an example using a picture of a house cat:

First let’s segment our image using 2 gaussian distributions;

Then we replace each pixel with the “average color” or the mean RGB values of the gaussian distribution it belongs to:

Similarly, if we increase the number of components to 8:

Our segmented image looks remarkably similar to the original, even though it uses only 8 colors!

4.4 Distance Metrics

1. **Euclidean Distance (L2 Distance):**
   - Classic distance metric, measures the straight-line distance between two points in Euclidean space.
   - Calculated as the square root of the sum of the squared differences in each dimension.
   - Formula: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ for 2D space.
   - Generalizes to higher dimensions by adding the squared differences in each dimension and taking the square root of the sum.
2. **Manhattan Distance (L1 Distance):**
   - Also known as city-block or taxicab distance.
   - Measures the sum of the absolute differences between the coordinates of two points.
   - Calculated as the sum of the absolute differences in each dimension.

- Formula: $d = |x_2 - x_1| + |y_2 - y_1|$ for 2D space.
- Particularly useful in high-dimensional spaces where distinguishing distances becomes challenging.

These distance metrics play crucial roles in clustering algorithms, where the choice of metric affects how clusters are formed based on proximity or similarity between data points.

3. **Cosine Similarity:**
   - Measures the cosine of the angle between two vectors.
   - Insensitive to the magnitude or scaling of the vectors, only the direction matters.
   - Formula: $\text{cosine\_similarity} = \frac{\text{dot product of vectors}}{\text{product of their norms}}$
   - Suitable for text data, where the relative importance of terms matters more than their frequency.
4. **Jaccard Distance:**
   - Measures dissimilarity between two sets by comparing the intersection and union of their elements.
   - Formula: $\text{Jaccard\_distance} = 1 - \frac{\text{intersection}}{\text{union}}$
   - Particularly useful for text data analysis, where sets of unique words are compared.

Each distance metric has its strengths and is suited for specific applications. Understanding the nature of the data and the clustering task is crucial in selecting the appropriate distance metric.

4.5 Hierarchical Agglomerative Clustering (HAC)

Hierarchical agglomerative clustering is a bottom-up approach where clusters are successively merged until convergence.

Hierarchical agglomerative clustering offers the advantage of providing a hierarchical structure that can be analyzed at different levels of granularity. However, it can be computationally expensive for large datasets due to its quadratic time complexity. Additionally, the choice of linkage criterion can significantly impact the resulting clusters.

Hierarchical agglomerative clustering involves merging clusters based on their average distances until a stopping criterion is met.

1. **Identifying Closest Pair:**
   - Begin with each point as its own cluster.
   - Identify the pair of points or clusters with the minimal distance.
   - Merge these two points or clusters into a single cluster.
2. **Continuing Merging:**
   - Repeat the process to find the next closest pair of points or clusters.
   - Merge them into a single cluster.
   - Continue this process until reaching a predefined stopping criterion.
3. **Linkage Criterion:**
   - The distance between clusters or between a cluster and a point is determined by the chosen linkage criterion.
   - Different linkage criteria include:
     - Single Linkage: Distance between the closest points in the two clusters.
     - Complete Linkage: Distance between the farthest points in the two clusters.
     - Average Linkage: Average distance between all pairs of points in the two clusters.
     - Ward's Linkage: Minimizes the variance when merging clusters.

4. **Hierarchical Structure:**
    - As clusters are merged, a hierarchical structure called a dendrogram is formed.
    - The dendrogram visually represents the merging process and can help determine the appropriate number of clusters.
5. **Stopping Criterion:**
    - The clustering process continues until reaching a stopping criterion, such as a predetermined number of clusters or a specified distance threshold.
    - Without a stopping criterion, all points would eventually be merged into a single cluster, which may not be meaningful.
6. **Pros and Cons of Linkage Types:**
    - Single Linkage: Can lead to clear separation but sensitive to noise.
    - Complete Linkage: Better at separating clusters with noise but may break apart larger clusters.
    - Average Linkage: Balances pros and cons of single and complete linkage.
    - Ward Linkage: Minimizes inertia similar to k-means, balancing pros and cons of other linkage types.

By considering the average distances between clusters and selecting an appropriate linkage type, hierarchical agglomerative clustering can effectively group data points into clusters while providing insights into the structure of the data through the dendrogram.

To implement hierarchical agglomerative clustering in Python, you can follow these steps using the scikit-learn library:

```
from sklearn.cluster import AgglomerativeClustering

# Create an instance of AgglomerativeClustering
agg = AgglomerativeClustering(n_clusters=3, affinity='euclidean',
linkage='ward')

# Fit the model to the data
agg.fit(X)

# Predict clusters for new data (if needed)
predicted_clusters = agg.fit_predict(new_data)
```

- Use the `AgglomerativeClustering` class from scikit-learn.
- Set parameters such as the number of clusters (`n_clusters`), distance metric (`affinity`), and linkage type (`linkage`).
- Fit the model to the data using the `fit` method.
- Predict clusters for new data points using the `fit_predict` method if needed.

4.6 DBSCAN and Mean Shift Clustering

DBSCAN Clustering:

In the DBSCAN clustering algorithm, the following inputs play a crucial role in determining the clusters:

1. **Distance Metric:**
    - Define the distance measure used to calculate similarity between data points.
    - Common distance metrics include Euclidean distance, Manhattan distance, etc.
2. **Epsilon (ε):**

- Epsilon determines the radius of the neighborhood around each point.
- Points within this distance from each other are considered neighbors.

3. **Min Samples (MinPts):**
   - This parameter determines the minimum number of points required to form a dense region or core point.
   - A core point must have at least MinPts neighbors (including itself) within the Epsilon distance.

Given these inputs, DBSCAN classifies points into three categories:

1. **Core Points:**
   - Points that have at least MinPts neighbors (including itself) within the Epsilon distance.
   - All clusters must contain at least one core point.
2. **Border Points (Density-Reachable):**
   - Points that are within the Epsilon neighborhood of a core point but do not have enough neighbors to be considered core points themselves.
   - These points are still part of the cluster as long as they are reachable from a core point.
3. **Noise:**
   - Points that are neither core points nor density-reachable.
   - These points are considered outliers or noise and do not belong to any cluster.

Clusters are formed by connecting core points and density-reachable points. The algorithm iteratively expands clusters by exploring neighboring points until no more points can be added to any cluster.

By defining the distance metric, Epsilon, and MinPts appropriately, DBSCAN can effectively identify clusters of arbitrary shapes and handle noisy data.

In the DBSCAN algorithm, the following steps are followed to cluster the data:

1. **Select a random point:** Begin with a random point in the dataset.
2. **Define Epsilon (ε) neighborhood:** Define a neighborhood around the selected point with a radius of Epsilon.
3. **Identify core points:** Determine if the number of points within the Epsilon neighborhood of the selected point (including itself) is greater than or equal to the MinPts parameter. If so, the point is classified as a core point.
4. **Expand cluster:** If a point is identified as a core point, expand the cluster by adding all points within its Epsilon neighborhood to the cluster.
5. **Identify border points:** Points within the Epsilon neighborhood of a core point but do not have enough neighbors to be considered core points themselves are classified as border points.
6. **Identify noise/outliers:** Points that do not belong to any cluster are classified as noise or outliers.
7. **Repeat:** Repeat the process for unvisited points until all points are assigned to a cluster or labeled as noise.

By adjusting the parameters such as Epsilon and MinPts, the algorithm can effectively identify clusters of different shapes and handle noisy data.

Strengths of DBSCAN:

- Automatically determines the number of clusters.
- Handles noise and identifies outliers.
- Effective for clustering data with arbitrary shapes.
- Does not require specifying the number of clusters beforehand.

Weaknesses of DBSCAN:

- Requires careful selection of parameters (Epsilon and MinPts).
- Difficulty in tuning parameters, especially in higher-dimensional space.
- May not perform well with clusters of varying densities.

In Python, DBSCAN can be implemented using the `DBSCAN` class from the `sklearn.cluster` module. The key parameters to adjust are `eps` (Epsilon) and `min_samples` (MinPts).

Mean Shift Clustering

In the mean shift clustering algorithm, the following steps are followed to cluster the data:

1. **Select a random point:** Begin with a random point in the dataset.
2. **Define a window:** Define a window around the selected point.
3. **Calculate the weighted mean:** Calculate the weighted mean of the points within the window, giving more weight to points closer to the original point.
4. **Shift the window centroid:** Shift the centroid of the window to the location of the calculated weighted mean.
5. **Repeat until convergence:** Repeat steps 3 and 4 until the window centroid no longer shifts, indicating convergence to a local density maximum (mode).
6. **Group points:** Assign all points that converge to the same mode to the same cluster.

Mean shift effectively identifies clusters by moving towards the densest regions of the data, iteratively updating the centroids based on the local density.

Strengths of Mean Shift:

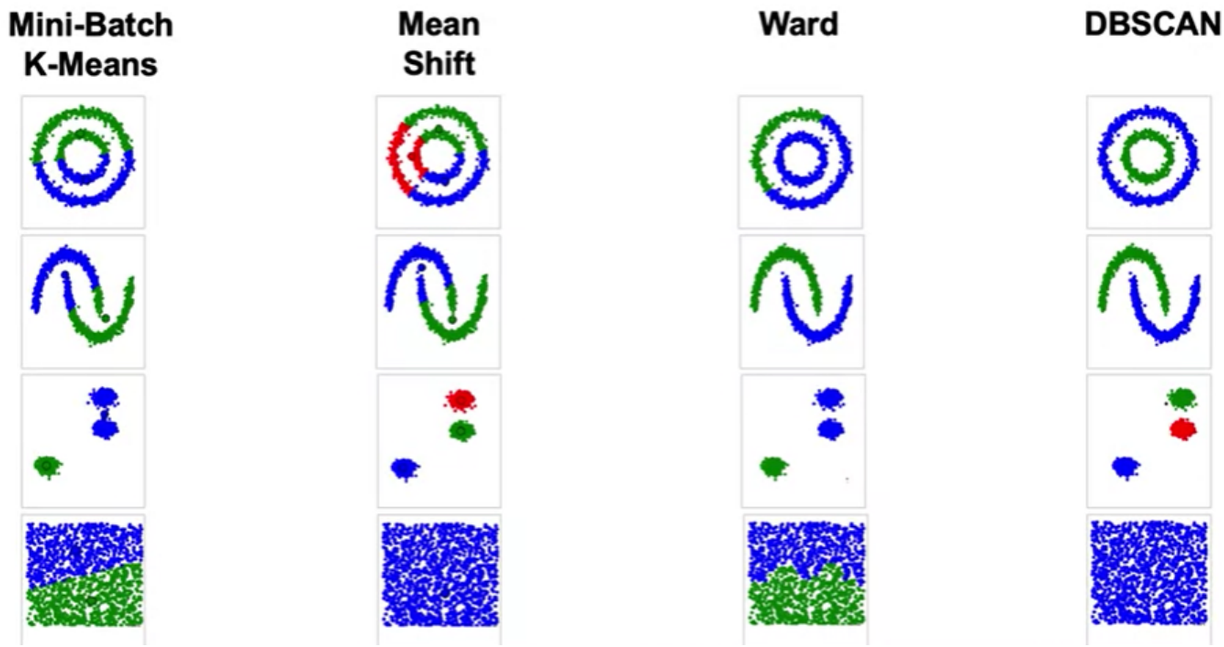
- Model-free: Does not assume the number or shape of clusters.
- Only requires tuning one parameter: the bandwidth.
- Robust to outliers: Outliers have minimal effect due to the use of the window size.
- Can handle clusters of arbitrary shapes.

Weaknesses of Mean Shift:

- Results depend heavily on the choice of bandwidth.
- Selection of bandwidth can be challenging and may require experimentation.
- Can be slow for large datasets due to its computational complexity, which is proportional to mn^2, where m is the number of iterations and n is the number of data points.

In Python, Mean Shift clustering can be implemented using the `MeanShift` class from the `sklearn.cluster` module. The key parameter to adjust is the bandwidth, which determines the size of the window used to calculate the local density.

4.7 Comparison of Clustering Algorithms



Method name	*K-means*	*Mean-shift*	*Hierarchical clustering*	*DBSCAN*
Parameters	Number of clusters	Bandwidth	Number of clusters	Neighborhood size
Scalability	Very large *n_samples* medium *n_clusters* with *MiniBatch code*	Not scalable with n_samples	Large *n_samples* and *n_clusters*	Very large *n_samples*, medium *n_clusters*
General use case	General purpose, even cluster size, flat geometry, not too many clusters	Many clusters, uneven cluster size, non-flat geometry	Many clusters, possibly connectivity constraints	Non-flat geometry, uneven cluster sizes, outlier detection
Applications	Find few clusters of roughly the same size	Can identify number of clusters, often used in video	Clusters may be of different size, does not identify outliers	Often used in computer vision applications

1. **K-means:**
   - **Pros:** Fast and scalable, especially with MiniBatch K-means. Works well with spherical clusters.
   - **Cons:** Requires pre-determination of the number of clusters. Not suitable for non-spherical clusters.
   - **Use Cases:** Useful when the number of clusters is known and when clusters are roughly spherical.
2. **Mean Shift:**
   - **Pros:** Automatically determines the number of clusters. Can handle uneven cluster sizes. Model-free.
   - **Cons:** Can be slow with large datasets. Limited to Euclidean distance. Tends to find spherical clusters.
   - **Use Cases:** Suitable when the number of clusters is unknown and when clusters have uneven sizes.
3. **Hierarchical Clustering (Ward):**
   - **Pros:** Provides a full hierarchy of clusters. Can handle clusters of different shapes. Offers flexibility in defining clusters.
   - **Cons:** Can be slow and memory-intensive with large datasets. Choosing the number of clusters can be subjective.
   - **Use Cases:** Useful for exploring subgroup structures within clusters and when a hierarchical view of clusters is needed.
4. **DBSCAN (Density-Based Spatial Clustering of Applications with Noise):**

- **Pros:** Does not require pre-determination of the number of clusters. Can handle clusters of different shapes and sizes. Good for outlier detection.
- **Cons:** Requires tuning of parameters (epsilon and min_samples). Difficulty in determining clusters of different densities.
- **Use Cases:** Suitable when the number of clusters is unknown, and when clusters are of similar density.

Each algorithm has its own strengths and weaknesses, and the choice of algorithm depends on factors such as the dataset characteristics, the number of clusters expected, and the desired shape of clusters. It's essential to consider these factors when selecting the appropriate clustering algorithm for a given business case.

4.8 Dimensionality Reduction

Dimensionality reduction is a class of unsupervised learning techniques aimed at reducing the number of features in our dataset while preserving the essential information.

The curse of dimensionality poses challenges in practical data analysis, as increasing the number of features can lead to worse model performance, poor distance measures, and increased outlier incidence. By reducing dimensionality, we aim to mitigate these challenges and create a more manageable representation of the dataset.

PCA offers a way to transform the original features into a lower-dimensional space while preserving as much variance as possible. This is achieved by selecting linear combinations of the original features intelligently.

NMF, on the other hand, focuses on decomposing the original data into positive values, providing another approach to dimensionality reduction.

Overall, dimensionality reduction techniques like PCA and NMF enable us to work with more manageable datasets without losing significant information, thus facilitating more efficient and effective data analysis and modeling.

Principal Component Analysis (PCA) is a powerful technique for reducing the dimensionality of a dataset while preserving most of its variance.

1. **Identifying Primary Vectors:**
   - PCA utilizes linear algebra tools to find the primary vectors along which the dataset exhibits the most variance.
   - The primary vector, also known as the primary right singular vector, accounts for the maximum amount of variance in any direction of the dataset.
2. **Decomposition into Orthogonal Vectors:**
   - Once the primary vector is identified, PCA decomposes the dataset into orthogonal vectors, where each vector is perpendicular to one another.
   - These vectors are determined through a mathematical process called singular value decomposition (SVD).
3. **Calculating Singular Values:**
   - The singular value decomposition yields three matrices: U, S, and V.
   - Matrix S, a diagonal matrix, stores the lengths of the vectors sorted from largest to smallest. Larger values indicate more important vectors.
4. **Selecting Principal Components:**
   - PCA identifies the principal components based on the singular values.
   - The first few principal components capture the majority of the variance in the dataset, allowing for dimensionality reduction.

5. **Transformation of the Dataset:**
    - To reduce dimensionality, PCA transforms the original dataset using the selected principal components.
    - The transformed dataset retains the most significant information while having fewer dimensions.
6. **Scaling Considerations:**
    - PCA is sensitive to scaling, so it's crucial to scale the data before applying PCA to ensure accurate results.
    - Scaling ensures that each feature contributes equally to the analysis and prevents any single feature from dominating the principal components.

In practice, you can implement PCA using libraries like scikit-learn. By specifying the desired number of components, you can perform PCA and obtain a transformed dataset with reduced dimensions. This allows for more efficient data analysis and visualization while retaining the essential information present in the original dataset.

Moving beyond linearity, we explore techniques for dealing with nonlinear features in dimensionality reduction.

1. **Nonlinear Features and PCA:**
    - Traditional PCA relies on linear transformations, which may fail to capture nonlinear features in the data.
    - When applying PCA directly to datasets with nonlinear features, the reduction in dimensionality may lead to loss of important information.
2. **Kernel PCA:**
    - Kernel PCA addresses the limitation of linear PCA by first mapping the data to a higher-dimensional space using kernel functions.
    - In the higher-dimensional space, nonlinear structures within the data become more apparent, enabling linear PCA to effectively reduce dimensionality.
    - The choice of kernel function and parameters, such as gamma, can influence the effectiveness of kernel PCA.
3. **Performing Kernel PCA in Python:**
    - In Python, kernel PCA can be implemented using the `KernelPCA` class from the `sklearn.decomposition` module.
    - After specifying the desired number of components and kernel type, the `.fit_transform()` method is used to transform the original dataset.
4. **Manifold Learning - MDS:**
    - Manifold learning techniques, such as Multidimensional Scaling (MDS), offer an alternative approach to dimensionality reduction.
    - Unlike PCA, which aims to preserve variance, MDS focuses on maintaining the geometric distances between data points.
    - MDS can be useful for visualizing high-dimensional data while preserving the pairwise distances between points.
5. **Performing MDS in Python:**
    - To perform MDS in Python, the `MDS` class from `sklearn.manifold` module can be used.
    - After creating an instance of the class and specifying the number of components, the `.fit_transform()` method is applied to the dataset.
6. **Other Manifold Learning Methods:**
    - Additional manifold learning methods include Isomap and t-SNE, which offer different strategies for preserving the underlying structure of high-dimensional data.

- Experimenting with various decomposition methods and evaluating their performance using visualization techniques can help determine the most suitable approach for a given dataset.

In practice, choosing the appropriate dimensionality reduction technique depends on the characteristics of the data and the specific goals of the analysis. Experimentation with different methods and careful evaluation of the results can lead to insights into the underlying structure of the data and facilitate downstream analysis tasks.

Non-negative Matrix Factorization (NMF) is introduced as another approach to reducing the dimensionality of data.

1. **Principle of NMF:**
   - NMF involves decomposing an original matrix containing only positive values into two matrices: W and H, where both W and H also contain only positive values.
2. **Applications of NMF:**
   - NMF is particularly useful for datasets with positive values, such as word count matrices in natural language processing or pixel intensity matrices in image processing.
   - In NLP, documents can be represented as matrices, where rows represent documents and columns represent words, with values indicating word counts.
   - NMF can extract topics from the terms matrix W and determine how these topics combine to reconstruct the original documents using the coefficients matrix H.
3. **Advantages of NMF:**
   - Since NMF works with only positive values, it cannot "undo" latent features and tends to produce more interpretable results.
   - NMF can handle various types of data, including text, images, and other non-interpretable data objects like videos and music.
4. **Implementing NMF in Python:**
   - NMF can be implemented using the `NMF` class from the `sklearn.decomposition` module.
   - The number of topics/components and the method of initialization can be specified when creating an instance of the `NMF` class.
   - Similar to other dimensionality reduction techniques, NMF can be fitted to the data using the `.fit()` method and transformed using the `.transform()` method.
5. **Choosing Dimensionality Reduction Techniques:**
   - The choice of dimensionality reduction technique depends on the nature of the data and the specific goals of the analysis.
   - PCA is suitable for linear combinations of features and aims to preserve variance.
   - Kernel PCA is similar to PCA but can capture nonlinear relationships.
   - Multidimensional Scaling (MDS) preserves distances between data points and is useful for visualizing clusters.
   - NMF is ideal for datasets with only positive values and produces interpretable representations.

Method	Use case
Principal Components Analysis (PCA)	Identify small number of transformed variables with different effects, preserving variance
Kernel PCA	Useful for situations with nonlinear relationships, but requires more computation than PCA
Multidimensional Scaling	Like PCA, but new (transformed features) are determined based on preserving distance between points, rather than explaining variance
Non-negative Matrix Factorization	Useful when you want to consider only positive values (word matrices, images)