**Unsupervised Learning**

**Chapter1**

**Clustering for Dataset Exploration**

A close-up of a white background

Description automatically generated

A close-up of a white background

Description automatically generated

A close-up of a purple flower

Description automatically generated

A close-up of a text

Description automatically generated

A white text with blue text

Description automatically generated

A screenshot of a computer

Description automatically generated

A yellow text on a white background

Description automatically generated

A diagram with different colored dots

Description automatically generated

A screenshot of a computer code

Description automatically generated

A close-up of a white background

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

**Cross-tabulation** in pandas is a method used to summarize the relationship between two (or more) **categorical variables** in the form of a table. It computes the frequency distribution of variables and is especially useful for exploring categorical data.

In pandas, cross-tabulation is done using the pd.crosstab() function.

**Key Features of pd.crosstab():**

1. **Input Parameters**:
   * index: Values to group by rows (usually one categorical variable).
   * columns: Values to group by columns (usually another categorical variable).
   * values: Array or column to aggregate (optional).
   * aggfunc: Aggregation function to apply if values is provided (e.g., sum, mean, etc.).
   * margins: Adds row and column totals (default is False).
2. **Output**:
   * A DataFrame where rows represent one variable's categories and columns represent the other variable's categories.

**Syntax:**

pd.crosstab(index, columns, values=None, aggfunc=None, margins=False, normalize=False)

**Example Usage:**

**1. Simple Frequency Cross-tabulation**

import pandas as pd

# Example data

data = {

"Gender": ["Male", "Female", "Female", "Male", "Male", "Female"],

"Preference": ["Tea", "Coffee", "Coffee", "Tea", "Coffee", "Tea"]

}

df = pd.DataFrame(data)

# Cross-tabulation

ct = pd.crosstab(df["Gender"], df["Preference"])

print(ct)

**Output:**

Preference Coffee Tea

Gender

Female 2 1

Male 1 2

**2. Adding Row and Column Totals (margins=True)**

ct = pd.crosstab(df["Gender"], df["Preference"], margins=True)

print(ct)

**Output:**

Preference Coffee Tea All

Gender

Female 2 1 3

Male 1 2 3

All 3 3 6

**3. Normalizing for Proportions**

ct = pd.crosstab(df["Gender"], df["Preference"], normalize=True)

print(ct)

**Output:**

Preference Coffee Tea

Gender

Female 0.3333 0.1667

Male 0.1667 0.3333

**4. Using Aggregation Functions (values and aggfunc)**

data = {

"Gender": ["Male", "Female", "Female", "Male", "Male", "Female"],

"Preference": ["Tea", "Coffee", "Coffee", "Tea", "Coffee", "Tea"],

"Score": [4, 5, 2, 3, 4, 5]

}

df = pd.DataFrame(data)

# Cross-tabulation with aggregation

ct = pd.crosstab(df["Gender"], df["Preference"], values=df["Score"], aggfunc="mean")

print(ct)

**Output:**

Preference Coffee Tea

Gender

Female 3.5 5

Male 4.0 3

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

A close-up of a text

Description automatically generated

A screenshot of a computer

Description automatically generated



A graph with a blue line

Description automatically generated

A graph with a line and a yellow dotted line

Description automatically generated with medium confidence

A white text with black text

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screen shot of a graph

Description automatically generated

A graph of a number of dots

Description automatically generated with medium confidence

A diagram with a blue arrow and a blue arrow

Description automatically generated with medium confidence

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

A yellow marker on a white background

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A close-up of a computer screen

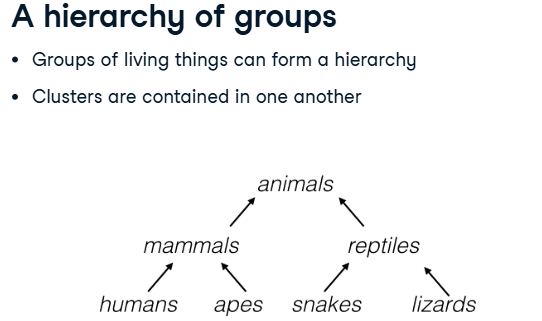
Description automatically generated

**Chapter2**

**Visualization with 1.Hierarchical Clustering and 2.t-SNE**

A close up of a map

Description automatically generated



A screenshot of a computer

Description automatically generated

Dendrogram:

A graph of a voting process

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

A diagram of a graph

Description automatically generated with medium confidence

A screenshot of a computer code

Description automatically generated

In hierarchical clustering, the number of operations is going to be n-1 and n= number of data.

**Arguments for linkage:**

1. **samples**:
   * A 2D array or matrix containing the data points to be clustered. Each row is a data point, and each column is a feature.
   * Example: If you have data about different flower species based on petal length and width, each row might represent one flower, and the columns would be the measurements.
2. **method**:
   * The linkage algorithm to use for clustering.
   * Common options:
     + 'complete': Uses the maximum distance between points in two clusters to determine cluster similarity.
     + 'single': Uses the minimum distance between points in two clusters.
     + 'average': Uses the average distance between points in two clusters.
   * 'complete' is used in your script, meaning the clusters will merge based on the farthest pair of points.

**Output:**

* The result, mergings, is a linkage matrix that describes how the clusters are formed. It’s used as input for visualizing a dendrogram.

**Dendrogram Parameters**

* **leaf\_rotation=90**: Rotates the leaf labels by 90 degrees for better readability.
* **leaf\_font\_size=6**: Adjusts the font size of the leaf labels.

A diagram of a diagram

Description automatically generated with medium confidence

A graph with a red line

Description automatically generated

A diagram showing clustering data

Description automatically generated with medium confidence

A graph with lines and a red line

Description automatically generated

A white background with black text

Description automatically generated

A graph of a cluster

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generated

A close-up of a text

Description automatically generated

A diagram of different types of data

Description automatically generated

A diagram of a graph

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A text on a white background

Description automatically generated

Here’s how **K-Means**, **Hierarchical Clustering**, and **t-SNE** handle adding new data:

**1. K-Means**

* **Can you add new data without retraining?**  
  Yes, with limitations.
* **Why?**  
  K-Means clusters are defined by centroids. When new data is added, you can assign the new points to the nearest centroid without recomputing the entire clustering. However, this assumes the original centroids are still valid and representative of the data.
* **When to recompute?**
  + If the new data significantly alters the distribution of the dataset.
  + If the clustering quality decreases because centroids no longer represent the data well.
* **Conclusion**: Efficient for adding new data in most cases.

**2. Hierarchical Clustering**

* **Can you add new data without recomputing?**  
  No.
* **Why?**  
  Hierarchical clustering builds a dendrogram by calculating pairwise distances between all points, which defines the hierarchical structure. Adding new data requires recalculating the entire distance matrix and rebuilding the hierarchy.
* **When to recompute?**  
  Every time new data is added.
* **Conclusion**: Hierarchical clustering is not suitable for dynamic datasets; it requires recomputation for every update.

**3. t-SNE (t-Distributed Stochastic Neighbor Embedding)**

* **Can you add new data without retraining?**  
  No.
* **Why?**  
  t-SNE is a dimensionality reduction technique, not a clustering algorithm. It optimizes the low-dimensional representation of a fixed dataset to preserve pairwise relationships. Adding new data requires recalculating the entire optimization to maintain consistency.
* **When to recompute?**  
  Every time new data is added.
* **Conclusion**: t-SNE is computationally expensive and not suitable for incremental updates.

**Summary Table**

| **Method** | **Can Add New Data Without Recomputing?** | **Comments** |
| --- | --- | --- |
| **K-Means** | Yes (with limitations) | Assign new points to nearest centroid; recompute if distribution changes. |
| **Hierarchical** | No | Requires recalculating the dendrogram from scratch. |
| **t-SNE** | No | Needs to recompute the entire optimization. |

For dynamic datasets where you frequently add new data, **K-Means** is the most practical choice.

A text on a white background

Description automatically generated

A screenshot of a computer screen

Description automatically generated

QUESTION: the "variety": ["Type A", "Type A", "Type B", "Type B", "Type C", "Type C"] is already mentioned in the dataset, so why is there a need for fitting and transforming? it's just like visualizing

**ChatGPT said:**

ChatGPT

That's a great observation! You're correct that if the **"variety"** labels are already in the dataset, you can directly use them to color or group your data visually. In this case, the **"variety"** column provides a pre-existing classification.

However, here's why we **fit and transform the data using t-SNE** even when we have labels:

**Purpose of t-SNE When Labels Are Present:**

1. **Data Structure Discovery**:
   * t-SNE isn't dependent on the labels; it focuses on relationships between the data points based on their features (e.g., width and length).
   * Labels might not always reflect natural clusters. t-SNE helps you see if the data **naturally forms clusters** that align with the labels or if there’s overlap.
2. **High-Dimensional Data**:
   * If your dataset had more than two features (e.g., width, length, color, texture), it would be impossible to plot all features directly in 2D. t-SNE reduces this to a visualizable form while preserving relationships.
3. **Unsupervised Approach**:
   * Even if labels exist, t-SNE treats the problem as **unsupervised**. It doesn’t use the labels during transformation—it simply reveals patterns in the feature space.
4. **Validation**:
   * After applying t-SNE, you can compare the **clusters from t-SNE** with your existing labels. If the clusters match the labels well, it validates your labeling process. If not, it could reveal potential misclassifications or new patterns.

**When You Don’t Need t-SNE:**

If your dataset is already 2D (like width vs. length) and the labels (e.g., "variety") are known, then yes, you can directly plot the data without t-SNE, as shown below:

**When t-SNE Adds Value:**

If your dataset has more than 2 dimensions, t-SNE is helpful to **reduce dimensionality** while preserving relationships. For instance, if you had:

* Width
* Length
* Color intensity
* Seed weight
* Texture score

Then t-SNE would transform all these features into 2D while keeping similar seeds close together.

**Conclusion:**

If you just want to **visualize existing labels in a 2D dataset**, you don’t need t-SNE. But if:

* Your dataset has many features, or
* You want to verify if clusters form **naturally** from the features,

**Chapter3**

**Decorrelating Your Data and Dimension Reduction**

A close-up of a white background

Description automatically generated

A close-up of a diagram

Description automatically generated

A graph of graphs and numbers

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

A graph of blue dots

Description automatically generated

A graph of components and red arrows

Description automatically generated

A screenshot of a computer component

Description automatically generated

In **Pandas**, the **Pearson correlation coefficient** is a statistical measure that evaluates the linear relationship between two continuous variables. It ranges from **-1** to **1**:

* **1**: Perfect positive linear correlation (as one variable increases, the other increases proportionally).
* **-1**: Perfect negative linear correlation (as one variable increases, the other decreases proportionally).
* **0**: No linear correlation (no linear relationship between the variables).

You can calculate the Pearson correlation between columns in a DataFrame using the .corr() method:

**Example:**

import pandas as pd

OR

from scipy.stats import pearsonr

# Sample data

data = {'A': [1, 2, 3, 4, 5],

'B': [2, 4, 6, 8, 10],

'C': [5, 4, 3, 2, 1]}

df = pd.DataFrame(data)

# Calculate Pearson correlation

correlation\_matrix = df.corr(method='pearson')

OR

correlation, pvalue = pearsonr(width, length)

print(correlation\_matrix)

OR

print(correlation)

A graph and chart with text

Description automatically generated

A white background with black text

Description automatically generated

A close-up of a text

Description automatically generated

A graph of data set

Description automatically generated with medium confidence

A white background with black text

Description automatically generated

A graph showing a graph showing a number of data

Description automatically generated with medium confidence

A graph and diagram of different colors

Description automatically generated with medium confidence

A graph of a number of blue and red bars

Description automatically generated

A screenshot of a computer program

Description automatically generated

A graph with blue and yellow text

Description automatically generated

A diagram of a bar graph

Description automatically generated

A white background with black text

Description automatically generated

A graph with blue bars and red text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

A close-up of a white background

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

The line svd = TruncatedSVD(n\_components=50) creates an instance of the TruncatedSVD class from the sklearn.decomposition module. Here's a detailed explanation of what happens:

**Truncated Singular Value Decomposition (SVD):**

* **Purpose**: It is a dimensionality reduction technique primarily used to reduce the number of features in a dataset while retaining the most significant information.
* **Why "Truncated"?**:
  + In full SVD, all singular values are computed. In **truncated SVD**, only the top n\_components singular values are retained.
  + This is computationally efficient when working with large datasets or sparse matrices.
* **n\_components=50**:
  + Specifies the number of singular values (or components) to keep.
  + These components represent the most important features, capturing the majority of the variance in the data.

**When to Use Truncated SVD?**

* It's often applied to reduce the dimensions of high-dimensional data before further processing, such as clustering or classification.
* Unlike PCA, it can work with sparse matrices (e.g., from text data after vectorization).

**Chapter4**

**Discovering Interpretable Features**

**NMF** (non-negative matrix factorization)

A yellow text on a white background

Description automatically generated

In the context of dimensionality reduction, **interpretability** refers to how easily humans can understand and explain the transformed features or components generated by the model. Let’s compare **PCA (Principal Component Analysis)** and **NMF (Non-Negative Matrix Factorization)** with respect to interpretability:

**Principal Component Analysis (PCA):**

* **How It Works**: PCA identifies directions (principal components) that maximize variance in the data. These components are linear combinations of the original features, which can include both positive and negative coefficients.
* **Interpretability**:
  + **Less Interpretable**: PCA components are often difficult to interpret because they are linear combinations of all original features with both positive and negative weights. For example, a PCA component might assign positive weight to one feature while assigning negative weight to another, which can obscure direct meaning.
  + **Orthogonality**: The components are orthogonal (uncorrelated), which is mathematically elegant but not inherently interpretable in terms of human-understandable relationships.

**Non-Negative Matrix Factorization (NMF):**

* **How It Works**: NMF decomposes the data into parts or components that are constrained to be non-negative. This is particularly useful when data naturally represents additive relationships (e.g., images, text frequencies).
* **Interpretability**:
  + **More Interpretable**: NMF components are easier to interpret because they represent additive combinations of the original features. For instance, in a topic modeling task, each NMF component (topic) represents a set of words with positive contributions, making it straightforward to understand what the component represents.
  + **Additive Nature**: Since the components and their weights are non-negative, they often align well with human intuition, especially when features represent counts, frequencies, or intensities.

**Key Differences in Interpretability:**

| **Feature** | **PCA** | **NMF** |
| --- | --- | --- |
| **Significance of Features** | Components involve both positive and negative contributions, making interpretation harder. | Components involve only positive contributions, making them additive and more intuitive. |
| **Data Structure** | Works well for global variance maximization, even if features lack direct interpretability. | Preserves part-based structures, making the output more aligned with human understanding. |
| **Use Case Examples** | Best for tasks where variance explanation is the priority (e.g., noise reduction, feature decorrelation). | Best for tasks needing interpretability (e.g., topic modeling, image decomposition). |

**When to Use Which?**

* Use **PCA** when:
  + The goal is to reduce dimensionality while retaining the maximum variance.
  + Interpretability is not a primary concern.
* Use **NMF** when:
  + Interpretability is crucial, especially when working with non-negative data (e.g., counts, pixel intensities, or text frequencies).
  + You need to uncover parts-based or additive representations.

A diagram of a function

Description automatically generated with medium confidence

A diagram of a diagram of a door

Description automatically generated with medium confidence

A screenshot of a computer code

Description automatically generated

A **csr\_matrix**, or **Compressed Sparse Row matrix**, is a data structure from the scipy.sparse module in Python that efficiently stores and manipulates sparse matrices. Sparse matrices are those where most of the elements are zero, and storing them in dense formats (like NumPy arrays) would waste memory and computation.

**Key Features of csr\_matrix:**

1. **Compressed Row Storage (CRS):**
   * The csr\_matrix stores data in a way that optimizes memory usage and allows efficient operations like matrix-vector multiplication.
   * It is particularly suited for row-wise operations.
2. **Three Main Arrays Used:**
   * **data**: Contains all the non-zero elements of the matrix.
   * **indices**: Stores the column indices corresponding to each non-zero element in data.
   * **indptr**: Points to the start of each row in the data and indices arrays.
3. **Memory Efficiency:**
   * Only non-zero values are stored, reducing memory usage significantly for large, sparse matrices.
   * For a dense matrix with many zeros, a sparse representation can reduce storage complexity.

**Structure:**

Consider a sparse matrix:

[[0, 0, 1, 0],

[2, 0, 0, 0],

[0, 3, 0, 4]]

The corresponding csr\_matrix representation:

* **data**: [1, 2, 3, 4] (the non-zero values)
* **indices**: [2, 0, 1, 3] (the column indices of the non-zero values)
* **indptr**: [0, 1, 2, 4] (index pointers for the start of each row in data and indices)

**How indptr Works:**

* Row 0 starts at index 0 in data (and ends before index 1).
* Row 1 starts at index 1 in data (and ends before index 2).
* Row 2 starts at index 2 in data (and ends before index 4).

**Advantages:**

1. **Memory Efficient**:
   * Only non-zero elements are stored, avoiding the overhead of storing zero elements in dense arrays.
2. **Fast Row Access**:
   * Operations like slicing rows or iterating over rows are very efficient.
3. **Matrix Multiplication**:
   * Optimized for matrix-vector and matrix-matrix multiplication.

**Common Operations:**

1. **Creation:**

from scipy.sparse import csr\_matrix

import numpy as np

# Create a dense matrix

dense\_matrix = np.array([

[0, 0, 1, 0],

[2, 0, 0, 0],

[0, 3, 0, 4]

])

# Convert to csr\_matrix

sparse\_matrix = csr\_matrix(dense\_matrix)

print(sparse\_matrix)

1. **Convert Back to Dense:**

dense\_again = sparse\_matrix.toarray()

print(dense\_again)

1. **Accessing Non-Zero Elements:**

print(sparse\_matrix.data) # Non-zero values

print(sparse\_matrix.indices) # Column indices

print(sparse\_matrix.indptr) # Row index pointers

1. **Matrix Operations:**

vector = np.array([1, 2, 3, 4])

result = sparse\_matrix.dot(vector)

print(result)

**When to Use csr\_matrix:**

* When working with large matrices where most elements are zero (e.g., document-term matrices in NLP, adjacency matrices in graph algorithms).
* When performing linear algebra or machine learning on sparse data.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer component

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A white background with black text

Description automatically generated

A close-up of a list of sites

Description automatically generated

Imagine you have a big puzzle made of many small pieces. Each piece contributes to the overall picture, but none of the pieces can "subtract" from the image—they can only "add" to it. **NMF** is a mathematical tool that breaks a large dataset (like the puzzle) into smaller, meaningful pieces (components) that only add up to recreate the original data.

In short:

* **NMF breaks data into parts** that are easy to understand and interpret.
* It works well when the data naturally has non-negative values (like word counts in documents, pixel intensities in images, etc.).

**How does NMF work?**

1. **Input Data (Matrix)**: You have a matrix where rows are "samples" (e.g., articles, images) and columns are "features" (e.g., word counts, pixel values). NMF tries to simplify this matrix into smaller pieces:
   * **Principal Components**: Groups or patterns that explain the data (e.g., topics in text data).
   * **Feature Combinations**: The weights or importance of these patterns for each sample.
2. **Output**:
   * **Two Matrices**:
     + **Components Matrix**: Each row represents a "principal component" or part (e.g., a topic, or a pattern in an image).
     + **Feature Combination Matrix**: Shows how much of each principal component is used to describe a specific sample.

**What are Principal Components in NMF?**

In NMF, **principal components** are the main building blocks or patterns found in the data. For example:

* In text analysis, a principal component could represent a **topic** (e.g., "sports," "politics").
* In image analysis, a principal component could represent a **shape** or **texture** (e.g., edges, colors).

These components are easy to interpret because:

* They only contain positive values (e.g., "Topic A is made of these 5 words").
* They represent **additive parts** of the data.

**What is a Feature Combination?**

The **feature combination** tells you how much of each principal component is present in a sample:

* In text data, it shows how strongly each topic appears in a document.
* In image data, it shows how much of each shape or pattern contributes to a picture.

Think of it like a recipe:

* The principal components are the ingredients (e.g., flour, sugar, eggs).
* The feature combination tells you how much of each ingredient to use for each recipe (e.g., "2 cups of flour, 1 cup of sugar").

**Example in Simple Words:**

**Imagine a Recipe Book:**

1. Each recipe is made of **ingredients** like flour, sugar, and butter.
2. NMF looks at the whole recipe book and figures out:
   * The **main ingredients** (principal components): Flour, sugar, butter.
   * The **proportions of ingredients** (feature combinations) for each recipe: Recipe 1 = 2 cups flour, 1 cup sugar; Recipe 2 = 1 cup flour, 2 cups butter.

**For a Real Dataset:**

* In text:
  + Principal components = **Topics** like "sports" or "movies."
  + Feature combinations = How much each topic appears in each article.
* In images:
  + Principal components = **Basic patterns** like edges or textures.
  + Feature combinations = How much of each pattern is in the image.

In summary:

* **NMF = Break data into understandable parts.**
* **Principal components = Building blocks (topics, patterns).**
* **Feature combinations = How much of each block is used for each sample.**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A diagram of components

Description automatically generated

A white and black text

Description automatically generated with medium confidence

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

The pd.values attribute in pandas is used to retrieve the underlying data of a pandas object (like a DataFrame or Series) as a **NumPy array**. Here's a detailed explanation:

**Usage**

1. **For a Series**:
   * Converts the one-dimensional data into a NumPy array.
   * Example:

import pandas as pd

series = pd.Series([1, 2, 3])

print(series.values) # Output: array([1, 2, 3])

1. **For a DataFrame**:
   * Converts the two-dimensional tabular data into a two-dimensional NumPy array.
   * Example:

df = pd.DataFrame([[1, 2], [3, 4]])

print(df.values) # Output: array([[1, 2],

# [3, 4]])

**Key Points**

* **Type Conversion**: values will attempt to convert all data to the same type, which might lead to upcasting (e.g., integers to floats if there's mixed data).
* **Deprecated Warning**: As of pandas v1.0.0, .values is **deprecated** in favor of .to\_numpy(). Using .to\_numpy() is now the preferred method for converting to NumPy arrays.

**When to Use .to\_numpy() Instead**

series = pd.Series([1, 2, 3])

array = series.to\_numpy()

print(array) # Output: array([1, 2, 3])

**Why Use .values or .to\_numpy()?**

1. If you need to work with data using NumPy operations.
2. When reshaping or applying functions that expect NumPy arrays.

**Building recommender systems using NMF**

A close-up of a white background

Description automatically generated

A yellow text on a white background

Description automatically generated

A screenshot of a computer code

Description automatically generated

A white paper with text and a graph

Description automatically generated

A screenshot of a computer code

Description automatically generated

A graph of a line and a line

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated