CAP 781
MACHINE LEARNING

Tanzeela Javid Kaloo (32638)
Assistant Professor
System And Architecture
Lovely Professional University

UNIT – III
Unsupervised Learning

Tanzeela Javid Kaloo (32638)
Assistant Professor
System And Architecture
Lovely Professional University

Content

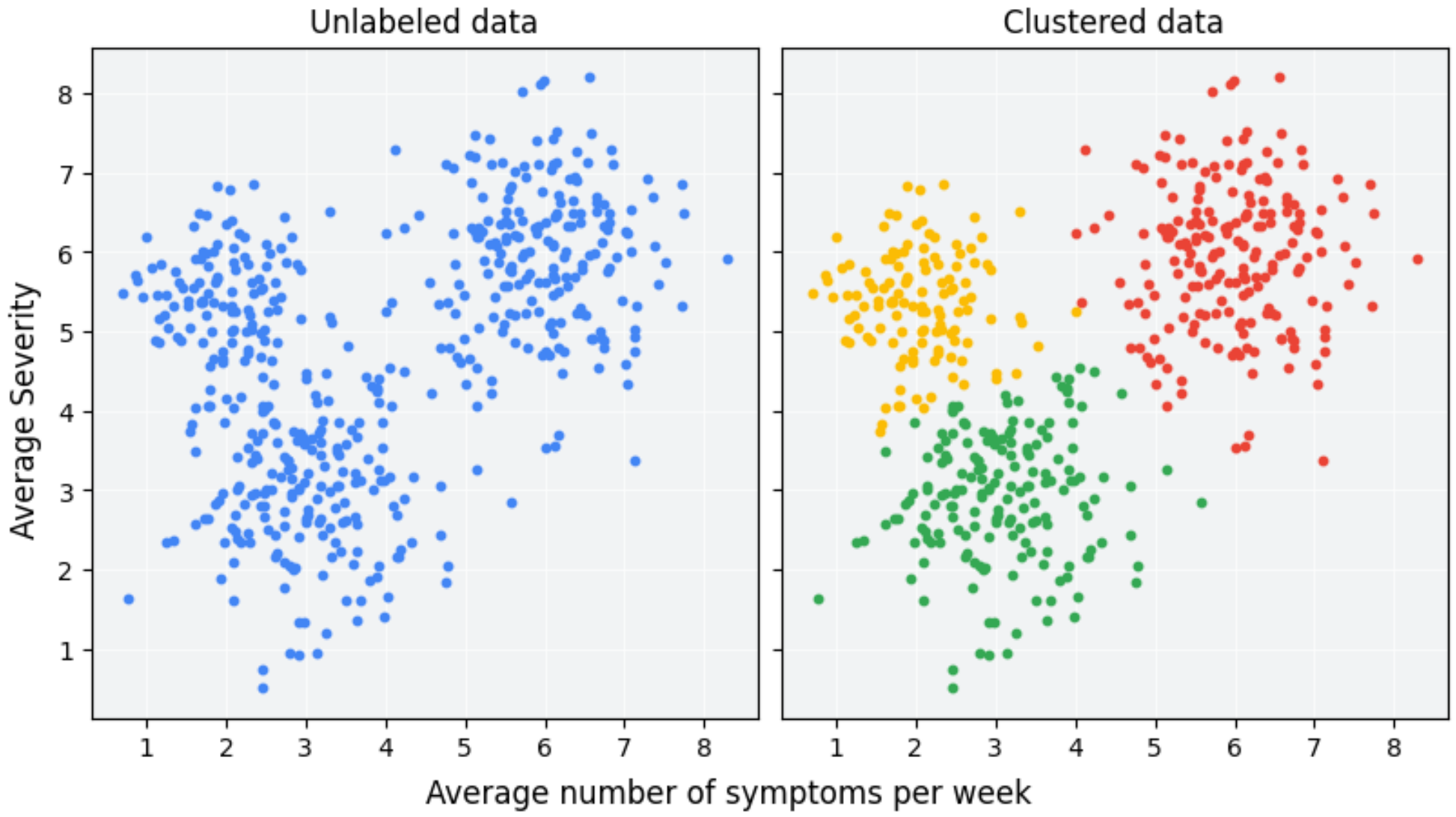
- Clustering
- K-Means
- Hierarchical Clustering
- Dimensionality Reduction
- Principal Component Analysis (PCA)
- Anomaly Detection
- Autoencoders and Feature Learning
- Case Studies and Applications of Unsupervised Learning

Objectives

- Describe clustering use cases in machine learning applications.
- Choose the appropriate similarity measure for an analysis.
- Cluster data with the k-means algorithm.
- Evaluate the quality of clustering results.
- Reduce dimensionality in clustering analysis with an autoencoder.

Clustering

- **Clustering** is an **unsupervised machine learning** technique used to group **unlabeled** examples based on their similarity.
- If the examples are labeled, the process is called **classification** instead of clustering.
- **Example:** A patient study evaluating a new treatment protocol.
- Patients report the **frequency** and **severity** of their symptoms per week.
- **Clustering analysis** helps group patients with **similar treatment responses** into clusters.
- This technique enables researchers to identify **patterns** in treatment responses without predefined labels.


Unlabeled data
Clustered data
Average Severity
Average number of symptoms per week
1
2
3
4
5
6
7
8

Clustering

- **Visual intuition** can sometimes suggest clusters in data, even without a formal definition of similarity.
- **Real-world applications** require an **explicit similarity measure** to compare data points.
- A **similarity measure** (or metric) is defined based on the dataset's features.
- When data has **only a few features**, visualizing and measuring similarity is **straightforward**.
- As the **number of features increases**, comparing and combining features becomes **more complex**.
- **Different similarity measures** may be **more or less appropriate** depending on the clustering scenario.

Clustering

- After clustering, each group is assigned a unique label called a **cluster ID**.
- Clustering is powerful because it can simplify large, complex datasets with many features to a single cluster ID.

Common Applications For Clustering

Clustering is useful in a variety of industries. Some common applications for clustering:

- Market segmentation
- Social network analysis
- Search result grouping
- Medical imaging
- Image segmentation
- Anomaly detection

Clustering Uses Cases

- Imputation
- Data Compression
- Privacy Preservation

Clustering Algorithms

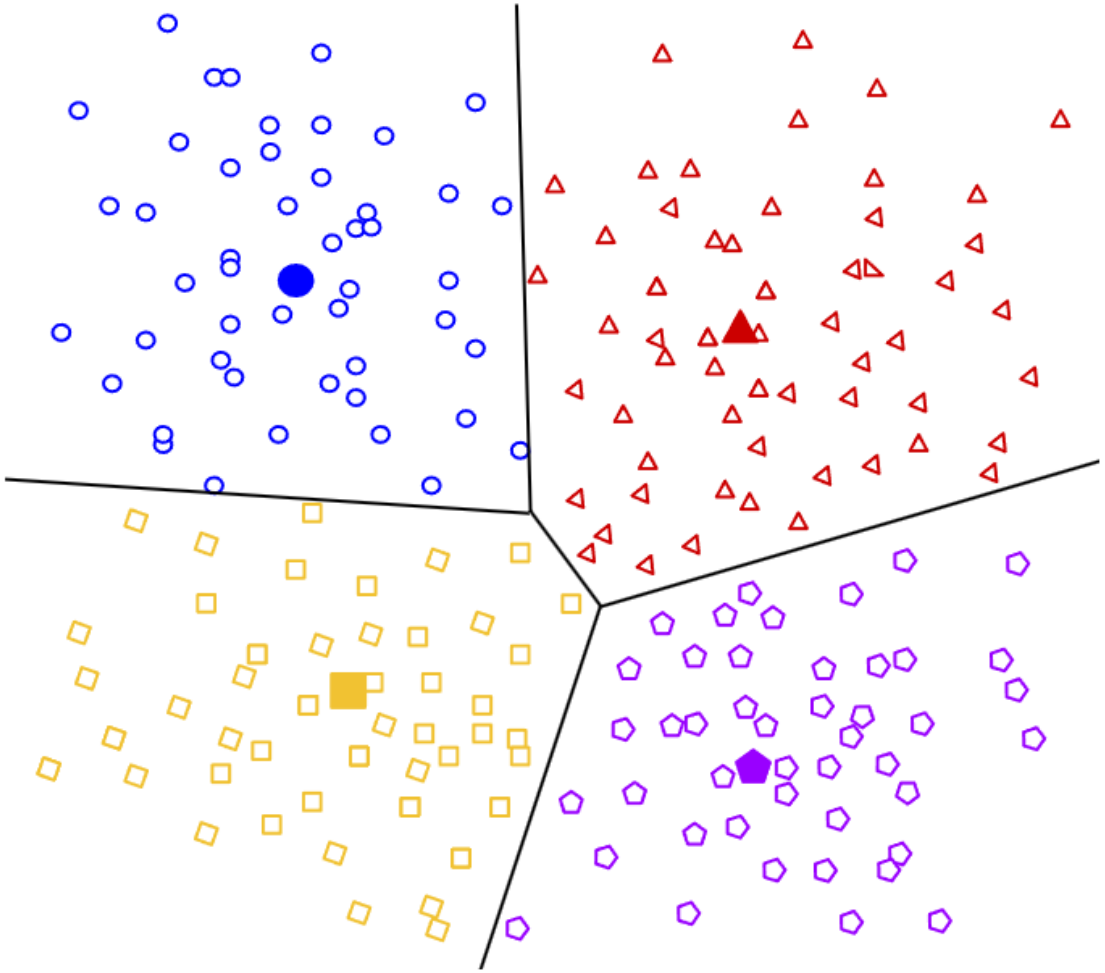
- **Machine learning datasets** can contain **millions of examples**, making scalability a key concern.
- **Not all clustering algorithms** scale efficiently to large datasets.
- Many clustering algorithms compute **pairwise similarity**, leading to **quadratic time complexity** $O(n^2)$.
- **Quadratic complexity** makes these algorithms **impractical** for very large datasets.
- **k-means clustering** has a **linear time complexity** $O(n)$, making it **more scalable**.
- **k-means algorithm** is the most popular due to its efficiency and scalability.

Types of Clustering

- Centroid-based clustering
- Density-based clustering
- Distribution-based clustering
- Hierarchical clustering

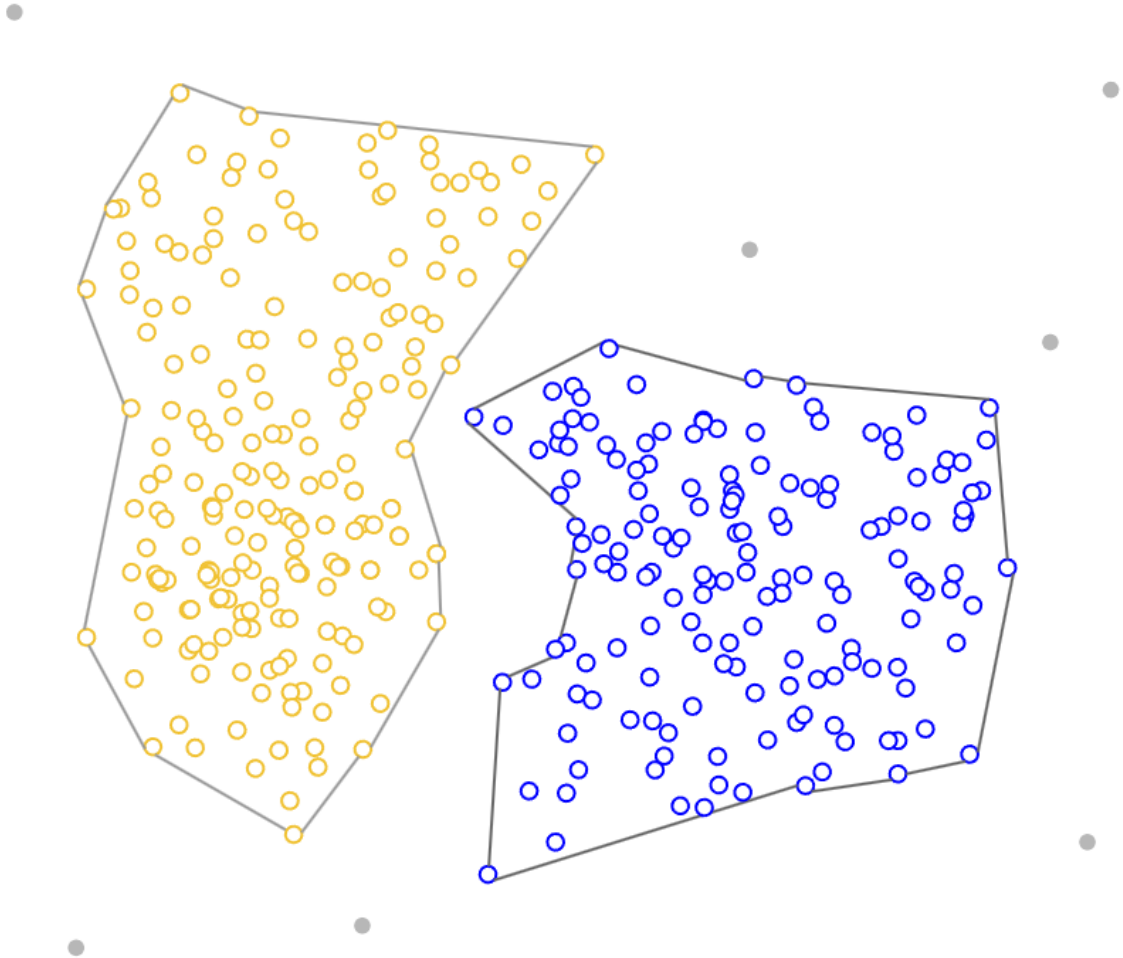
Centroid-based clustering

The **centroid** of a cluster is the arithmetic mean of all the points in the cluster. **Centroid-based clustering** organizes the data into non-hierarchical clusters. Centroid-based clustering algorithms are efficient but sensitive to initial conditions and outliers. Of these, k-means is the most widely used. It requires users to define the number of centroids, *k*, and works well with clusters of roughly equal size.



Density-based clustering

Density-based clustering connects contiguous areas of high example density into clusters. This allows for the discovery of any number of clusters of any shape. Outliers are not assigned to clusters. These algorithms have difficulty with clusters of different density and data with high dimensions.



Distribution-based clustering

This clustering approach assumes data is composed of probabilistic distributions, such as **Gaussian distributions**. In Figure 3, the distribution-based algorithm clusters data into three Gaussian distributions. As distance from the distribution's center increases, the probability that a point belongs to the distribution decreases. The bands show that decrease in probability. When you're not comfortable assuming a particular underlying distribution of the data, you should use a different algorithm.

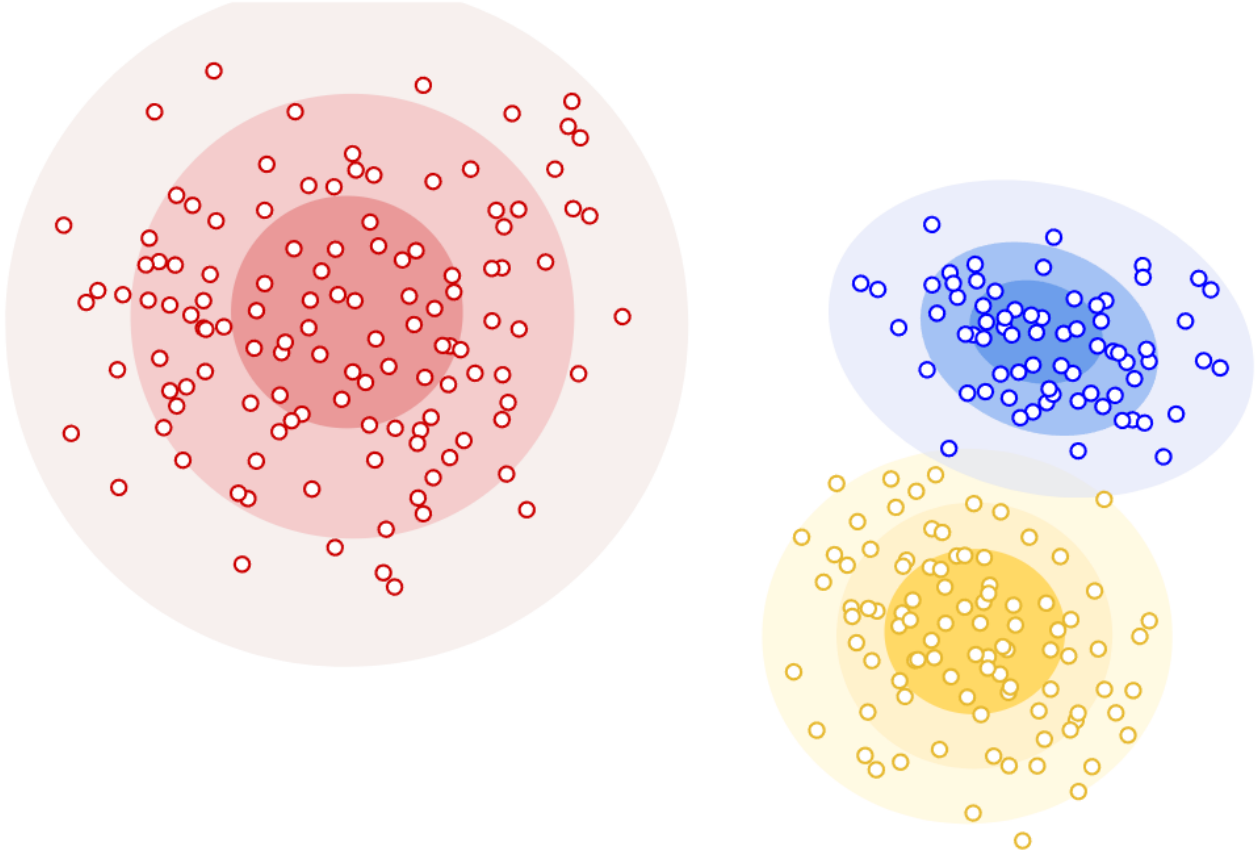
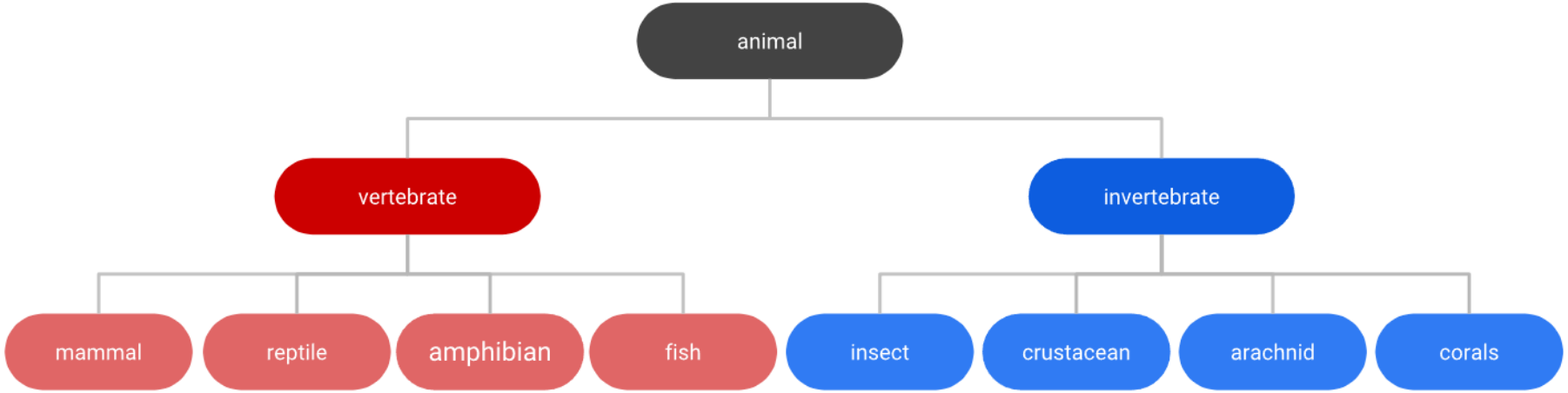
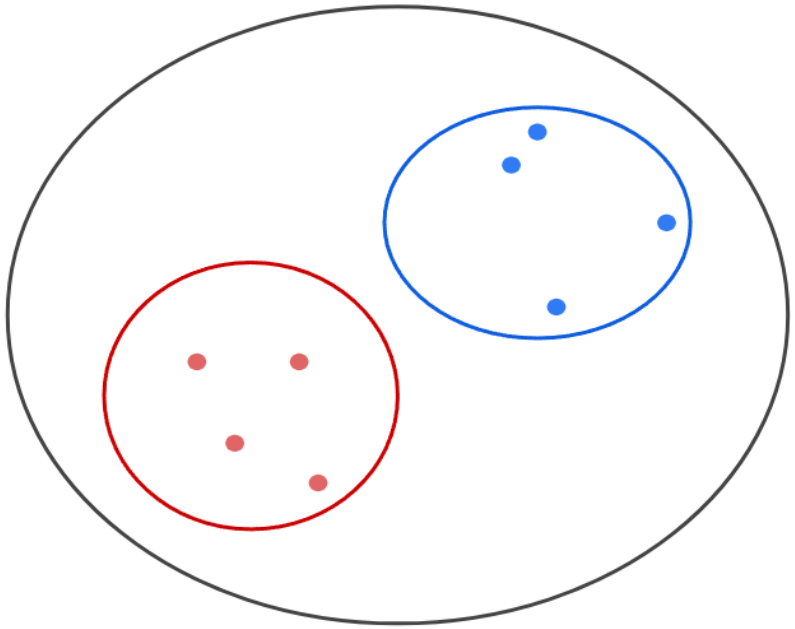


Figure 3: Example of distribution-based clustering.



Hierarchical clustering creates a tree of clusters. Hierarchical clustering, not surprisingly, is well suited to hierarchical data, such as taxonomies.

Any number of clusters can be chosen by cutting the tree at the right level.



Clustering workflow

- To cluster your data, you'll follow these steps:
  - Prepare data.
  - Create similarity metric.
  - Run clustering algorithm.
  - Interpret results and adjust your clustering.

Preparing Data

•**Clustering** requires calculating **similarity** between data points by combining all feature values into a **numeric measure**.
•**Feature scaling** is essential to ensure fair comparisons between features.
•Scaling can be achieved through **normalization**, **transformation**, or **creating quantiles**.
•**Normalization** adjusts feature values to a common scale, typically between 0 and 1.
•**Transformation** applies mathematical functions (e.g., log, square root) to adjust feature distributions.
•**Quantiles** divide data into equal-sized groups, making it useful when the **distribution is unknown**.
•If unsure about the **data distribution**, using **quantiles** is a safe default approach.

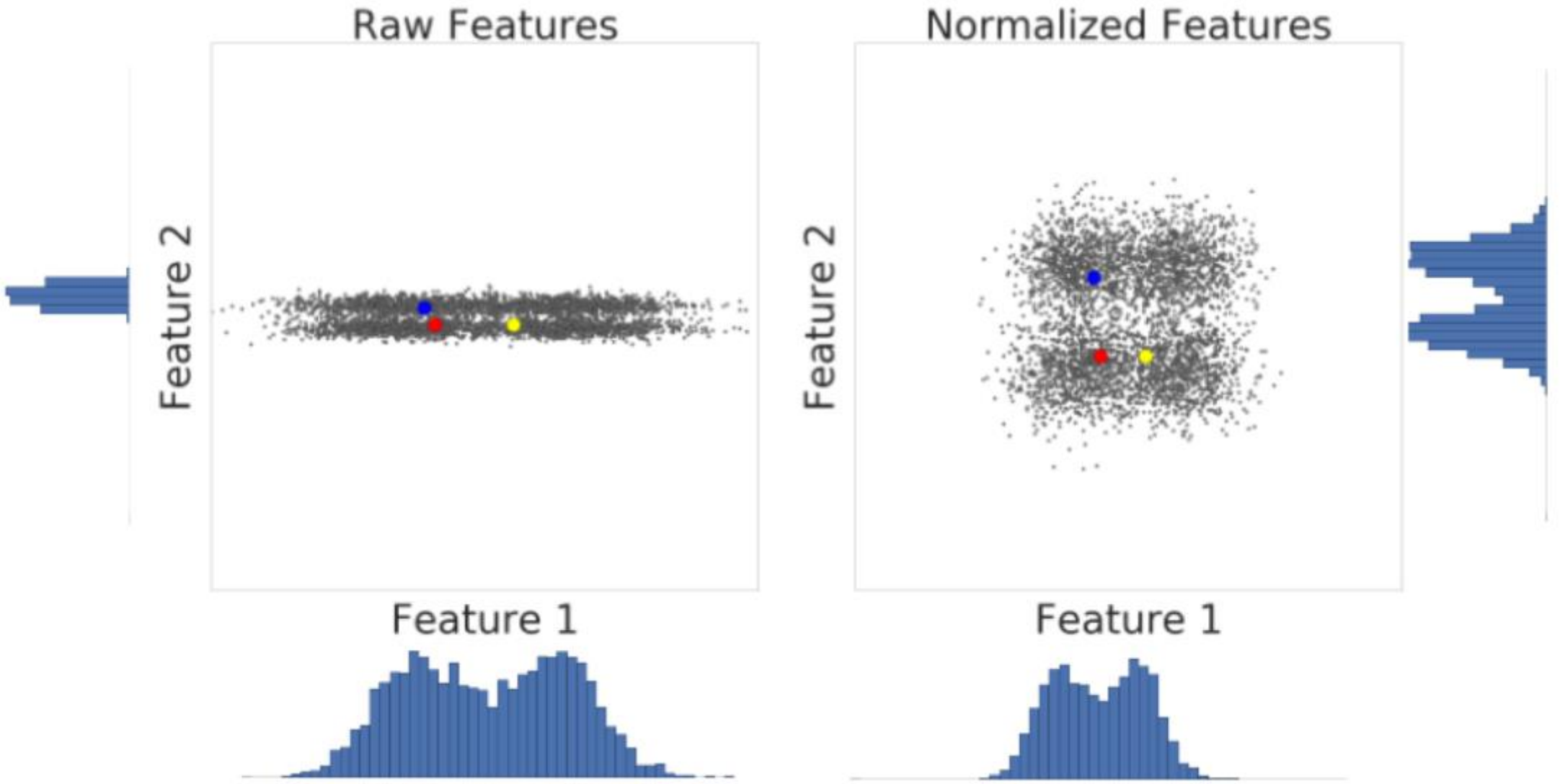


Figure 1: A comparison of feature data before and after normalization.

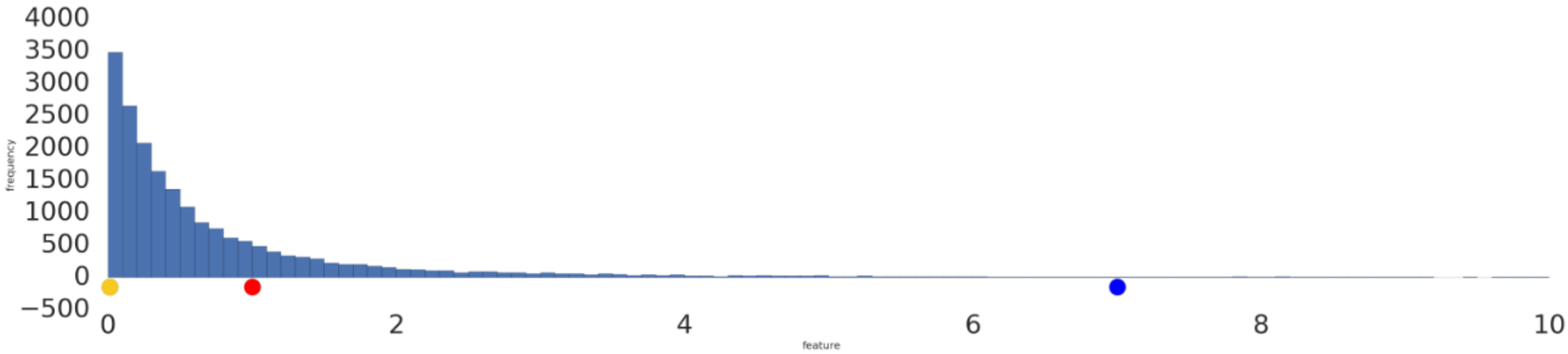


Figure 2: A power law distribution.

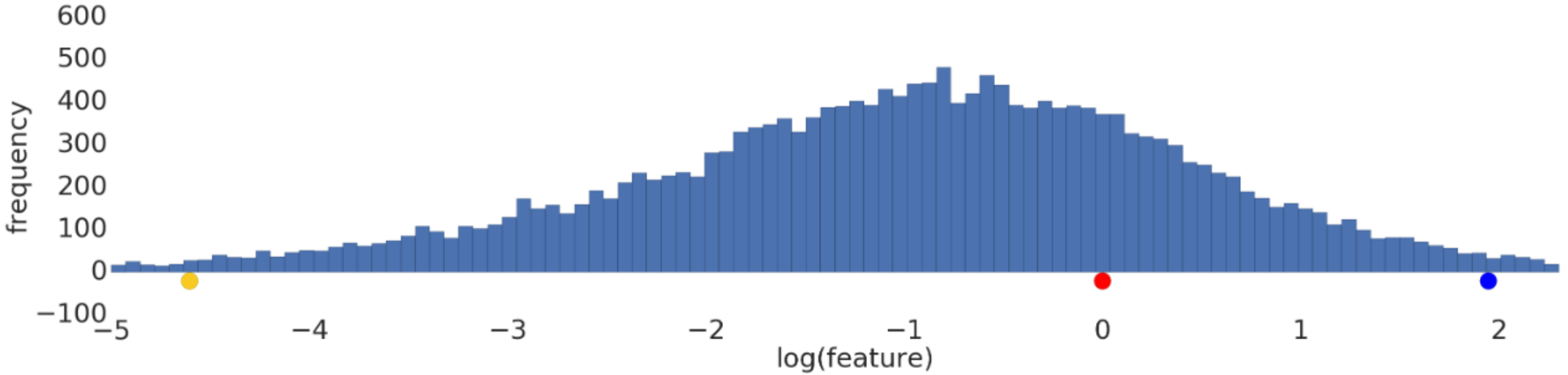


Figure 3: A log transform of Figure 2.

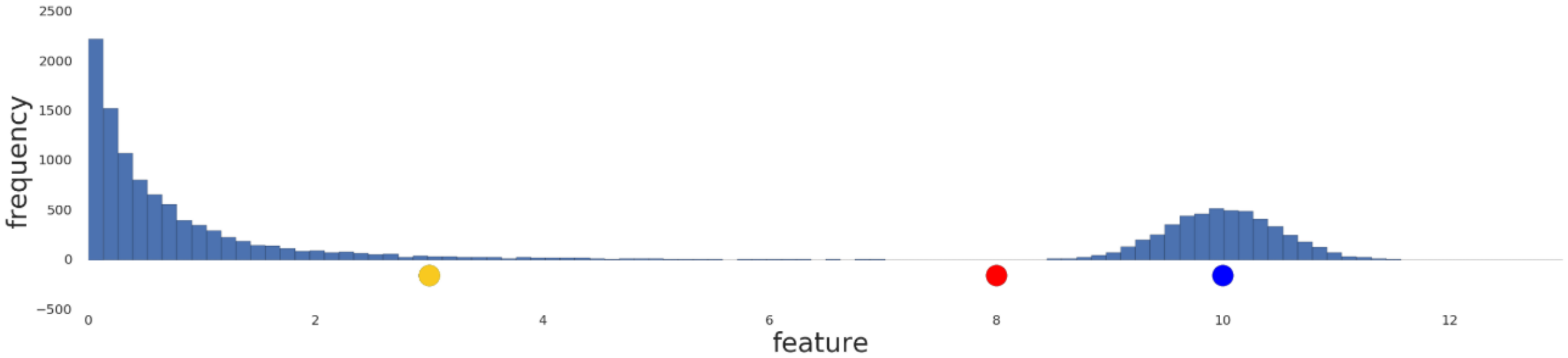


Figure 4: An uncategorizable distribution prior to any preprocessing.

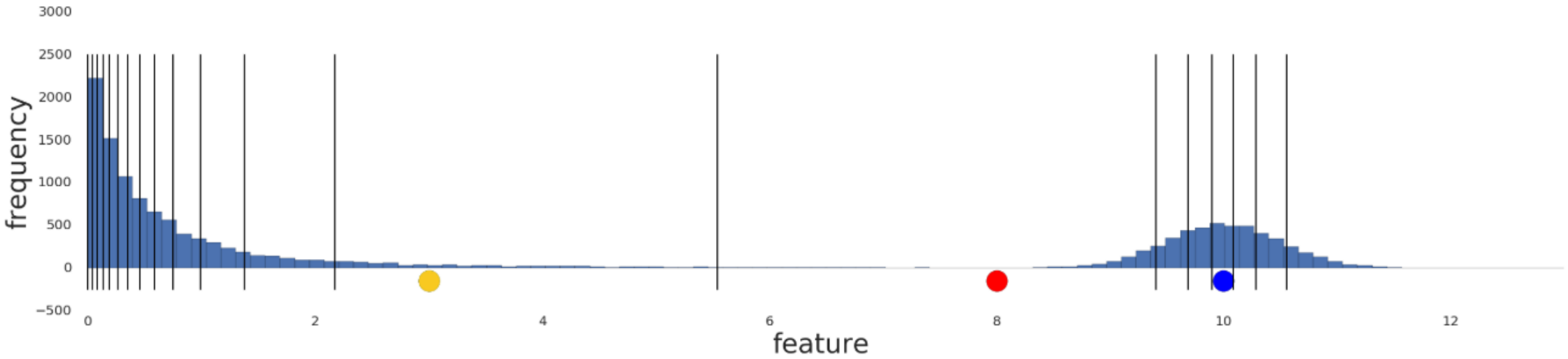
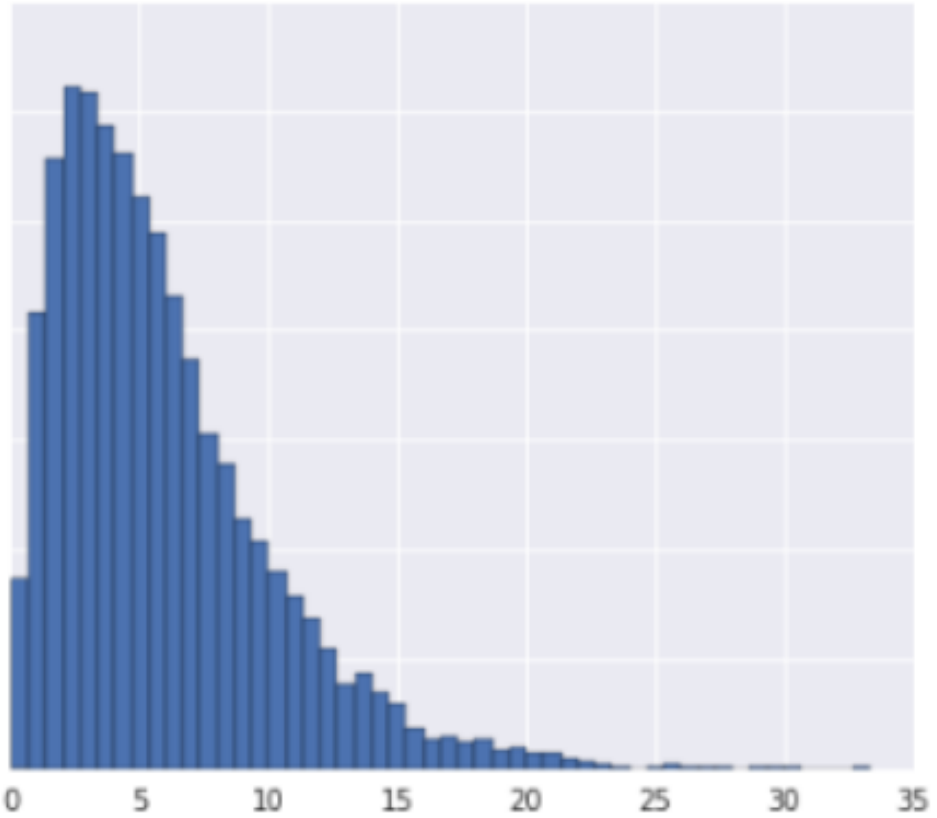


Figure 5: The distribution in Figure 4 after conversion into 20 quantiles.

Question one



Question two

K-Means

K-means is often referred to as Lloyd's algorithm

The k-means algorithm divides a set of N samples X into K disjoint clusters C, each described by the mean μ_j of the samples in the cluster. The means are commonly called the cluster "centroids"; note that they are not, in general, points from X, although they live in the same space.

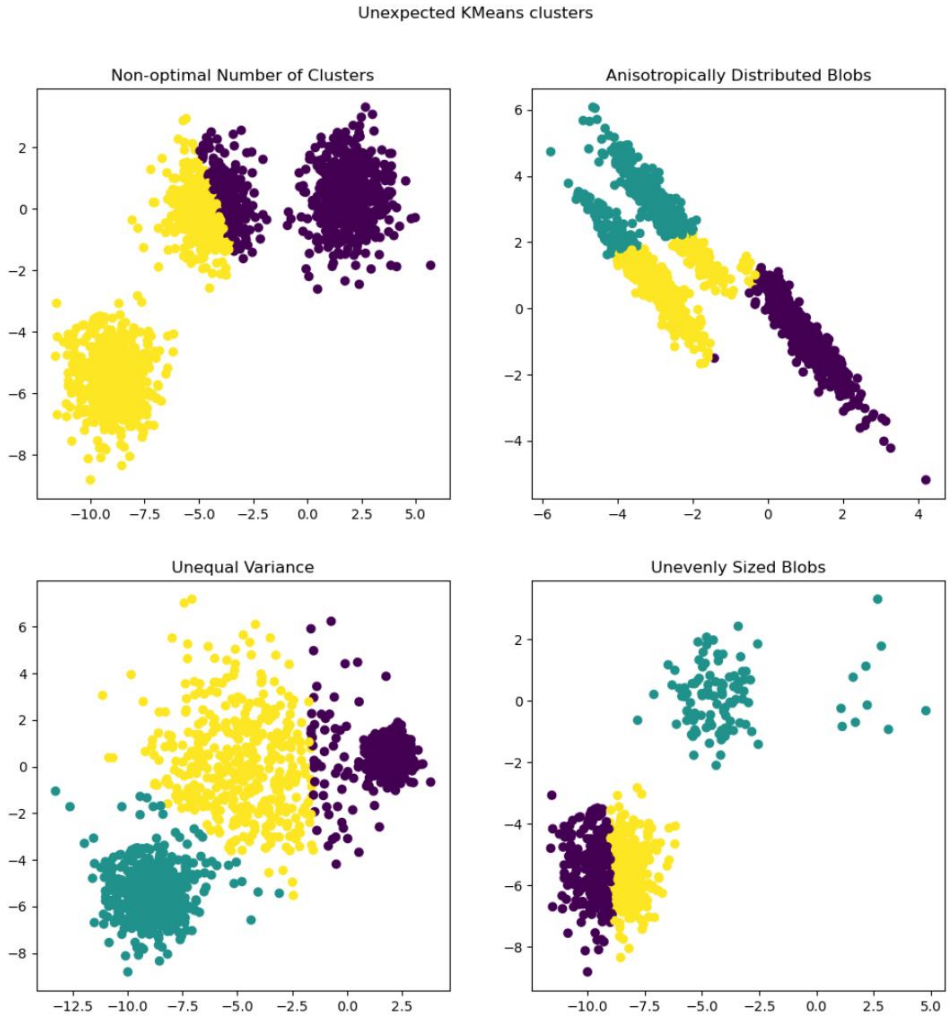
The K-means algorithm aims to choose centroids that minimise the **inertia**, or **within-cluster sum-of-squares criterion**:

$$\sum_{i=0}^{n} \min_{\mu_j \in C} (||x_i - \mu_j||^2)$$

Drawbacks of Inertia

Inertia can be recognized as a measure of how internally coherent clusters are. It suffers from various drawbacks:

- Inertia makes the assumption that clusters are convex and isotropic, which is not always the case. It responds poorly to elongated clusters, or manifolds with irregular shapes.
- Inertia is not a normalized metric: we just know that lower values are better and zero is optimal. But in very high-dimensional spaces, Euclidean distances tend to become inflated (this is an instance of the so-called "curse of dimensionality"). Running a dimensionality reduction algorithm such as Principal component analysis (PCA) prior to k-means clustering can alleviate this problem and speed up the computations.


Unexpected KMeans clusters
Non-optimal Number of Clusters
Anisotropically Distributed Blobs
Unequal Variance
Unevenly Sized Blobs

k-means clustering algorithm

The algorithm follows these steps:

1. Provide an initial guess for k, which can be revised later. For this example, we choose $k = 3$.
2. Randomly choose k centroids.

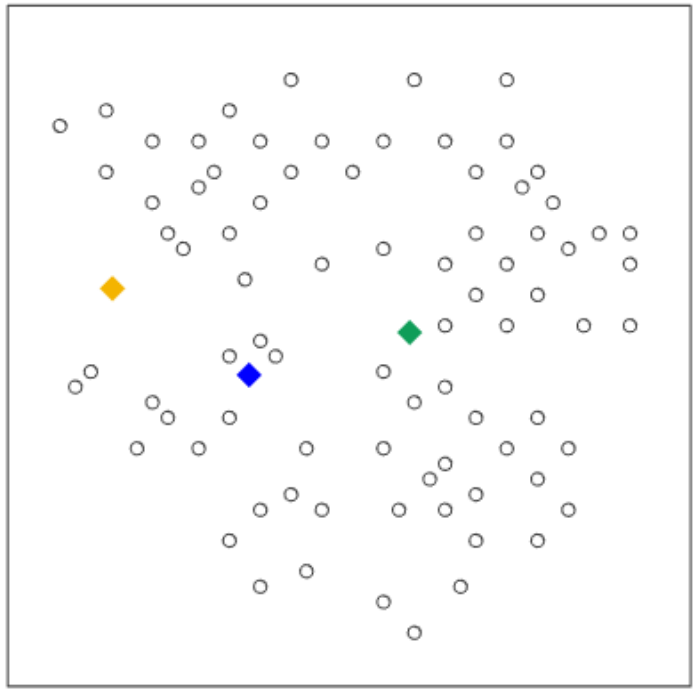


Figure 1: k-means at initialization.

3. Assign each point to the nearest centroid to get k initial clusters.

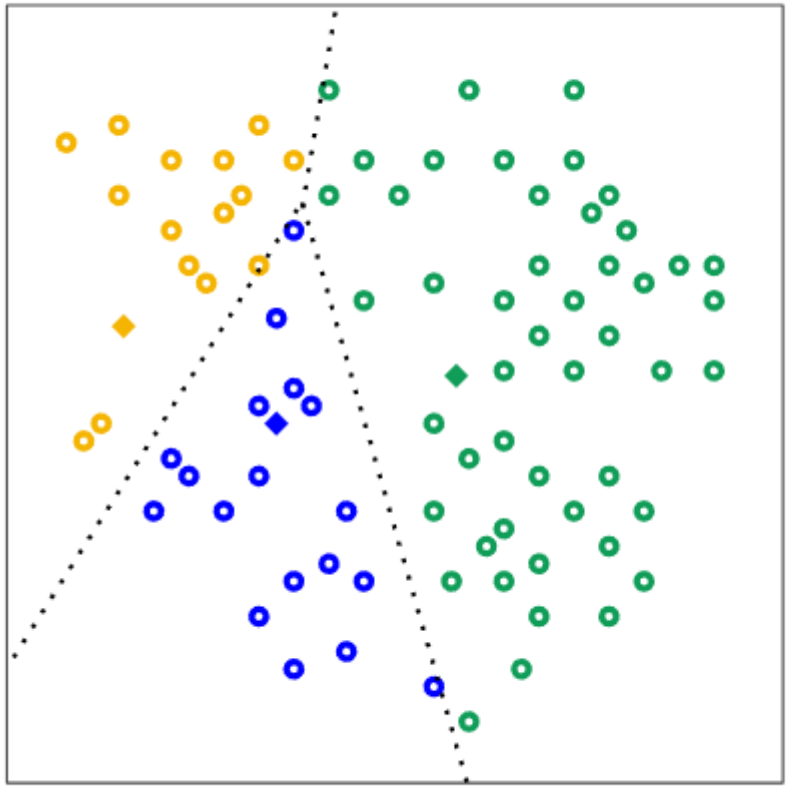


Figure 2: Initial clusters.

4. For each cluster, calculate a new centroid by taking the mean position of all points in the cluster. The arrows in Figure 4 show the change in centroid positions.

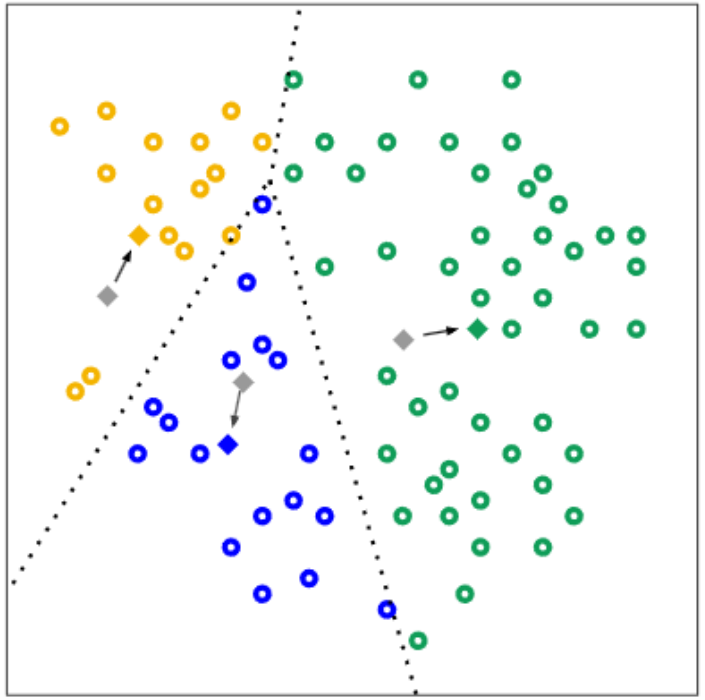


Figure 3: Recomputed centroids.

5. Reassign each point to the nearest new centroid.

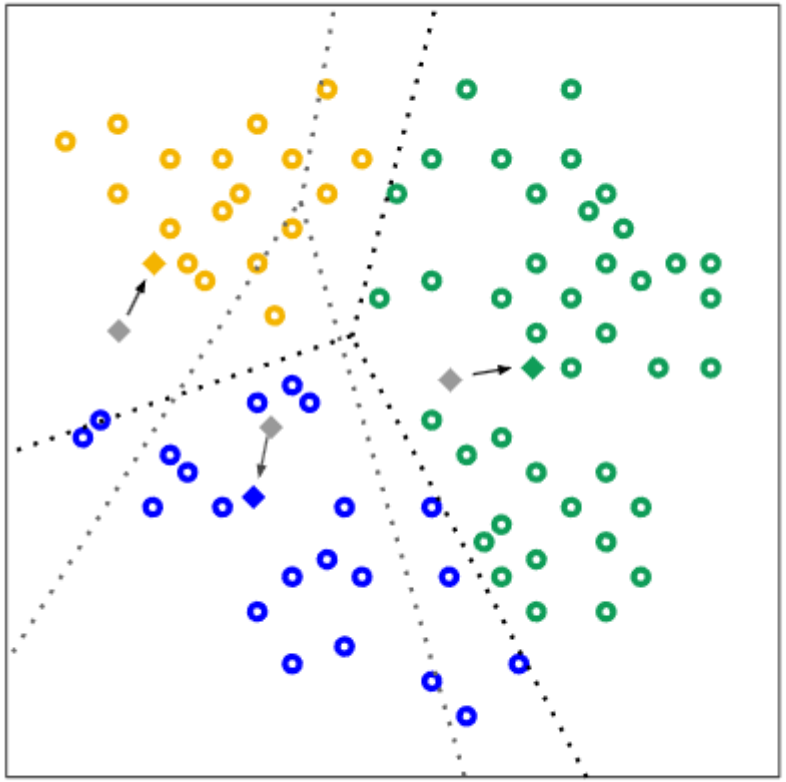


Figure 4: Clusters after reassignment.

6. Repeat steps 4 and 5, recalculating centroids and cluster membership, until points no longer change clusters. In the case of large datasets, you can stop the algorithm before convergence based on other criteria.

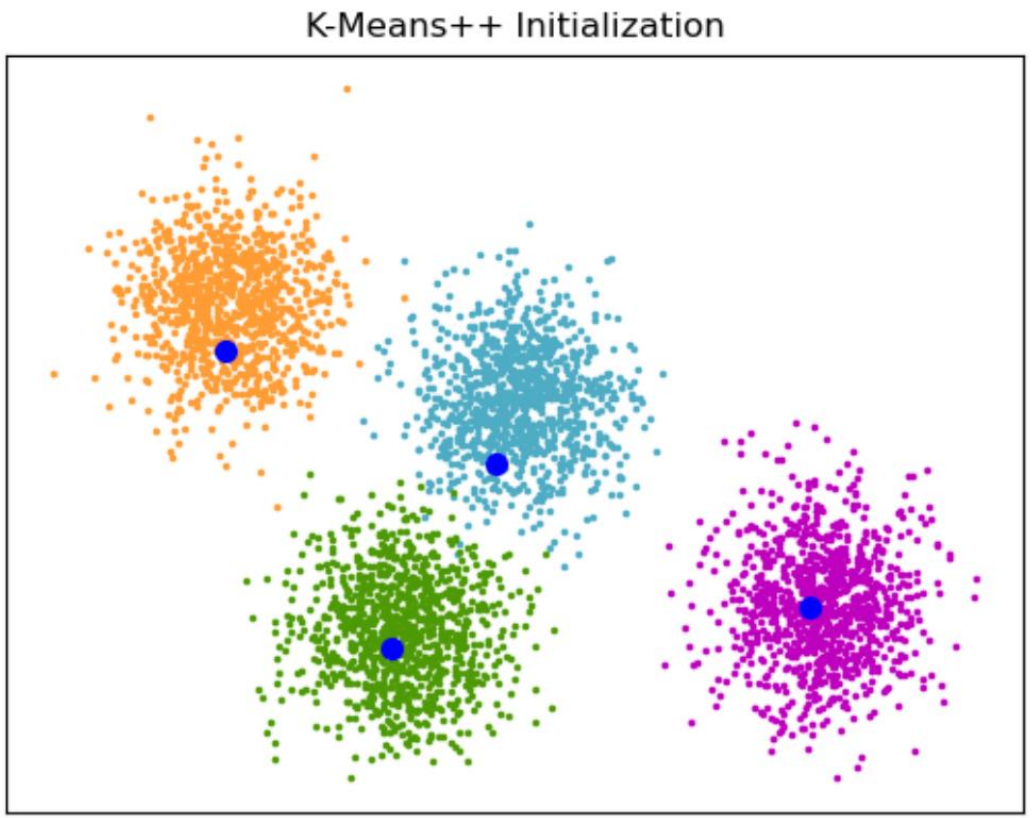
Because the centroid positions are initially chosen at random, k-means can return significantly different results on successive runs. To solve this problem, run k-means multiple times and choose the result with the best quality metrics.

k-means++

Given enough time, K-means will always converge, however this may be to a local minimum. This is highly dependent on the initialization of the centroids. As a result, the computation is often done several times, with different initializations of the centroids. One method to help address this issue is the k-means++ initialization scheme, which has been implemented in scikit-learn (**use the init='k-means++' parameter**). This initializes the centroids to be (generally) distant from each other, leading to probably better results than random initialization.

An example to show the output of the `sklearn.cluster.kmeans_plusplus` function for generating initial seeds for clustering.

K-Means++ is used as the default initialization for K-means.



Similarity Measures

- k-means assigns points to their closest centroid. But what does "closest" mean?

- To apply k-means to feature data, you will need to define a measure of similarity that combines all the feature data into a single numeric value, called a **manual similarity measure**.

As just shown, k-means assigns points to their closest centroid. But what does "closest" mean?

To apply k-means to feature data, you will need to define a measure of similarity that combines all the feature data into a single numeric value, called a **manual similarity measure**.

Consider a shoe dataset. If that dataset has shoe size as its only feature, you can define the similarity of two shoes in terms of the difference between their sizes. The smaller the numerical difference between sizes, the greater the similarity between shoes.

If that shoe dataset had two numeric features, size and price, you can combine them into a single number representing similarity. First scale the data so both features are comparable:

- Size (s): Shoe size probably forms a Gaussian distribution. Confirm this. Then normalize the data.
- Price (p): The data is probably a Poisson distribution. Confirm this. If you have enough data, convert the data to quantiles and scale to $[0, 1]$.

Next, combine the two features by calculating the root mean squared error (RMSE). This rough measure of similarity is given by $\sqrt{\frac{(s_i - s_j)^2 + (p_i - p_j)^2}{2}}$.

For a simple example, calculate similarity for two shoes with US sizes 8 and 11, and prices 120 and 150. Since we don't have enough data to understand the distribution, we'll scale the data without normalizing or using quantiles.

Action	Method
Scale the size.	Assume a maximum possible shoe size of 20. Divide 8 and 11 by the maximum size 20 to get 0.4 and 0.55.
Scale the price.	Divide 120 and 150 by the maximum price 150 to get 0.8 and 1.
Find the difference in size.	$0.55 - 0.4 = 0.15$
Find the difference in price.	$1 - 0.8 = 0.2$
Calculate the RMSE.	$\sqrt{\frac{0.2^2 + 0.15^2}{2}} = 0.17$

Intuitively, your similarity measure should increase when feature data is more similar. Instead, your similarity measure (RMSE) actually decreases. Make your similarity measure follow your intuition by subtracting it from 1.

$$\text{Similarity} = 1 - 0.17 = 0.83$$

- What if that dataset included both shoe size and shoe color? Color is categorical data.
- Categorical data is harder to combine with the numerical size data. It can be:

  - Single-valued (univalent), such as a car's color ("white" or "blue" but never both)
  - Multi-valued (multivalent), such as a movie's genre (a movie can be both "action" and "comedy," or only "action")

If univalent data matches, for example in the case of two pairs of blue shoes, the similarity between the examples is 1. Otherwise, similarity is 0.

Multivalent data, like movie genres, is harder to work with. If there are a fixed set of movie genres, similarity can be calculated using the ratio of common values, called **Jaccard similarity**. Example calculations of Jaccard similarity:

- [“comedy”,”action”] and [“comedy”,”action”] = 1
- [“comedy”,”action”] and [“action”] = ½
- [“comedy”,”action”] and [“action”, "drama"] = ⅓
- [“comedy”,”action”] and [“non-fiction”,”biographical”] = 0

Jaccard similarity is not the only possible manual similarity measure for categorical data. Two other examples:

- **Postal codes** can be converted into latitude and longitude before calculating Euclidean distance between them.
- **Color** can be converted into numeric RGB values, with differences in values combined into Euclidean distance.

In general, a manual similarity measure must directly correspond to actual similarity. If your chosen metric does not, then it isn't encoding the information you want it to encode.

As the complexity of data increases, it becomes harder to create a manual similarity measure. In that situation, switch to a **supervised similarity measure**, where a supervised machine learning model calculates similarity.

Evaluating results

Because clustering is unsupervised, no ground truth is available to verify results. The absence of truth complicates assessments of quality. Moreover, real-world datasets typically don't offer obvious clusters of examples as in the example shown in Figure 1.

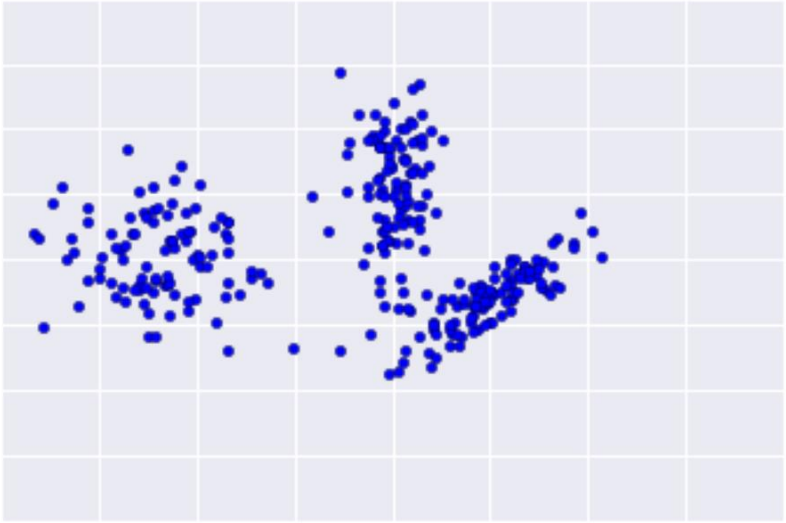


Figure 1: An ideal data plot. Real-world data rarely looks like this.

Instead, real-world data often looks more like Figure 2, making it difficult to visually assess clustering quality.

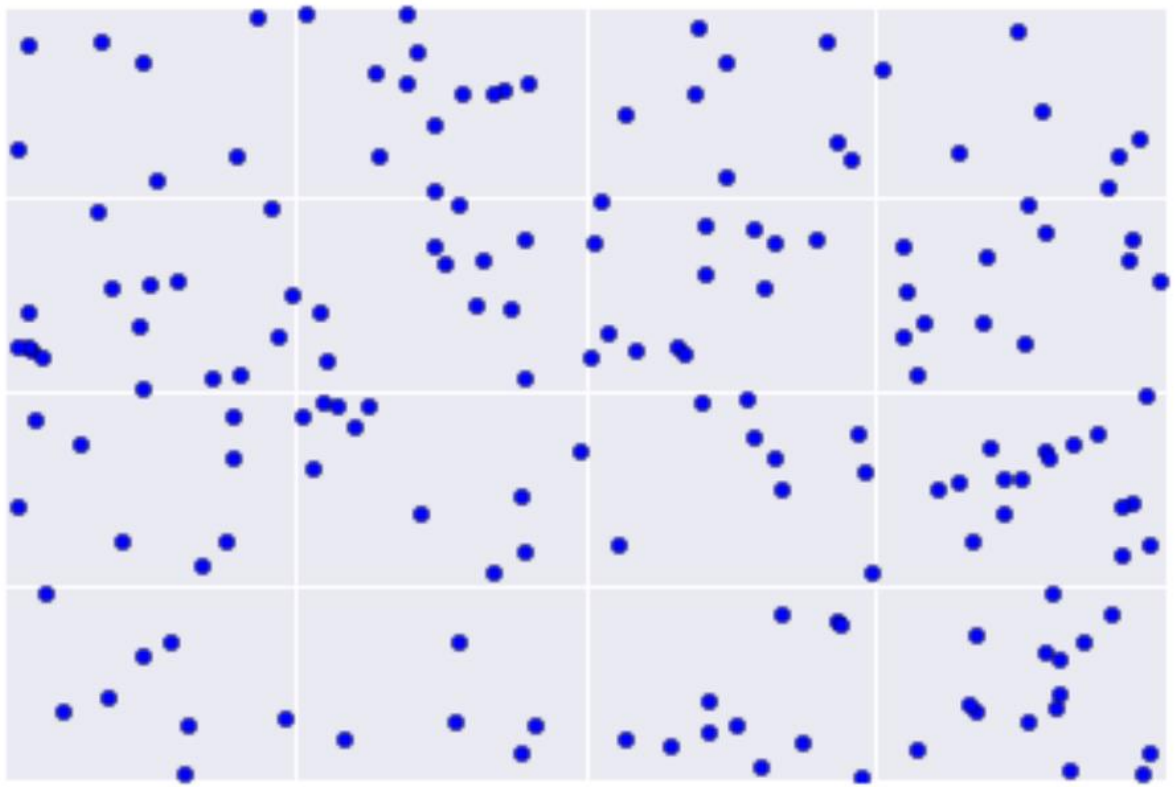
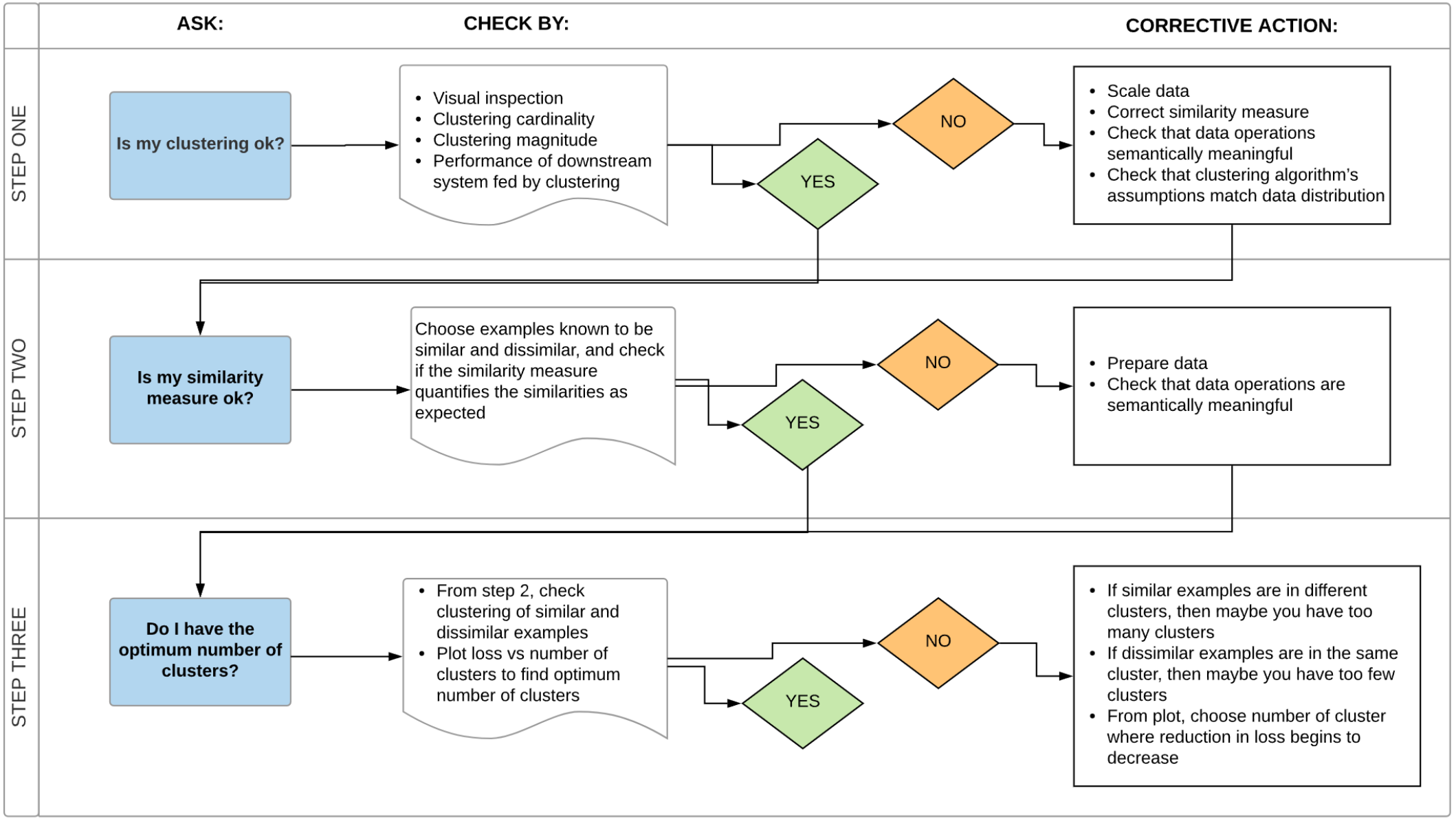


Figure 2: A more realistic data plot


ASK:
CHECK BY:
CORRECTIVE ACTION:
STEP ONE
Is my clustering ok?
Visual inspection
Clustering cardinality
Clustering magnitude
Performance of downstream system fed by clustering
YES
NO
Scale data
Correct similarity measure
Check that data operations semantically meaningful
Check that clustering algorithm's assumptions match data distribution
STEP TWO
Is my similarity measure ok?
Choose examples known to be similar and dissimilar, and check if the similarity measure quantifies the similarities as expected
YES
NO
Prepare data
Check that data operations are semantically meaningful
STEP THREE
Do I have the optimum number of clusters?
From step 2, check clustering of similar and dissimilar examples
Plot loss vs number of clusters to find optimum number of clusters
YES
NO
If similar examples are in different clusters, then maybe you have too many clusters
If dissimilar examples are in the same cluster, then maybe you have too few clusters
From plot, choose number of cluster where reduction in loss begins to decrease

Step 1: Assess quality of clustering

First check that the clusters look as you'd expect, and that examples you consider similar to each other appear in the same cluster.

Then check these commonly-used metrics (not an exhaustive list):

- Cluster cardinality
- Cluster magnitude
- Downstream performance

Cluster cardinality

Cluster cardinality is the number of examples per cluster. Plot the cluster cardinality for all clusters and investigate clusters that are major outliers. In Figure 2, this would be cluster 5.

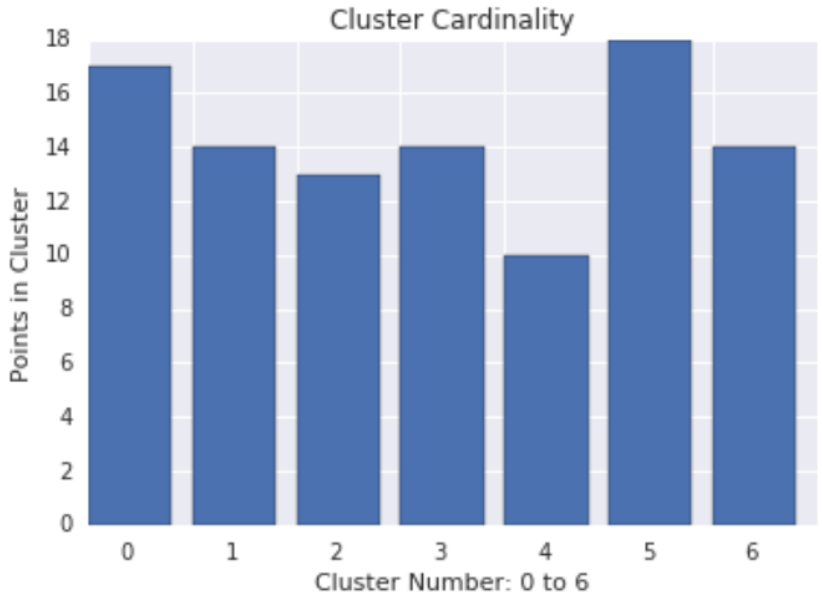


Figure 2: Cardinality of several clusters.

Cluster magnitude

Cluster magnitude is the sum of distances from all examples in a cluster to the cluster's centroid. Plot cluster magnitude for all clusters and investigate outliers. In Figure 3, cluster 0 is an outlier.

Also consider looking at the maximum or average distance of examples from centroids, by cluster, to find outliers.

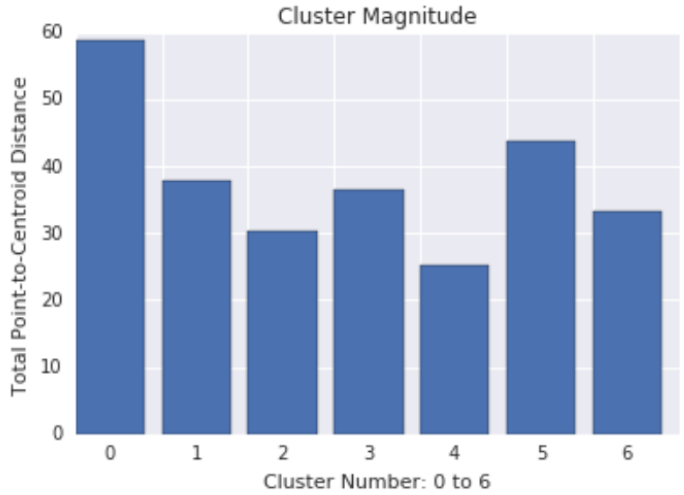


Figure 3: Magnitude of several clusters.

Magnitude versus cardinality

You may have noticed that a higher cluster cardinality corresponds to a higher cluster magnitude, which makes intuitive sense, since the more points in a cluster (cardinality), the greater the probable sum of the distances of those points from the centroid (magnitude). You can also identify anomalous clusters by looking for ones where this relationship between cardinality and magnitude is very different than for other clusters. In Figure 4, fitting a line to the plot of cardinality and magnitude suggests that cluster 0 is anomalous. (Cluster 5 is also far from the line, but if cluster 0 were omitted, the new fitted line would be much closer to cluster 5.)

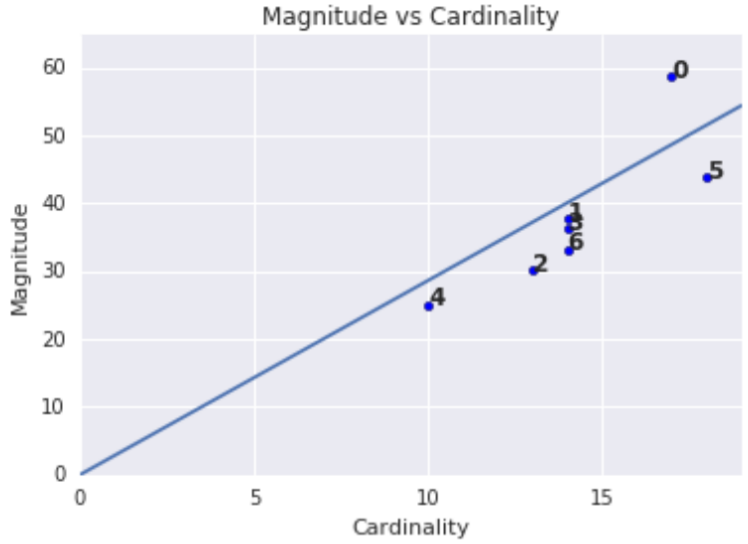


Figure 4: Cardinality vs. magnitude for the previously shown clusters.

Downstream performance

Since clustering outputs are often used in downstream ML systems, see if downstream model performance improves when your clustering process changes. This offers a real-world evaluation of the quality of your clustering results, although it can be complex and expensive to conduct this kind of test.

Step 2: Reassess your similarity measure

Your clustering algorithm is only as good as your similarity measure. Make sure your similarity measure returns sensible results. A quick check is to identify pairs of examples known to be more or less similar. Calculate the similarity measure for each pair of examples, and compare your results to your knowledge: pairs of similar examples should have a higher similarity measure than pairs of dissimilar examples.

The examples you use to spot-check your similarity measure should be representative of the dataset, so you can be confident that your similarity measure holds for all your examples. The performance of your similarity measure, whether manual or supervised, must be consistent across your dataset. If your similarity measure is inconsistent for some examples, those examples won't be clustered with similar examples.

If you find examples with inaccurate similarity scores, then your similarity measure probably doesn't fully capture the feature data that distinguishes those examples. Experiment with your similarity measure until it returns more accurate and consistent results.

Step 3: Find the optimal number of clusters

k-means requires you to decide the number of clusters k beforehand. How do you determine an optimal k? Try running the algorithm with increasing values of k and note the sum of all cluster magnitudes. As k increases, clusters become smaller, and the total distance of points from centroids decreases. We can treat this total distance as a loss. Plot this distance against the number of clusters.

As shown in Figure 5, above a certain k, the reduction in loss becomes marginal with increasing k. Consider using the k where the slope first has a drastic change, which is called the elbow method. For the plot shown, the optimal k is approximately 11. If you prefer more granular clusters, you can choose a higher k, consulting this plot.

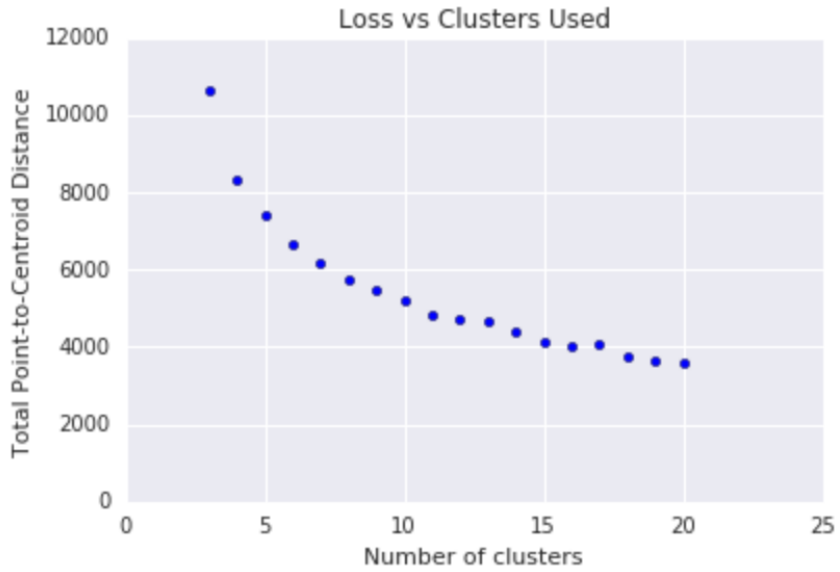


Figure 5: Loss versus number of clusters

Advantages of k-means

- 👍 **Relatively simple to implement.**
- 👍 **Scales to large data sets.**
- 👍 **Always converges.**
- 👍 **Allows warm-starting the positions of centroids.**
- 👍 **Smoothly adapts to new examples.**
- 👍 **Can be generalized to clusters of different shapes and sizes, such as elliptical clusters.**

Generalizing k-means

A straightforward implementation of k-means can struggle with clusters of different densities and sizes. The left side of Figure 1 shows the clusters we'd expect to see, while the right side shows the clusters proposed by k-means.

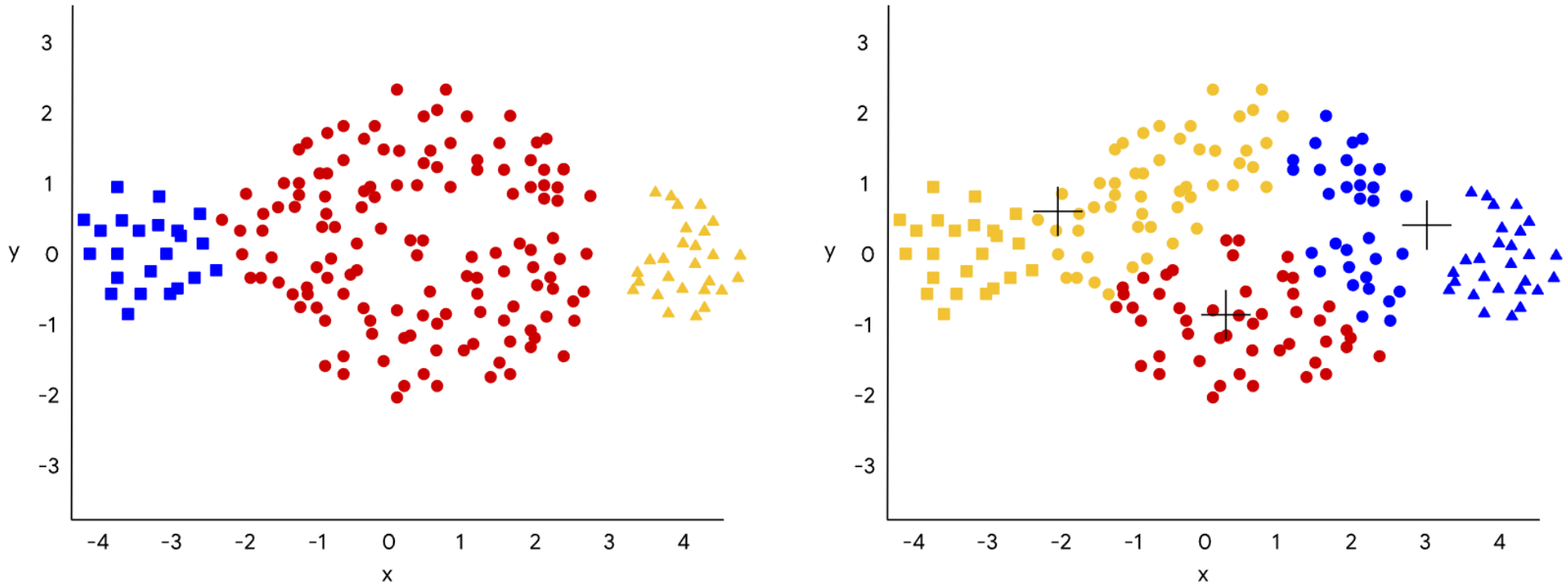


Figure 1: Ungeneralized k-means example.

For better performance on imbalanced clusters like the ones shown in Figure 1, you can generalize, that is, adapt, k-means. Figure 2 shows three different datasets clustered with two different generalizations. The first dataset shows k-means without generalization, while the second and third allow for clusters to vary in width.

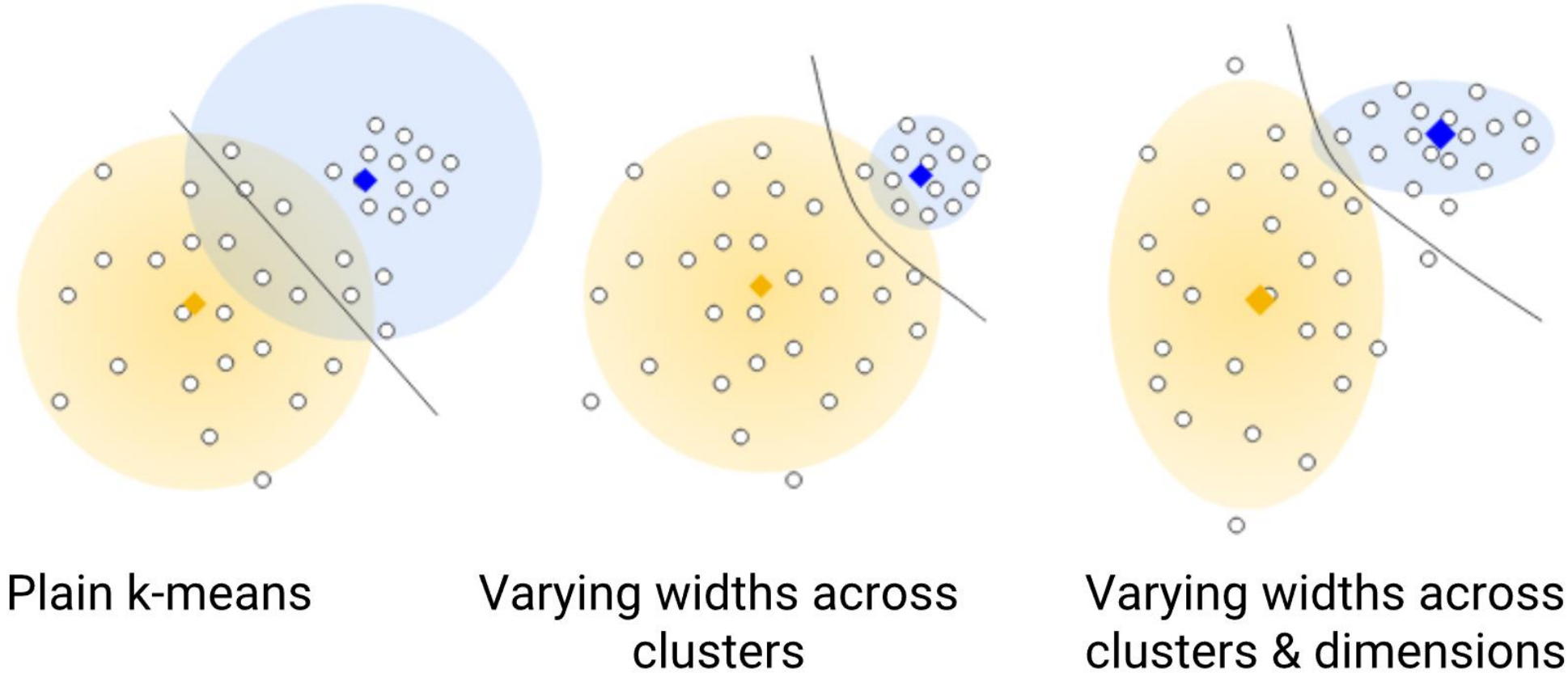


Figure 2: k-means clustering with and without generalization.

Disadvantages of k-means

👎 **k must be chosen manually.**

👎 **Results depend on initial values.**

For low k, you can mitigate this dependence by running k-means several times with different initial values and picking the best result. As k increases, you need **k-means seeding** to pick better initial centroids For a full discussion of k-means seeding, see "A Comparative Study of Efficient Initialization Methods for the K-means Clustering Algorithm," by M. Emre Celebi, Hassan A. Kingravi, and Patricio A. Vela.

👎 **Difficulty clustering data of varying sizes and densities without generalization.**

👎 **Difficulty clustering outliers.**

Centroids can be dragged by outliers, or outliers might get their own cluster instead of being ignored. Consider removing or clipping outliers before clustering.

👎 **Difficulty scaling with number of dimensions.**

As the number of dimensions in the data increases, a distance-based similarity measure converges to a constant value between any given examples. Reduce dimensionality either by using **PCA** on the feature data or by using **spectral clustering** to modify the clustering algorithm.

Curse of dimensionality and spectral clustering

In these three plots, notice how, as dimensions increase, the standard deviation in distance between examples shrinks relative to the mean distance between examples. This convergence means that k-means becomes less effective at distinguishing between examples as the dimensionality of the data increases. This is referred to as the **curse of dimensionality**.

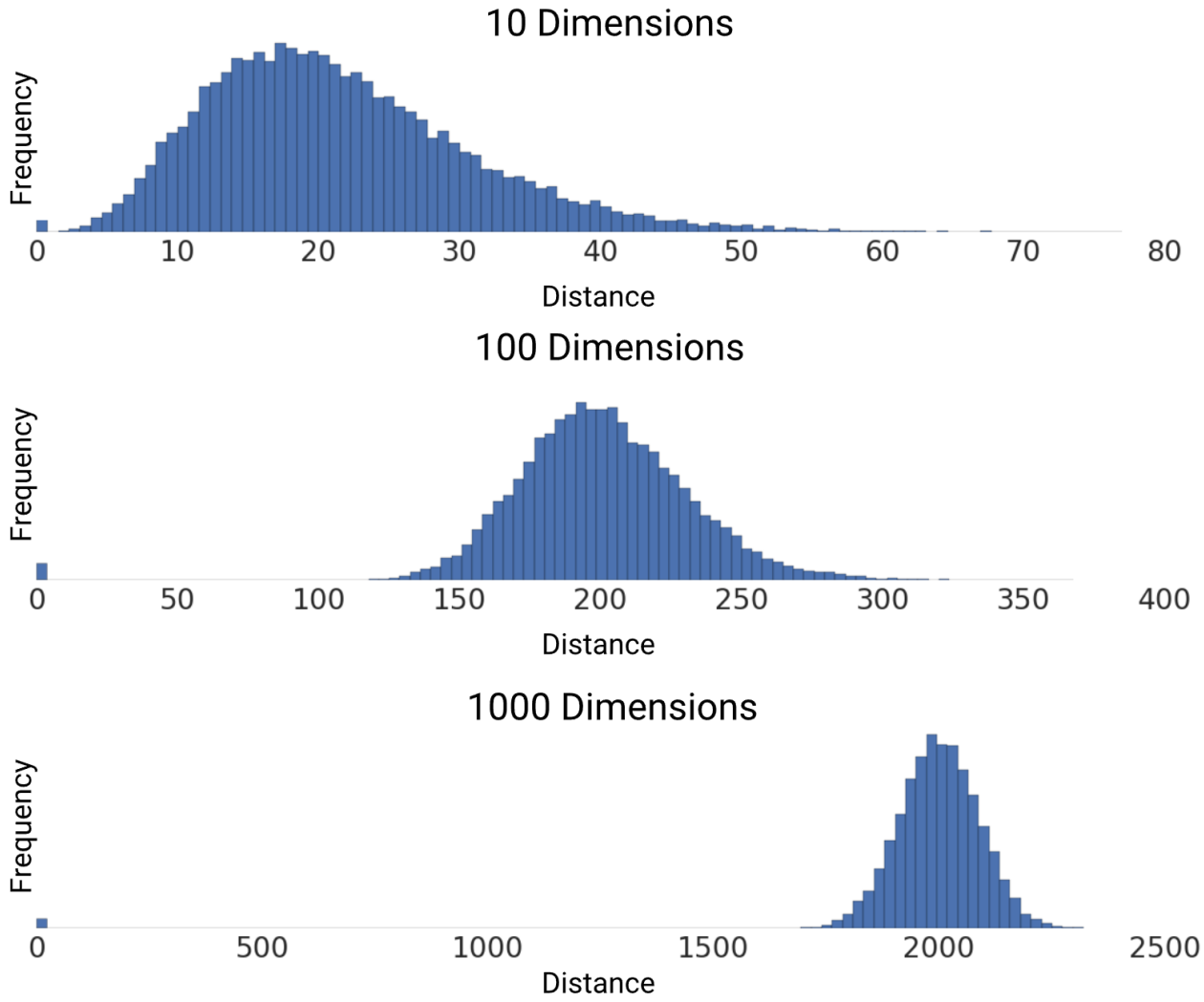


Figure 3: A demonstration of the curse of dimensionality. Each plot shows the pairwise distances between 200 random points.

You can avoid this diminishment in performance with **spectral clustering**, which adds pre-clustering steps to the algorithm. To perform spectral clustering:

1. Reduce the dimensionality of feature data by using PCA.
2. Project all data points into the lower-dimensional subspace.
3. Cluster the data in this subspace using your chosen algorithm.

Principal Component Analysis (PCA)

- Lab

- 13.PCA.ipynb

```
---
title: "Dimension Reduction"
subtitle: "Principle Component Analysis"
author: "Tanzeela Javid Kaloo" | Assistant Professor LPU
date: "Febuary 21, 2025"
---
```

USArrest Data

USArrest Data contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. Also the percent of the population living in urban areas is given.

A data frame with 50 observations on 4 variables:

- `Assault` : Assault arrests (per 100,000),
- `Murder` : Murder arrests (per 100,000),
- `Rape` : Rape arrests (per 100,000), and
- `UrbanPop` : The percent of the population in each state living in the urban areas.

[1]:
```
#required libraries
import pandas as pd
df = pd.read_excel('datasets/USArrests.xlsx', index_col=0, header=0)
df.head(10)
```

[1]:

	Assault	Murder	Rape	UrbanPop
Alabama	236	13.2	21.2	58
Alaska	263	10.0	44.5	48
Arizona	294	8.1	31.0	80
Arkansas	190	8.8	19.5	50
California	276	9.0	40.6	91
Colorado	204	7.9	38.7	78
Connecticut	110	3.3	11.1	77

Comparison of Manual and Supervised Measures

This table describes when to use a manual or supervised similarity measure depending on your requirements.

Requirement	Manual	Supervised
Eliminates redundant information in correlated features?	No, you need to investigate any correlations between features.	Yes, DNN eliminates redundant information.
Gives insight into calculated similarities?	Yes	No, embeddings cannot be deciphered.
Suitable for small datasets with few features?	Yes.	No, small datasets don't provide enough training data for a DNN.
Suitable for large datasets with many features?	No, manually eliminating redundant information from multiple features and then combining them is very difficult.	Yes, the DNN automatically eliminates redundant information and combines features.

Creating a supervised similarity measure

Here's an overview of the process to create a supervised similarity measure:

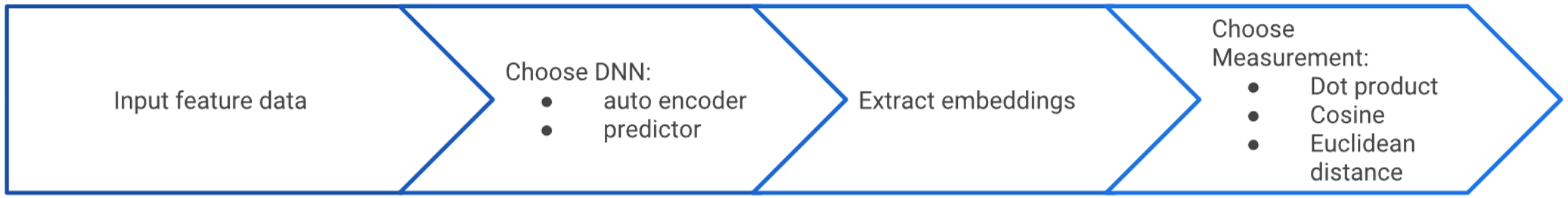


Figure 1: Steps to create a supervised similarity measure.

Choose DNN based on training labels

Reduce your feature data to lower-dimensional embeddings by training a DNN that uses the same feature data both as input and as the labels. For example, in the case of house data, the DNN would use the features—such as price, size, and postal code—to predict those features themselves.

Autoencoder

A DNN that learns embeddings of input data by predicting the input data itself is called an autoencoder. Because an autoencoder's hidden layers are smaller than the input and output layers, the autoencoder is forced to learn a compressed representation of the input feature data. Once the DNN is trained, extract the embeddings from the smallest hidden layer to calculate similarity.

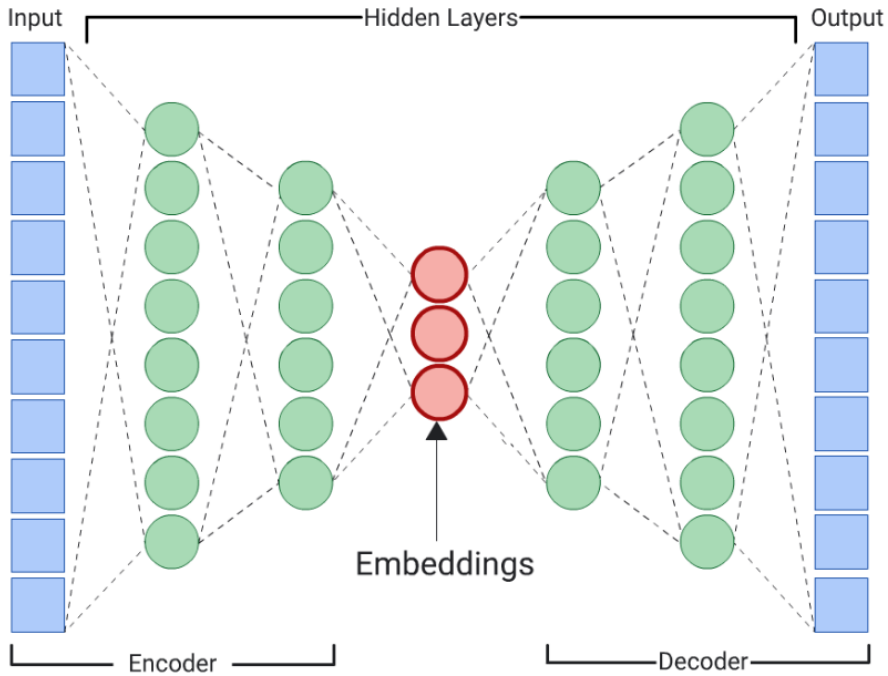


Figure 2: Autoencoder architecture.

Predictor

An autoencoder is the simplest choice to generate embeddings. However, an autoencoder isn't the optimal choice when certain features could be more important than others in determining similarity. For example, in house data, assume price is more important than postal code. In such cases, use only the important feature as the training label for the DNN. Since this DNN predicts a specific input feature instead of predicting all input features, it is called a **predictor** DNN. Embeddings should usually be extracted from the last embedding layer.

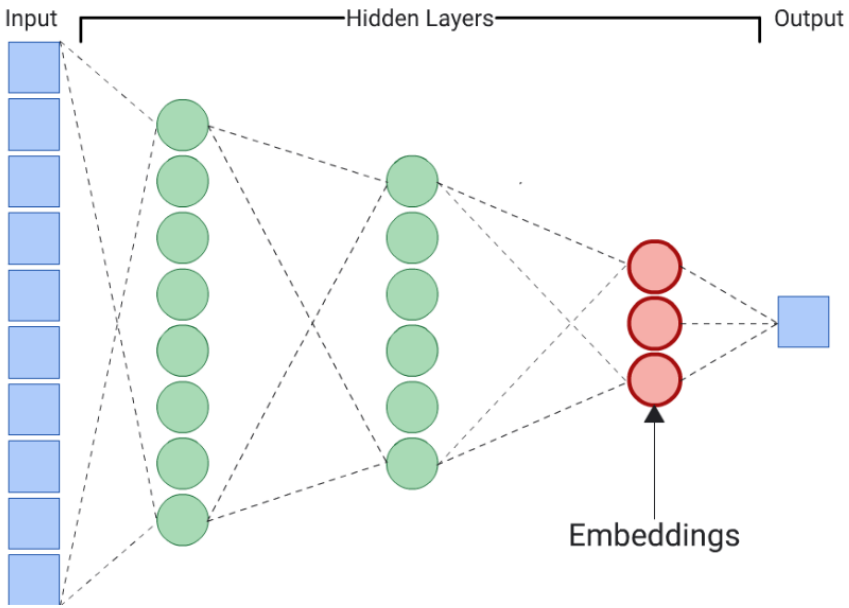


Figure 3: Predictor architecture.

When choosing a feature to be the label:

- Prefer numerical to categorical features because loss is easier to calculate and interpret for numeric features.
- Remove the feature that you use as the label from the input to the DNN, or else the DNN will use that feature to perfectly predict the output. (This is an extreme example of label leakage.)

Depending on your choice of labels, the resulting DNN is either an autoencoder or a predictor.

Measuring similarity from embeddings

You now have embeddings for any pair of examples. A supervised similarity measure takes these embeddings and returns a number measuring their similarity. Remember that embeddings are vectors of numbers. To find the similarity between two vectors $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_n]$, choose one of these three similarity measures:

Measure	Meaning	Formula	As similarity increases, this measure...
Euclidean distance	Distance between ends of vectors	$\sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_N - b_N)^2}$	Decreases
Cosine	Cosine of angle θ between vectors	$\frac{a^T b}{\lvert a\rvert \cdot \lvert b\rvert}$	Increases
Dot product	Cosine multiplied by lengths of both vectors	$a_1 b_1 + a_2 b_2 + \ldots + a_n b_n = \lvert a\rvert\lvert b\rvert cos(\theta)$	Increases. Also increases with length of vectors.

Choosing a similarity measure

In contrast to the cosine, the dot product is proportional to the vector length. This is important because examples that appear very frequently in the training set (for example, popular YouTube videos) tend to have embedding vectors with large lengths. If you want to capture popularity, then choose dot product. However, the risk is that popular examples may skew the similarity metric. To balance this skew, you can raise the length to an exponent $\alpha \ < 1$ to calculate the dot product as $|a|^{\alpha} |b|^{\alpha} \cos(\theta)$.

To better understand how vector length changes the similarity measure, normalize the vector lengths to 1 and notice that the three measures become proportional to each other.

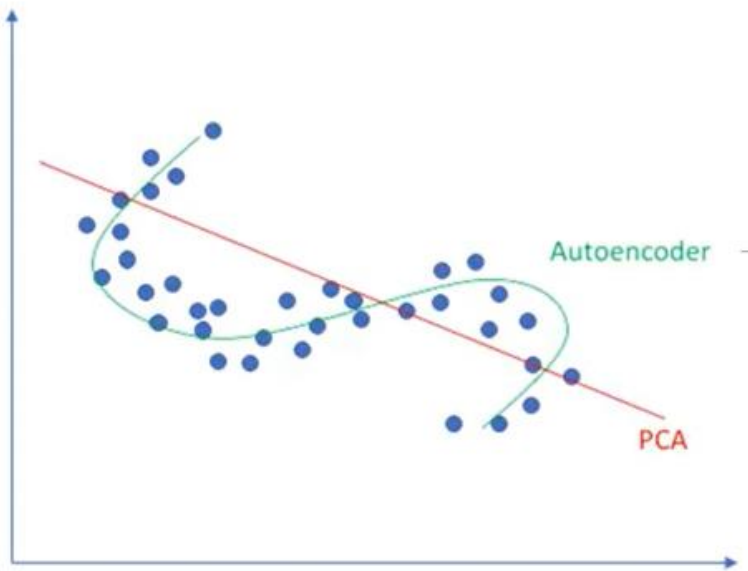
Review of similarity measures

A similarity measure quantifies the similarity between a pair of examples, relative to other pairs of examples. The two types, manual and supervised, are compared below:

Type	How to create	Best for	Implications
Manual	Manually combine feature data.	Small datasets with features that are straightforward to combine.	Gives insight into results of similarity calculations. If feature data changes, you must manually update the similarity measure.
Supervised	Measure distance between embeddings generated by a supervised DNN.	Large datasets with hard-to-combine features.	Gives no insight into results. However, a DNN can automatically adapt to changing feature data.

PCA vs Autoencoders

Linear vs nonlinear dimensionality reduction



Non-linear Transformations
Non-linear activation function and multiple layers

Convolutional Layers
An autoencoder doesn't have to learn dense layers

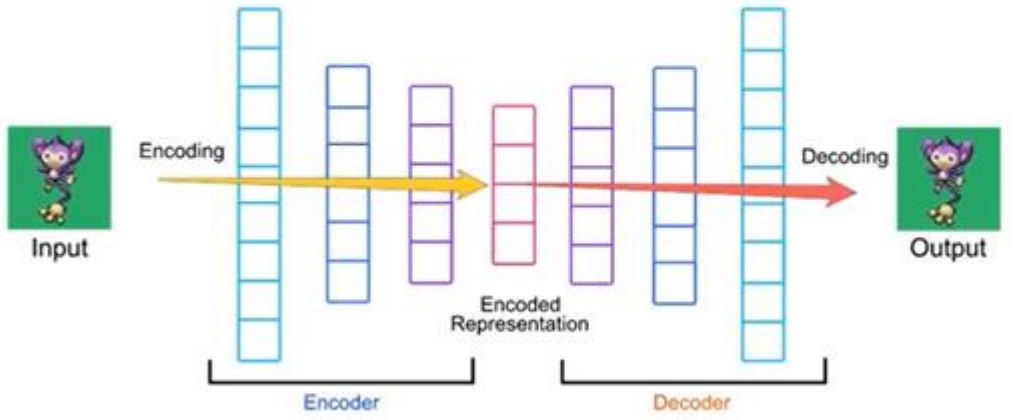
Higher Efficiency
More efficient to learn several layers with an autoencoder

Multiple Transformations
It gives a representation as the output of each layer

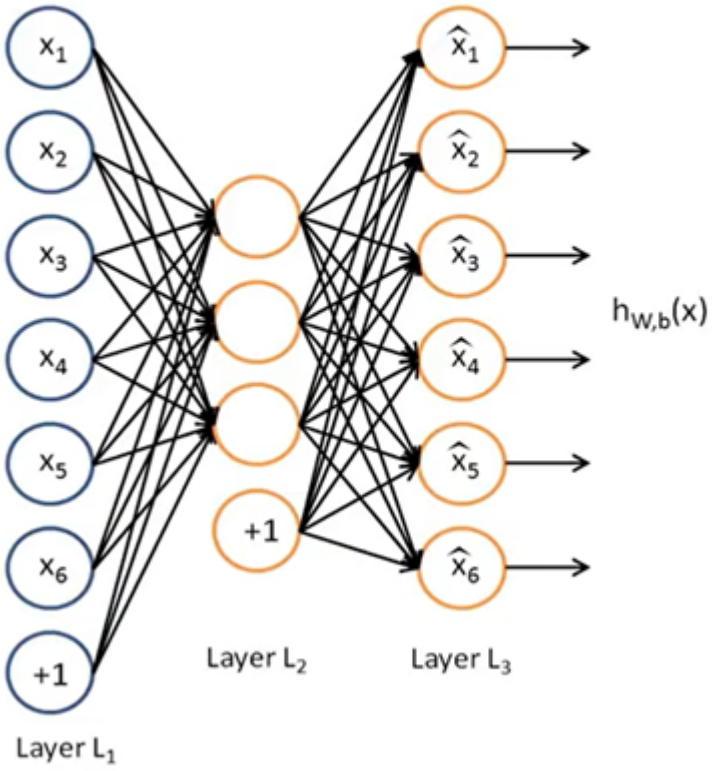
Why Autoencoders?

Introduction to Autoencoders

An ***autoencoder*** neural network is an unsupervised Machine learning algorithm that applies backpropagation, setting the target values to be equal to the inputs.



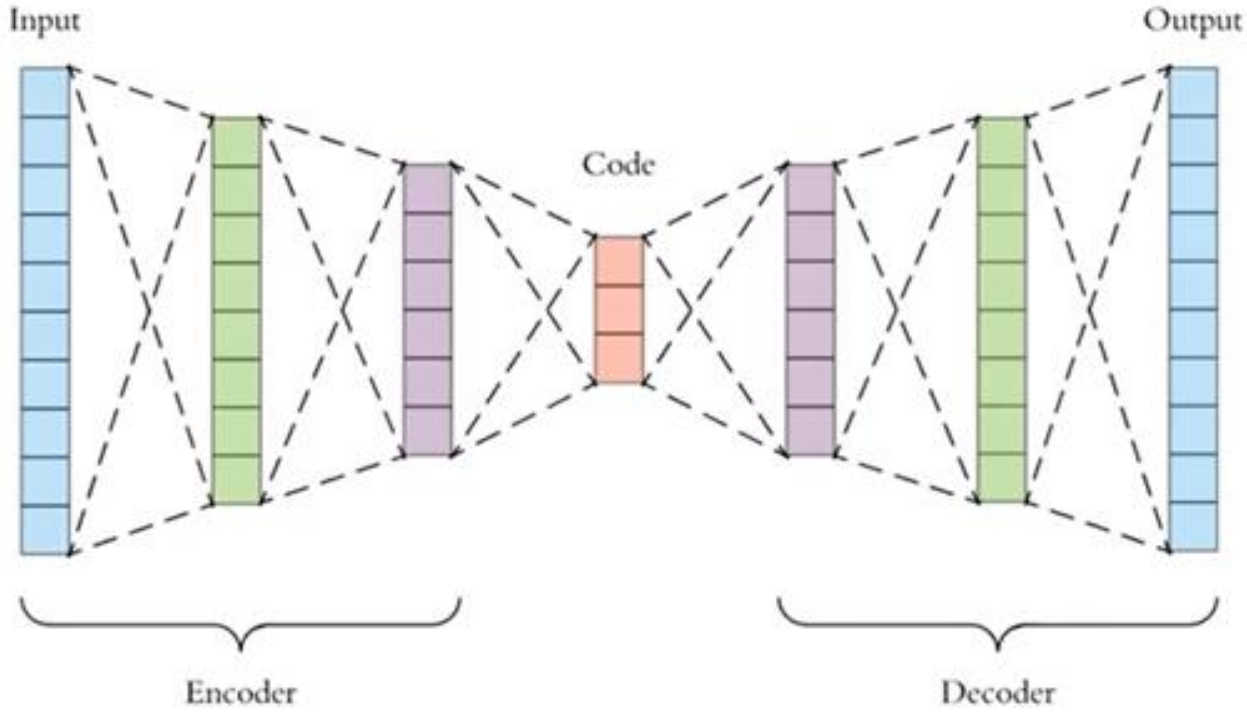
Autoencoders

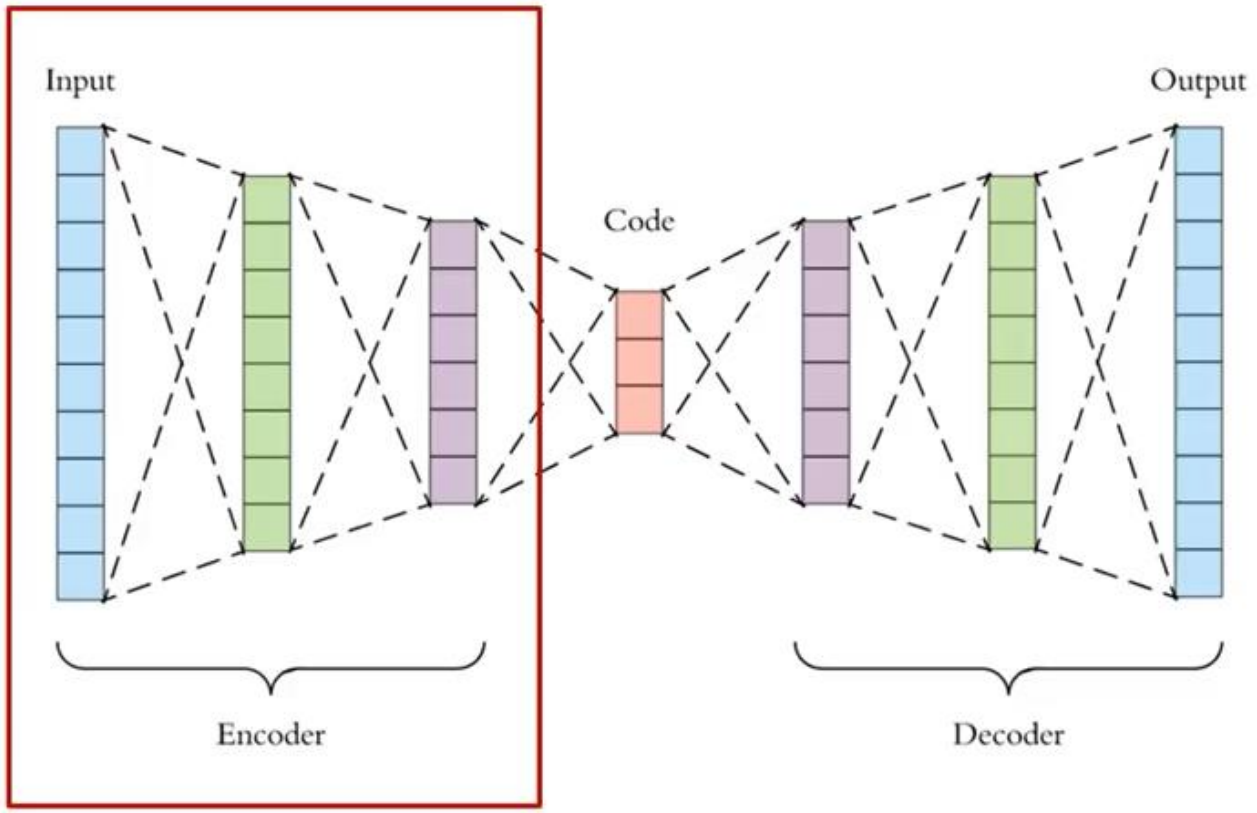


Key Facts about Autoencoders

- It is an unsupervised ML algorithm similar to PCA
- It minimizes the same objective function as PCA
- It is a neural network
- The neural network's target output is its input

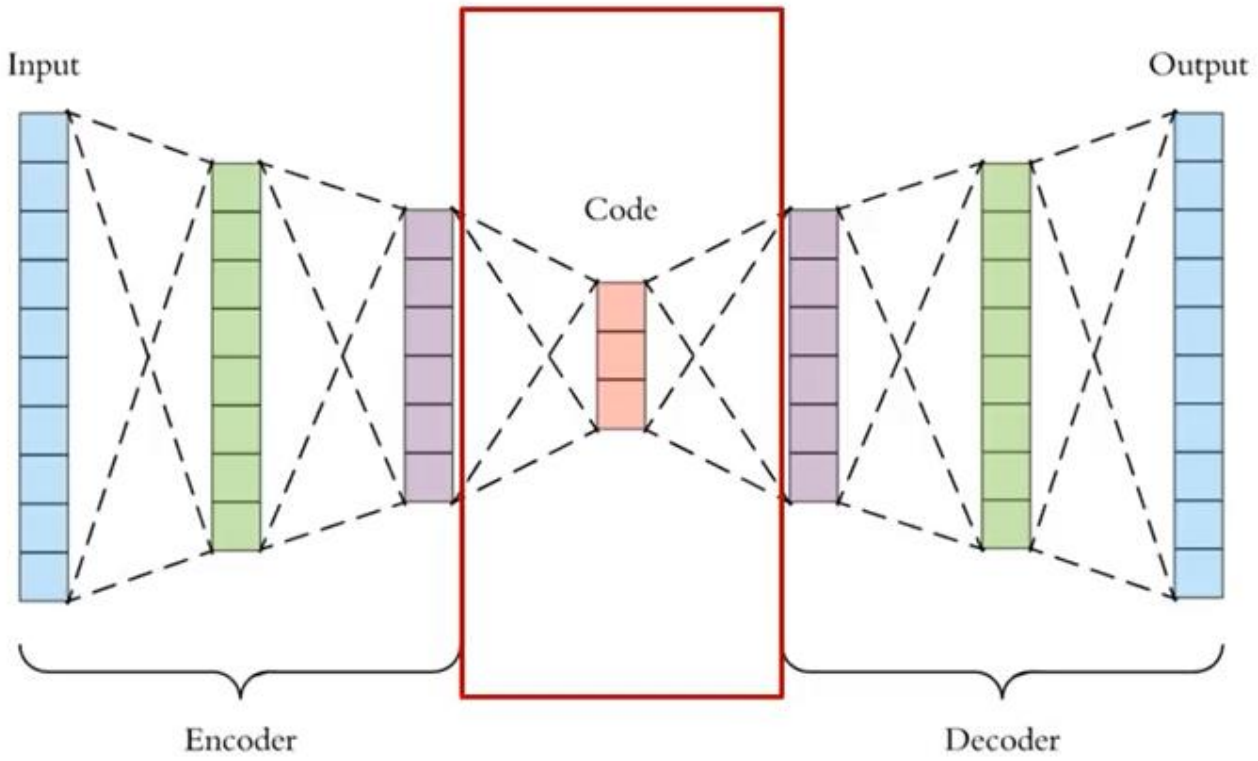
Components of Autoencoders





01 **Encoder**

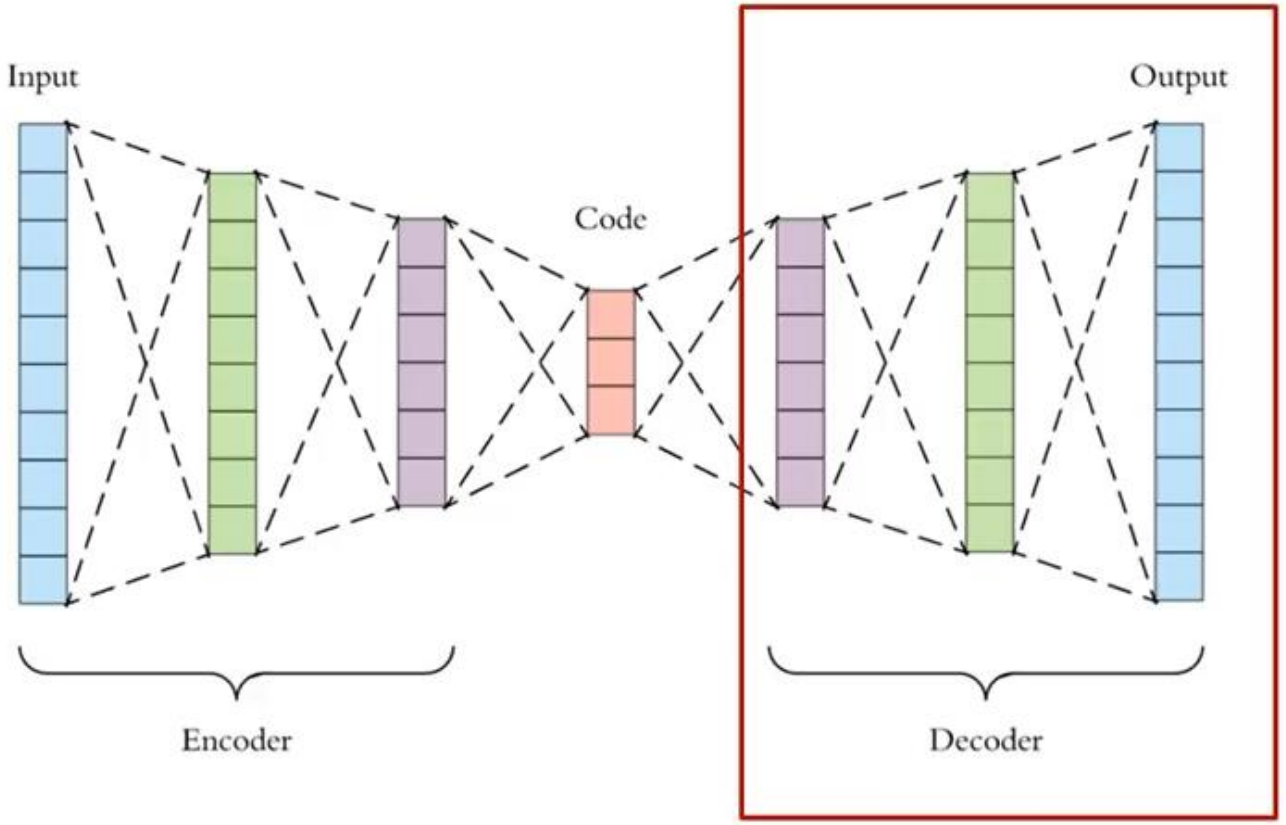
This is the part of the network that compresses the input into a latent space representation.



02

Code

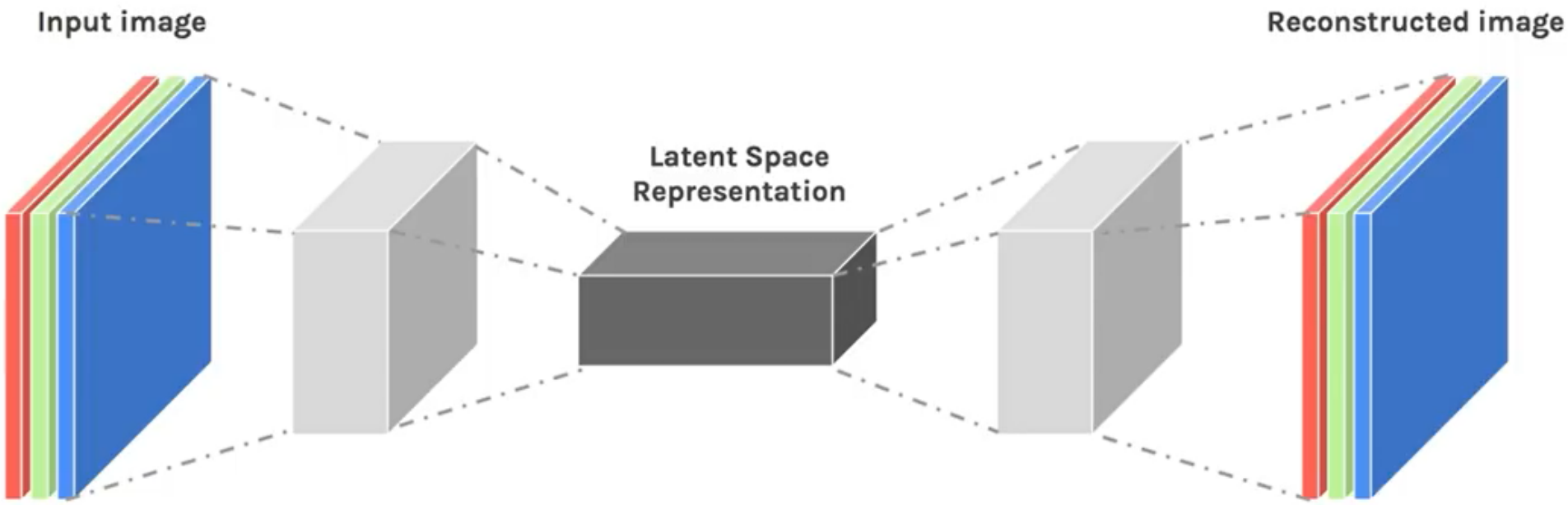
This is the part of the network represents the compressed input that is fed to the decoder



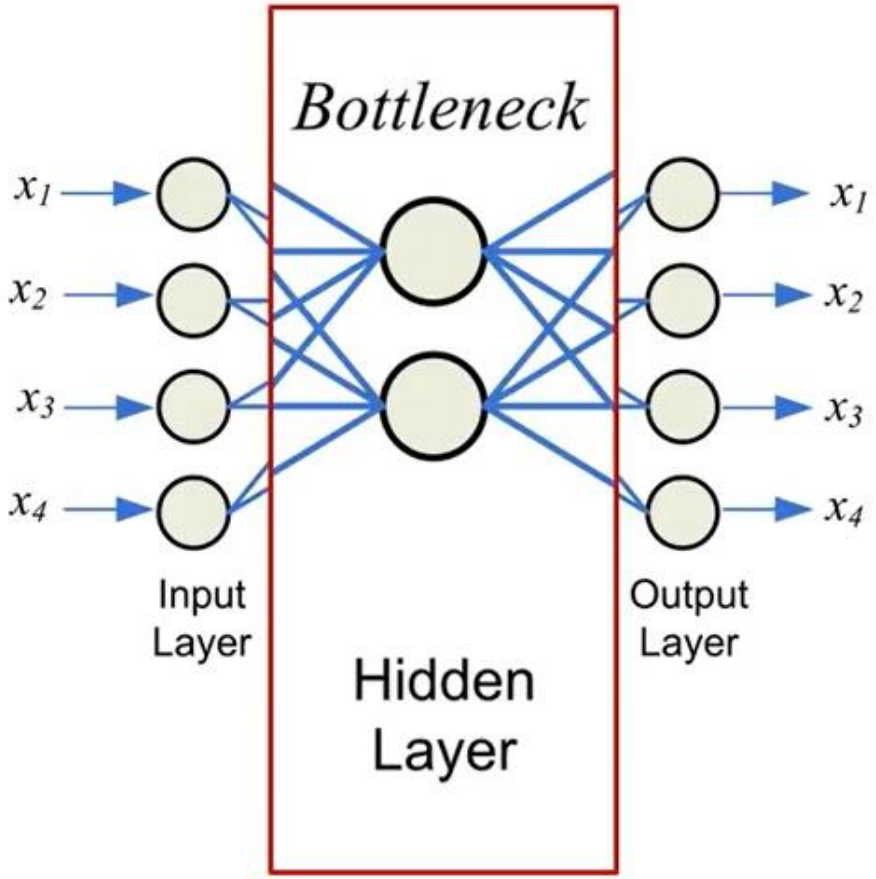
03 Decoder

This part aims to reconstruct the input from the latent space representation

Architecture of Autoencoders

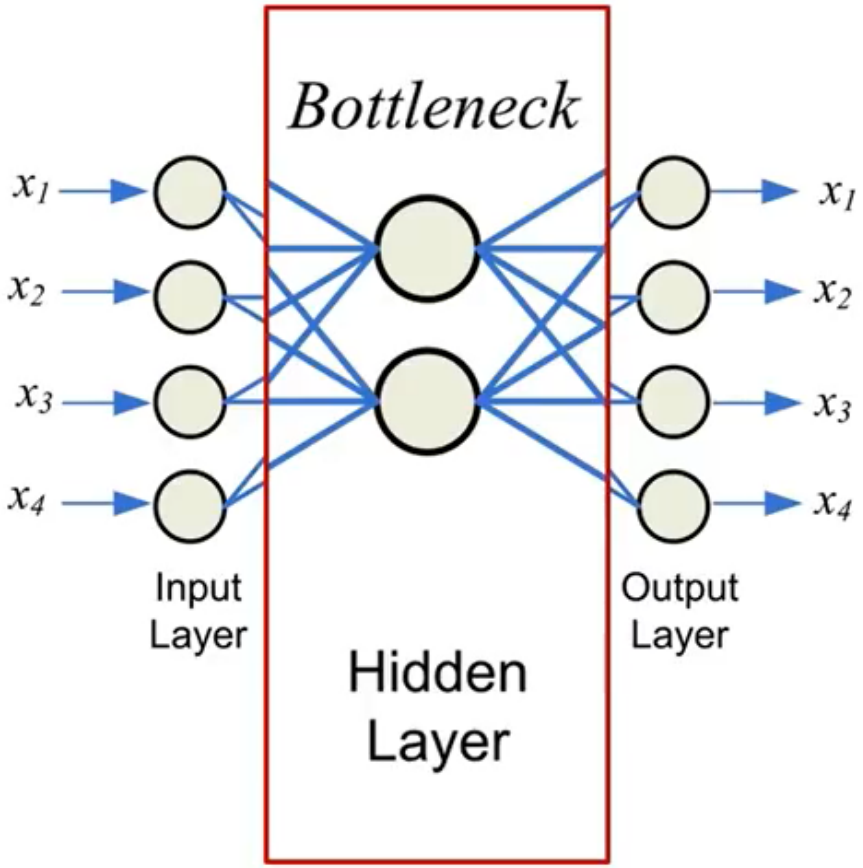


Architecture of Autoencoders



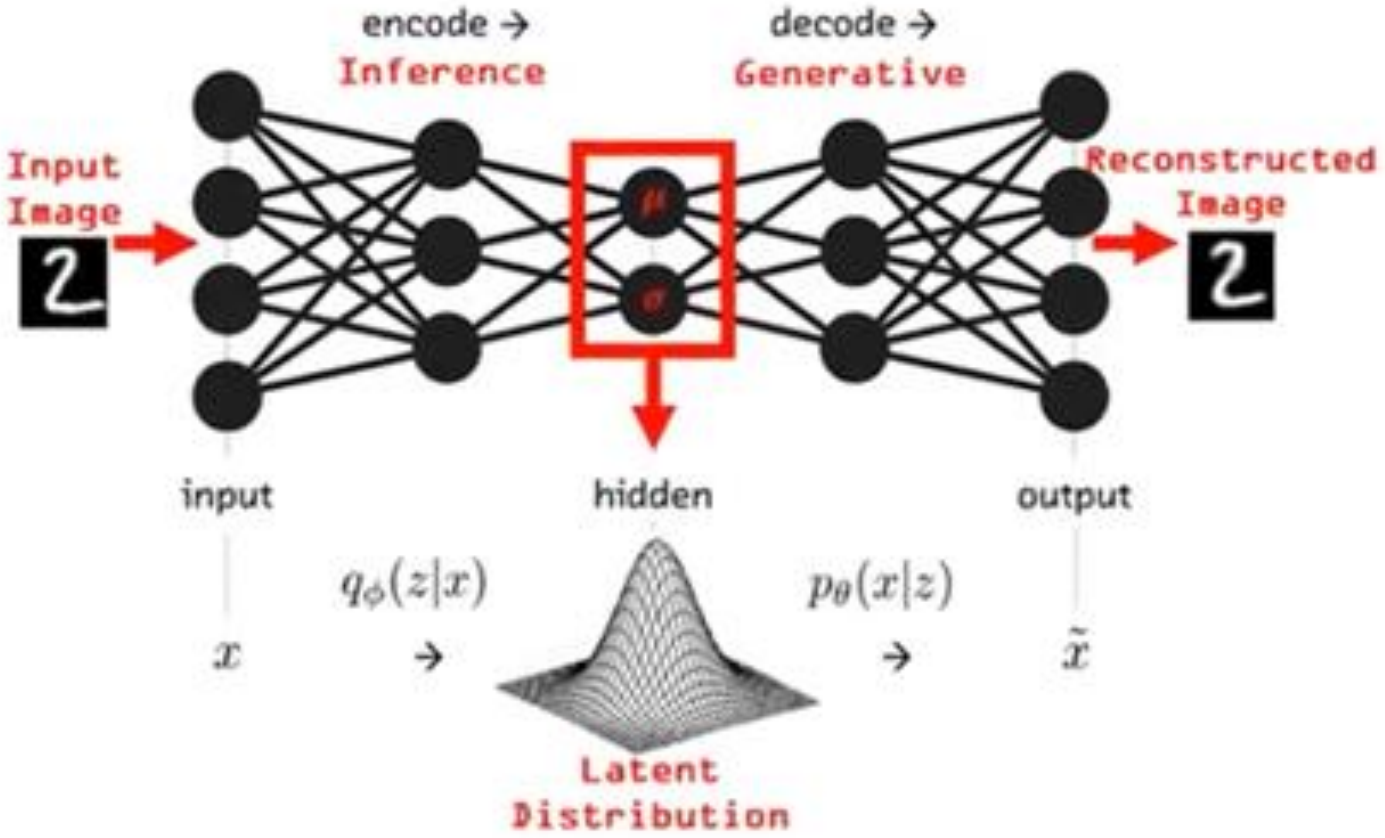
Bottleneck approach is an approach to for deciding which aspects of observed data are relevant information and what aspects can be thrown away

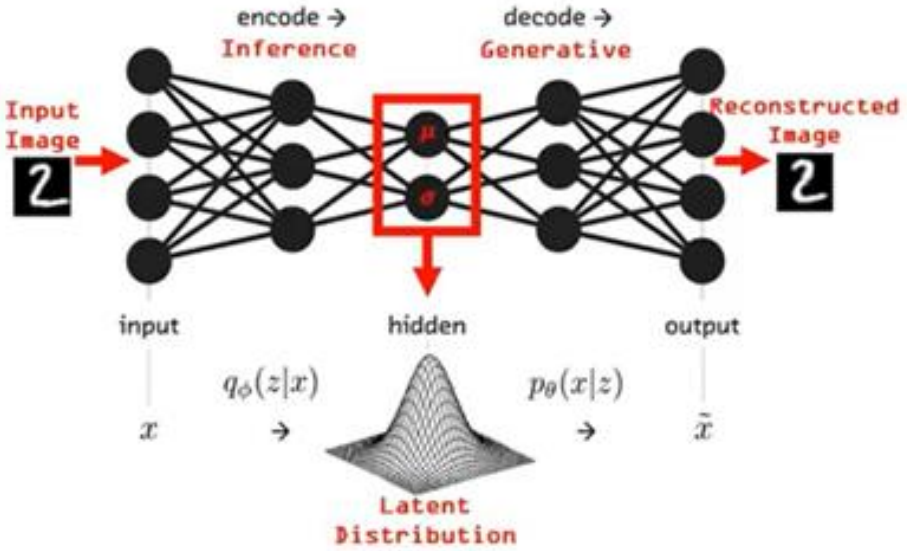
Architecture of Autoencoders

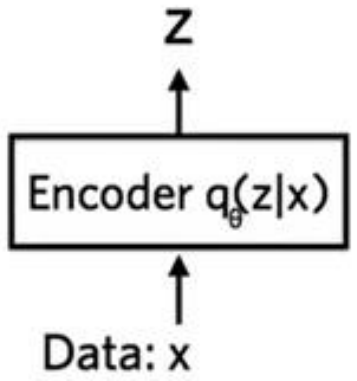


Bottleneck approach is an approach to for deciding which aspects of observed data are relevant information and what aspects can be thrown away

- Compactness of representation, measured as the compressibility
- Representation retains about some behaviourally relevant variables

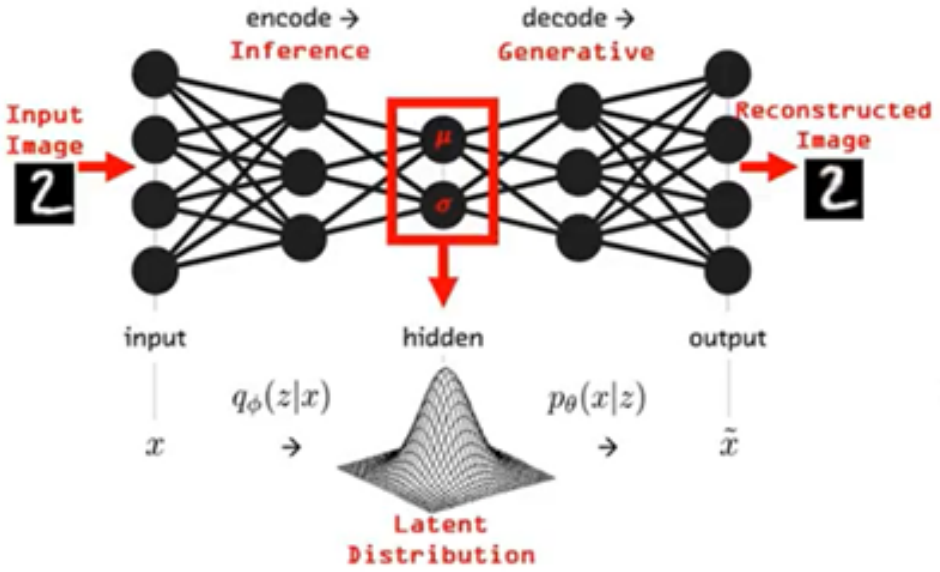

encode →
Inference
decode →
Generative
Input
Image
Reconstructed
Image
input
hidden
output
$q_\phi(z|x)$
$p_\theta(x|z)$
x
$\tilde{x}$
Latent
Distribution

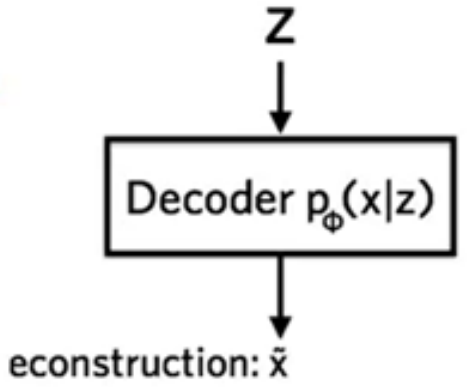




Encoder

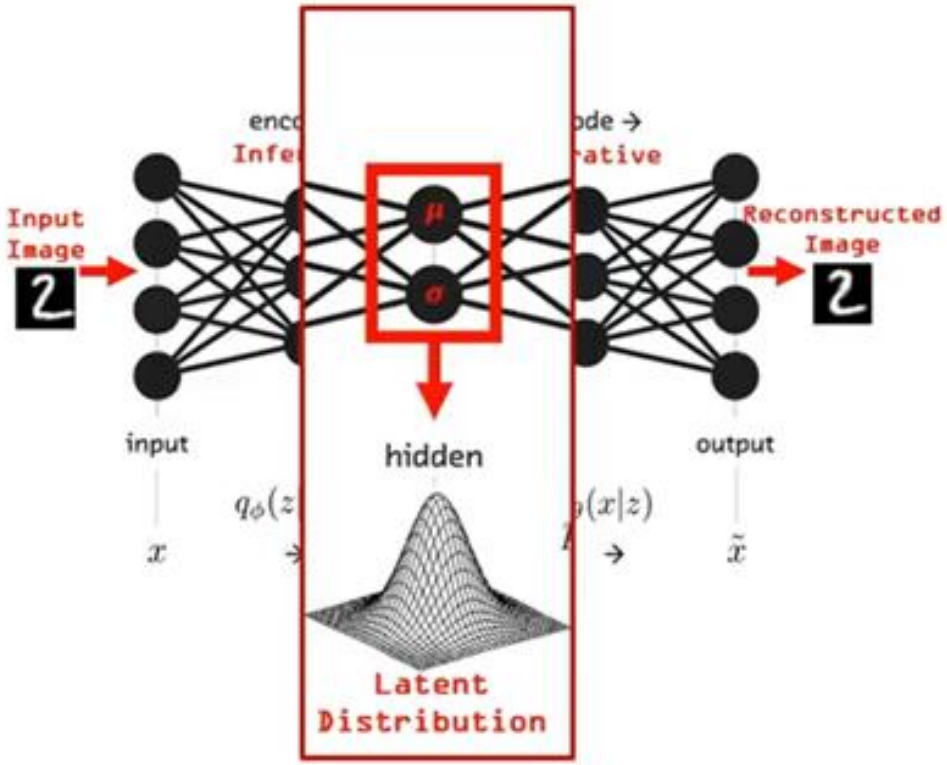
In the neural net world, the encoder is a neural network that outputs a representation *z* of data *x*.





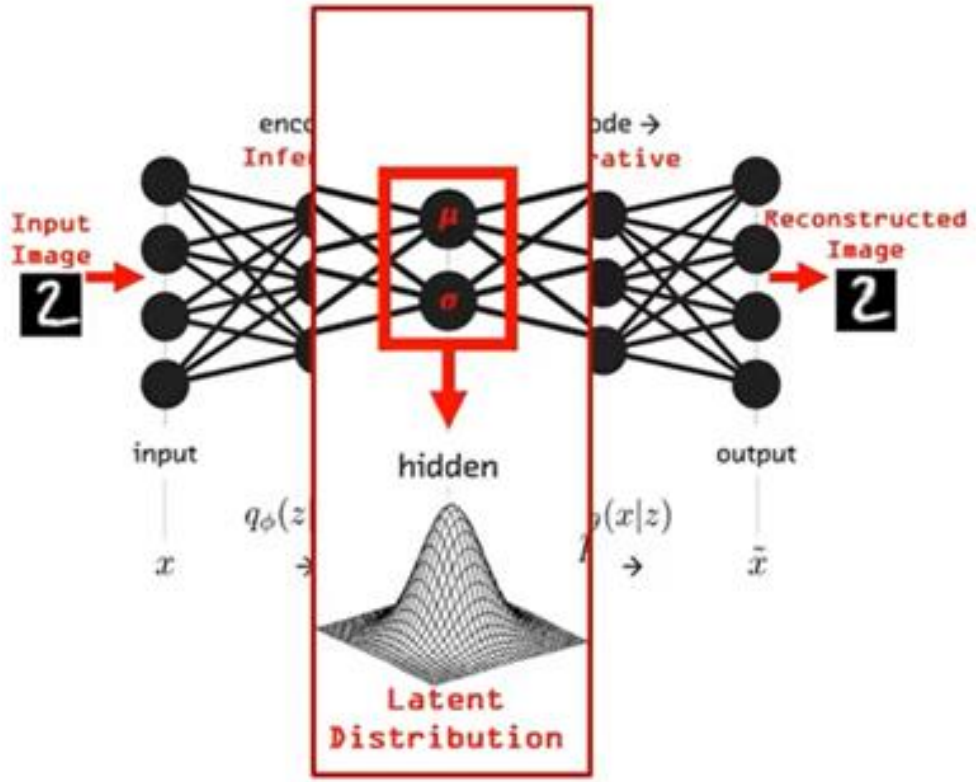
Decoder

In deep learning, the decoder is a neural net that learns to reconstruct the data *x* given a representation *z*.



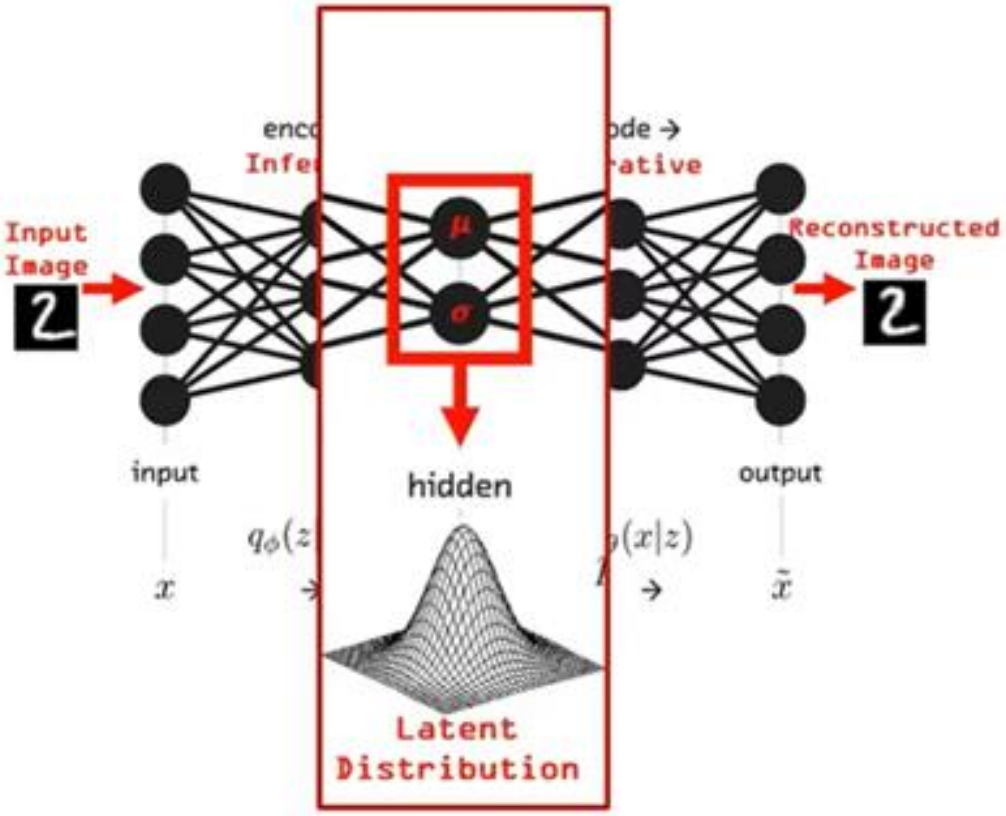
Loss Function

$$l_i(\theta,\phi) = -E_{z\sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(z))$$



$$l_i(\theta,\phi) = -E_{z\sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(z))$$

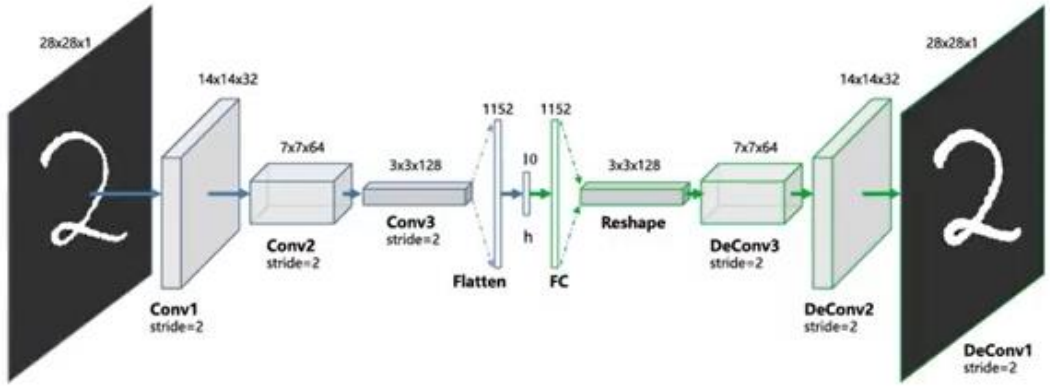
Reconstruction Loss



$$l_i(\theta,\phi) = -E_{z\sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(z))$$

Regularizer

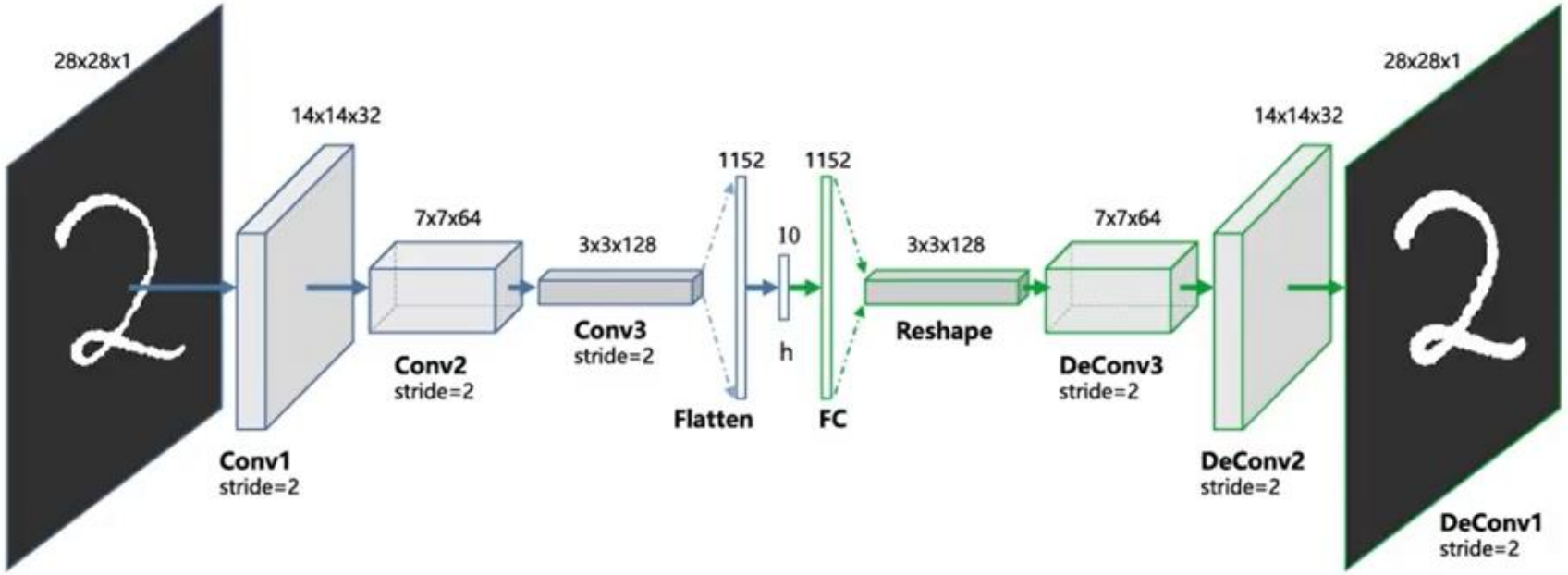
Convolution Autoencoders



Convolution Autoencoders use the convolution operator to learn to encode the input in a set of simple signals and then try to reconstruct the input from them.

$$f(t) * g(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau) g(t-\tau) d\tau$$

Convolution Autoencoders



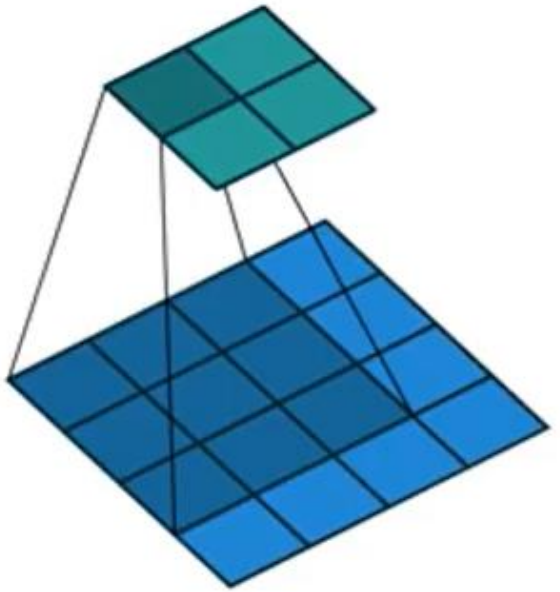
Convolution Autoencoders

In the 2D discrete space, the convolution operation is defined as:

$$O(i,j) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} F(u,v) I(i-u, j-v)$$

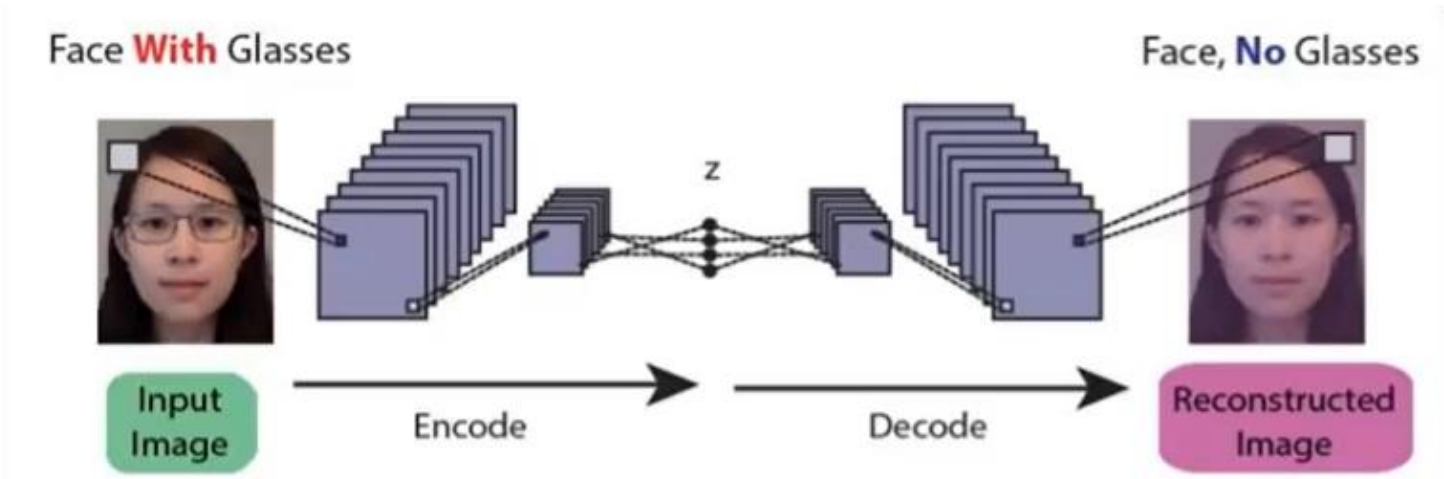
In the image domain where the signals are finite, this formula becomes:

$$O(i,j) = \sum_{u=-2k-1}^{2k+1} \sum_{v=-2k-1}^{2k+1} F(u,v) I(i-u, j-v)$$



Use Case of CAE:

1. Image Reconstruction
2. Image Colorization
3. Advanced Applications



learns to **remove noise** or **reconstruct** missing parts

Noisy Version is converted to clean version

the network fills the gaps in the image

Use Case of CAE:

1. Image Reconstruction
2. **Image Colorization**
3. Advanced Applications

- maps **circles** and **squares** from an image to the same image but with Colors
- Purple is formed sometimes because of **blend** of colors, where network hesitates between circle or square.

Use Case of CAE:

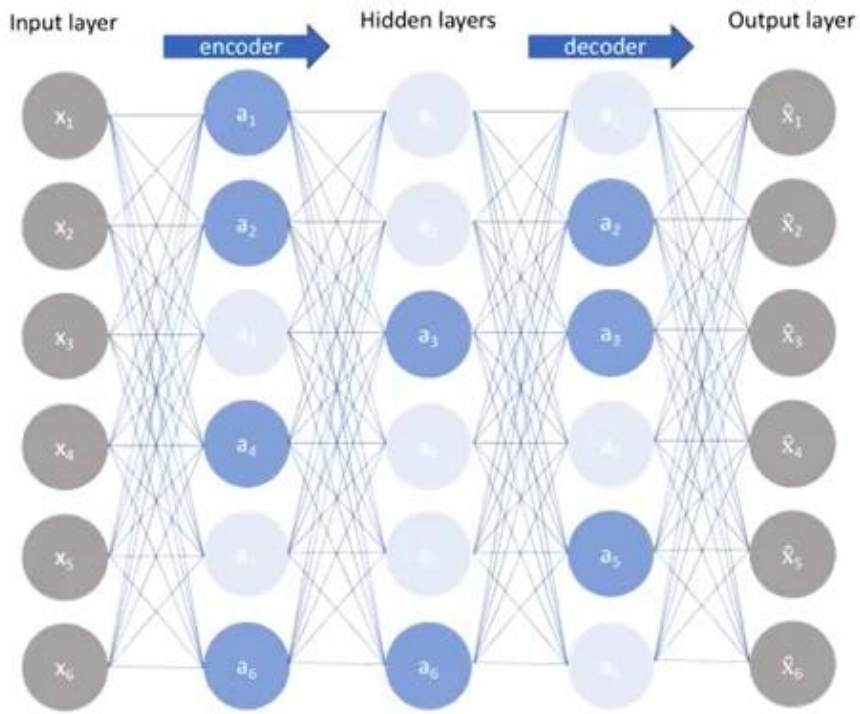
1. Image Reconstruction
2. Image Colorization
3. **Advanced Applications**

Fully image colorization

Latent space clustering

Generating higher resolution images

Sparse Autoencoders



Sparse autoencoders offer us an alternative method for introducing an information bottleneck without requiring a reduction in the number of nodes at our hidden layers

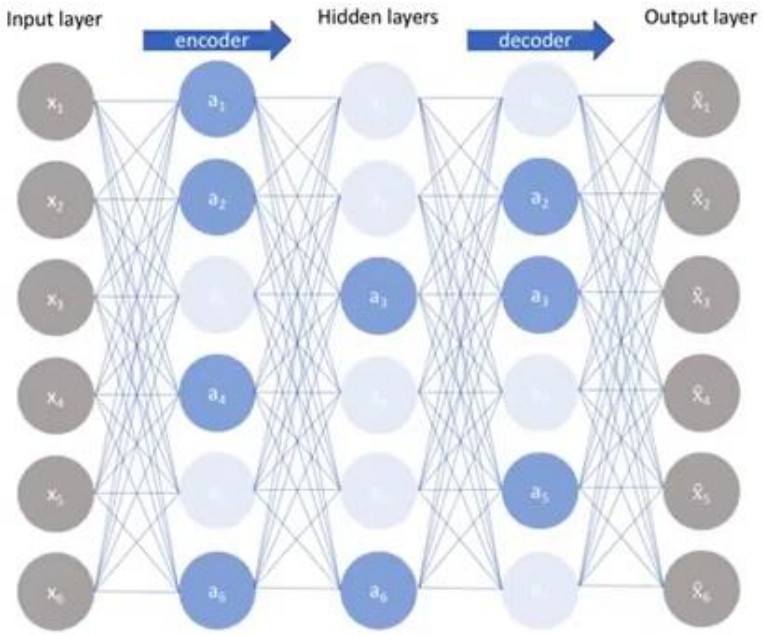
Sparse Autoencoders

L1 Regularization

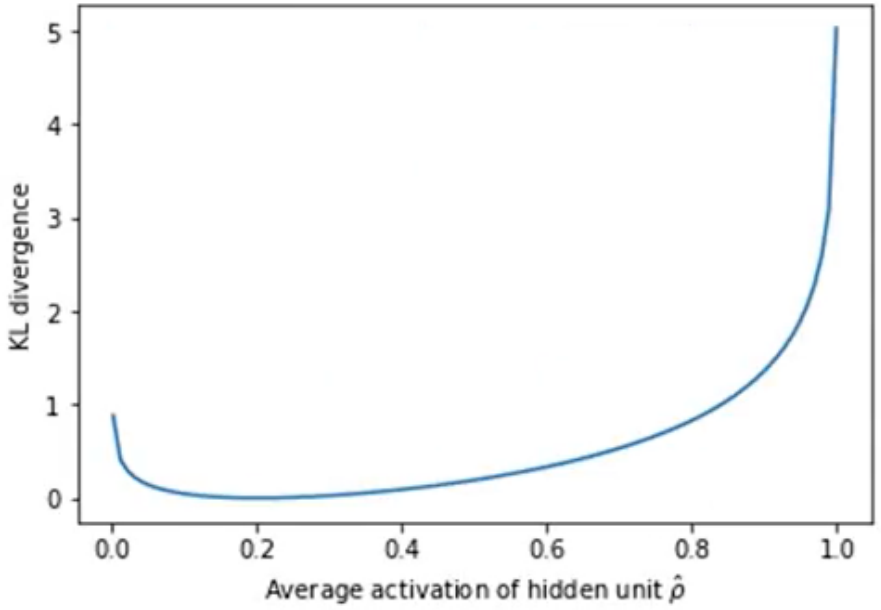
$$\mathcal{L}\left(x, \hat{x}\right) + \lambda \sum_{i} \left| a_i^{(h)} \right|$$

KL Divergence

$$\mathcal{L}\left(x, \hat{x}\right) + \sum_{j} KL\left(\rho || \hat{\rho}_j\right)$$



Sparse Autoencoders



$$\sum_{j=1}^{l^{(h)}} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$$

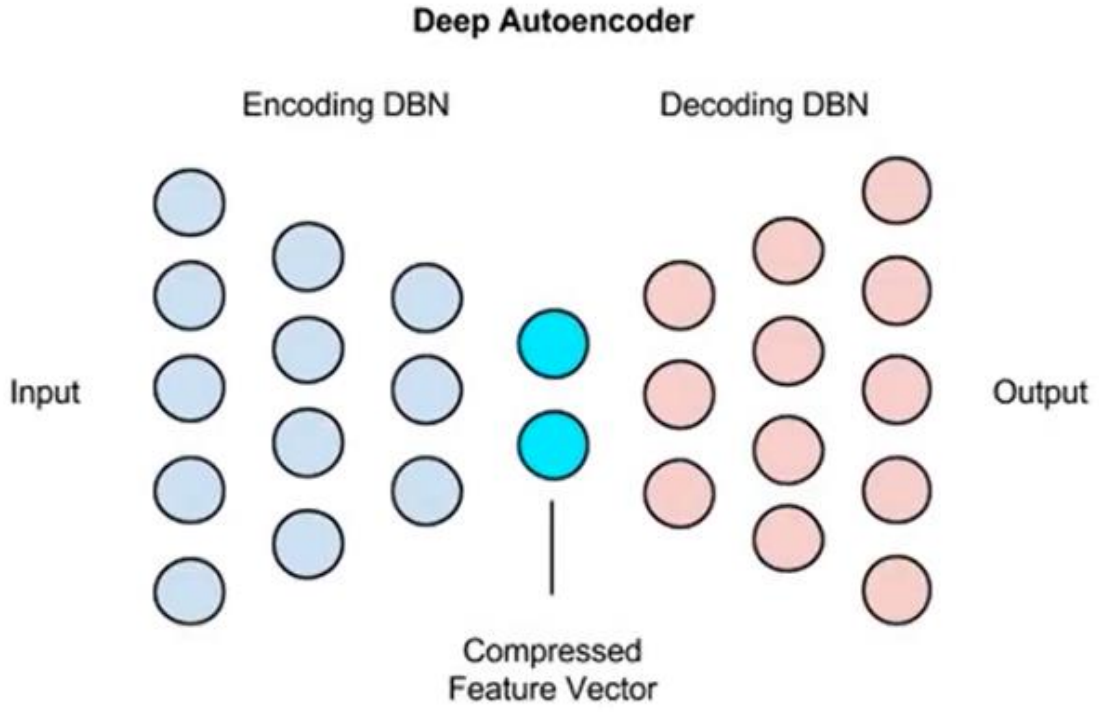
L1 Regularization

$$\mathcal{L}(x, \hat{x}) + \lambda \sum_i \left| a_i^{(h)} \right|$$

KL Divergence

$$\mathcal{L}(x, \hat{x}) + \sum_j KL\left(\rho || \hat{\rho}_j\right)$$

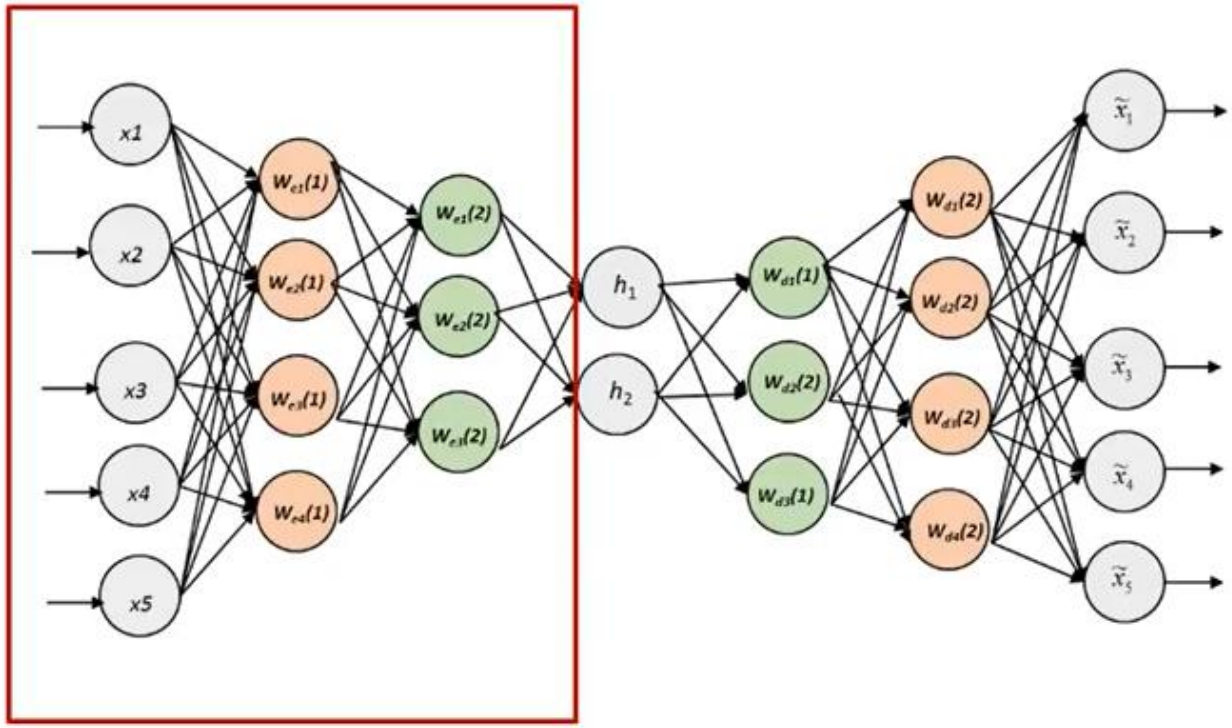
Deep Autoencoders



A ***deep autoencoder*** is composed of two, symmetrical deep-belief networks-

- First four or five shallow layers representing the encoding half of the net
- second set of four or five layers that make up the decoding half.

Deep Autoencoders

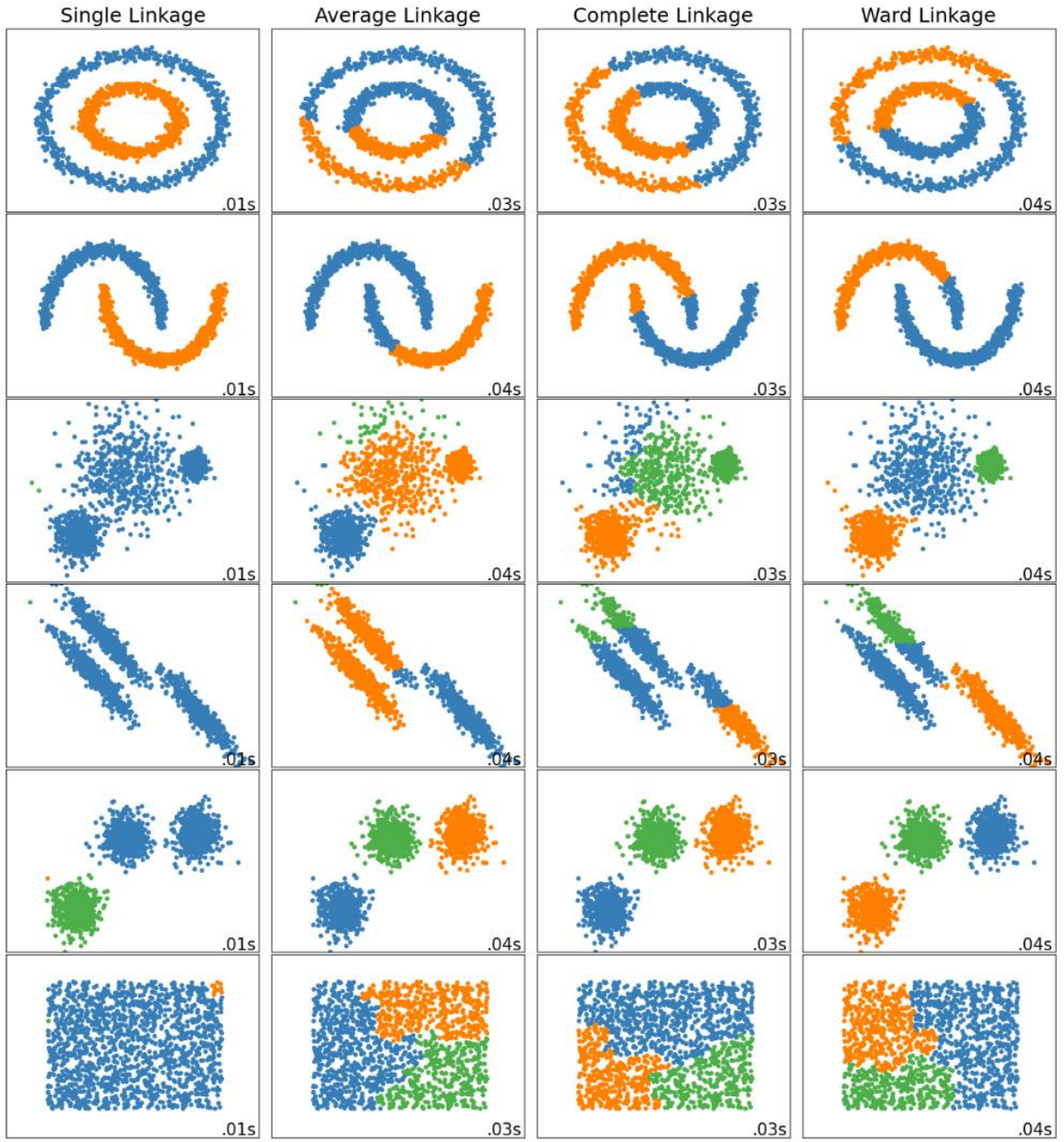


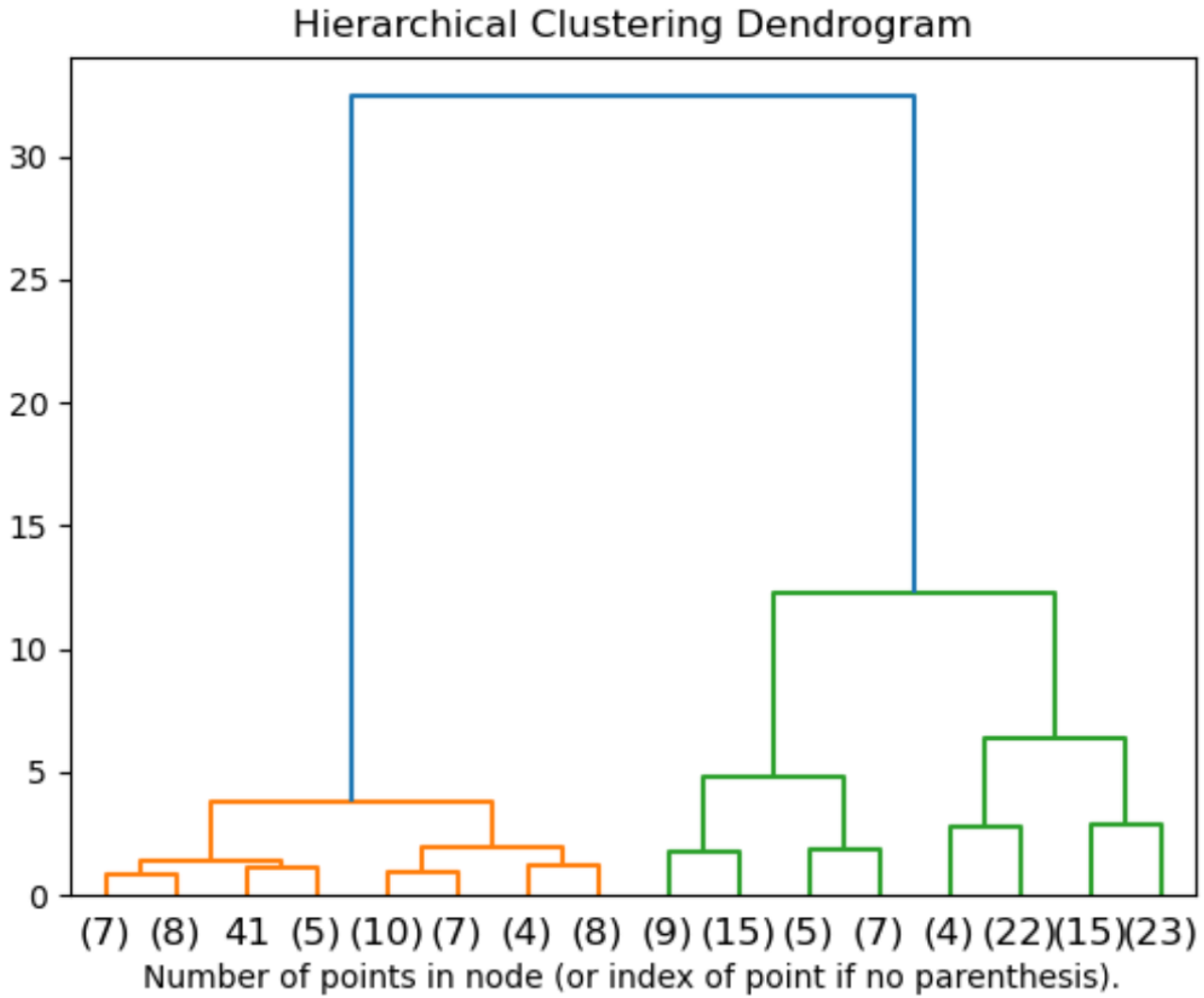
The first layer of the Deep Autoencoder learns first-order features in the raw input such as edges in an image

Hierarchical clustering

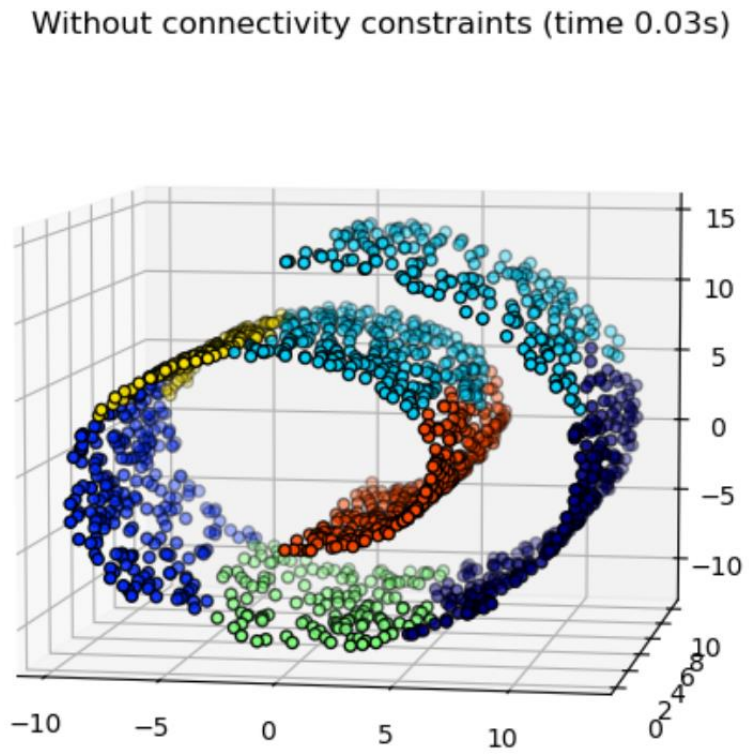
- Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively.
- This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample.
- The '**AgglomerativeClustering**' object performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together. The linkage criteria determines the metric used for the merge strategy:
  - Ward minimizes the sum of squared differences within all clusters
  - **Maximum** or **complete linkage** minimizes the maximum distance between observations of pairs of clusters.
  - Average Linkage minimizes the average of the distances between all observations of pairs of clusters
  - Single Linkage minimizes the distance between the closest observations of pairs of clusters.

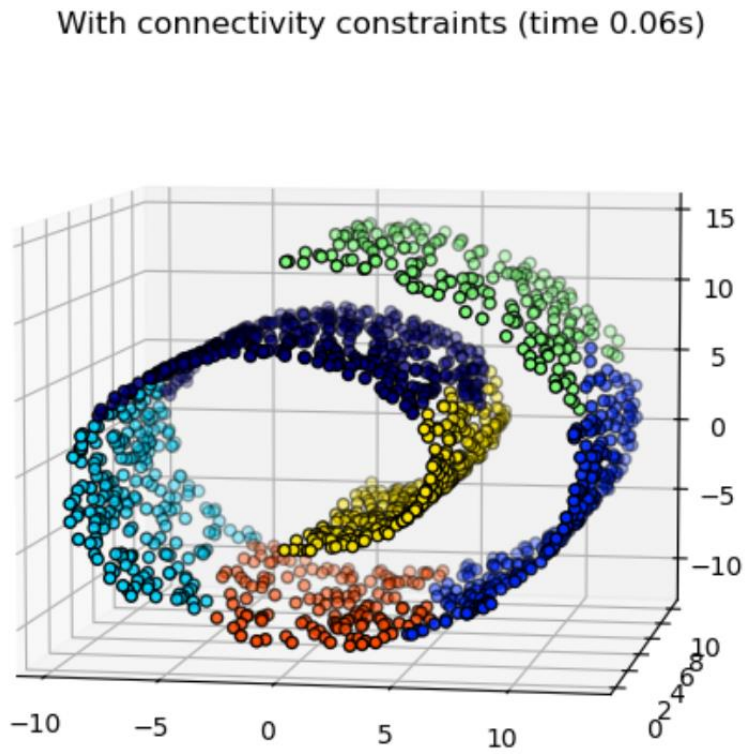
Different linkage type: Ward, complete, average, and single linkage




Hierarchical Clustering Dendrogram
30
25
20
15
10
5
0
(7) (8) 41 (5) (10) (7) (4) (8) (9) (15) (5) (7) (4) (22) (15) (23)
Number of points in node (or index of point if no parenthesis).

Adding connectivity constraints to Agglomerative Clustering





An interesting aspect of `AgglomerativeClustering` is that connectivity constraints can be added to this algorithm (only adjacent clusters can be merged together), through a connectivity matrix that defines for each sample the neighboring samples following a given structure of the data. For instance, in the swiss-roll example below, the connectivity constraints forbid the merging of points that are not adjacent on the swiss roll, and thus avoid forming clusters that extend across overlapping folds of the roll.

These constraint are useful to impose a certain local structure, but they also make the algorithm faster, especially when the number of the samples is high.

Varying the metric

Single, average and complete linkage can be used with a variety of distances (or affinities), in particular Euclidean distance (*l2*), Manhattan distance (or Cityblock, or *l1*), cosine distance, or any precomputed affinity matrix.

- *l1* distance is often good for sparse features, or sparse noise: i.e. many of the features are zero, as in text mining using occurrences of rare words.
- *cosine* distance is interesting because it is invariant to global scalings of the signal.

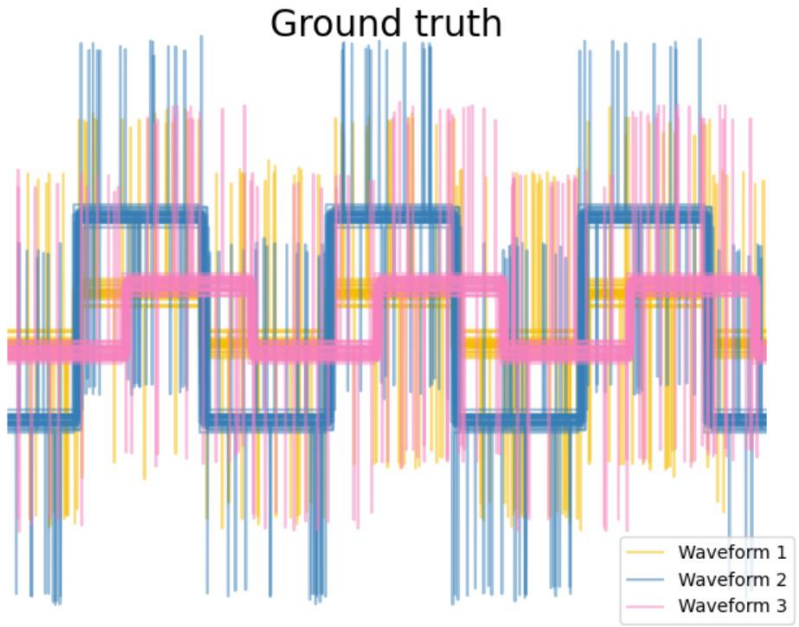
The guidelines for choosing a metric is to use one that maximizes the distance between samples in different classes, and minimizes that within each class.

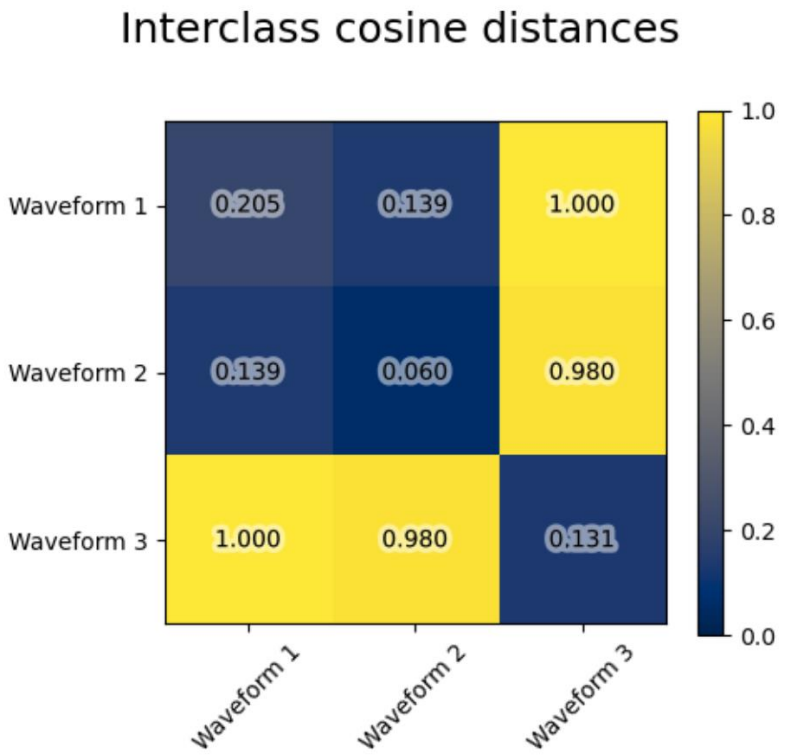
Agglomerative clustering with different metrics

Demonstrates the effect of different metrics on the hierarchical clustering.

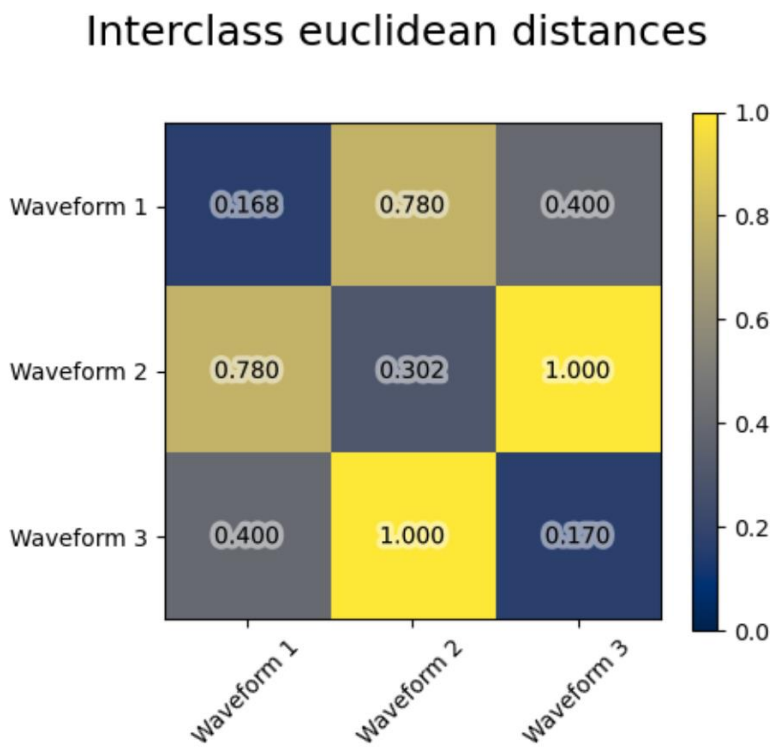
The example is engineered to show the effect of the choice of different metrics. It is applied to waveforms, which can be seen as high-dimensional vector. Indeed, the difference between metrics is usually more pronounced in high dimension (in particular for euclidean and cityblock).

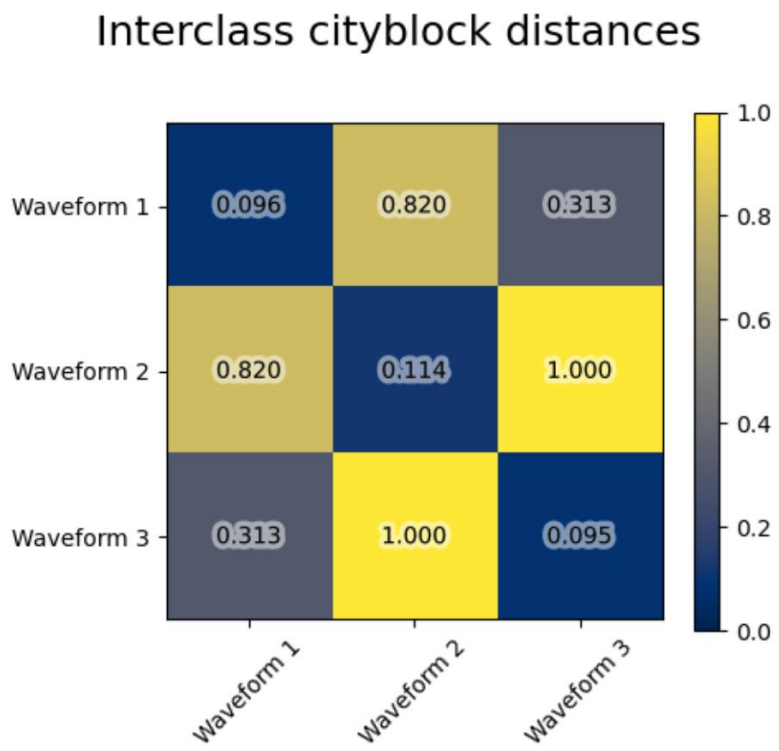
We generate data from three groups of waveforms. Two of the waveforms (waveform 1 and waveform 2) are proportional one to the other. The cosine distance is invariant to a scaling of the data, as a result, it cannot distinguish these two waveforms. Thus even with no noise, clustering using this distance will not separate out waveform 1 and 2.

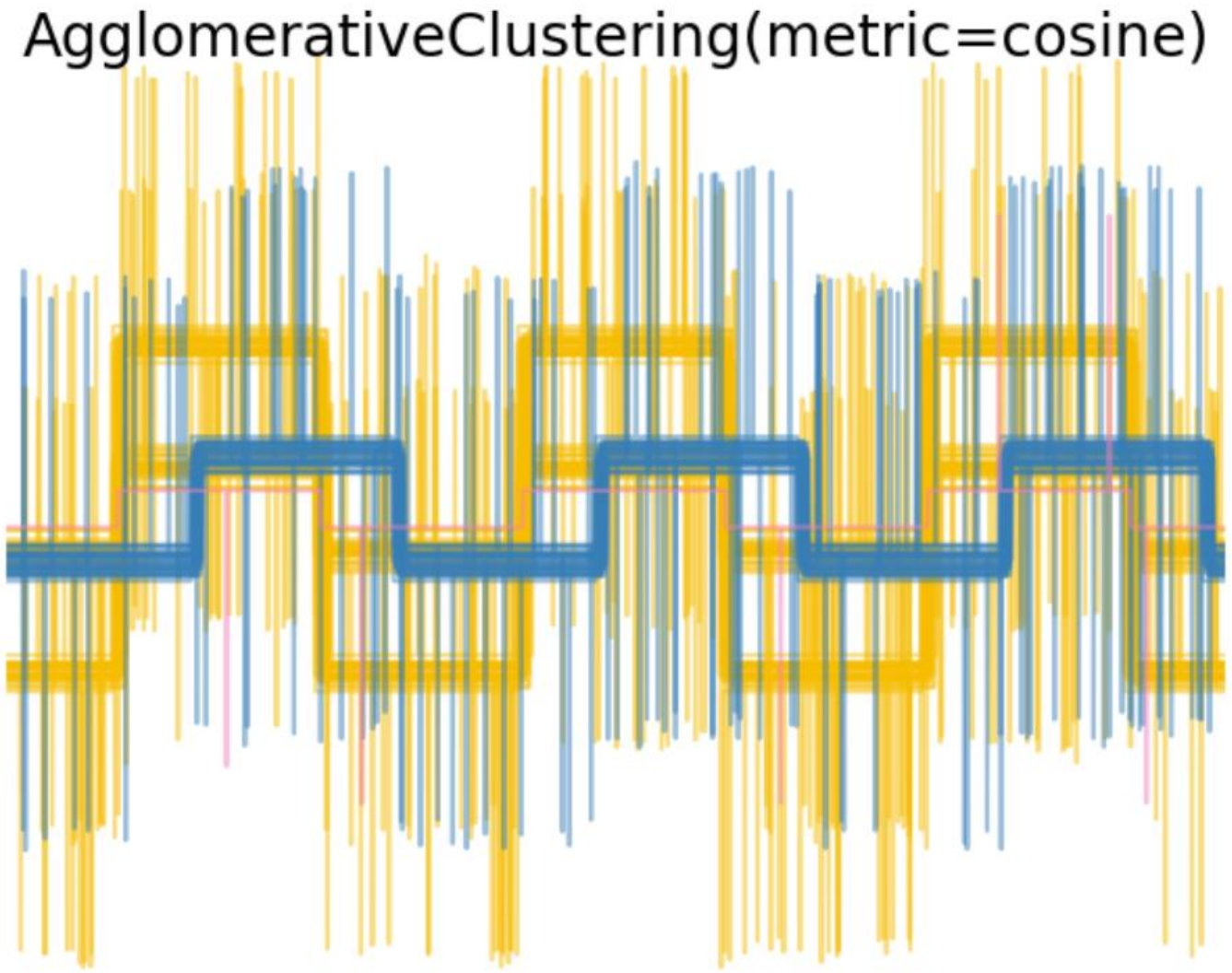


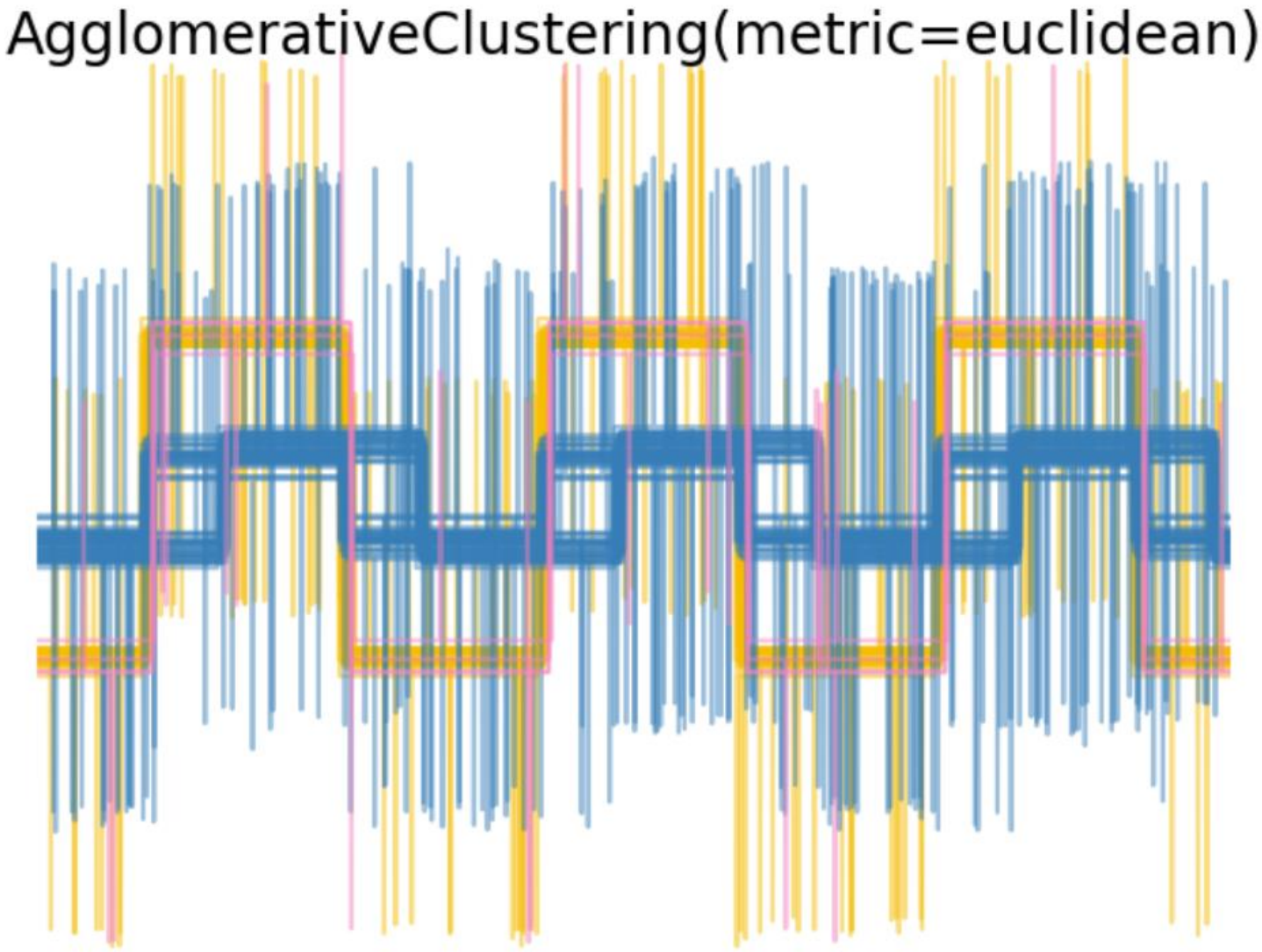


We add observation noise to these waveforms. We generate very sparse noise: only 6% of the time points contain noise. As a result, the l1 norm of this noise (ie “cityblock” distance) is much smaller than it’s l2 norm (“euclidean” distance). This can be seen on the inter-class distance matrices: the values on the diagonal, that characterize the spread of the class, are much bigger for the Euclidean distance than for the cityblock distance.






AgglomerativeClustering(metric=cosine)


AgglomerativeClustering(metric=euclidean)


AgglomerativeClustering(metric=cityblock)

When we apply clustering to the data, we find that the clustering reflects what was in the distance matrices. Indeed, for the Euclidean distance, the classes are ill-separated because of the noise, and thus the clustering does not separate the waveforms. For the cityblock distance, the separation is good and the waveform classes are recovered. Finally, the cosine distance does not separate at all waveform 1 and 2, thus the clustering puts them in the same cluster.