
Machine
Learning
for
Astrophysics

Unsupervised Learning

Outline

Introduction to ML & Supervised ML

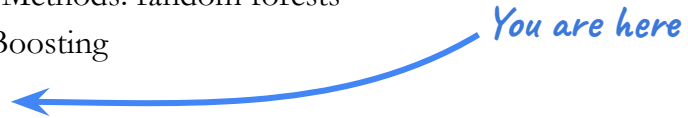
- Introduction
- Regression, Regularization
- Classification, Logistic Regression
- Bias/Variance trade-off

Supervised ML strikes back

- Support Vector Machines
- Gaussian Processes
- Nearest Neighbors
- Ensemble Methods: random forests
- Gradient Boosting

Unsupervised ML

- Clustering: KMeans, DBScan, GMM, Agglomerative Clustering
- Anomaly Detection
- Dimensionality Reduction:
  - linear: PCA, NMF, ICA
  - manifold learning: LLE, IsoMap, t-SNE
- Self-Organizing Maps



Deep Learning

- A few notes on HPC and GPUs (*maybe*)
- Basics of NN: computation graphs
- Training a NN: *forth*- and *back*-propagation
- Optimization Algorithms
- Tensorflow
- Transfer Learning
- Autoencoders
- Bayesian NN, Probabilistic BNN

Deep Learning, The Revenge

- (*hints on*) Reinforcement Learning
- Convolutional Neural Networks
- ResNet, Inception Module and MobileNet
- Generative Adversarial Networks
- (*hints on*) Recurrent Neural Networks
- Variational Autoencoders
- Transformers

The Holy Trinity of Machine Learning

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

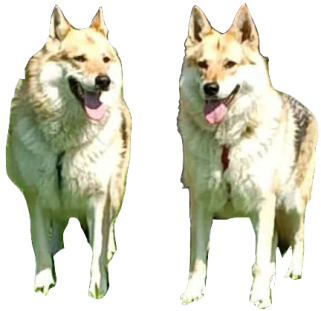
Goal:
learn a function that maps
$x \rightarrow y$

"This thing is a dog"

Unsupervised Learning

Data: x
there are no labels, only data x

Goal:
learn underlying structure of x



"These two things look alike"

Reinforcement Learning

Data: *state-action* pairs

Goal:
learn a policy π
maximizing future rewards

"Cuddling this thing will make you happy"

Unsupervised Learning

Data: x
there are no labels, only data x

Goal:

learn underlying structure of x

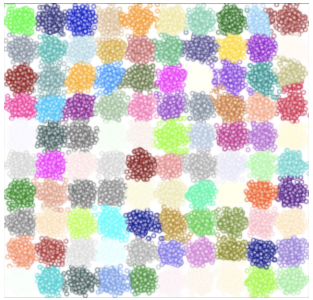
"These two things look alike"

There are three main categories of algorithms in unsupervised learning.

Clustering

Automatic grouping of similar objects into sets. *"Similar"* here means *"close in feature space"*.

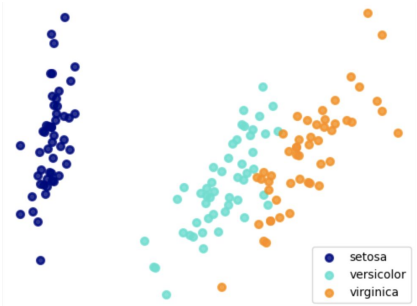
e.g. KMeans, DBScan, Hierarchical Clustering, GMM...



Dimensionality Reduction

Reducing the number of random variables to consider, either for visualization, or flexibility.

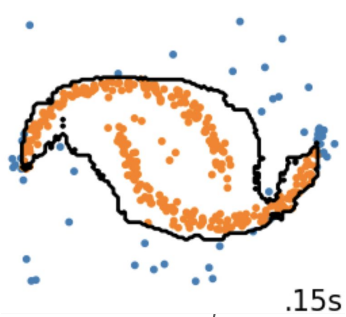
e.g. PCA, NNMF, IsoMap, LDA, t-SNE, SOMs...



Anomaly Detection

Automatically identify if a (new) observation is an anomaly, and/or clean datasets accordingly.

e.g. density estimation, isolation forest, Local Outlier Factor, SVMs...



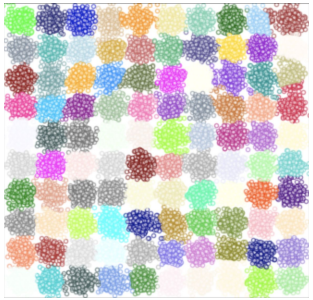
While the usual applications of these algorithms fall within those three categories, they are so versatile that it is not uncommon to see, e.g. *KMeans* applied as a dimensionality reduction algorithm; *DBScan* for anomaly detection; *t-SNE* or *PCA* for clustering; *SOMs* for anomaly detection…

There are three main categories of algorithms in unsupervised learning.

Clustering

Automatic grouping of similar objects into sets. *"Similar"* here means *"close in feature space"*.

e.g. KMeans, DBScan, Hierarchical Clustering, GMM...



Clustering means grouping unlabeled examples, a first step to understand a data set in a machine learning system.

Before you can *group* similar examples, you first need to *find* similar examples. This **similarity** between examples can be measured by combining the examples' feature data into a metric, called a **similarity measure**.

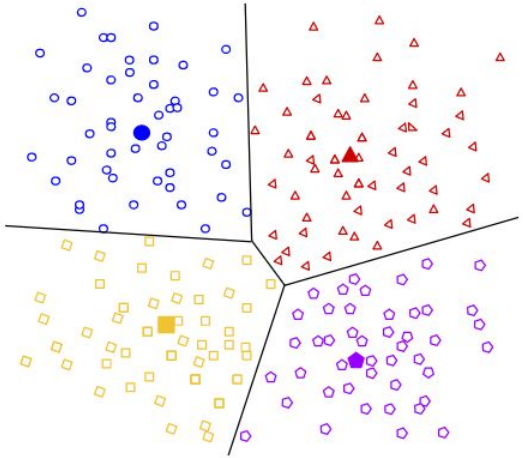
When each example is defined by one or two features, it's easy to measure **similarity**, e.g., find close stars in a 3D feature space (RA, DEC and distance). As the number of features increases, creating a similarity measure becomes more complex, and as such clustering.

There are roughly four type of clustering algorithms:

Centroid-based clustering:

Centroid-based algorithms are based on how distant each example is from a (presumed) cluster center. Centroid-based algorithms are efficient but sensitive to initial conditions and outliers.

e.g. KMeans



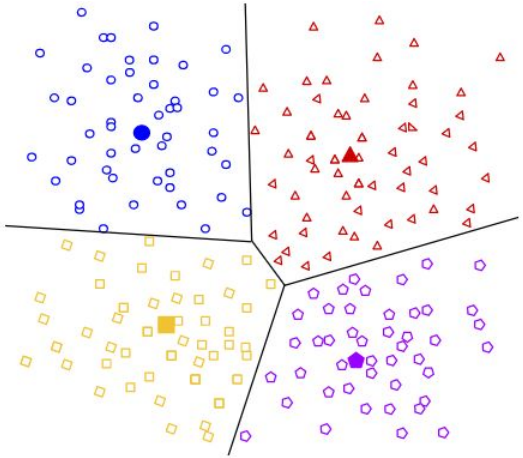
Clustering Types

There are roughly four type of clustering algorithms:

Centroid-based clustering:

Centroid-based algorithms are based on how distant each example is from a (presumed) cluster center.
Centroid-based algorithms are efficient but sensitive to initial conditions and outliers.

e.g. KMeans



Density-based clustering:

Connecting areas of high example density into clusters, allowing for for arbitrary-shaped distributions.
They have difficulties on higher dimensions, and on data of varying density.
By design, these algorithms do not assign outliers to clusters.

e.g. DBScan

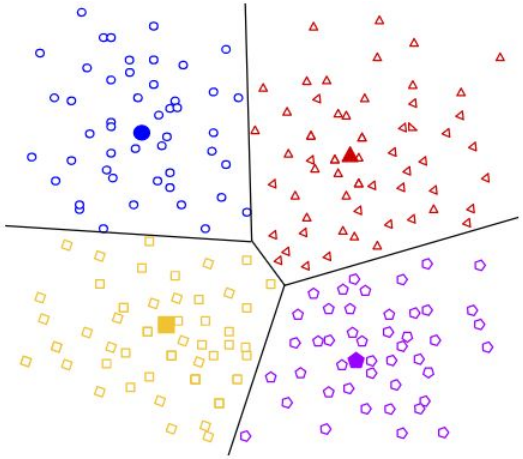
Clustering Types

There are roughly four type of clustering algorithms:

Centroid-based clustering:

Centroid-based algorithms are based on how distant each example is from a (presumed) cluster center.
Centroid-based algorithms are efficient but sensitive to initial conditions and outliers.

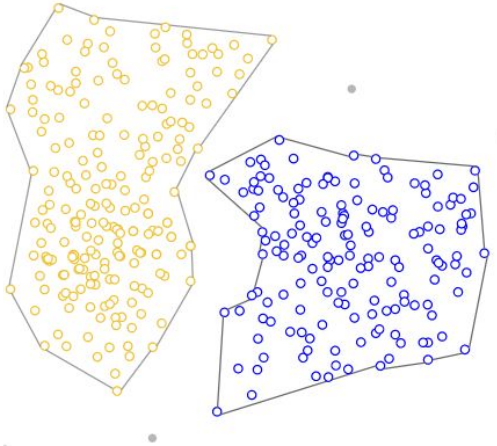
e.g. KMeans



Density-based clustering:

Connecting areas of high example density into clusters, allowing for for arbitrary-shaped distributions.
They have difficulties on higher dimensions, and on data of varying density.
By design, these algorithms do not assign outliers to clusters.

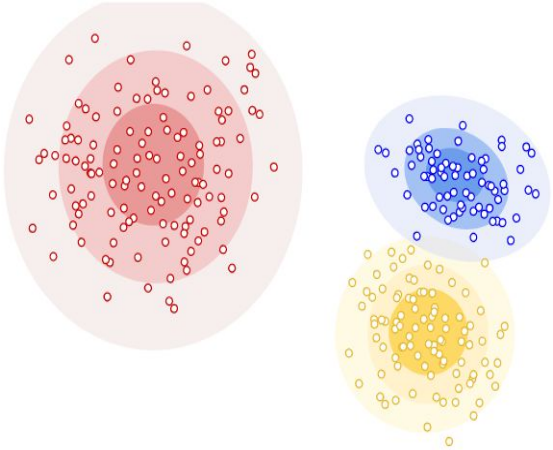
e.g. DBScan



Distribution-based clustering

This clustering approach assumes data is composed of distributions (e.g. Gaussian distributions).
When you do not know the type of distribution in your data, you should use a different algorithm.

e.g. Gaussian Mixture Models



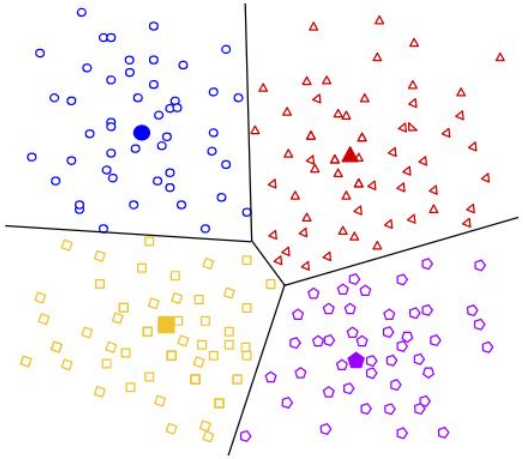
Clustering Types

There are roughly four type of clustering algorithms:

Centroid-based clustering:

Centroid-based algorithms are based on how distant each example is from a (presumed) cluster center.
Centroid-based algorithms are efficient but sensitive to initial conditions and outliers.

e.g. KMeans



Density-based clustering:

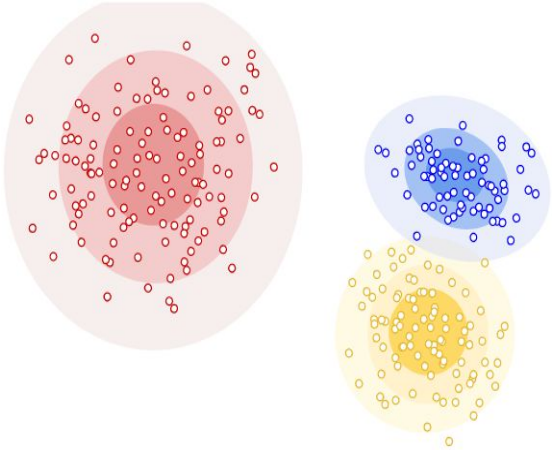
Connecting areas of high example density into clusters, allowing for for arbitrary-shaped distributions.
They have difficulties on higher dimensions, and on data of varying density.
By design, these algorithms do not assign outliers to clusters.

e.g. DBScan

Distribution-based clustering

This clustering approach assumes data is composed of distributions (e.g. Gaussian distributions).
When you do not know the type of distribution in your data, you should use a different algorithm.

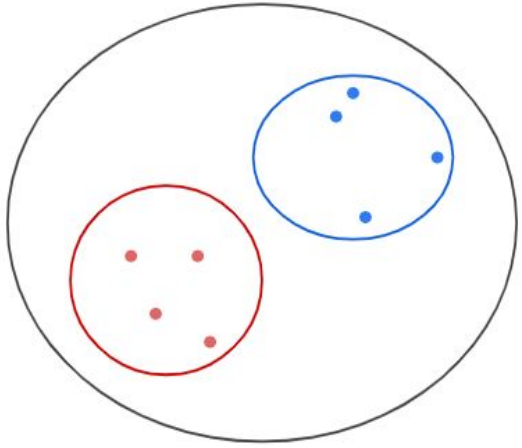
e.g. Gaussian Mixture Models



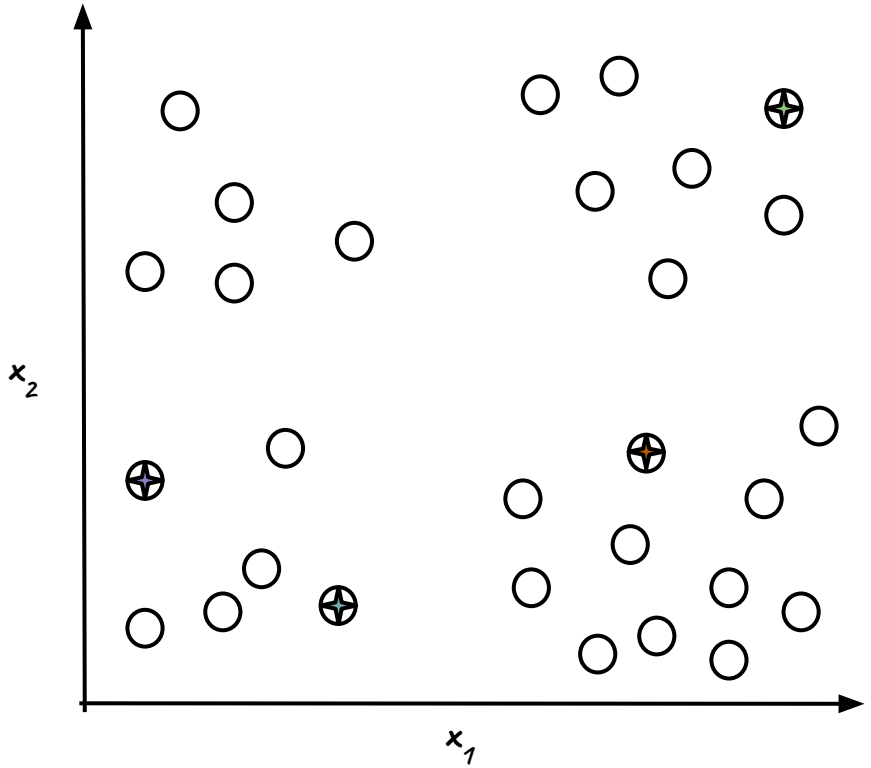
Hierarchical clustering

Hierarchical clustering creates a tree of clusters. Hierarchical clustering, not surprisingly, is well suited to hierarchical data.
Any number of clusters can be chosen by cutting the tree at the right level.

e.g. AgglomerativeClustering



Let's start with the easiest clustering algorithms available on the market: *KMeans*. Here is an example.

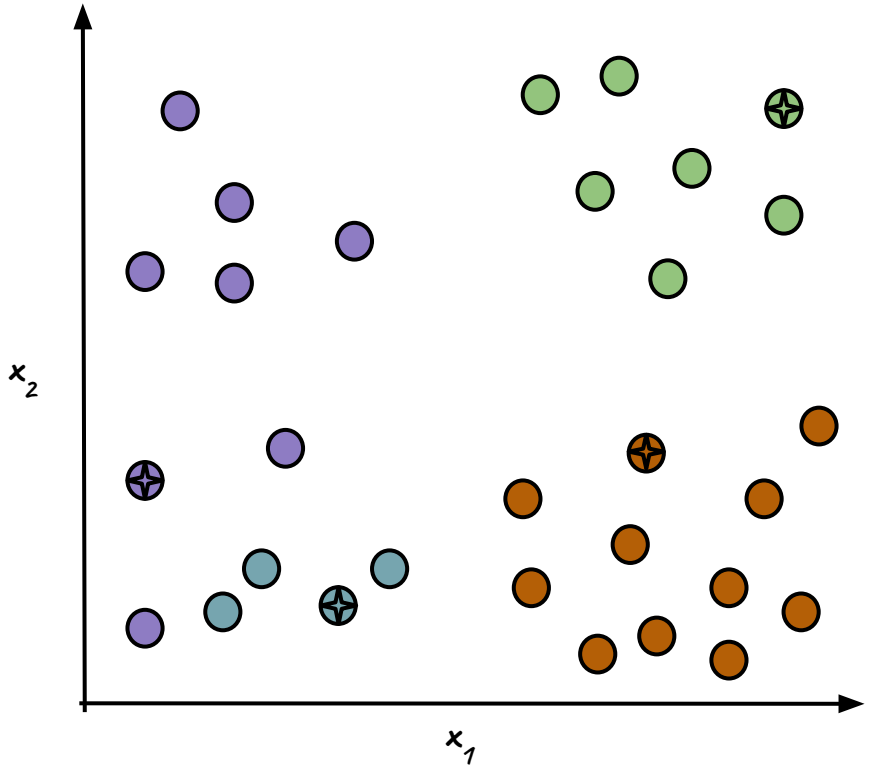


https://k-means-explorable.vercel.app/

The number of clusters is a hyperparameter of *KMeans*, so in the example on the left our gut feeling tells us that there are 4 clusters.
The algorithm works this way:

1) first guess where the cluster centres might be (randomly picked between the example, *"random initialization"*)

Let's start with the easiest clustering algorithms available on the market: *KMeans*. Here is an example.

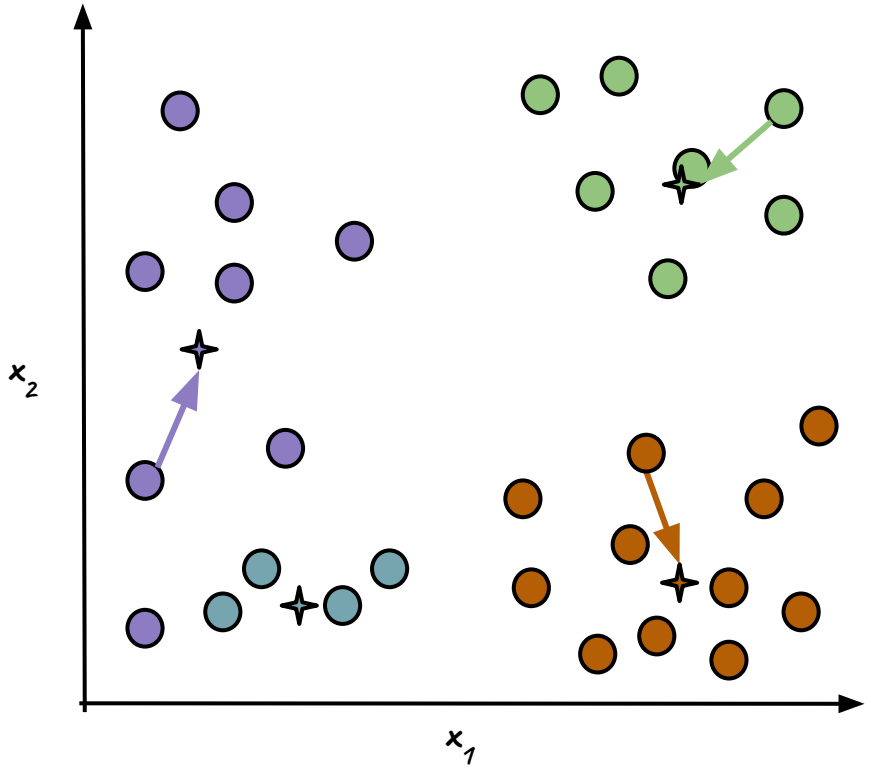


https://k-means-explorable.vercel.app/

The number of clusters is a hyperparameter of *KMeans*, so in the example on the left our gut feeling tells us that there are 4 clusters.
The algorithm works this way:

1) first guess where the cluster centres might be (randomly picked between the example, *"random initialization"*)
2) assign points to the closest centroid cluster (depending on the chosen metric)

Let's start with the easiest clustering algorithms available on the market: *KMeans*. Here is an example.

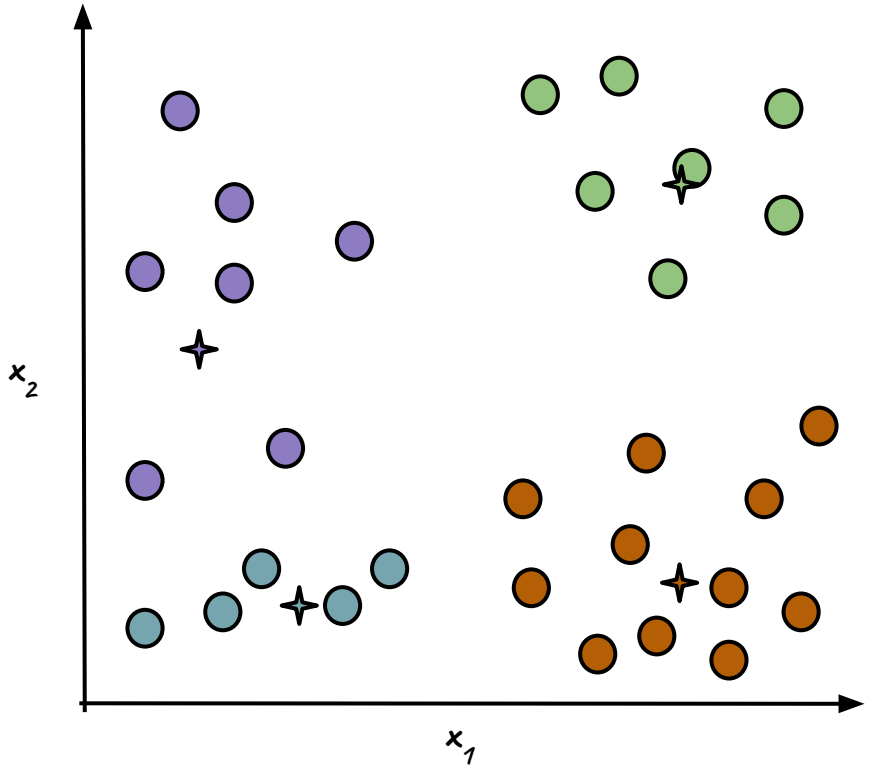


https://k-means-explorable.vercel.app/

The number of clusters is a hyperparameter of *KMeans*, so in the example on the left our gut feeling tells us that there are 4 clusters.
The algorithm works this way:

1) first guess where the cluster centres might be (randomly picked between the example, *"random initialization"*)
2) assign points to the closest centroid cluster (depending on the chosen metric)
3) recompute the centroids as the average positions (**means**) of the points belonging to the four clusters

Let's start with the easiest clustering algorithms available on the market: *KMeans*. Here is an example.



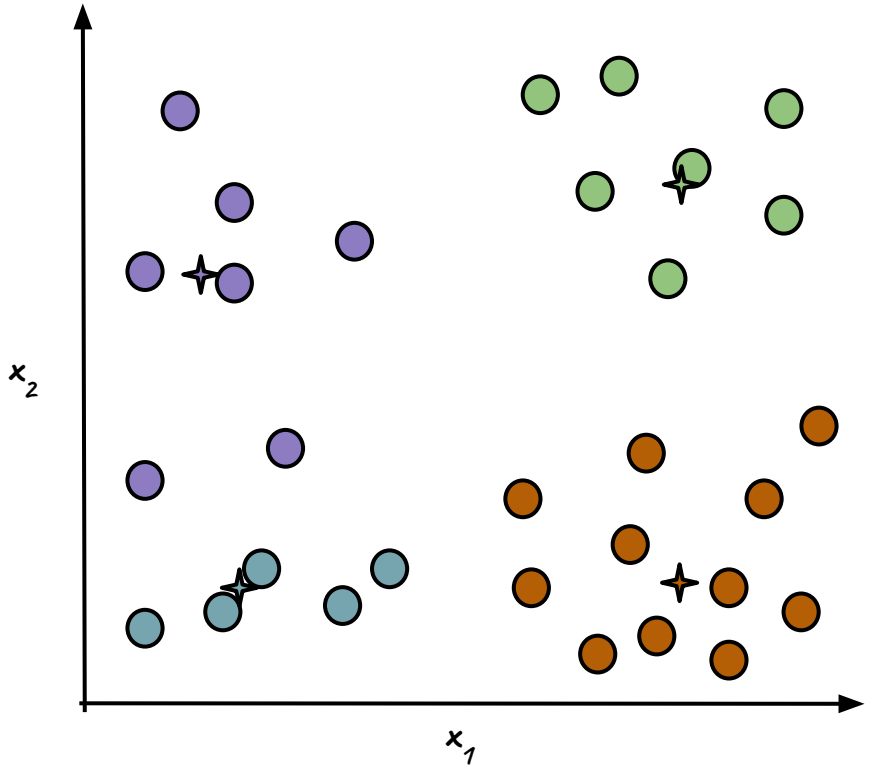
https://k-means-explorable.vercel.app/

The number of clusters is a hyperparameter of *KMeans*, so in the example on the left our gut feeling tells us that there are 4 clusters.
The algorithm works this way:

1) first guess where the cluster centres might be (randomly picked between the example, *"random initialization"*)
2) assign points to the closest centroid cluster (depending on the chosen metric)
3) recompute the centroids as the average positions (**means**) of the points belonging to the four clusters

Repeat 2) and 3) until convergence.

Let's start with the easiest clustering algorithms available on the market: *KMeans*. Here is an example.



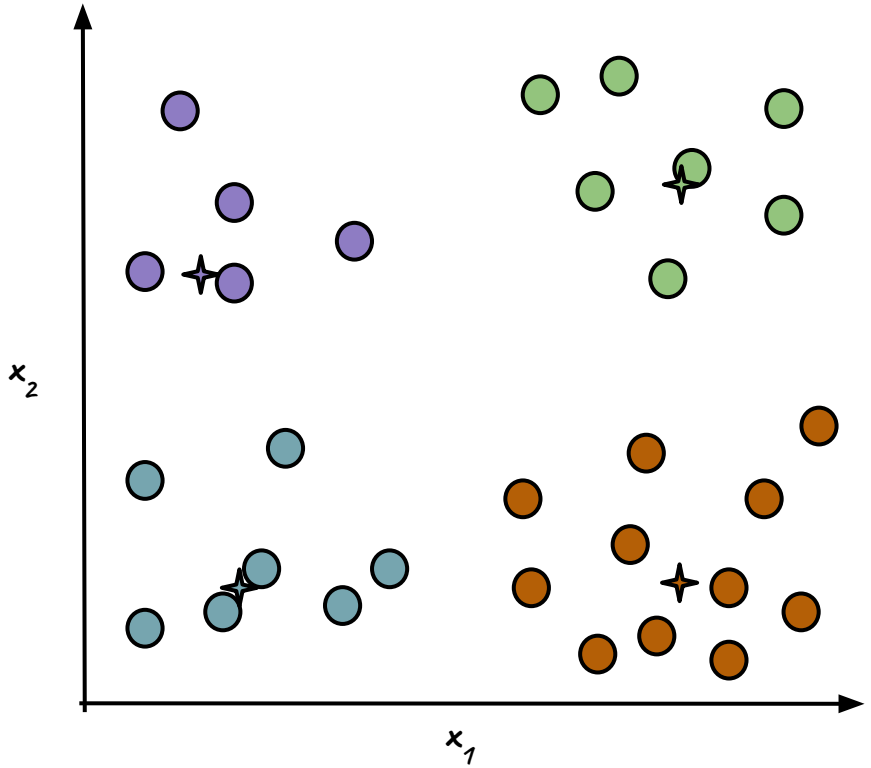
https://k-means-explorable.vercel.app/

The number of clusters is a hyperparameter of *KMeans*, so in the example on the left our gut feeling tells us that there are 4 clusters.
The algorithm works this way:

1) first guess where the cluster centres might be (randomly picked between the example, *"random initialization"*)
2) assign points to the closest centroid cluster (depending on the chosen metric)
3) recompute the centroids as the average positions (**means**) of the points belonging to the four clusters

Repeat 2) and 3) until convergence.

Let's start with the easiest clustering algorithms available on the market: *KMeans*. Here is an example.



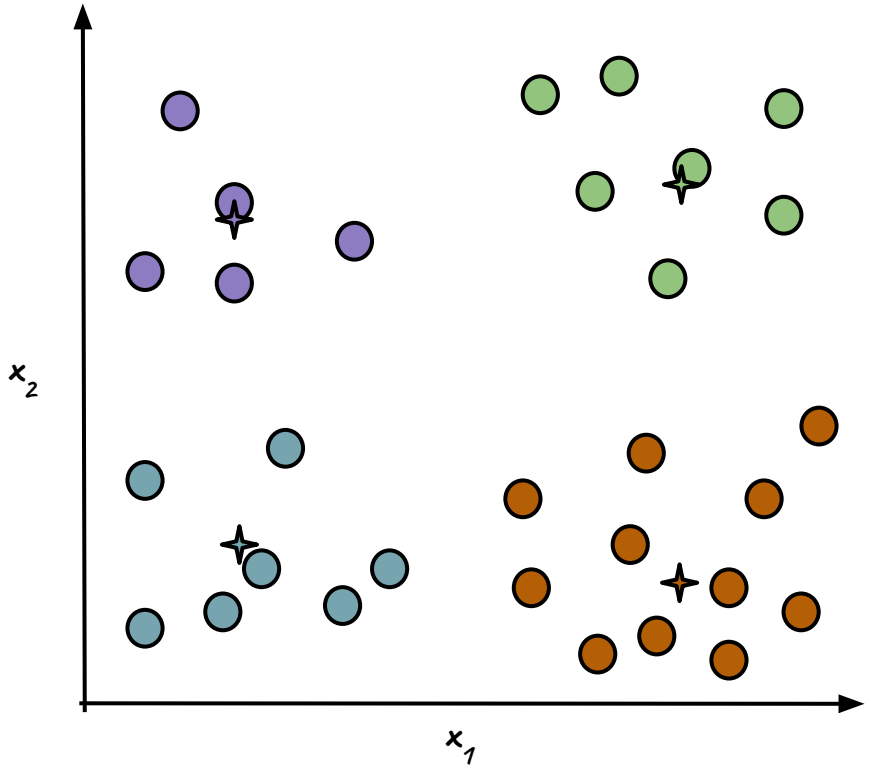
https://k-means-explorable.vercel.app/

The number of clusters is a hyperparameter of *KMeans*, so in the example on the left our gut feeling tells us that there are 4 clusters.
The algorithm works this way:

1) first guess where the cluster centres might be (randomly picked between the example, *"random initialization"*)
2) assign points to the closest centroid cluster (depending on the chosen metric)
3) recompute the centroids as the average positions (**means**) of the points belonging to the four clusters

Repeat 2) and 3) until convergence.

Let's start with the easiest clustering algorithms available on the market: *KMeans*. Here is an example.



https://k-means-explorable.vercel.app/

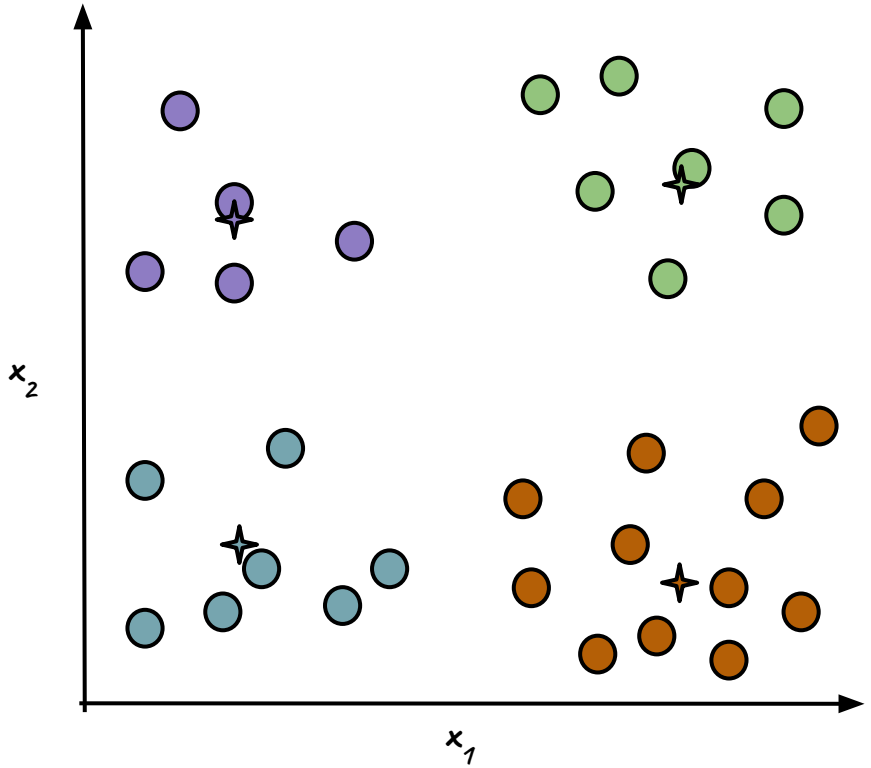
The number of clusters is a hyperparameter of *KMeans*, so in the example on the left our gut feeling tells us that there are 4 clusters.
The algorithm works this way:

1) first guess where the cluster centres might be (randomly picked between the example, *"random initialization"*)
2) assign points to the closest centroid cluster (depending on the chosen metric)
3) recompute the centroids as the average positions (**means**) of the points belonging to the four clusters

Repeat 2) and 3) until convergence

→ when cluster centroids do not move anymore

Let's start with the easiest clustering algorithms available on the market: *KMeans*. Here is an example.



https://k-means-explorable.vercel.app/

The number of clusters is a hyperparameter of *KMeans*, so in the example on the left our gut feeling tells us that there are 4 clusters.
The algorithm works this way:

1) first guess where the cluster centres might be (randomly picked between the example, *"random initialization"*)
2) assign points to the closest centroid cluster (depending on the chosen metric)
3) recompute the centroids as the average positions (*means*) of the points belonging to the four clusters

Repeat 2) and 3) until convergence

→ when cluster centroids do not move anymore

Generalizing for k clusters:

for i = 1 to m:

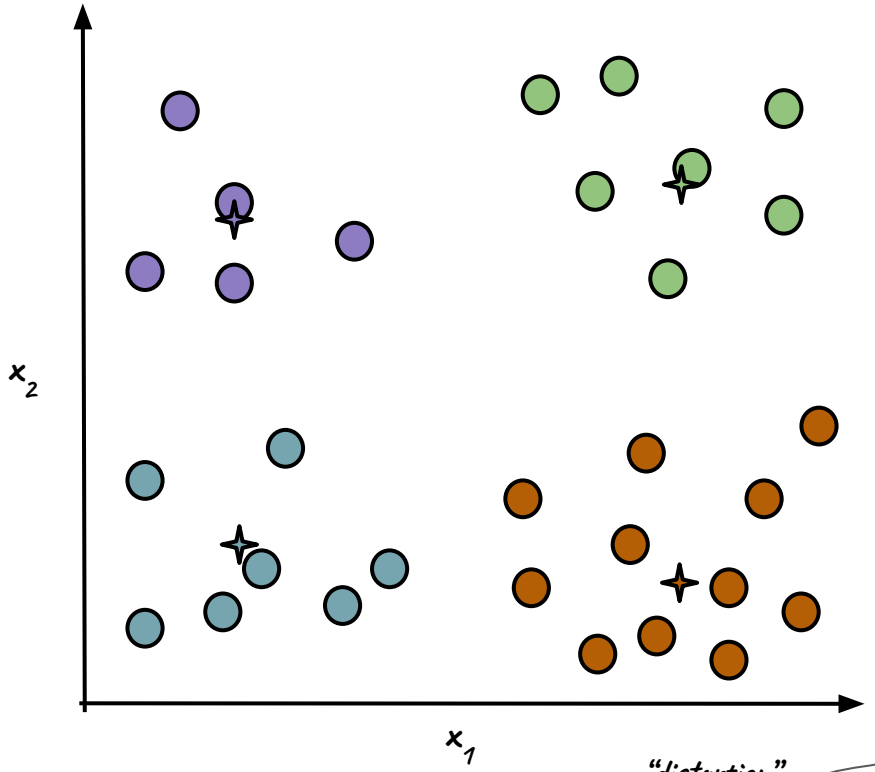
$c^{(i)}$ = index (from 1 to K) of cluster centroid closest to $x^{(i)}$

min $||x^{(i)} - \mu_k||^2$ *(also called the* L_2 *norm)*

for k = 1 to K:

μ_k = average (mean) of points assigned to cluster k

Let's start with the easiest clustering algorithms available on the market: *KMeans*. Here is an example.



The number of clusters is a hyperparameter of *KMeans*, so in the example on the left our gut feeling tells us that there are 4 clusters.
The algorithm works this way:

1) first guess where the cluster centres might be (randomly picked between the example, *"random initialization"*)
2) assign points to the closest centroid cluster (depending on the chosen metric)
3) recompute the centroids as the average positions (**means**) of the points belonging to the four clusters

Repeat 2) and 3) until convergence

→ when cluster centroids do not move anymore

Generalizing for **k** clusters:

for i = 1 to m:

$c^{(i)}$ = index (from 1 to K) of cluster centroid closest to $x^{(i)}$

min $||x^{(i)} - \mu_k||^2$ *(also called the* L_2 *norm)*

for k = 1 to K:

μ_k = average (mean) of points assigned to cluster **k**

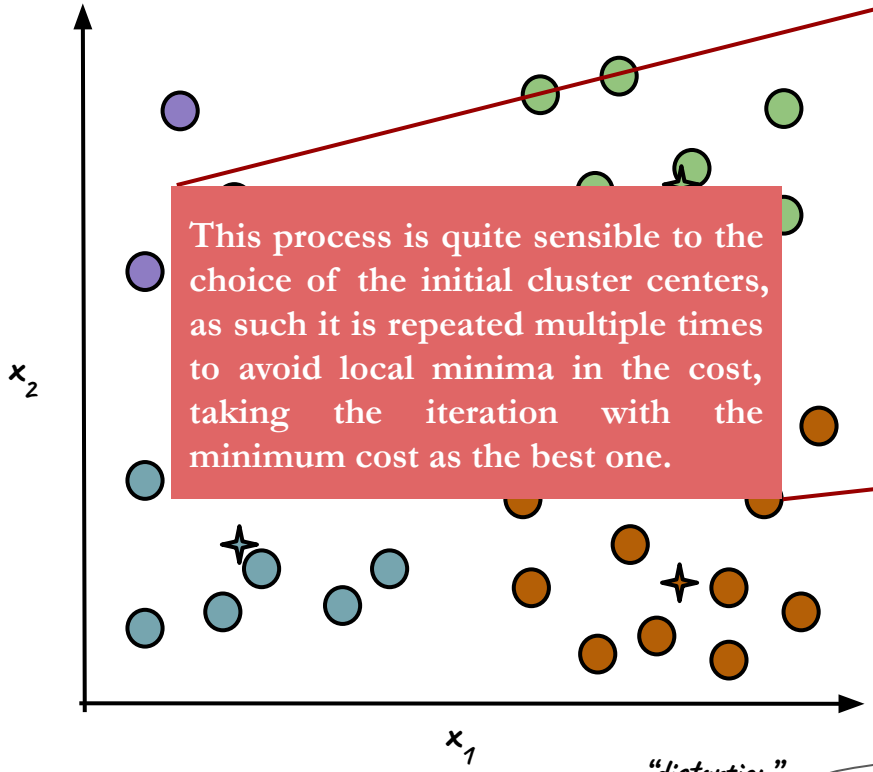
The *KMeans* cost function (here, for Euclidean distance) is:

"distortion" →

$$J(c^{(1)} \dots c^{(m)}, \mu_1 \dots \mu_m) = \frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - \mu_c^{(i)}||^2$$

https://k-means-explorable.vercel.app/

Let's start with the easiest clustering algorithms available on the market: *KMeans*. Here is an example.



This process is quite sensible to the choice of the initial cluster centers, as such it is repeated multiple times to avoid local minima in the cost, taking the iteration with the minimum cost as the best one.

The number of clusters is a hyperparameter of *KMeans*, so in the example on the left our gut feeling tells us that there are 4 clusters.

The algorithm works this way:

1) first guess where the cluster centres might be (randomly picked between the example, *"random initialization"*)
2) assign points to the closest centroid cluster (depending on the chosen metric)
3) recompute the centroids as the average positions (*means*) of the points belonging to the four clusters

Repeat 2) and 3) until convergence

→ when cluster centroids do not move anymore

Generalizing for *k* clusters:

for i = 1 to m:

$c^{(i)}$ = index (from 1 to K) of cluster centroid closest to $x^{(i)}$

min $||x^{(i)} - \mu_k||^2$ *(also called the* L_2 *norm)*

for k = 1 to K:

μ_k = average (mean) of points assigned to cluster *k*

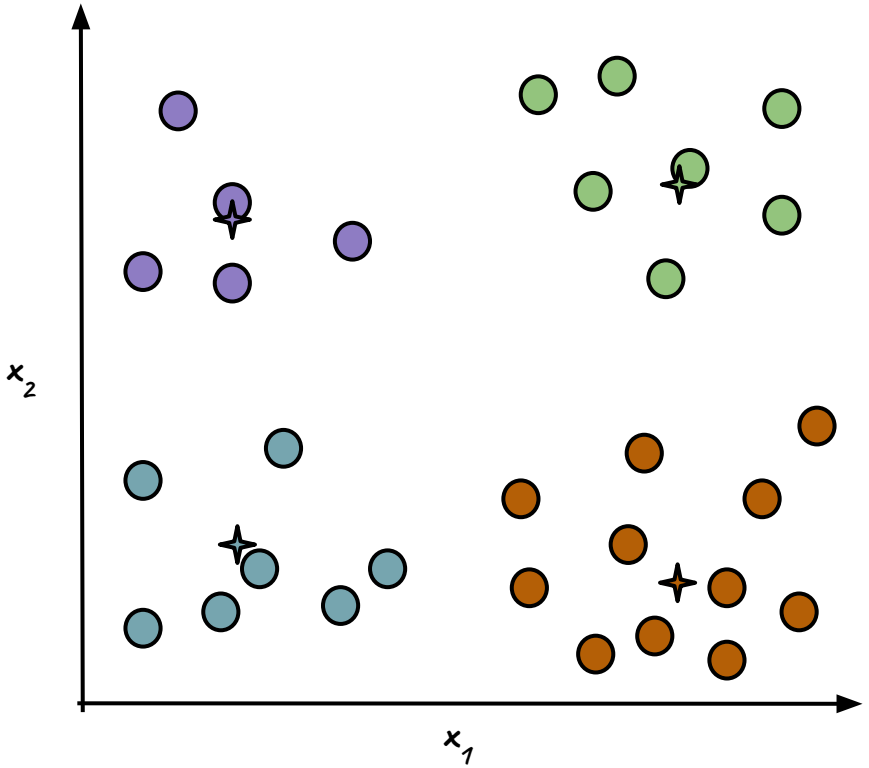
The *KMeans* cost function (here, for Euclidean distance) is:

"distortion" →

$$J(c^{(1)} \dots c^{(m)}, \mu_1 \dots \mu_m) = \frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - \mu_c^{(i)}||^2$$

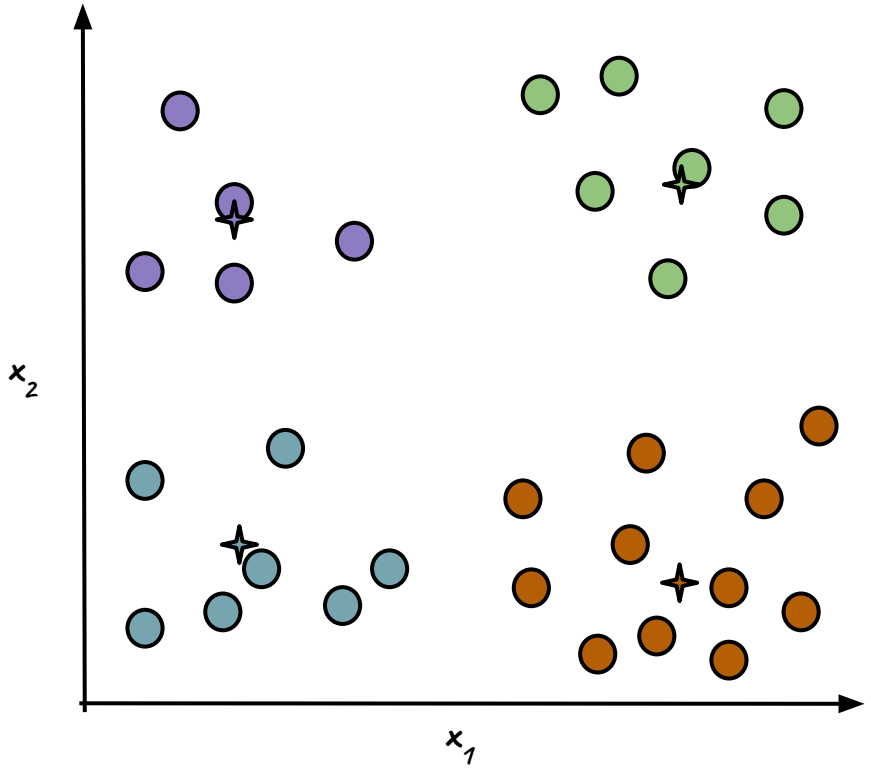
https://k-means-explorable.vercel.app/

KMeans it's easy, it's simple, it might lead to good results, but with some big caveats attached:



https://k-means-explorable.vercel.app/

KMeans it's easy, it's simple, it might lead to good results, but with some big caveats attached:



https://k-means-explorable.vercel.app/

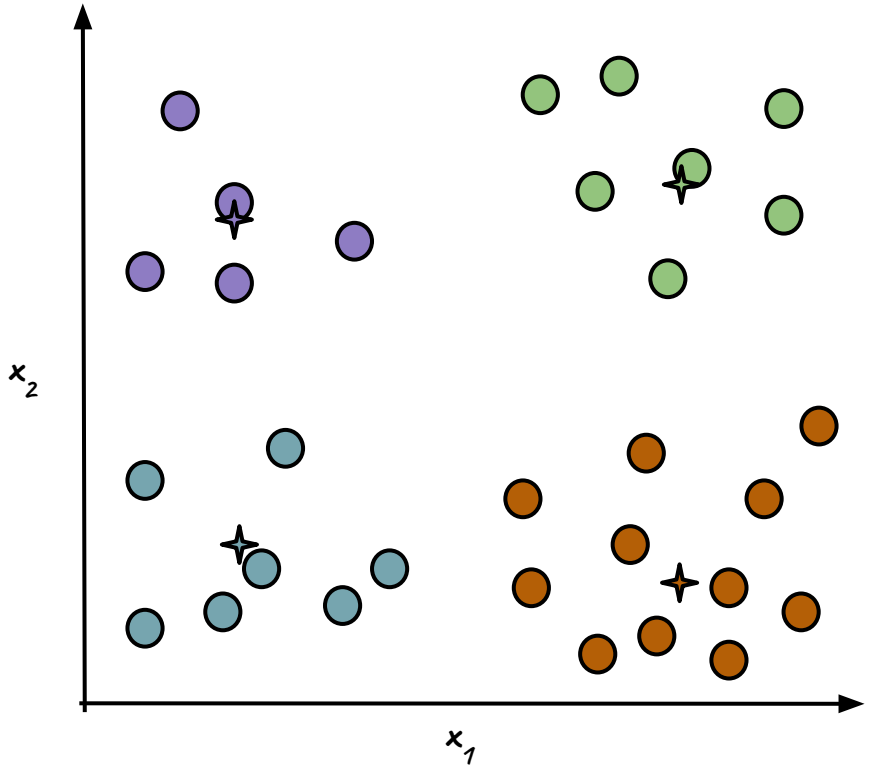
Advantages:

- quite efficient: scales as O(*tkn*), where *n* is number of objects, *k* the number clusters, and *t* the number of iterations. Usually, $k, t << n$.
- easy to understand, easy to use

Disadvantages:

- the concept of average is applicable for numerical values, for categorical values is not that straightforward
- need to know the *k* number of cluster in advance
- not great results for noisy data; outliers might cause a mess
- the chosen metric greatly influences the quality of the clustering: Euclidean metric will favor circularly symmetric clusters, and might have a hard time in finding elongated ones

KMeans it's easy, it's simple, it might lead to good results, but with some big caveats attached:



https://k-means-explorable.vercel.app/

Advantages:

- quite efficient: scales as O(*tkn*), where *n* is number of objects, *k* the number clusters, and *t* the number of iterations. Usually, $k, t << n$.
- easy to understand, easy to use

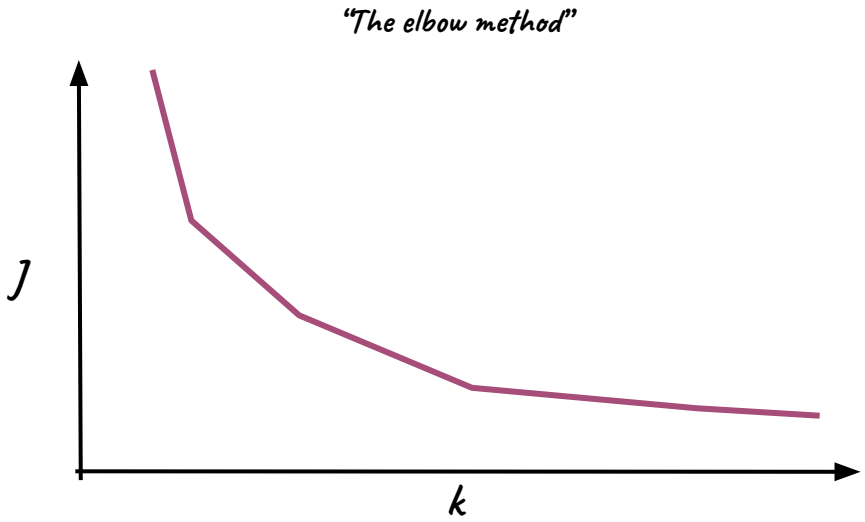
Disadvantages:

- the concept of average is applicable for numerical values, for categorical values is not that straightforward
- need to know the *k* number of cluster in advance
- not great results for noisy data; outliers might cause a mess
- the chosen metric greatly influences the quality of the clustering: Euclidean metric will favor circularly symmetric clusters, and might have a hard time in finding elongated ones

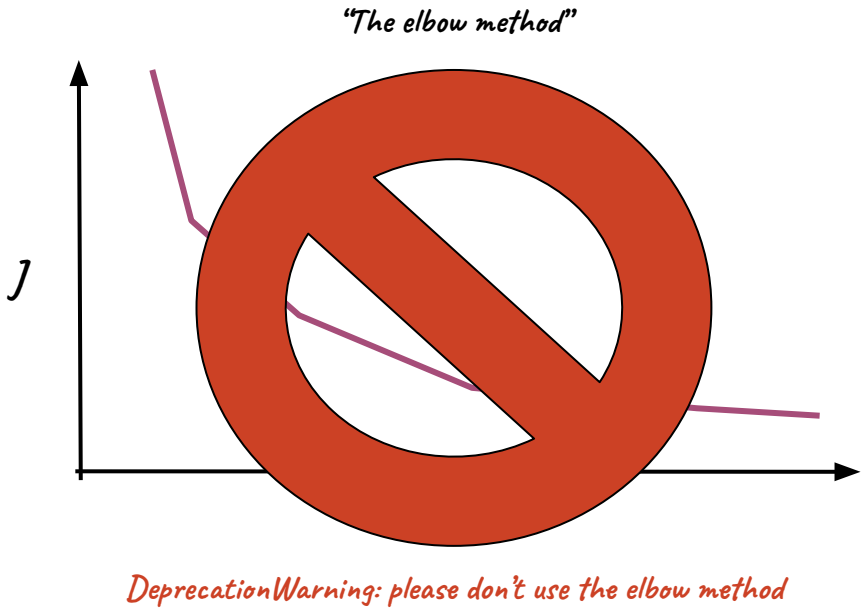
Mini-batch KMeans: same mini-batch intuition of the mini-batch gradient descent: at each iteration find the cluster centroids from a random subsample of the examples set. Noisy cost descent, but computationally efficient.

How do you choose in advance the number of clusters? There are mainly three methods.
The first is easy: gut feeling for the optimal number of clusters for your data.

How do you choose in advance the number of clusters? There are mainly three methods.
The first is easy: gut feeling for the optimal number of clusters for your data.



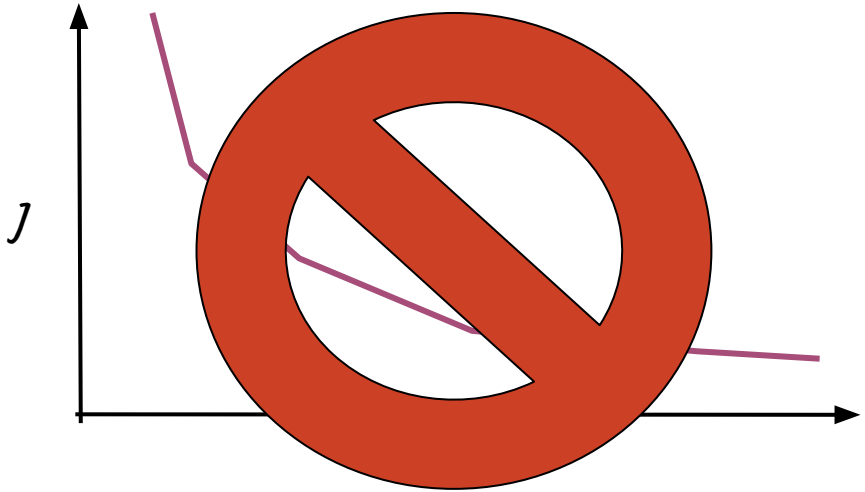
How do you choose in advance the number of clusters? There are mainly three methods.
The first is easy: gut feeling for the optimal number of clusters for your data.



DeprecationWarning: please don't use the elbow method

How do you choose in advance the number of clusters? There are mainly three methods.
The first is easy: gut feeling for the optimal number of clusters for your data.

"The elbow method"



DeprecationWarning: please don't use the elbow method

Optimize a metric measurement wrt k number of clusters

Remember, this is **unsupervised** learning. You do not know the ground truth (or at least, the model does not). You cannot compare the labels found by the clustering algorithms to some true values.
But still, you can define some metrics to evaluate the quality of your results… just remember that these are based on some pre-assumptions on how your data should be.

e.g. the *silhouette score, Calinski-Harabasz index, the Davies-Bouldin score*

Silhouette is an "internal" score, used to assess the optimal number of clusters (and thus the quality of clustering), by measuring how close a point assigned to a cluster is to the points in the neighboring ones.

It assigns a coefficient to each example:

$$s = \frac{b - a}{max(a, b)}$$

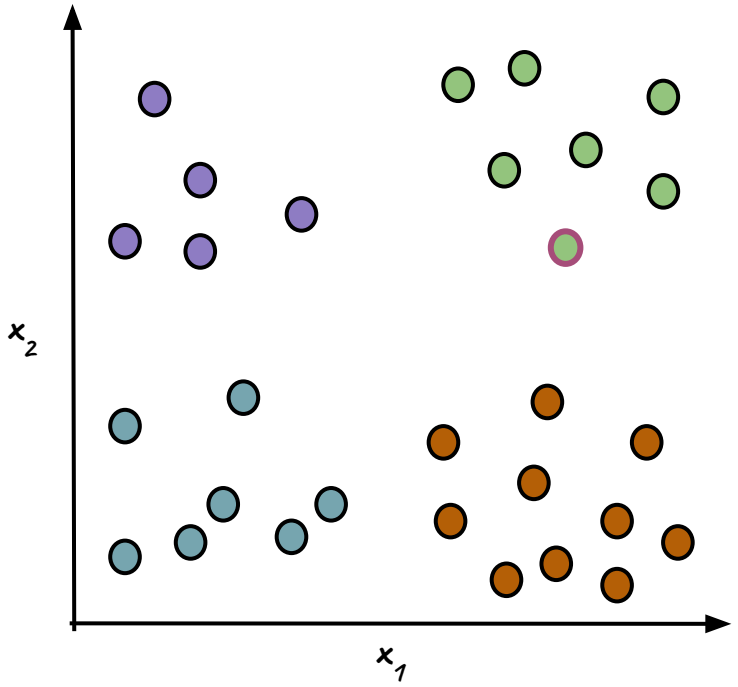
Mean distance between the example and all the other examples in the **nearest** *cluster*

Mean distance between the example and all the other examples in the **same** *cluster*

The coefficient is always in the range [-1, 1]:

- *negative values*: the example has been assigned to the wrong cluster ❌
- *close to zero*: the example is on or very close to the boundary between two clusters
- *close to +1*: the example is far away from the neighboring clusters ✅

Silhouette score is the coefficients average. It is evaluated for different number of clusters k, looking for the one with the highest score.



Centroid-based clustering - Silhouette score

Silhouette is an "internal" score, used to assess the optimal number of clusters (and thus the quality of clustering), by measuring how close a point assigned to a cluster is to the points in the neighboring ones.

It assigns a coefficient to each example:

$$s = \frac{b - a}{max(a, b)}$$

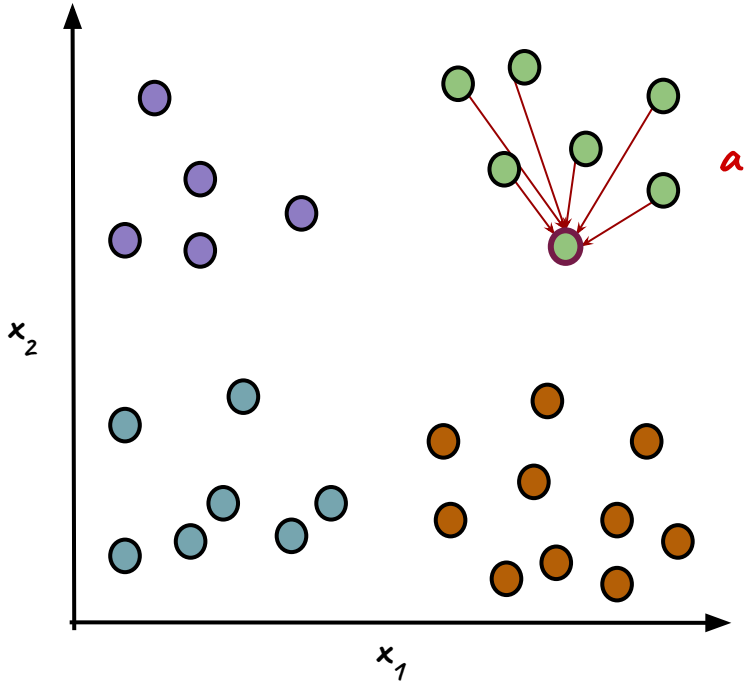
Mean distance between the example and all the other examples in the **nearest** *cluster*

Mean distance between the example and all the other examples in the **same** *cluster*

The coefficient is always in the range [-1, 1]:

- *negative values*: the example has been assigned to the wrong cluster ❌
- *close to zero*: the example is on or very close to the boundary between two clusters
- *close to +1*: the example is far away from the neighboring clusters ✅

Silhouette score is the coefficients average. It is evaluated for different number of clusters k, looking for the one with the highest score.



Silhouette is an "internal" score, used to assess the optimal number of clusters (and thus the quality of clustering), by measuring how close a point assigned to a cluster is to the points in the neighboring ones.

It assigns a coefficient to each example:

$$s = \frac{b - a}{max(a, b)}$$

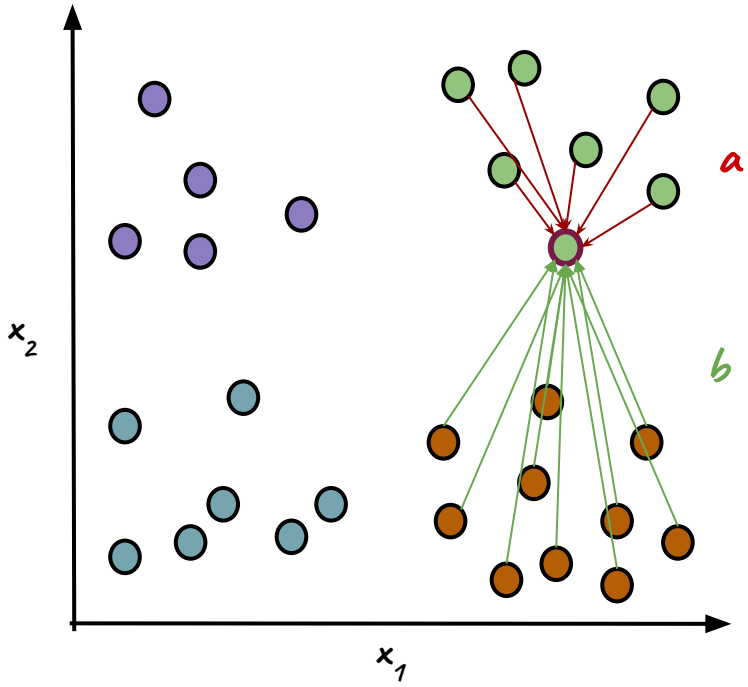
Mean distance between the example and all the other examples in the **nearest** *cluster*

Mean distance between the example and all the other examples in the **same** *cluster*

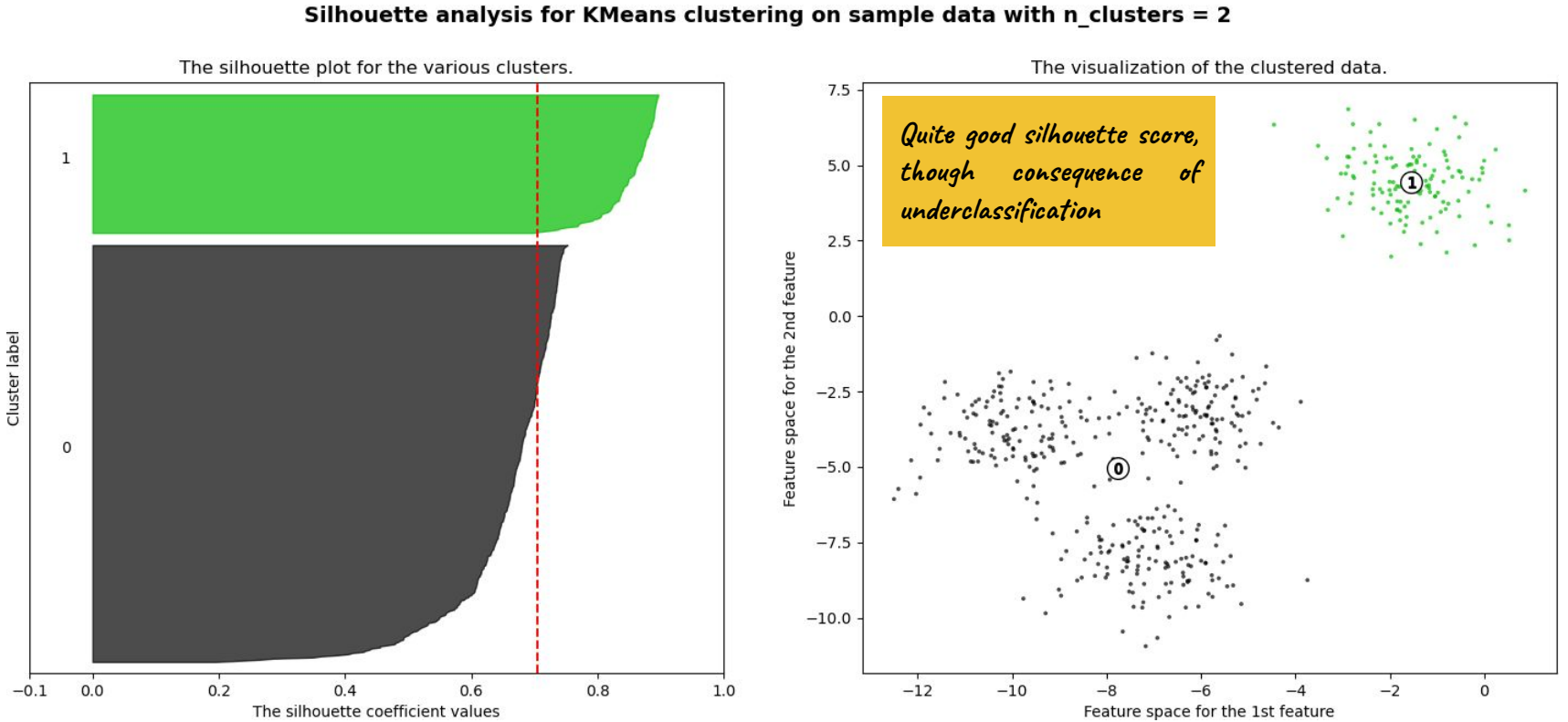
The coefficient is always in the range [-1, 1]:

- *negative values*: the example has been assigned to the wrong cluster ❌
- *close to zero*: the example is on or very close to the boundary between two clusters
- *close to +1*: the example is far away from the neighboring clusters ✅

Silhouette score is the coefficients average. It is evaluated for different number of clusters k, looking for the one with the highest score.

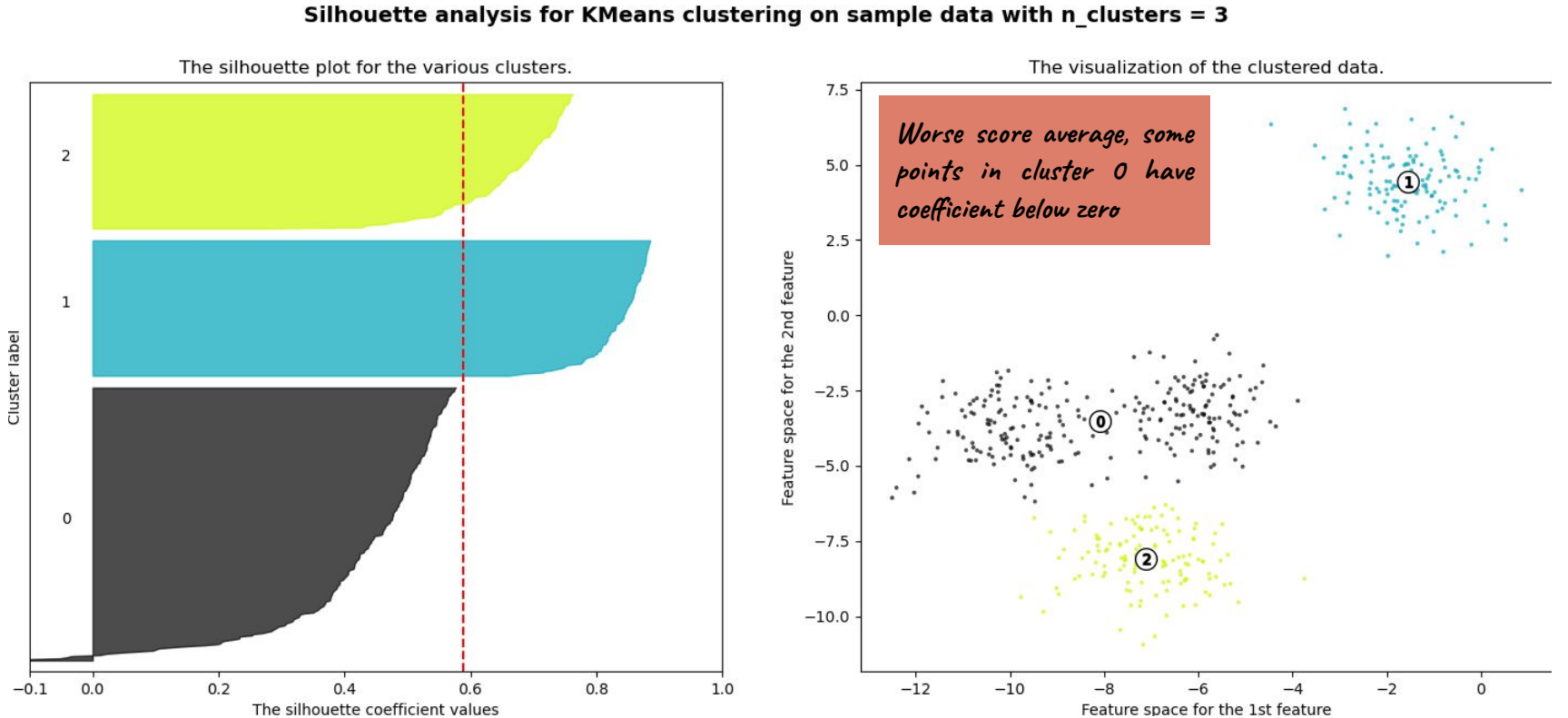


Silhouette is an "internal" score, used to assess the optimal number of clusters (and thus the quality of clustering), by measuring how close a point assigned to a cluster is to the points in the neighboring ones.



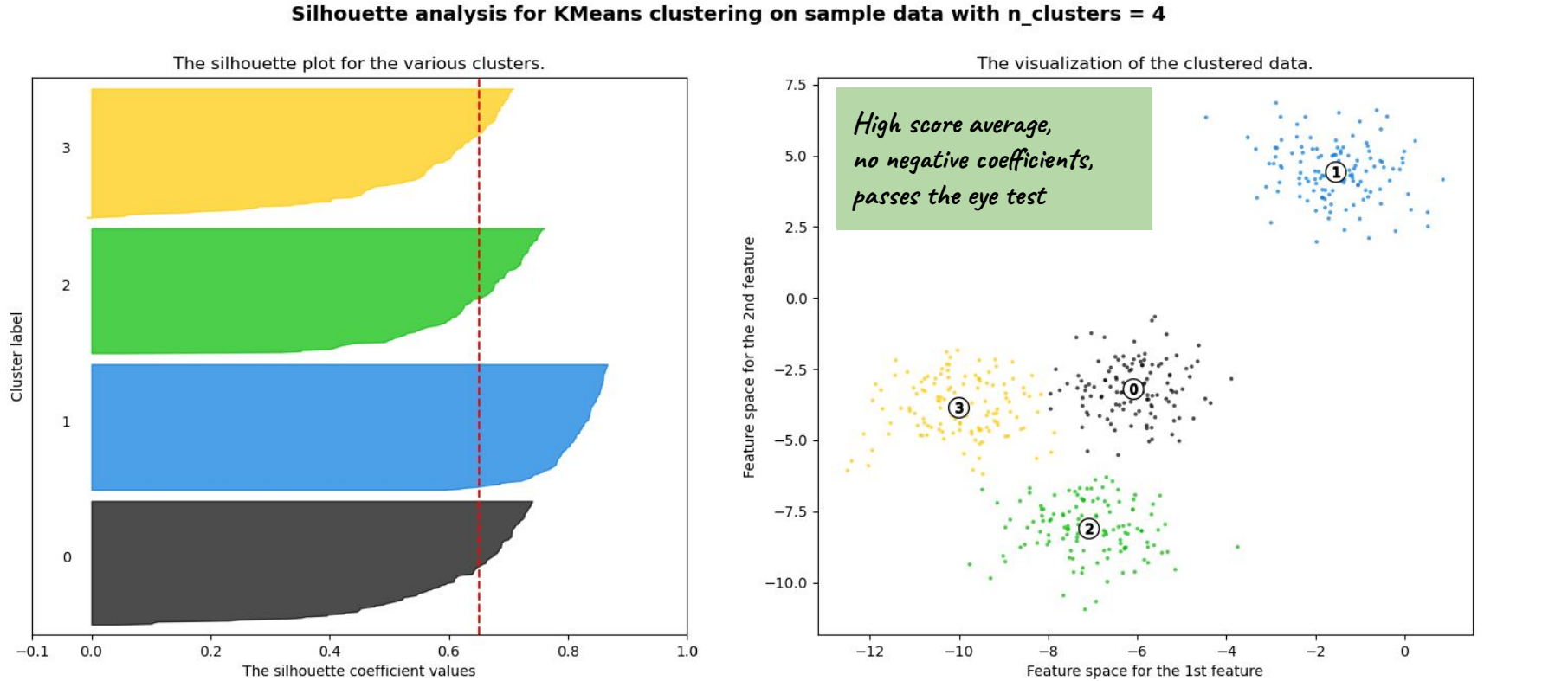
https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html

Silhouette is an "internal" score, used to assess the optimal number of clusters (and thus the quality of clustering), by measuring how close a point assigned to a cluster is to the points in the neighboring ones.



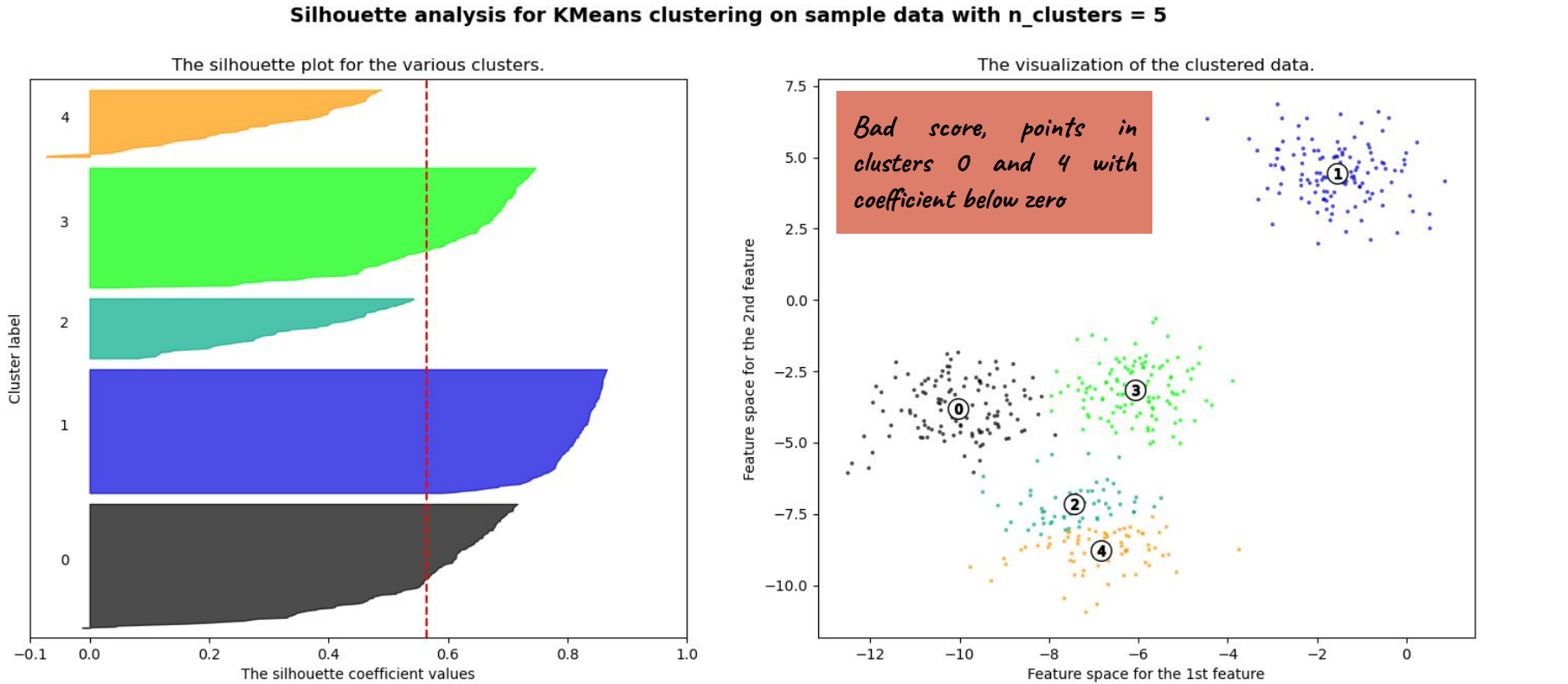
https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html

Silhouette is an "internal" score, used to assess the optimal number of clusters (and thus the quality of clustering), by measuring how close a point assigned to a cluster is to the points in the neighboring ones.



https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html

Silhouette is an "internal" score, used to assess the optimal number of clusters (and thus the quality of clustering), by measuring how close a point assigned to a cluster is to the points in the neighboring ones.



https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html

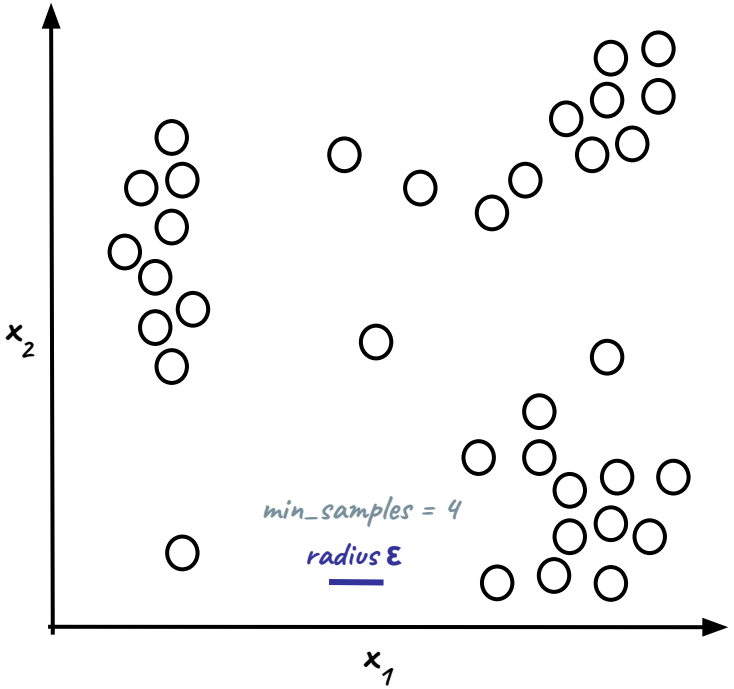
Density-based clustering - DBScan

DBScan stands for **Density-Based Spatial Clustering of Applications with Noise**. The idea is that a cluster is a contiguous region of high examples density, separated from other such clusters by contiguous regions of low example density. The examples in the separating regions of low point density are typically considered outliers.

DBScan stands for **Density-Based Spatial Clustering of Applications with Noise**. The idea is that a cluster is a contiguous region of high examples density, separated from other such clusters by contiguous regions of low example density. The examples in the separating regions of low point density are typically considered outliers.

The main hyperparameters are:

- the maximum **radius ε** between two examples to be considered as in the *neighborhood* of the other.
- the **minimum number of samples** in a *neighborhood* for a point to be considered as a **core point**, including the point itself.

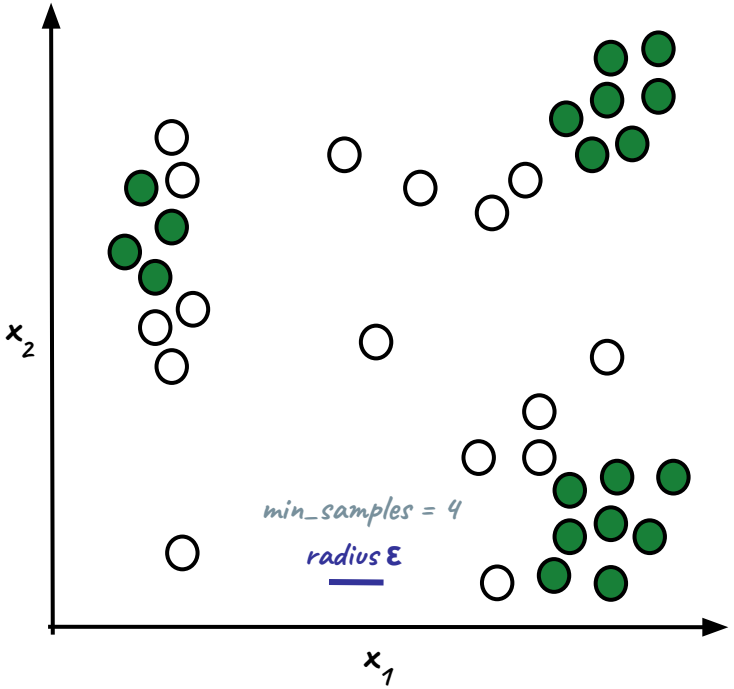


Density-based clustering - DBScan

DBScan stands for **Density-Based Spatial Clustering of Applications with Noise**. The idea is that a cluster is a contiguous region of high examples density, separated from other such clusters by contiguous regions of low example density. The examples in the separating regions of low point density are typically considered outliers.

The main hyperparameters are:

- the maximum **radius ε** between two examples to be considered as in the *neighborhood* of the other.
- the **minimum number of samples** in a *neighborhood* for a point to be considered as a **core point**, including the point itself.



What is a core point? In DBScan, an example is classified as:

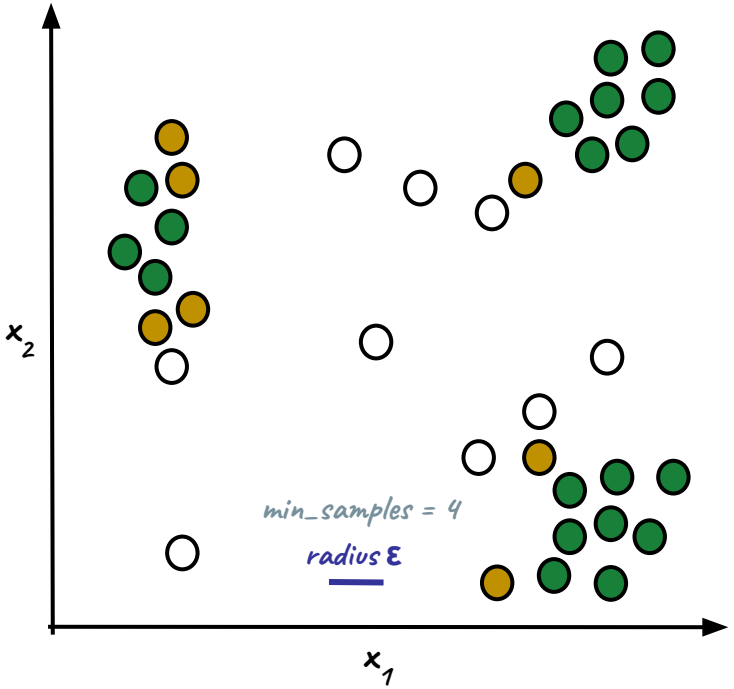
- a **core point**, if at least **min_samples** points are within **ε**

Density-based clustering - DBScan

DBScan stands for **Density-Based Spatial Clustering of Applications with Noise**. The idea is that a cluster is a contiguous region of high examples density, separated from other such clusters by contiguous regions of low example density. The examples in the separating regions of low point density are typically considered outliers.

The main hyperparameters are:

- the maximum **radius ε** between two examples to be considered as in the *neighborhood* of the other.
- the **minimum number of samples** in a *neighborhood* for a point to be considered as a **core point**, including the point itself.



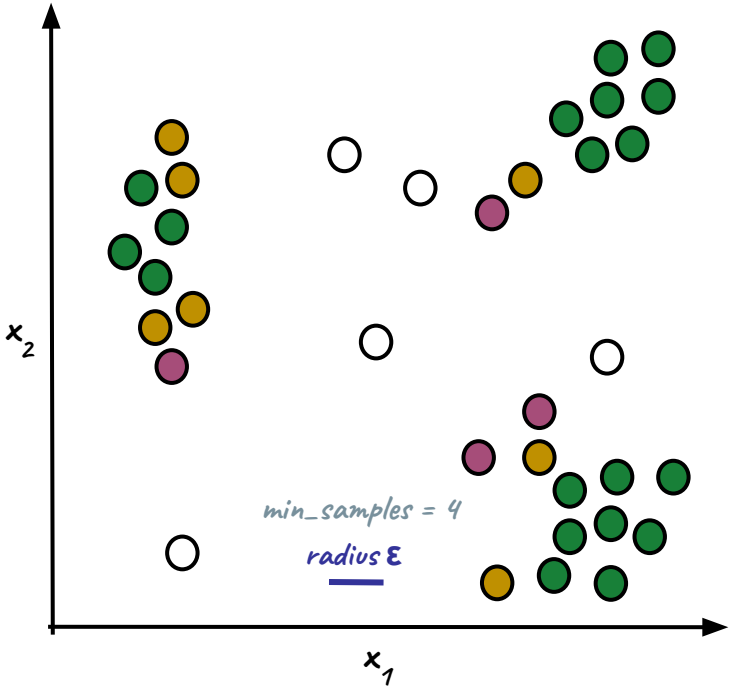
What is a core point? In DBScan, an example is classified as:

- a **core point**, if at least **min_samples** points are within **ε**
- a **directly reachable point**, if it is within a distance **ε** from a core point

Density-based clustering - DBScan

DBScan stands for **Density-Based Spatial Clustering of Applications with Noise**. The idea is that a cluster is a contiguous region of high examples density, separated from other such clusters by contiguous regions of low example density. The examples in the separating regions of low point density are typically considered outliers.
The main hyperparameters are:

- the maximum **radius ε** between two examples to be considered as in the *neighborhood* of the other.
- the **minimum number of samples** in a *neighborhood* for a point to be considered as a **core point**, including the point itself.



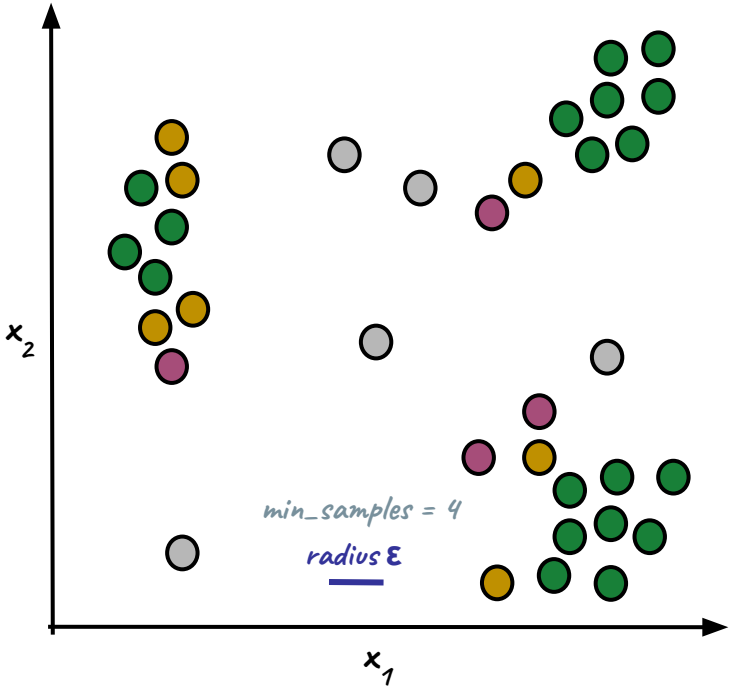
What is a core point? In DBScan, an example is classified as:

- a **core point**, if at least **min_samples** points are within **ε**
- a **directly reachable point**, if it is within a distance **ε** from a core point
- a **reachable point**, if there is a path/chain of **core points** and no more than one last **directly reachable point** within **ε**

Density-based clustering - DBScan

DBScan stands for **Density-Based Spatial Clustering of Applications with Noise**. The idea is that a cluster is a contiguous region of high examples density, separated from other such clusters by contiguous regions of low example density. The examples in the separating regions of low point density are typically considered outliers.
The main hyperparameters are:

- the maximum **radius ε** between two examples to be considered as in the *neighborhood* of the other.
- the **minimum number of samples** in a *neighborhood* for a point to be considered as a **core point**, including the point itself.



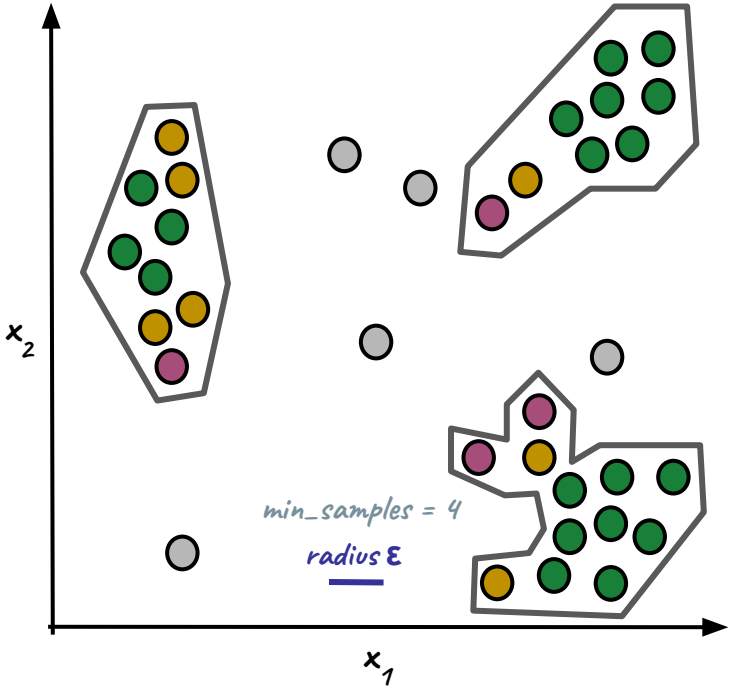
What is a core point? In DBScan, an example is classified as:

- a **core point**, if at least **min_samples** points are within **ε**
- a **directly reachable point**, if it is within a distance **ε** from a core point
- a **reachable point**, if there is a path/chain of **core points** and no more than one last **directly reachable point** within **ε**
- **outliers:** all the unreachable points

Density-based clustering - DBScan

DBScan stands for **Density-Based Spatial Clustering of Applications with Noise**. The idea is that a cluster is a contiguous region of high examples density, separated from other such clusters by contiguous regions of low example density. The examples in the separating regions of low point density are typically considered outliers.
The main hyperparameters are:

- the maximum *radius* ε between two examples to be considered as in the *neighborhood* of the other.
- the *minimum number of samples* in a *neighborhood* for a point to be considered as a *core point*, including the point itself.



What is a core point? In DBScan, an example is classified as:

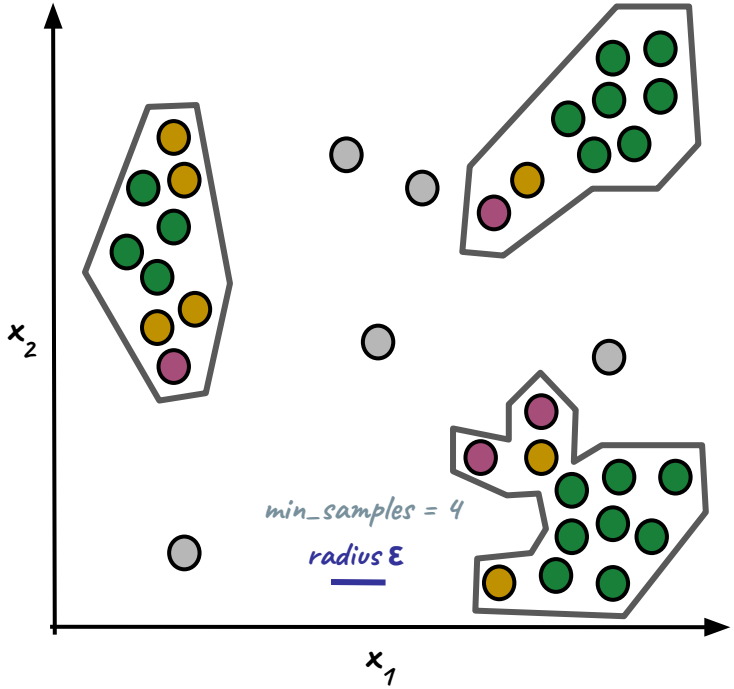
- a *core point*, if at least *min_samples* points are within ε
- a *directly reachable point*, if it is within a distance ε from a core point
- a *reachable point*, if there is a path/chain of *core points* and no more than one last *directly reachable point* within ε
- *outliers:* all the unreachable points

Core points form a cluster together with all the reachable points. *Reachable points* (*directly* or not) form the edge of the cluster, as they cannot be used to reach more than one other point.

Density-based clustering - DBScan

DBScan stands for **Density-Based Spatial Clustering of Applications with Noise**. The idea is that a cluster is a contiguous region of high examples density, separated from other such clusters by contiguous regions of low example density. The examples in the separating regions of low point density are typically considered outliers.
The main hyperparameters are:

- the maximum **radius ε** between two examples to be considered as in the *neighborhood* of the other.
- the *minimum number of samples* in a *neighborhood* for a point to be considered as a **core point**, including the point itself.



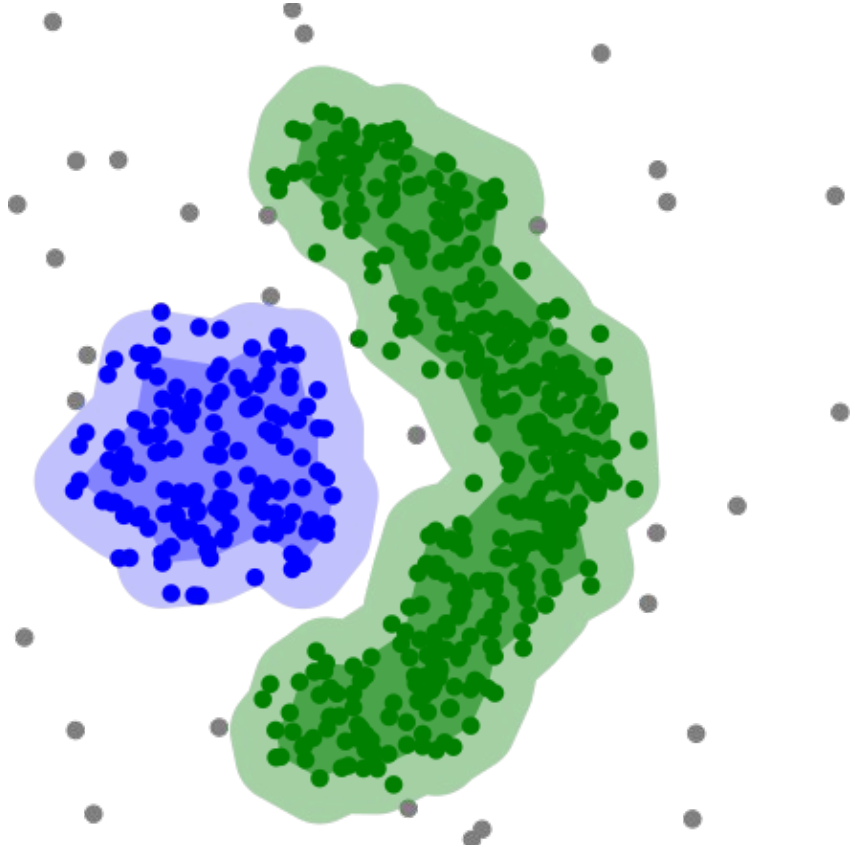
What is a core point? In DBScan, an example is classified as:

- a **core point**, if at least *min_samples* points are within **ε**
- a **directly reachable point**, if it is within a distance **ε** from a core point
- a **reachable point**, if there is a path/chain of **core points** and no more than one last **directly reachable point** within **ε**
- *outliers:* all the unreachable points

Core points form a cluster together with all the reachable points. **Reachable points** (**directly** or not) form the edge of the cluster, as they cannot be used to reach more than one other point.

min_samples primarily controls the algorithm tolerance towards noise (on noisy and large data sets it may be desirable to increase it)
ε must be chosen appropriately, given the data set and distance metric, and usually cannot be left at the default value.

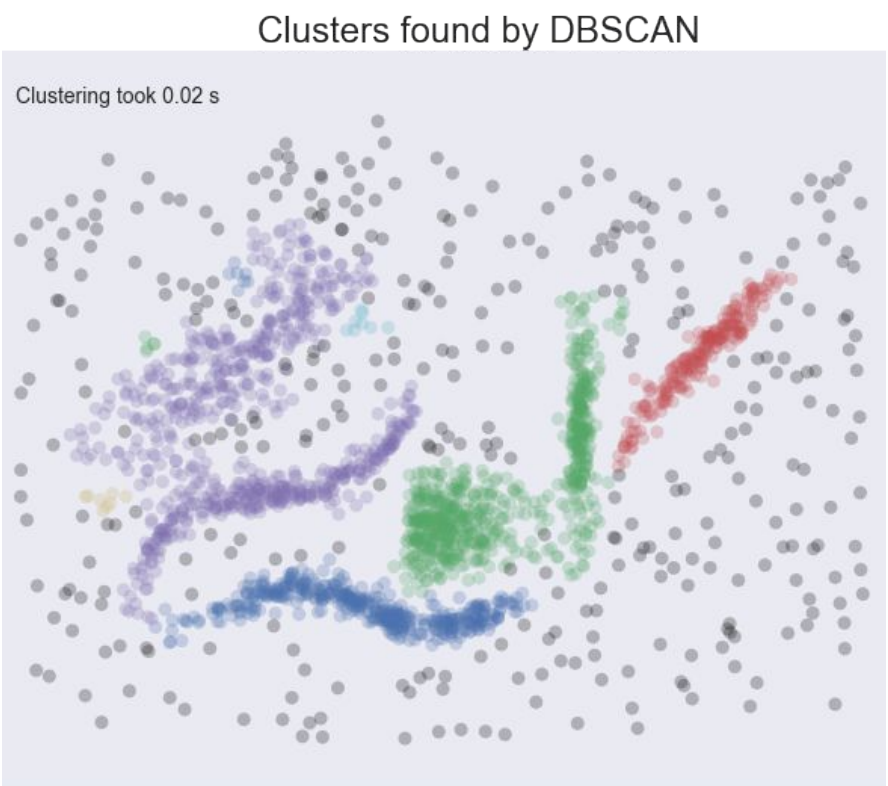
DBScan it's extremely powerful, and does not even come with huge caveats attached:



Advantages:

- The number of clusters is not a model hyperparameter, as in KMeans, but is directly inferred from the algorithm.
- Works with every possible shape, even concentric clusters.
- Identify outliers, and is able to adapt to noisy examples
- It's simple: it works with just two main parameters.

DBScan it's extremely powerful, and does not even come with huge caveats attached:



Advantages:

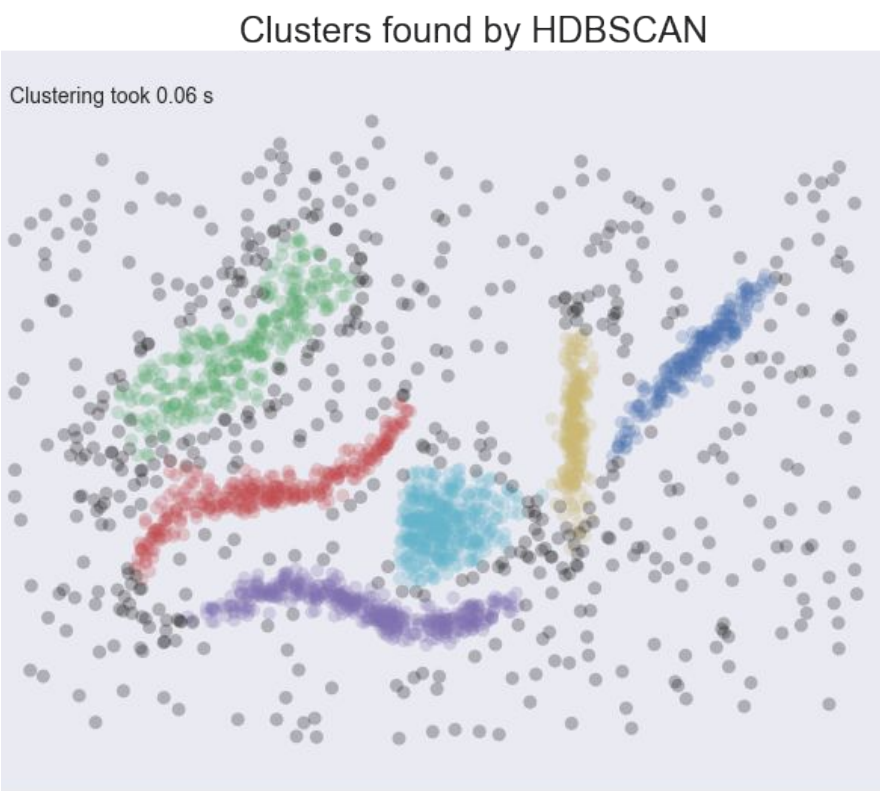
- The number of clusters is not a model hyperparameter, as in KMeans, but is directly inferred from the algorithm.
- Works with every possible shape, even concentric clusters.
- Identify outliers, and is able to adapt to noisy examples
- It's simple: it works with just two main parameters.

Disadvantages:

- Sensible to the chosen distance metric used (and that's normal), and heavily affected by the curse of dimensionality.
- Works fine for not too much dissimilar datasets in terms of example densities, does not work at all for datasets with huge discrepancies in densities.
- Choosing the radius ε is harder than it seems, especially for not well understood datasets.

https://nbviewer.org/github/scikit-learn-contrib/hdbscan/blob/master/notebooks/Comparing%20Clustering%20Algorithms.ipynb

DBScan it's extremely powerful, and does not even come with huge caveats attached:



Advantages:

- The number of clusters is not a model hyperparameter, as in *KMeans*, but is directly inferred from the algorithm.
- Works with every possible shape, even concentric clusters.
- Identify outliers, and is able to adapt to noisy examples
- It's simple: it works with just two main parameters.

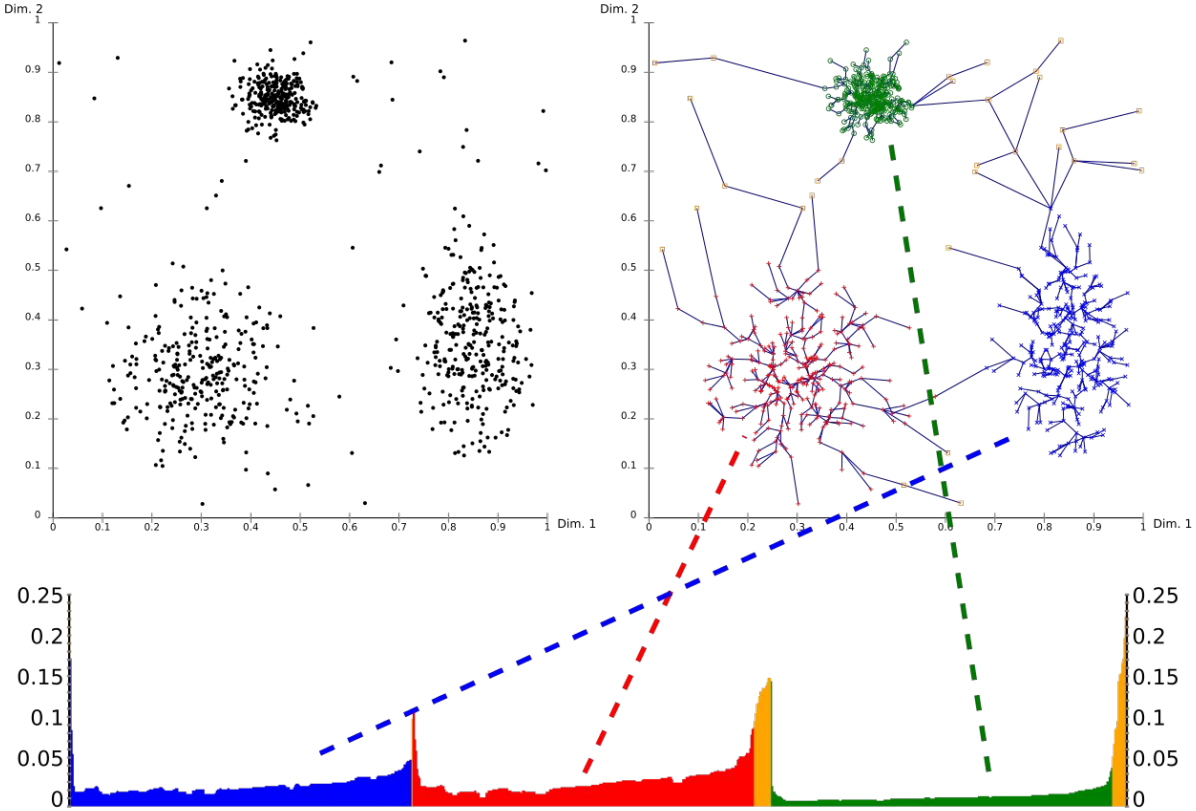
Disadvantages:

- Sensible to the chosen distance metric used (and that's normal), and heavily affected by the curse of dimensionality.
- Works fine for not too much dissimilar datasets in terms of example densities, does not work at all for datasets with huge discrepancies in densities.
- Choosing the radius **ε** is harder than it seems, especially for not well understood datasets.

Hierarchical DBScan: run *DBScan* over varying **ε**, and integrates the result to find the most stable solution.
It returns a good clustering straight away with little or no parameter tuning; the primary parameter, minimum cluster size, is intuitive and easy to select.

https://nbviewer.org/github/scikit-learn-contrib/hdbscan/blob/master/notebooks/Comparing%20Clustering%20Algorithms.ipynb

OPTICS: Ordering points to identify the clustering structure. It relaxes the **ε** requirement from a fixed value to a range of values. Then it builds a *reachability graph*, a 2D plot between the ordered OPTICS processed points and the *reachability distance*, with clusters showing up as valleys. The deeper the valley, the denser the cluster.

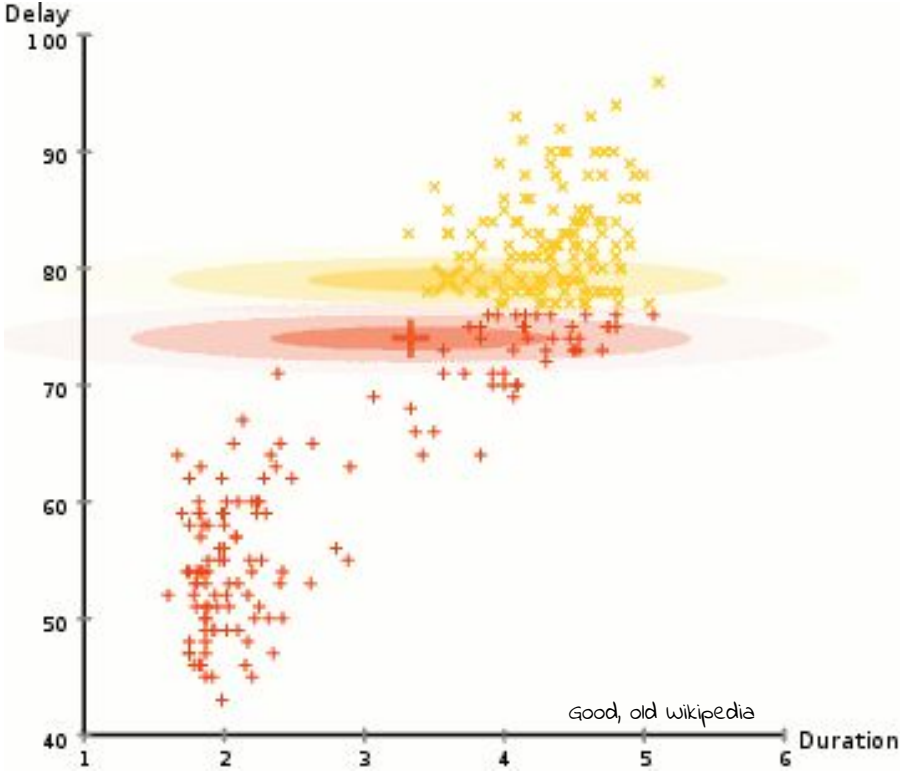


if you want my personal opinion, OPTICS plays in a league of its own, for results and versatility, but you'll see with your eyes in a little while

Distribution-based clustering - Gaussian Mixture Models

If somehow you are confident about the distribution from which your dataset might have been sampled, then *distribution-based clustering* might be the ideal solution for you.

Gaussian Mixture Models (GMM) assumes that the examples are generated from a mixture of a finite number of Gaussian, with unknown parameters. In a sense, they generalize *KMeans* to include the covariance structure of the data, as well as the centers of the latent Gaussians.



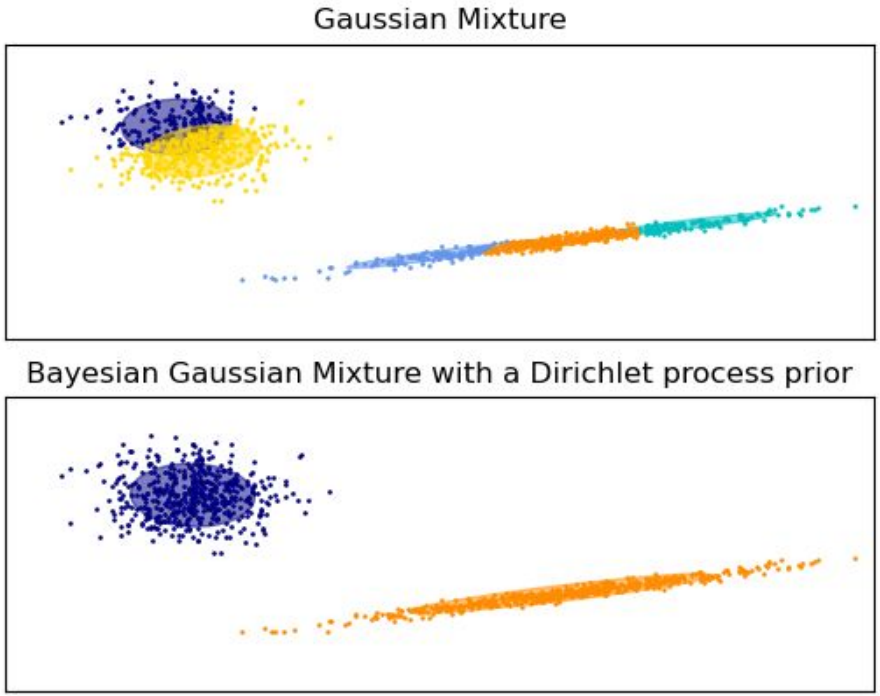
The most commonly used technique to estimate the mixture model's parameters is the ***expectation-maximization*** algorithm.

I have no intention to get deep into the mathematical details; the algorithm in a nutshell works in a iterative fashion:

- start with an initialization step, which assigns model parameters to reasonable values based on the data, or randomly centered on data points (learned from ***KMeans***)
- compute for each point a probability of being generated by each component of the model ***(expectation)***
- maximize the expectations with respect to the model parameters ***(maximization)***
- iterate over the ***expectation*** and ***maximization*** steps until the parameters' estimates converge

Distribution-based clustering - Gaussian Mixture Models

If somehow you are confident about the distribution from which your dataset might have been sampled, then *distribution-based clustering* might be the ideal solution for you.
Gaussian Mixture Models (GMM) assumes that the examples are generated from a mixture of a finite number of Gaussian, with unknown parameters. In a sense, they generalize *KMeans* to include the covariance structure of the data, as well as the centers of the latent Gaussians.



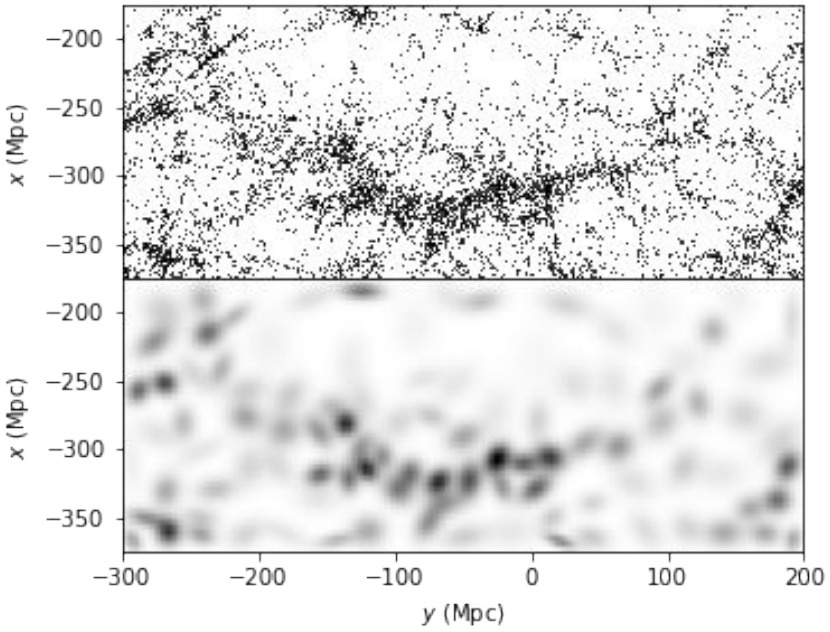
The most commonly used technique to estimate the mixture model's parameters is the ***expectation-maximization*** algorithm.
I have no intention to get deep into the mathematical details; the algorithm in a nutshell works in a iterative fashion:

- start with an initialization step, which assigns model parameters to reasonable values based on the data, or randomly centered on data points (learned from ***KMeans***)
- compute for each point a probability of being generated by each component of the model ***(expectation)***
- maximize the expectations with respect to the model parameters ***(maximization)***
- iterate over the ***expectation*** and ***maximization*** steps until the parameters' estimates converge

Variational Bayesian Gaussian Mixture add a form of regularization to the algorithm by integrating information coming from parameters' prior distributions.

Distribution-based clustering - Gaussian Mixture Models

If somehow you are confident about the distribution from which your dataset might have been sampled, then *distribution-based clustering* might be the ideal solution for you.
Gaussian Mixture Models (GMM) assumes that the examples are generated from a mixture of a finite number of Gaussian, with unknown parameters. In a sense, they generalize *KMeans* to include the covariance structure of the data, as well as the centers of the latent Gaussians.

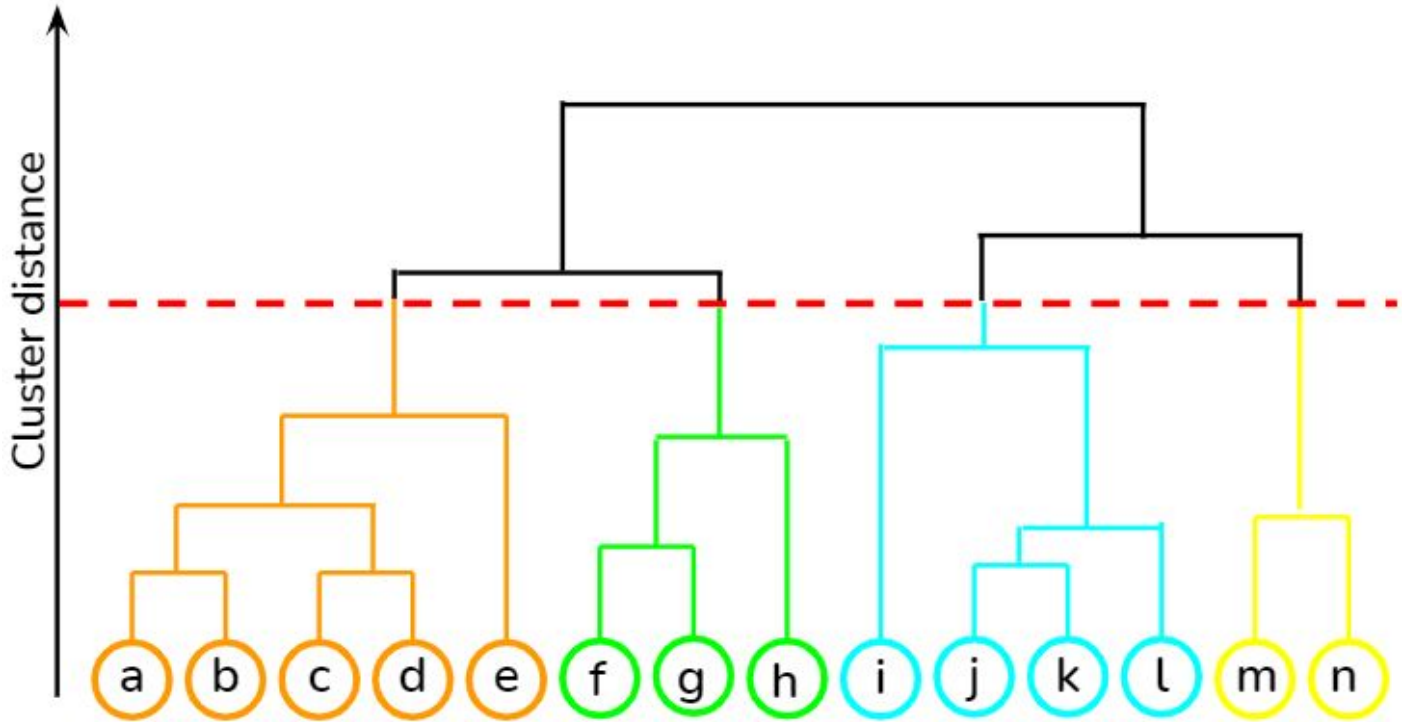


The most commonly used technique to estimate the mixture model's parameters is the ***expectation-maximization*** algorithm.
I have no intention to get deep into the mathematical details; the algorithm in a nutshell works in a iterative fashion:

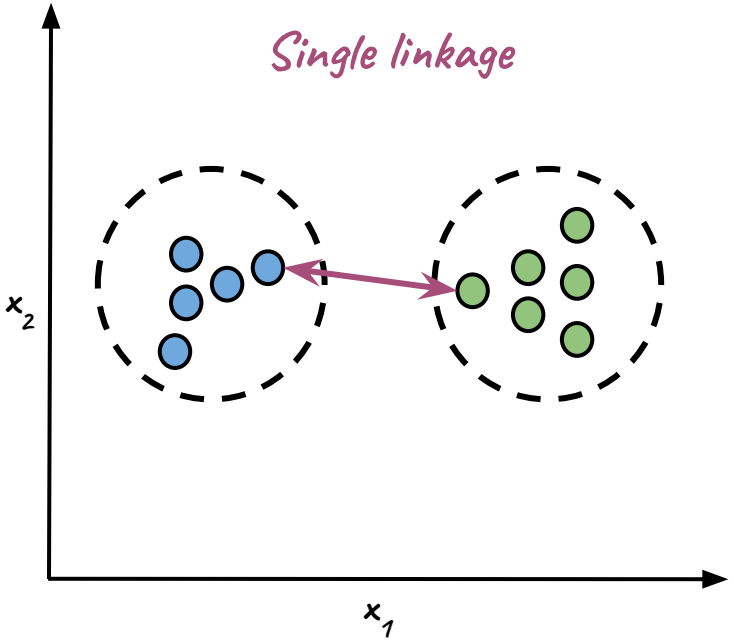
- start with an initialization step, which assigns model parameters to reasonable values based on the data, or randomly centered on data points (learned from ***KMeans***)
- compute for each point a probability of being generated by each component of the model ***(expectation)***
- maximize the expectations with respect to the model parameters ***(maximization)***
- iterate over the ***expectation*** and ***maximization*** steps until the parameters' estimates converge

Variational Bayesian Gaussian Mixture add a form of regularization to the algorithm by integrating information coming from parameters' prior distributions.

Hierarchical clustering builds *nested* clusters by merging (or splitting) subclusters successively. This hierarchy is represented as a tree (*"dendrogram"*), whose root is the unique cluster that gathers all the samples, and the leaves being the clusters with only one sample.
There are two main approaches: bottom-up (*Agglomerative Clustering*) and top-down (*Divisive Clustering*), with the former being the most widely used approach for clustering.



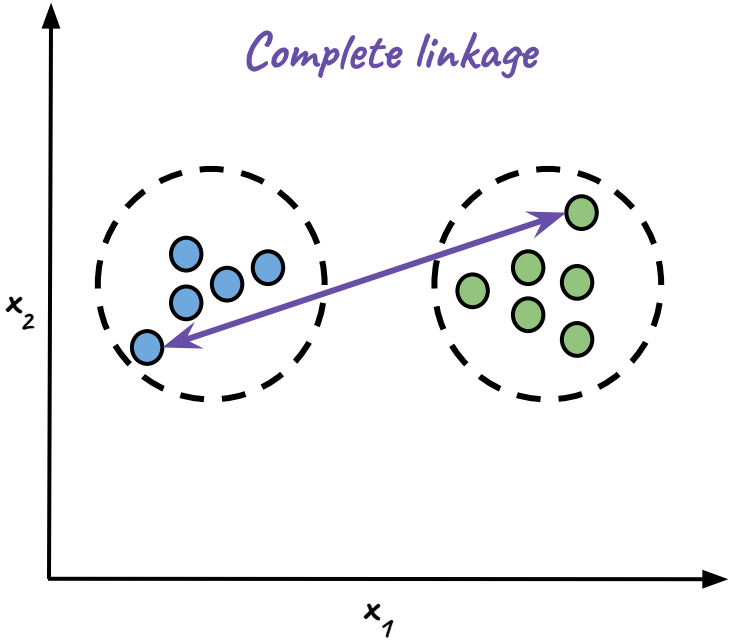
Hierarchical clustering builds *nested* clusters by merging (or splitting) subclusters successively. This hierarchy is represented as a tree (*"dendrogram"*), whose root is the unique cluster that gathers all the samples, and the leaves being the clusters with only one sample.
There are two main approaches: bottom-up (*Agglomerative Clustering*) and top-down (*Divisive Clustering*), with the former being the most widely used approach for clustering.



With *Agglomerative Clustering* each example starts in its own cluster, and clusters are successively merged together, by iteratively joining the two closest data points/clusters, until one big cluster is formed
The merging sequence is chosen from the similarity between examples/clusters. This is measured as a distance, and the merge strategy is determined via a ***linkage*** criterion:

- *Single linkage:* the distance between clusters is defined by the distance between their closest members.

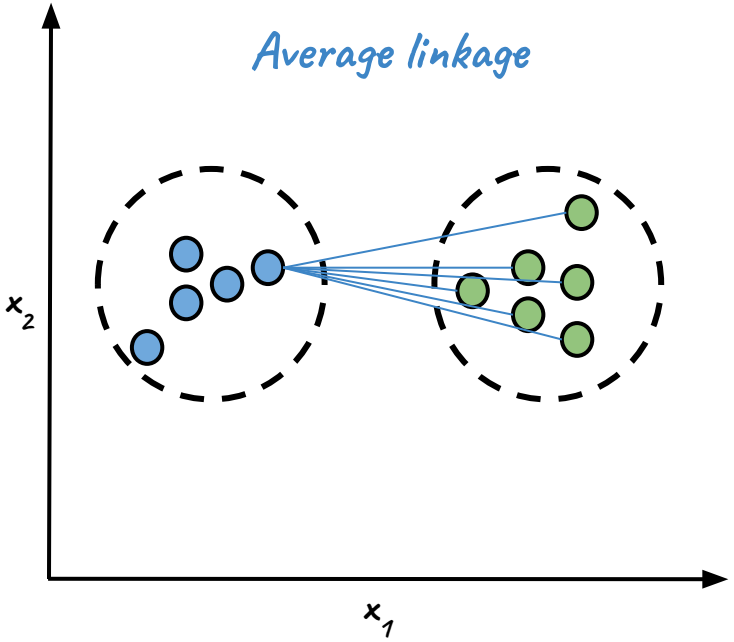
Hierarchical clustering builds *nested* clusters by merging (or splitting) subclusters successively. This hierarchy is represented as a tree (*"dendrogram"*), whose root is the unique cluster that gathers all the samples, and the leaves being the clusters with only one sample.
There are two main approaches: bottom-up (*Agglomerative Clustering*) and top-down (*Divisive Clustering*), with the former being the most widely used approach for clustering.



With *Agglomerative Clustering* each example starts in its own cluster, and clusters are successively merged together, by iteratively joining the two closest data points/clusters, until one big cluster is formed
The merging sequence is chosen from the similarity between examples/clusters. This is measured as a distance, and the merge strategy is determined via a ***linkage*** criterion:

- *Single linkage:* the distance between clusters is defined by the distance between their closest members.
- *Complete linkage:* the distance between clusters is defined by the distance between their furthest members..

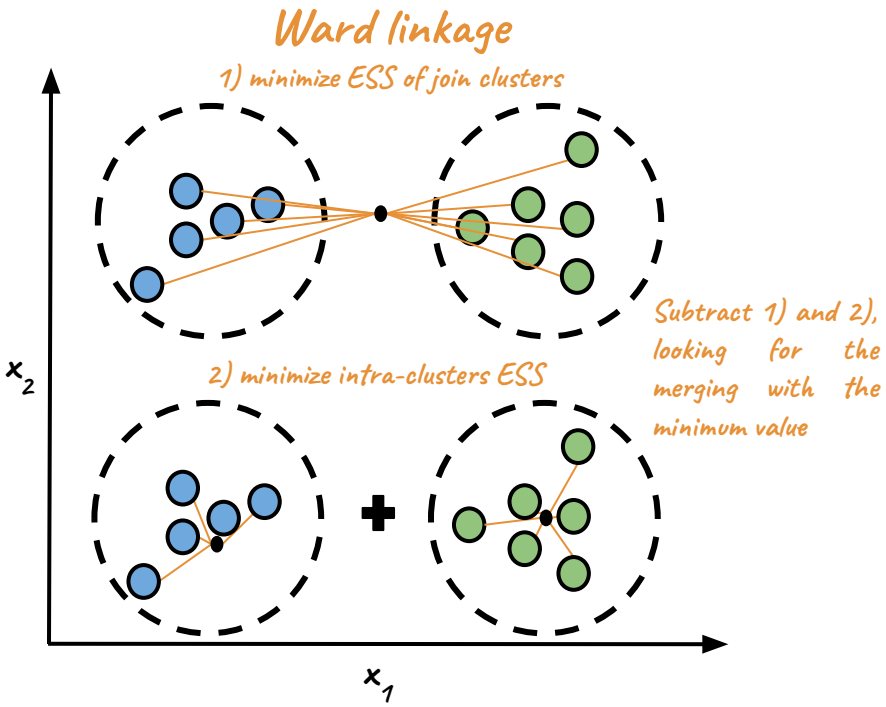
Hierarchical clustering builds *nested* clusters by merging (or splitting) subclusters successively. This hierarchy is represented as a tree (*"dendrogram"*), whose root is the unique cluster that gathers all the samples, and the leaves being the clusters with only one sample.
There are two main approaches: bottom-up (*Agglomerative Clustering*) and top-down (*Divisive Clustering*), with the former being the most widely used approach for clustering.



With *Agglomerative Clustering* each example starts in its own cluster, and clusters are successively merged together, by iteratively joining the two closest data points/clusters, until one big cluster is formed
The merging sequence is chosen from the similarity between examples/clusters. This is measured as a distance, and the merge strategy is determined via a ***linkage*** criterion:

- *Single linkage:* the distance between clusters is defined by the distance between their closest members.
- *Complete linkage:* the distance between clusters is defined by the distance between their furthest members..
- *Average linkage:* minimizes the average of the distances between all observations of pairs of clusters.

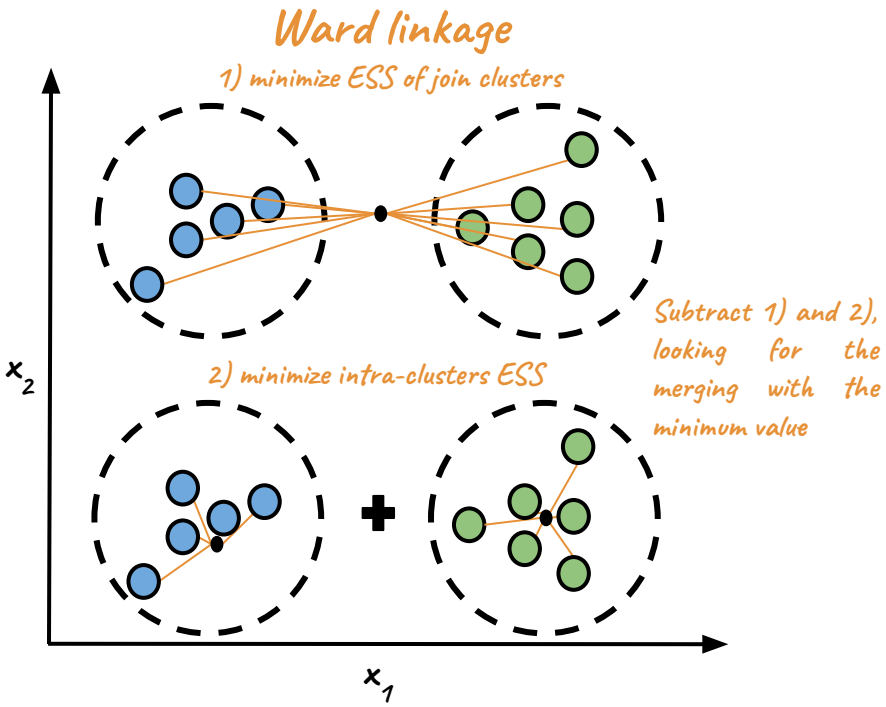
Hierarchical clustering builds *nested* clusters by merging (or splitting) subclusters successively. This hierarchy is represented as a tree (*"dendrogram"*), whose root is the unique cluster that gathers all the samples, and the leaves being the clusters with only one sample.
There are two main approaches: bottom-up (*Agglomerative Clustering*) and top-down (*Divisive Clustering*), with the former being the most widely used approach for clustering.



With *Agglomerative Clustering* each example starts in its own cluster, and clusters are successively merged together, by iteratively joining the two closest data points/clusters, until one big cluster is formed
The merging sequence is chosen from the similarity between examples/clusters. This is measured as a distance, and the merge strategy is determined via a ***linkage*** criterion:

- *Single linkage:* the distance between clusters is defined by the distance between their closest members.
- *Complete linkage:* the distance between clusters is defined by the distance between their furthest members..
- *Average linkage:* minimizes the average of the distances between all observations of pairs of clusters.
- *Ward linkage:* combines cluster where increase in within cluster variance is to the smallest degree. It computes the sum of squares error (ESS), and successively chooses the next clusters based on the smallest one.

Hierarchical clustering builds *nested* clusters by merging (or splitting) subclusters successively. This hierarchy is represented as a tree (*"dendrogram"*), whose root is the unique cluster that gathers all the samples, and the leaves being the clusters with only one sample.
There are two main approaches: bottom-up (*Agglomerative Clustering*) and top-down (*Divisive Clustering*), with the former being the most widely used approach for clustering.

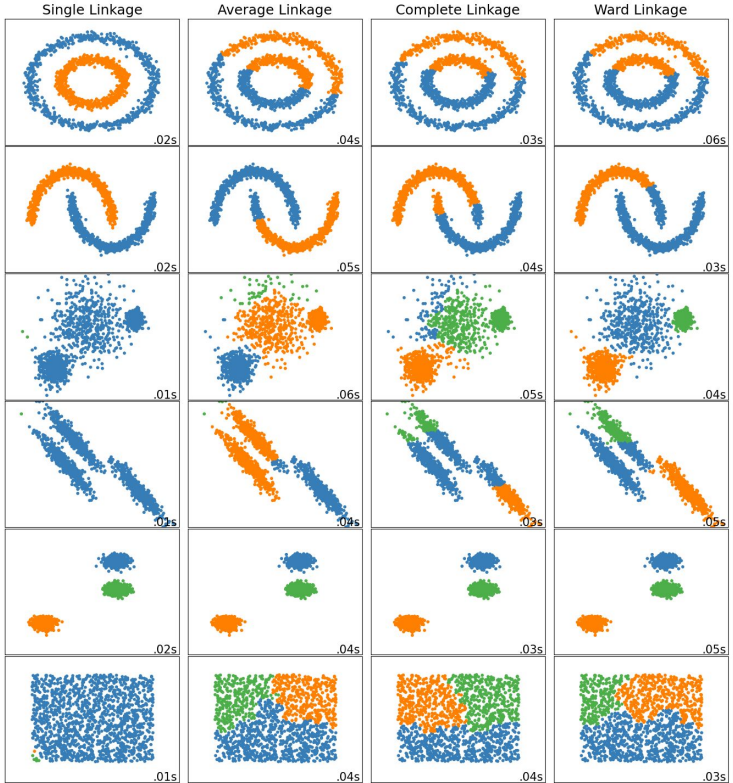


With *Agglomerative Clustering* each example starts in its own cluster, and clusters are successively merged together, by iteratively joining the two closest data points/clusters, until one big cluster is formed
The merging sequence is chosen from the similarity between examples/clusters. This is measured as a distance, and the merge strategy is determined via a ***linkage*** criterion:

- *Single linkage:* the distance between clusters is defined by the distance between their closest members.
- *Complete linkage:* the distance between clusters is defined by the distance between their furthest members..
- *Average linkage:* minimizes the average of the distances between all observations of pairs of clusters.
- *Ward linkage:* combines cluster where increase in within cluster variance is to the smallest degree. It computes the sum of squares error (ESS), and successively chooses the next clusters based on the smallest one.

Divisive Clustering is all of the above, but backwards.

Hierarchical clustering builds *nested* clusters by merging (or splitting) subclusters successively. This hierarchy is represented as a tree (*"dendrogram"*), whose root is the unique cluster that gathers all the samples, and the leaves being the clusters with only one sample.
There are two main approaches: bottom-up (*Agglomerative Clustering*) and top-down (*Divisive Clustering*), with the former being the most widely used approach for clustering.

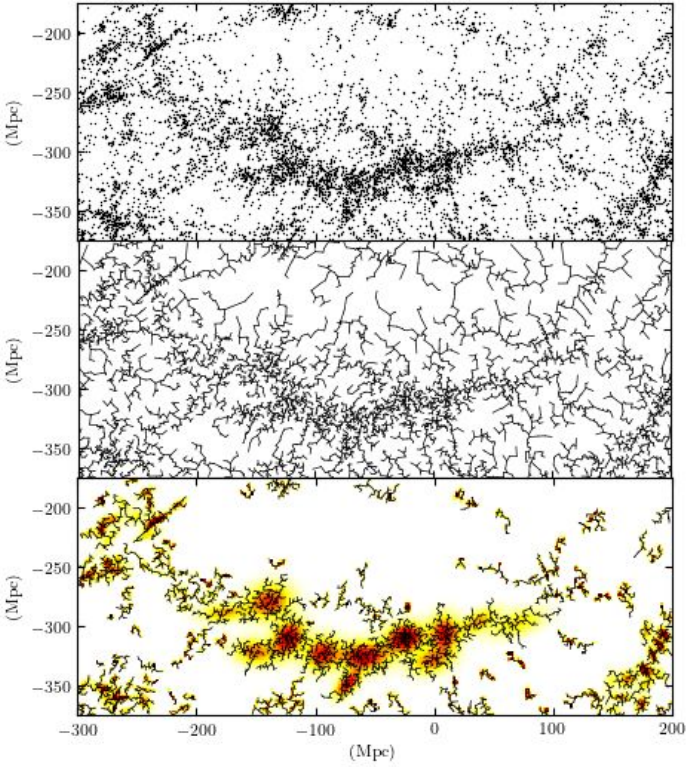


With *Agglomerative Clustering* each example starts in its own cluster, and clusters are successively merged together, by iteratively joining the two closest data points/clusters, until one big cluster is formed.
The merging sequence is chosen from the similarity between examples/clusters. This is measured as a distance, and the merge strategy is determined via a ***linkage*** criterion:

- *Single linkage:* the distance between clusters is defined by the distance between their closest members.
- *Complete linkage:* the distance between clusters is defined by the distance between their furthest members..
- *Average linkage:* minimizes the average of the distances between all observations of pairs of clusters.
- *Ward linkage:* combines cluster where increase in within cluster variance is to the smallest degree. It computes the sum of squares error (ESS), and successively chooses the next clusters based on the smallest one.

Divisive Clustering is all of the above, but backwards.

Hierarchical clustering builds *nested* clusters by merging (or splitting) subclusters successively. This hierarchy is represented as a tree (*"dendrogram"*), whose root is the unique cluster that gathers all the samples, and the leaves being the clusters with only one sample.
There are two main approaches: bottom-up (*Agglomerative Clustering*) and top-down (*Divisive Clustering*), with the former being the most widely used approach for clustering.

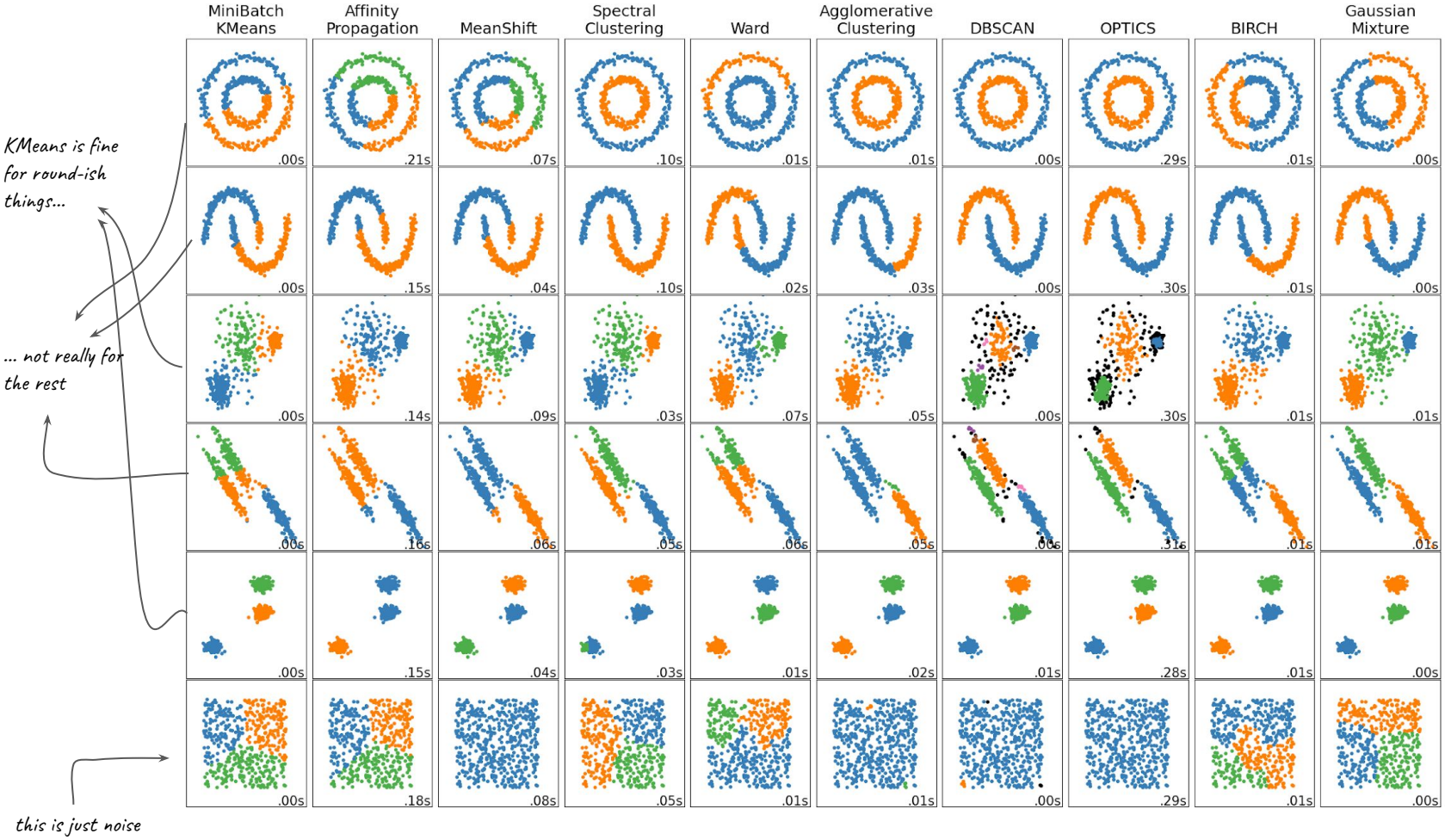


With *Agglomerative Clustering* each example starts in its own cluster, and clusters are successively merged together, by iteratively joining the two closest data points/clusters, until one big cluster is formed.
The merging sequence is chosen from the similarity between examples/clusters. This is measured as a distance, and the merge strategy is determined via a ***linkage*** criterion:

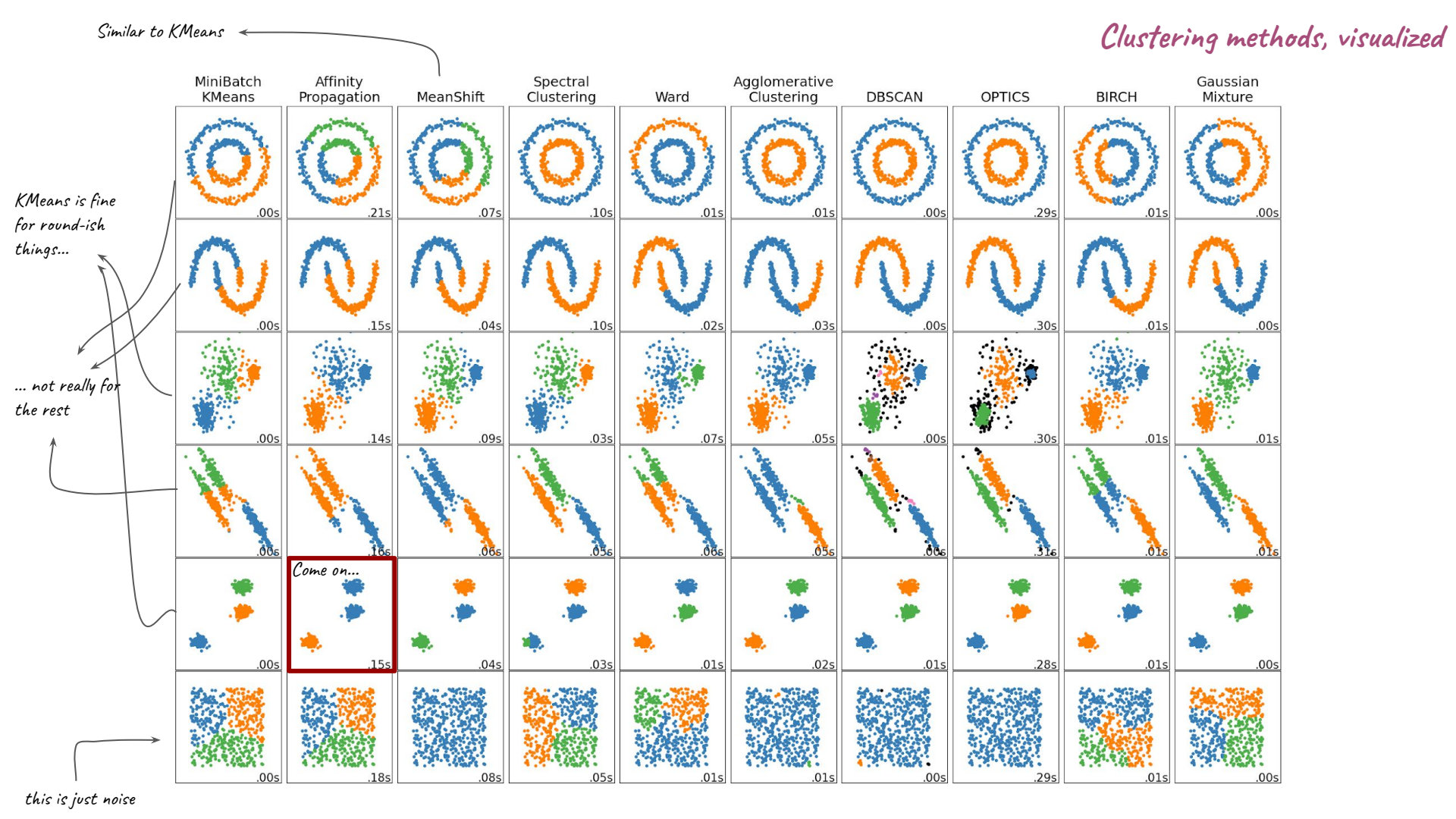
- *Single linkage:* the distance between clusters is defined by the distance between their closest members.
- *Complete linkage:* the distance between clusters is defined by the distance between their furthest members..
- *Average linkage:* minimizes the average of the distances between all observations of pairs of clusters.
- *Ward linkage:* combines cluster where increase in within cluster variance is to the smallest degree. It computes the sum of squares error (ESS), and successively chooses the next clusters based on the smallest one.

Divisive Clustering is all of the above, but backwards.

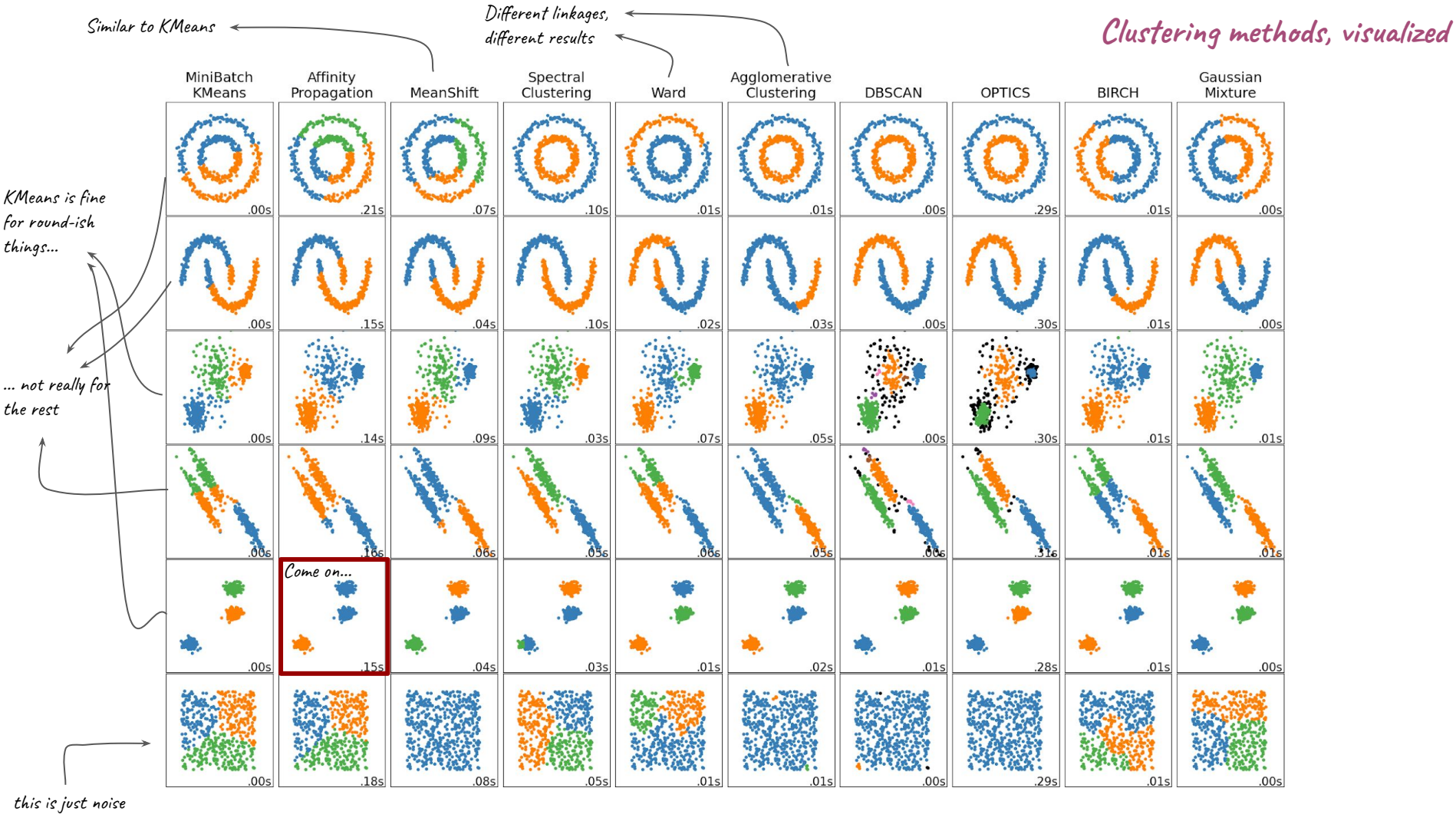
Clustering methods, visualized



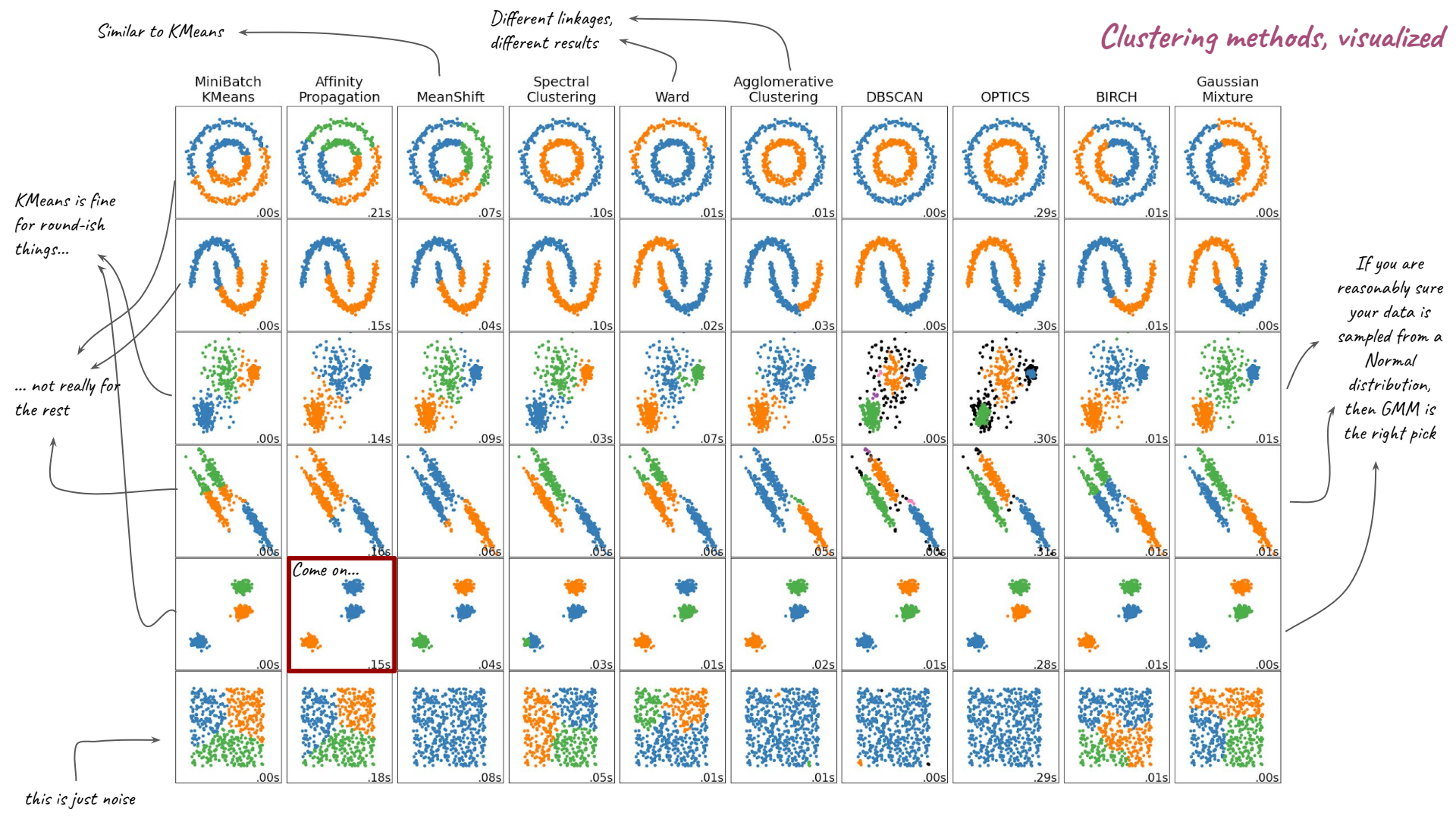
Clustering methods, visualized



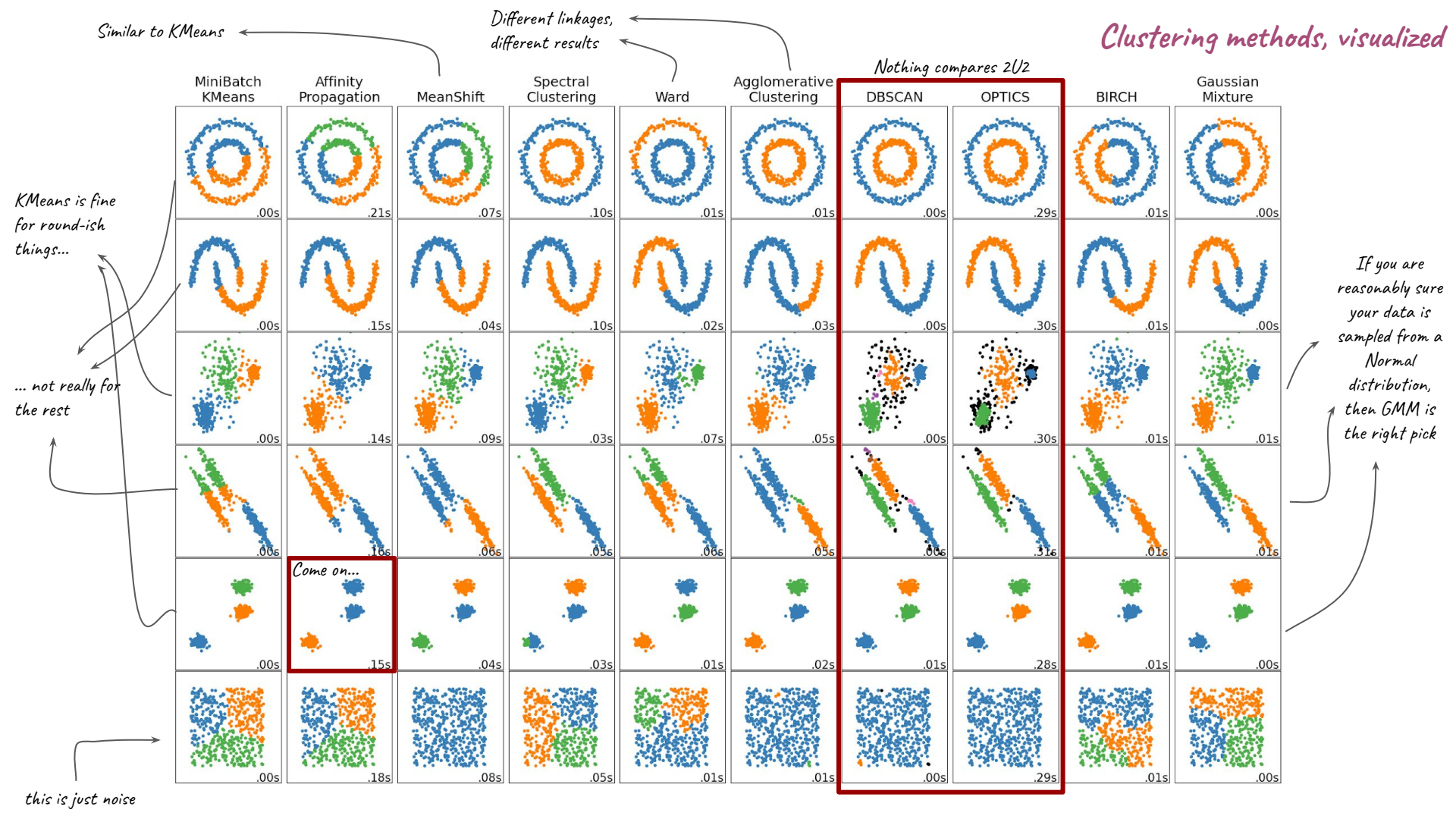
Clustering methods, visualized



Clustering methods, visualized



Clustering methods, visualized

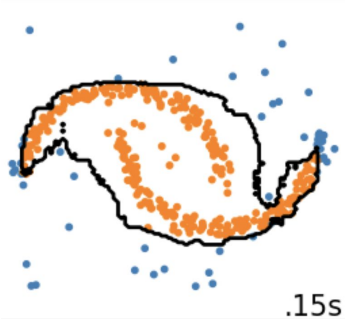


There are three main categories of algorithms in unsupervised learning.

Anomaly Detection

Automatically identify if a (new) observation is an anomaly, and/or clean datasets accordingly.

e.g. density estimation, isolation forest, Local Outlier Factor, SVMs...



Anomaly Detection

An **anomaly detection** algorithm is able to look at an unlabeled dataset, and automatically identify *novelties/outliers**, and raise a warning or clean datasets accordingly, or select them for future analysis. In a nutshell, it discriminates between the bulk of the data distribution - whatever the functional form it is - and what is not part of that bulk.

Yes, **DBScan** is *also* an **anomaly detection** algorithm, if you consider the outliers as anomalies.

Yes, also **GMM** could be seen as an **anomaly detection** algorithm, if you cut everything over a threshold.

I told you, the boundaries between classes and algorithms are not fixed, at the contrary, these are easily crossed.

Anomaly Detection

An **anomaly detection** algorithm is able to look at an unlabeled dataset, and automatically identify *novelties/outliers**, and raise a warning or clean datasets accordingly, or select them for future analysis. In a nutshell, it discriminates between the bulk of the data distribution - whatever the functional form it is - and what is not part of that bulk.
Yes, **DBScan** is *also* an **anomaly detection** algorithm, if you consider the outliers as anomalies.
Yes, also **GMM** could be seen as an **anomaly detection** algorithm, if you cut everything over a threshold.
I told you, the boundaries between classes and algorithms are not fixed, at the contrary, these are easily crossed.

outlier

observations far from the others in the "training" dataset

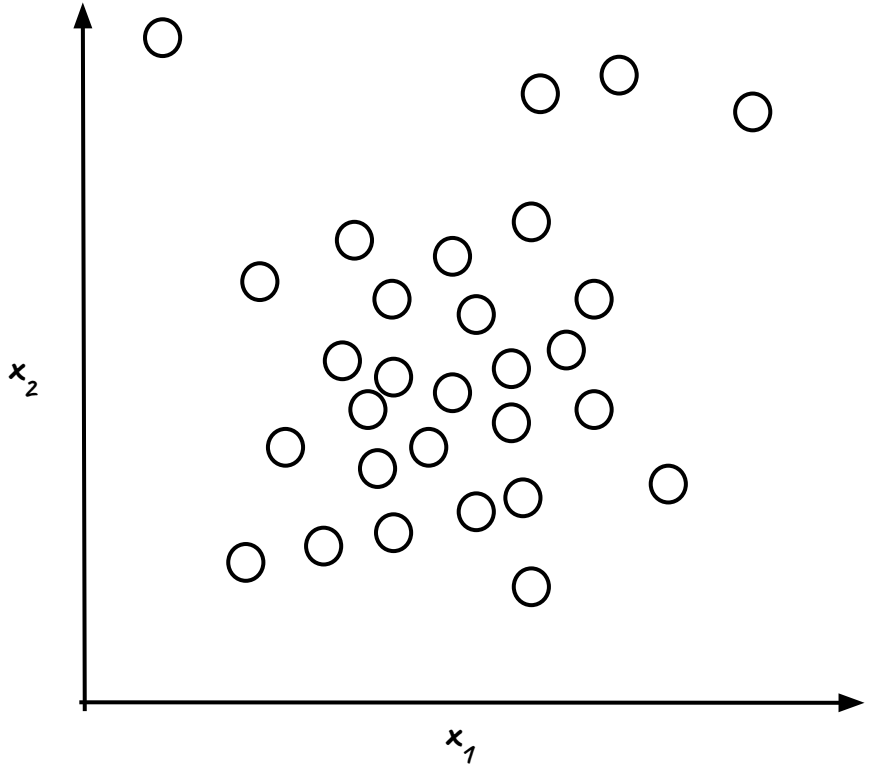
→ "unsupervised anomaly detection"

novelty

we want to know if a NEW observation is an outlier, with respect to a "training" dataset that is NOT polluted with outliers

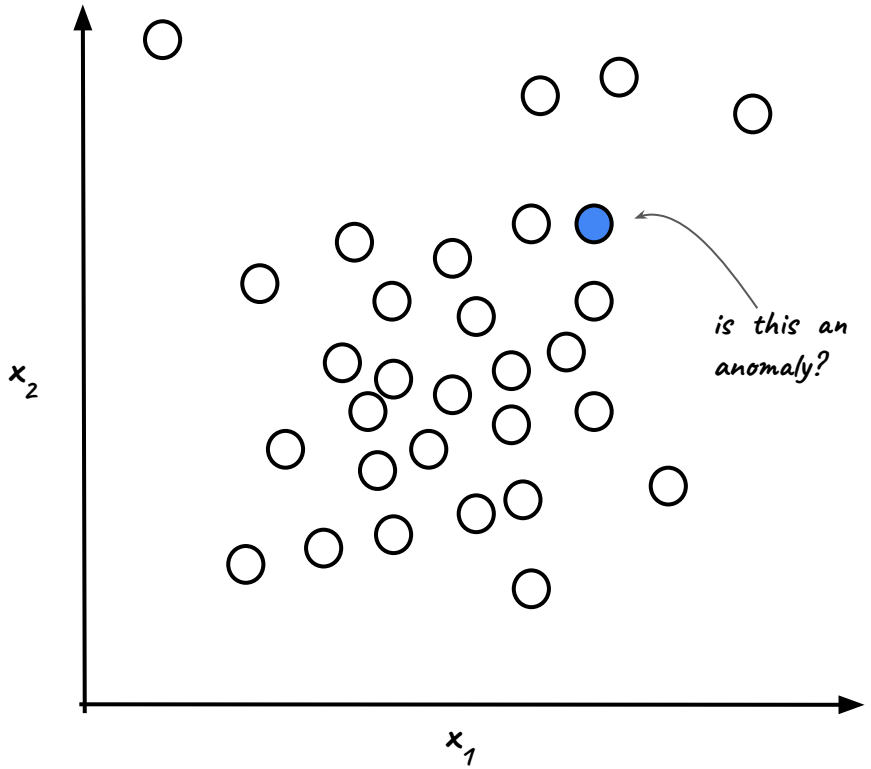
→ "semi-supervised anomaly detection"

An *anomaly detection* algorithm is able to look at an unlabeled dataset, and automatically identify *novelties/outliers*, and raise a warning or clean datasets accordingly, or select them for future analysis. In a nutshell, it discriminates between the bulk of the data distribution - whatever the functional form it is - and what is not part of that bulk.



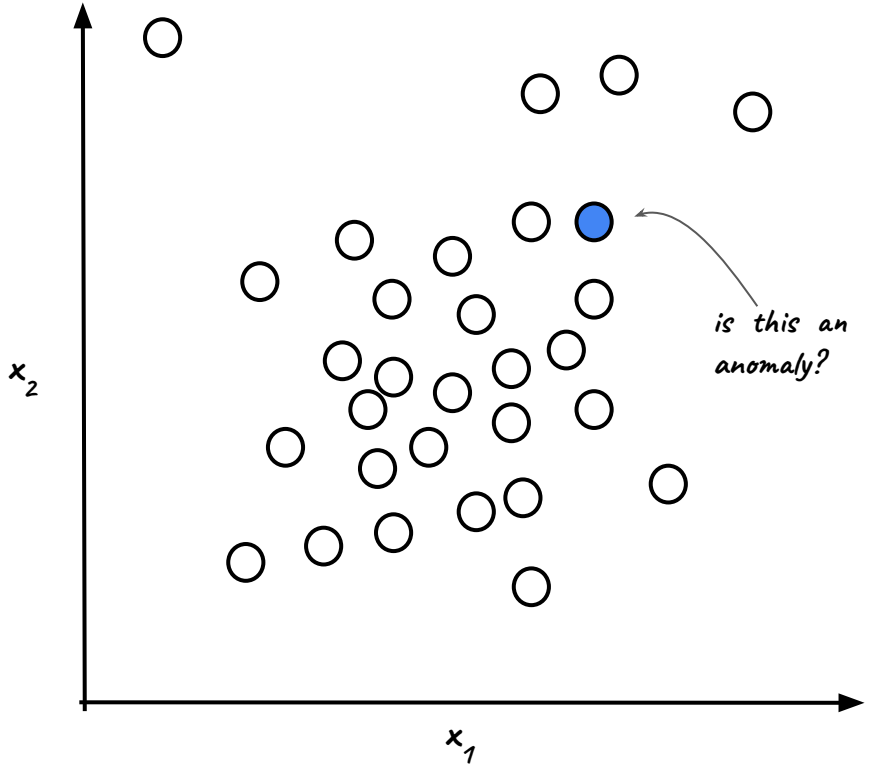
Let's take this simple example, with a dataset $\{x^{(1)}, ..., x^{(m)}\}$

An **anomaly detection** algorithm is able to look at an unlabeled dataset, and automatically identify *novelties/outliers*, and raise a warning or clean datasets accordingly, or select them for future analysis. In a nutshell, it discriminates between the bulk of the data distribution - whatever the functional form it is - and what is not part of that bulk.



Let's take this simple example, with a dataset $\{x^{(1)}, \ldots, x^{(m)}\}$
Now, a *new* test example x_{test} arrives.
What I want is an algorithm that automatically tells me if the new x_{test} example is similar or different from the bulk of the previous distribution → is an anomaly or not.

An **anomaly detection** algorithm is able to look at an unlabeled dataset, and automatically identify *novelties/outliers*, and raise a warning or clean datasets accordingly, or select them for future analysis. In a nutshell, it discriminates between the bulk of the data distribution - whatever the functional form it is - and what is not part of that bulk.

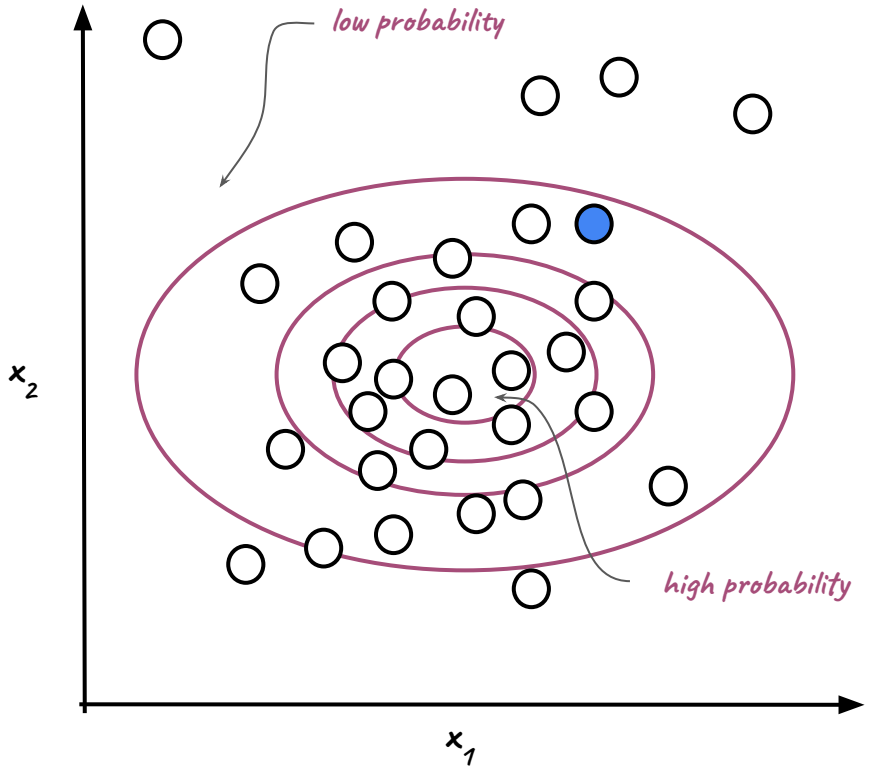


Let's take this simple example, with a dataset $\{x^{(1)}, \ldots, x^{(m)}\}$
Now, a *new* test example x_{test} arrives.
What I want is an algorithm that automatically tells me if the new x_{test} example is similar or different from the bulk of the previous distribution → is an anomaly or not.

The most common way to do anomaly detection is through a technique called **density estimation**. It works this way: first, you build a $p(x)$ model for the probability of x in the feature space, then when a new x_{test} arrives:

- if $p(x_{test}) < \varepsilon$ → flag x_{test} anomaly
- if $p(x_{test}) > \varepsilon$ → it looks fine

An *anomaly detection* algorithm is able to look at an unlabeled dataset, and automatically identify *novelties/outliers*, and raise a warning or clean datasets accordingly, or select them for future analysis. In a nutshell, it discriminates between the bulk of the data distribution - whatever the functional form it is - and what is not part of that bulk.



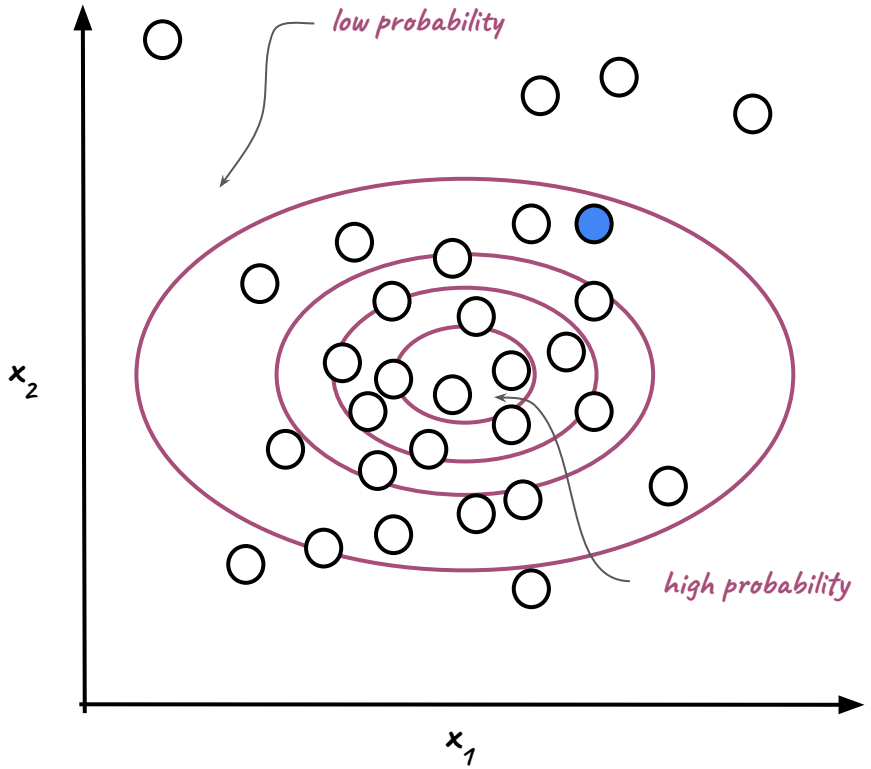
Let's take this simple example, with a dataset $\{x^{(1)}, \ldots, x^{(m)}\}$
Now, a *new* test example x_{test} arrives.
What I want is an algorithm that automatically tells me if the new x_{test} example is similar or different from the bulk of the previous distribution → is an anomaly or not.

The most common way to do anomaly detection is through a technique called *density estimation*. It works this way: first, you build a $p(x)$ model for the probability of x in the feature space, then when a new x_{test} arrives:

- if $p(x_{test}) < \varepsilon$ → flag x_{test} anomaly
- if $p(x_{test}) > \varepsilon$ → it looks fine

So, in this case, the answer is …

An *anomaly detection* algorithm is able to look at an unlabeled dataset, and automatically identify *novelties/outliers*, and raise a warning or clean datasets accordingly, or select them for future analysis. In a nutshell, it discriminates between the bulk of the data distribution - whatever the functional form it is - and what is not part of that bulk.



Let's take this simple example, with a dataset $\{x^{(1)}, \ldots, x^{(m)}\}$
Now, a *new* test example x_{test} arrives.
What I want is an algorithm that automatically tells me if the new x_{test} example is similar or different from the bulk of the previous distribution → is an anomaly or not.

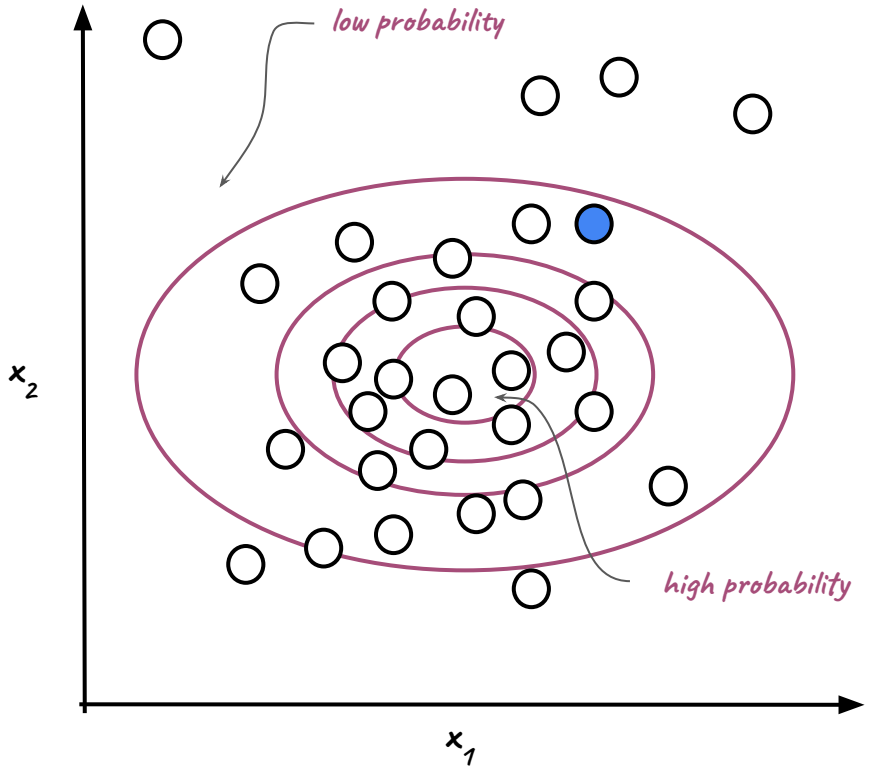
The most common way to do anomaly detection is through a technique called *density estimation*. It works this way: first, you build a $p(x)$ model for the probability of x in the feature space, then when a new x_{test} arrives:

- if $p(x_{test}) < \varepsilon$ → flag x_{test} anomaly
- if $p(x_{test}) > \varepsilon$ → it looks fine

So, in this case, the answer is … it depends the ε threshold.

How do you measure $p(x)$?

An *anomaly detection* algorithm is able to look at an unlabeled dataset, and automatically identify *novelties/outliers*, and raise a warning or clean datasets accordingly, or select them for future analysis. In a nutshell, it discriminates between the bulk of the data distribution - whatever the functional form it is - and what is not part of that bulk.



Given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$, and assuming that each feature is distributed normally with $x_i \sim N(\mu_i, \sigma_i^2)$, the probability $p(x)$ is computed as:

$$p(x) = p(x_1; \mu_1, \sigma_1^2)p(x_2; \mu_2, \sigma_2^2)\ldots p(x_n; \mu_n, \sigma_n^2)$$

$$= \prod_{j=1}^{N} p(x_j; \mu_j, \sigma_j)$$

which in practice is a multivariate Gaussian

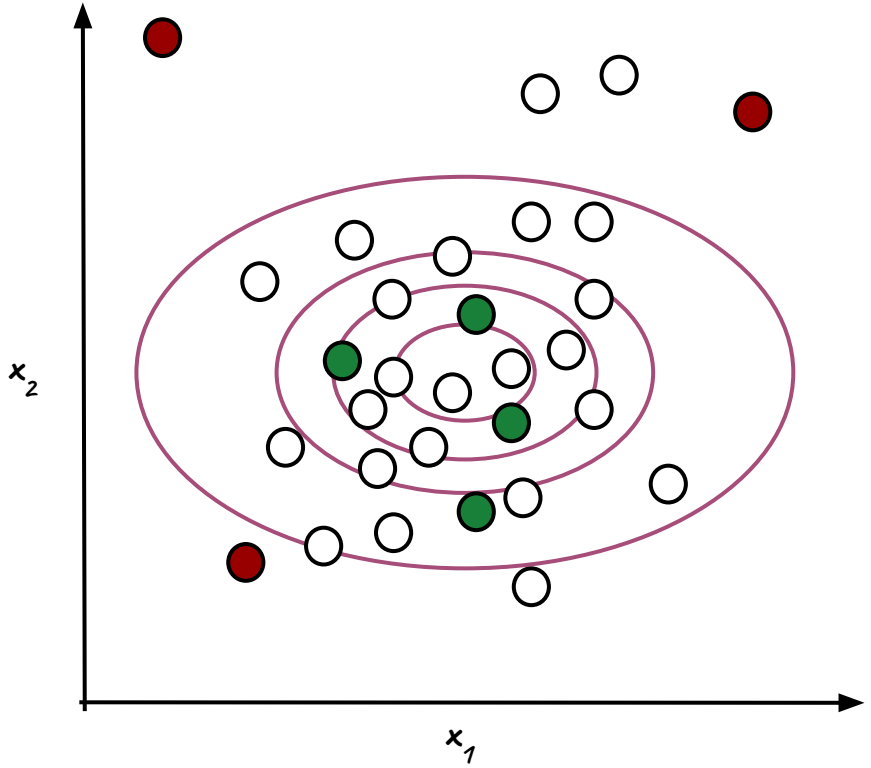
So, in order to identify anomalies, you just fit the parameters:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)} \qquad \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$

modelling every feature as Normally distributed.
Given a new example x_{test}, compute $p(x_{test})$, and decide based on a threshold if it is an anomaly or not.

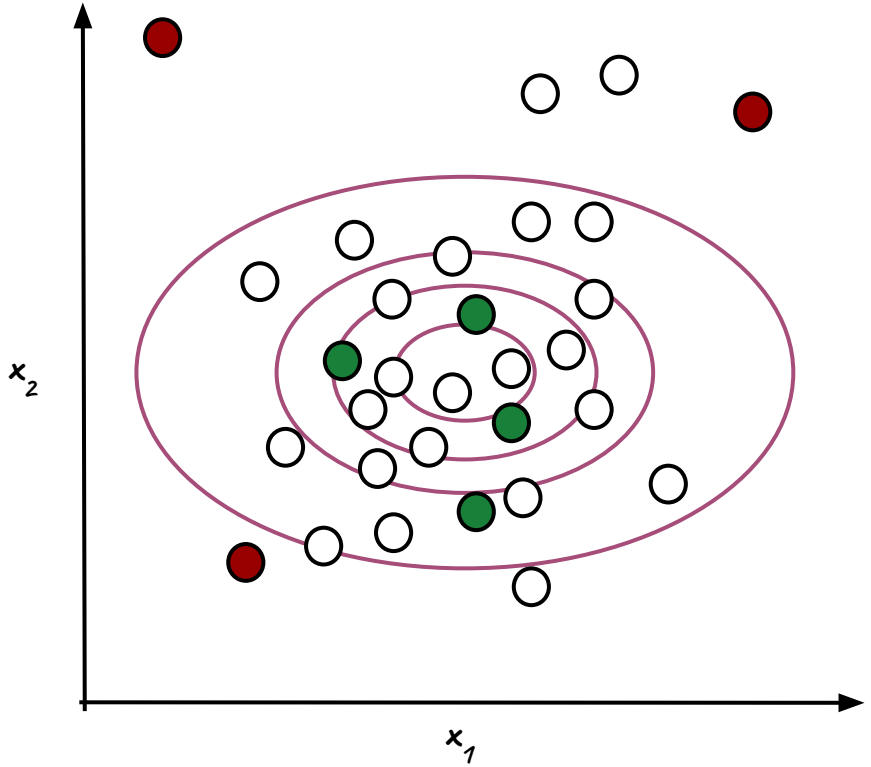
As you can imagine, this gets easily generalized by directly modelling the dataset with multivariate Gaussians, e.g. *Gaussian Mixture Models*, storing the results and placing a threshold.

An *anomaly detection* algorithm is able to look at an unlabeled dataset, and automatically identify *novelties/outliers*, and raise a warning or clean datasets accordingly, or select them for future analysis. In a nutshell, it discriminates between the bulk of the data distribution - whatever the functional form it is - and what is not part of that bulk.



Remember: *outlier detection* to clean the training set, *novelty detection* trained on the cleaned dataset and applied to **NEW** data.

An *anomaly detection* algorithm is able to look at an unlabeled dataset, and automatically identify *novelties/outliers*, and raise a warning or clean datasets accordingly, or select them for future analysis. In a nutshell, it discriminates between the bulk of the data distribution - whatever the functional form it is - and what is not part of that bulk.



Remember: *outlier detection* to clean the training set, *novelty detection* trained on the cleaned dataset and applied to **NEW** data.

When building an anomaly detection algorithm with density estimation, it is much easier to make a decision if somehow we have some labeled examples on which to cross-validate/dev and test the algorithm, then deployed in production if everything works fine.

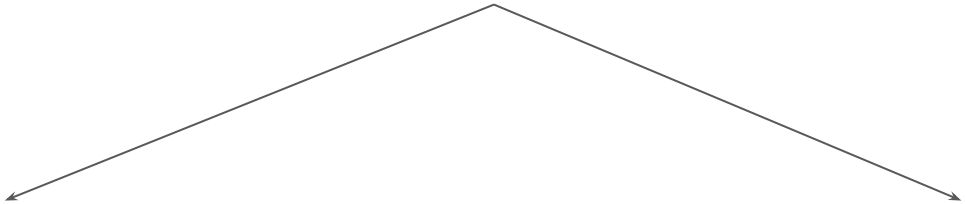
Sometimes it happens that you have a bulk of unlabeled data with a fraction of labeled data. This is called "*weakly supervised learning*".
In this case, the best practice is to:

- train the anomaly detection algorithm on the **unlabeled** data
- cross-validate/dev the algorithm on the **labeled** data

and evaluate the model with the usual classification metrics I've told you about in the first lesson

Why not using supervised learning altogether?

But, wait a minute, I have some labeled examples… why am I not using a supervised learning algorithm instead?



Anomaly detection

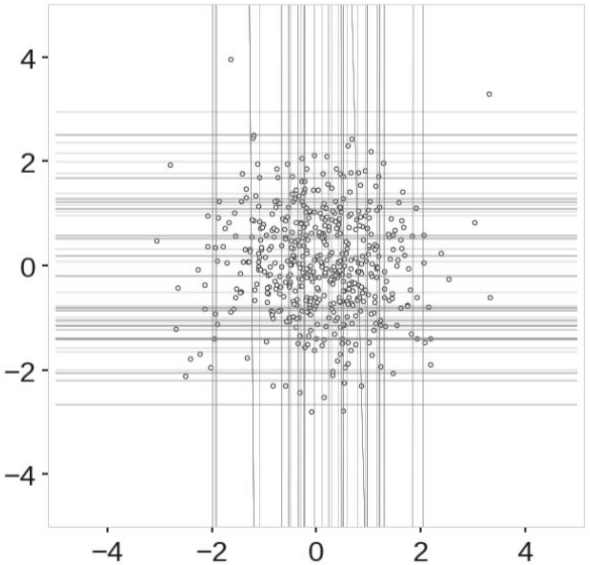
- very small number of positive examples (0–20 is common)
- a very large number of negative examples
- many different types of anomaly: it is hard for any algorithm to learn from a very few handful of positive examples what the anomalies look like, as future anomalies might be different from past ones

Supervised learning

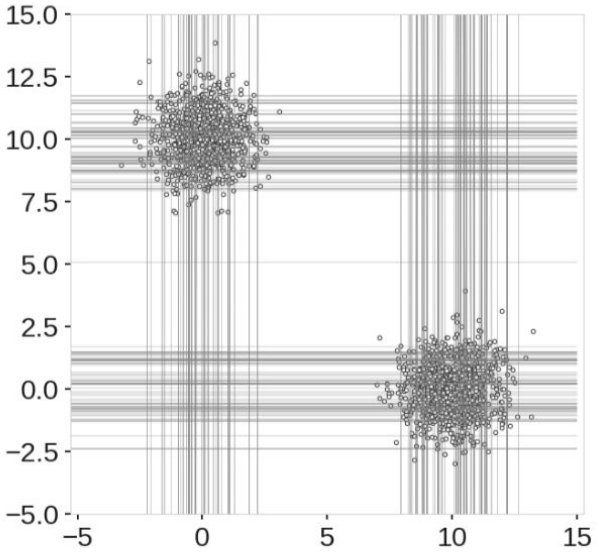
- large number of positive and negative examples
- enough positive examples for algorithm to make sense of what positive examples are; future examples are likely to be similar to ones in training

Isolation Forest, as you can guess by the name, has lots in common with the *random forest* algorithm we saw last week.

Isolation Forest, as you can guess by the name, has lots in common with the *random forest* algorithm we saw last week.
It isolates observations by *randomly* selecting a feature, and *randomly* splitting them with a value between the maximum and minimum values of the selected feature, until a *max_depth* is reached, or there is only one point in the final node. This recursive random partitioning is represented by a tree structure, and the *path length* from the root node to the terminating node is equal to the number of splittings.
The number of splittings required to isolate a sample is *lower* for *outliers*, and *higher* for *inliers*. When a forest of random trees collectively produce shorter path lengths for particular examples, they are highly likely to be anomalies.



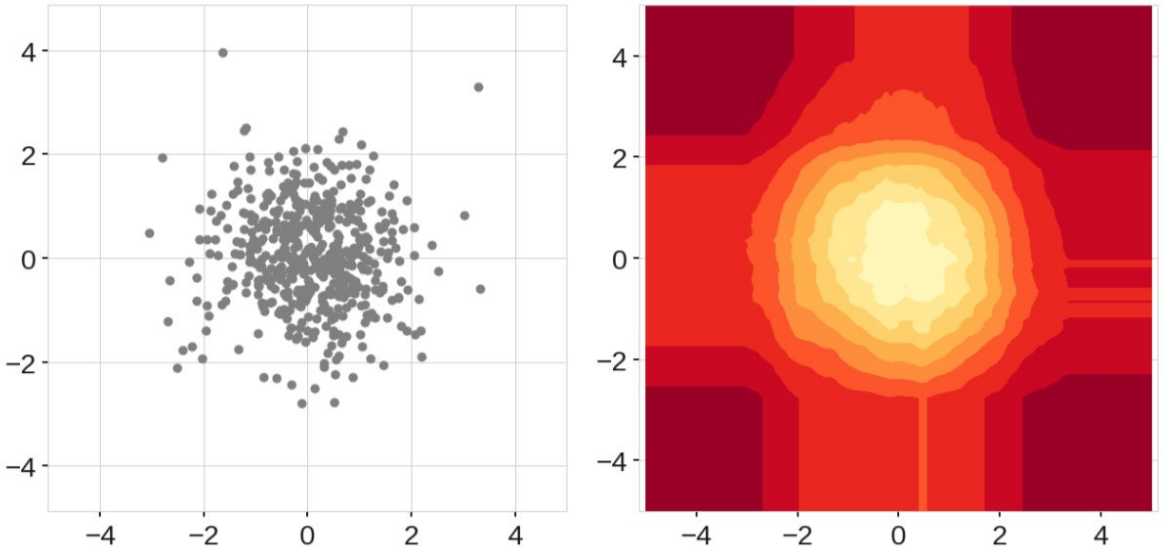
(a) Single blob



(b) Multiple Blobs

https://www.analyticsvidhya.com/blog/2021/07/anomaly-detection-using-isolation-forest-a-complete-guide/

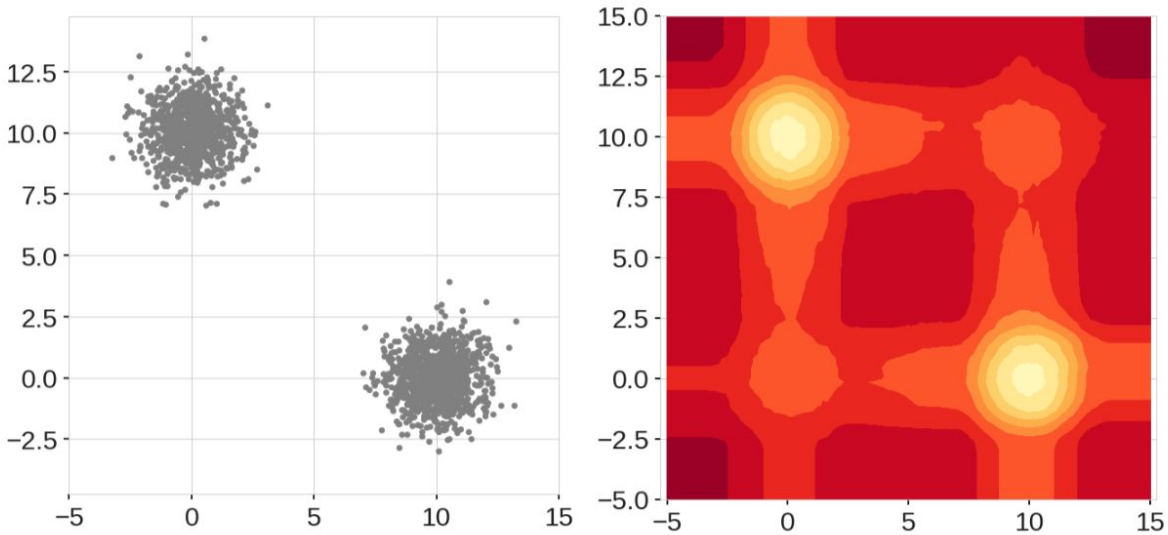
Isolation Forest, as you can guess by the name, has lots in common with the **random forest** algorithm we saw last week.
It isolates observations by *randomly* selecting a feature, and *randomly* splitting them with a value between the maximum and minimum values of the selected feature, until a **max_depth** is reached, or there is only one point in the final node. This recursive random partitioning is represented by a tree structure, and the **path length** from the root node to the terminating node is equal to the number of splittings.
The number of splittings required to isolate a sample is **lower** for *outliers*, and **higher** for *inliers*. When a forest of random trees collectively produce shorter path lengths for particular examples, they are highly likely to be anomalies.



(a) Normally Distributed Data (b) Anomaly Score Map

https://www.analyticsvidhya.com/blog/2021/07/anomaly-detection-using-isolation-forest-a-complete-guide/

Isolation Forest, as you can guess by the name, has lots in common with the **random forest** algorithm we saw last week.
It isolates observations by *randomly* selecting a feature, and *randomly* splitting them with a value between the maximum and minimum values of the selected feature, until a **max_depth** is reached, or there is only one point in the final node. This recursive random partitioning is represented by a tree structure, and the **path length** from the root node to the terminating node is equal to the number of splittings.
The number of splittings required to isolate a sample is **lower** for *outliers*, and **higher** for *inliers*. When a forest of random trees collectively produce shorter path lengths for particular examples, they are highly likely to be anomalies.



(a) Two normally distributed clusters

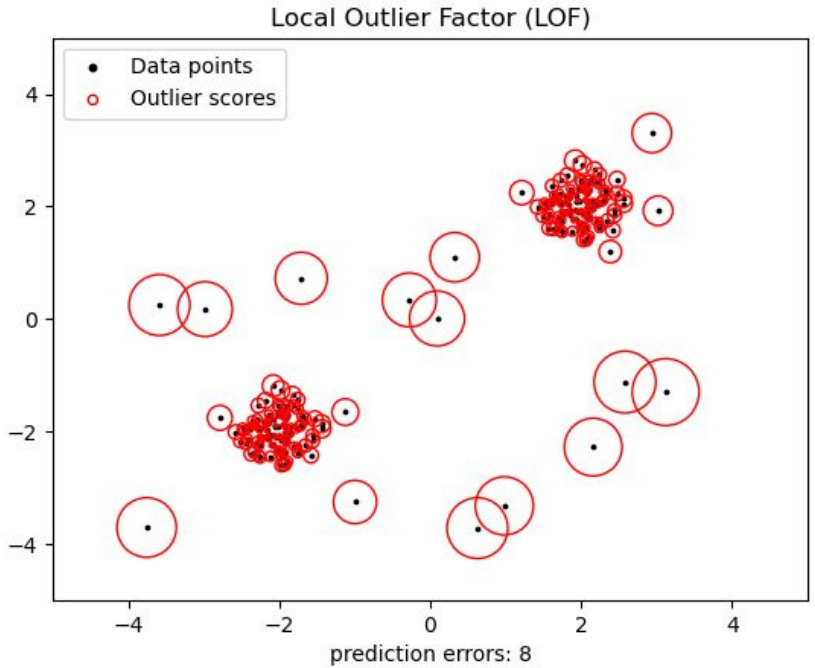
(b) Anomaly Score Map

https://www.analyticsvidhya.com/blog/2021/07/anomaly-detection-using-isolation-forest-a-complete-guide/

Local Outlier Fraction is basically *k-Nearest Neighbors* used as an anomaly detection algorithm.

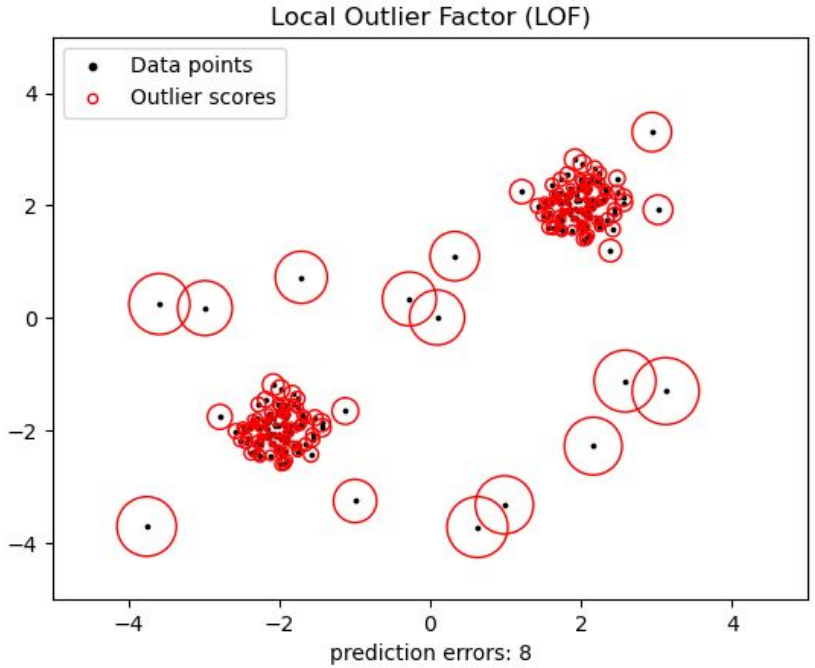
Local Outlier Fraction is basically *k-Nearest Neighbors* used as an anomaly detection algorithm.
LOF computes a score (the *local outlier factor*) measuring the local density deviation of a given data point with respect to its neighbors, trying to detect the examples living in a region of feature space of lower density than their neighbors.
The local density is obtained from *kNN*. *LOF* score is equal to the ratio of the average local density of its *kNN*, and its own local density.
An inlier is expected to have a local density similar to that of its neighbors, while outliers are expected to have much smaller local density.

Local Outlier Fraction is basically *k-Nearest Neighbors* used as an anomaly detection algorithm.
LOF computes a score (the *local outlier factor*) measuring the local density deviation of a given data point with respect to its neighbors, trying to detect the examples living in a region of feature space of lower density than their neighbors.
The local density is obtained from *kNN*. *LOF* score is equal to the ratio of the average local density of its *kNN*, and its own local density.
An inlier is expected to have a local density similar to that of its neighbors, while outliers are expected to have much smaller local density.



https://scikit-learn.org/stable/auto_examples/neighbors/plot_lof_outlier_detection.html

Local Outlier Fraction is basically *k-Nearest Neighbors* used as an anomaly detection algorithm.
LOF computes a score (the *local outlier factor*) measuring the local density deviation of a given data point with respect to its neighbors, trying to detect the examples living in a region of feature space of lower density than their neighbors.
The local density is obtained from *kNN*. *LOF* score is equal to the ratio of the average local density of its *kNN*, and its own local density.
An inlier is expected to have a local density similar to that of its neighbors, while outliers are expected to have much smaller local density.



https://scikit-learn.org/stable/auto_examples/neighbors/plot_lof_outlier_detection.html

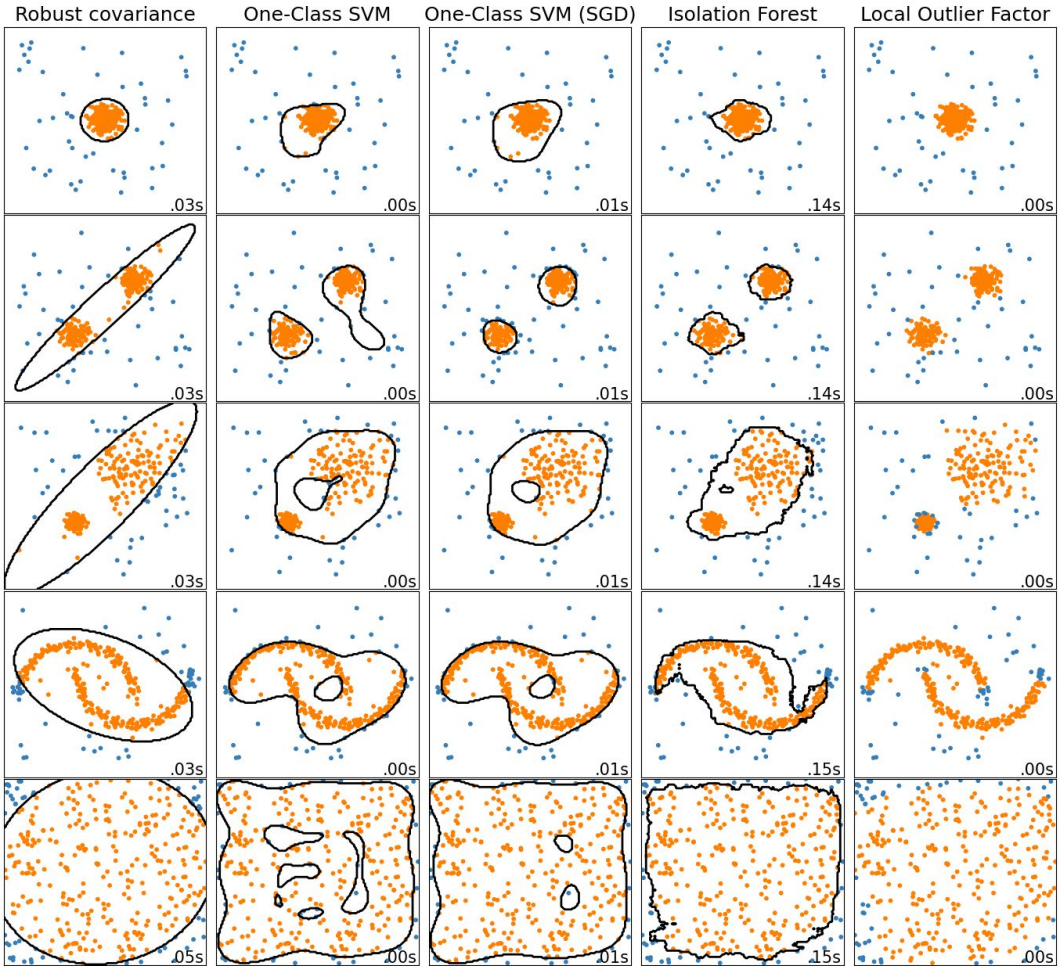
k should be:

- higher than the minimum number of objects a cluster has to contain, so that other objects can be *local outliers* relative to this cluster
- smaller than the maximum number of close by objects that can potentially be local outliers.

In practice, such information is generally not available (at least, if you do not have a physical motivation behind), and taking *k=20* appears to work well in general.

The strength of the LOF algorithm is that it takes both local and global properties of datasets into consideration: it can perform well even in datasets where abnormal samples have different underlying densities. The question is not how isolated the sample is, but how isolated it is *with respect to* the surrounding neighborhood.

Anomaly detection: same data, different results

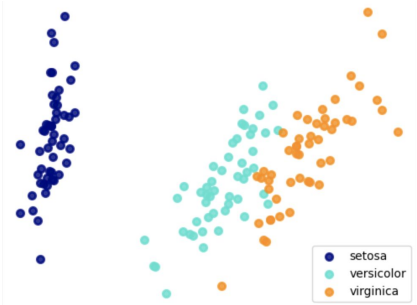


There are three main categories of algorithms in unsupervised learning.

Dimensionality Reduction

Reducing the number of random variables to consider, either for visualization, or flexibility.

e.g. PCA, NNMF, IsoMap, LDA, t-SNE, SOMs...



https://visxai-dimensionality-reduction-1dbad0a67a092b007c526a45.vercel.app/

Dimensionality reduction algorithms are codes that, well, reduce the dimensionality of a dataset of examples.

Dimensionality reduction algorithms are codes that, well, reduce the dimensionality of a dataset of examples.
They serve three main purposes:

- *visualization*, e.g. reduce a 18 dimensional feature space, that even Dr. Manhattan has some troubles in imagining, to a simple 2D space, under the assumption that similar objects will tend to stay closer in a reduced version of the feature space.

Dimensionality reduction algorithms are codes that, well, reduce the dimensionality of a dataset of examples.
They serve three main purposes:

- *visualization*, e.g. reduce a 18 dimensional feature space, that even Dr. Manhattan has some troubles in imagining, to a simple 2D space, under the assumption that similar objects will tend to stay closer in a reduced version of the feature space.
- *cleaning/clustering*. as above, but instead of just looking at the dataset, exploit the fact similar objects group together in the reduced feature space.

Dimensionality reduction algorithms are codes that, well, reduce the dimensionality of a dataset of examples.
They serve three main purposes:

- *visualization*, e.g. reduce a 18 dimensional feature space, that even Dr. Manhattan has some troubles in imagining, to a simple 2D space, under the assumption that similar objects will tend to stay closer in a reduced version of the feature space.
- *cleaning/clustering*. as above, but instead of just looking at the dataset, exploit the fact similar objects group together in the reduced feature space.
- actual *data compression*: these algorithms reduce the dataset weight while keeping the maximum possible amount of information, e.g. PCA is able to retain 94% of the information in a stellar spectra with just the first 10 eigenspectra, with full information contained in a thousand eigenspectra: that's a factor 100 data compression.

Dimensionality reduction algorithms are codes that, well, reduce the dimensionality of a dataset of examples.
They serve three main purposes:

- *visualization*, e.g. reduce a 18 dimensional feature space, that even Dr. Manhattan has some troubles in imagining, to a simple 2D space, under the assumption that similar objects will tend to stay closer in a reduced version of the feature space.
- *cleaning/clustering*. as above, but instead of just looking at the dataset, exploit the fact similar objects group together in the reduced feature space.
- actual *data compression*: these algorithms reduce the dataset weight while keeping the maximum possible amount of information, e.g. PCA is able to retain 94% of the information in a stellar spectra with just the first 10 eigenspectra, with full information contained in a thousand eigenspectra: that's a factor 100 data compression.

And the reason why these have been developed and employed is one and one only:

THE CURSE OF DIMENSIONALITY

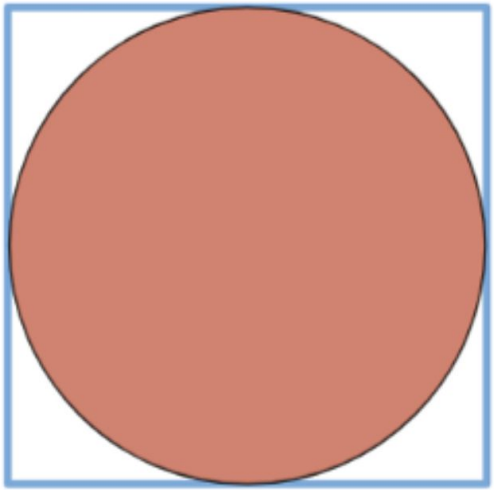
a concept dating back to 1957 that had the blessing of being dubbed with a **fantastic** name
especially if you consider that scientist and software engineers are usually really bad at naming things.

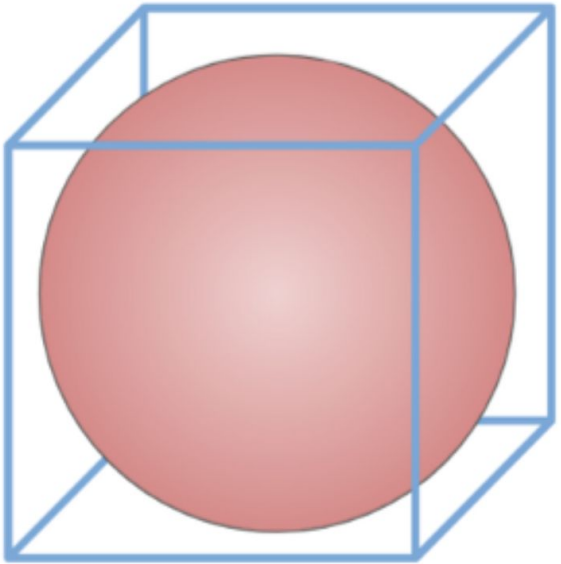
What is the **"curse of dimensionality"**?

Take a look at this example: a circle inscribed in a square on the left, and a sphere inscribed in a cube on the right.

Notice how the area *outside* of the circle/sphere grows larger and larger as the number of dimensions increase, up to the point where most of the volume of a D-dimensional hypercube actually reside in the corners.

$$f_D = \frac{V_D(r)}{(2r)^D} = \frac{\pi^{D/2}}{D2^{D-1}\Gamma(D/2)}$$





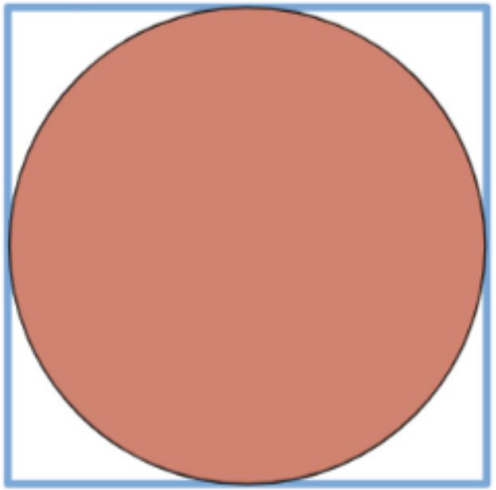
What is the *"curse of dimensionality"*?

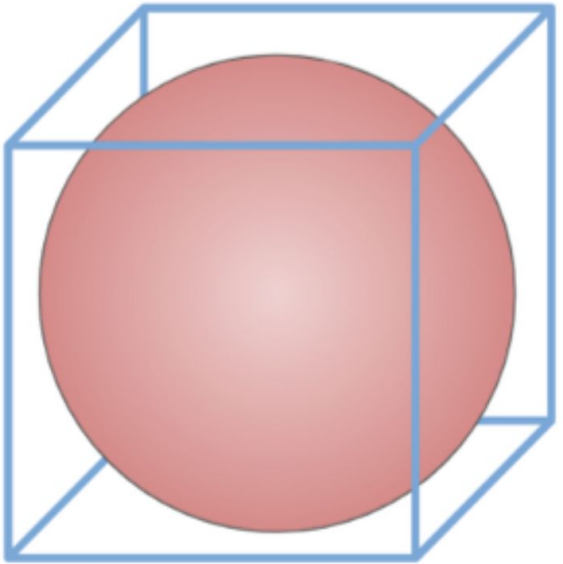
Take a look at this example: a circle inscribed in a square on the left, and a sphere inscribed in a cube on the right.

Notice how the area *outside* of the circle/sphere grows larger and larger as the number of dimensions increase, up to the point where most of the volume of a D-dimensional hypercube actually reside in the corners.

$$f_D = \frac{V_D(r)}{(2r)^D} = \frac{\pi^{D/2}}{D2^{D-1}\Gamma(D/2)}$$

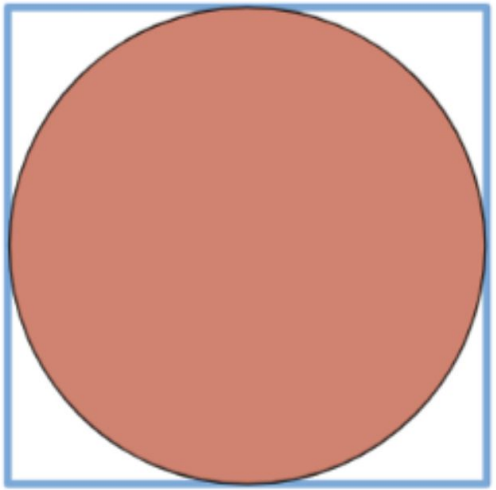
The more dimensions your data span, the more points are needed to uniformly sample the space, up to the point where it becomes mathematically impossible to sample *any* region of the space.

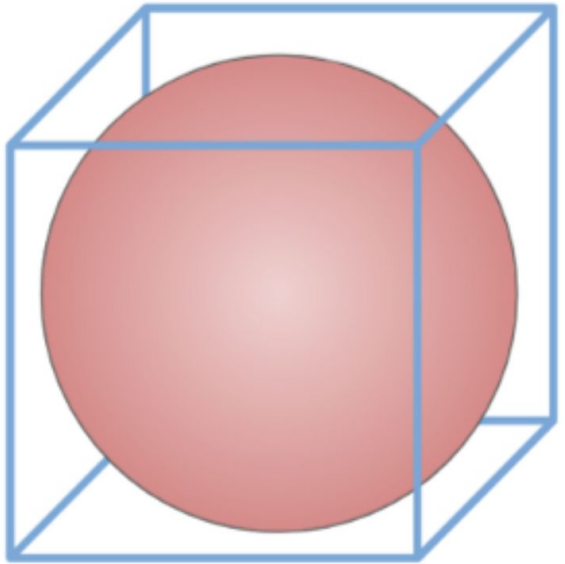




What is the *"curse of dimensionality"*?
Take a look at this example: a circle inscribed in a square on the left, and a sphere inscribed in a cube on the right.
Notice how the area *outside* of the circle/sphere grows larger and larger as the number of dimensions increase, up to the point where most of the volume of a D-dimensional hypercube actually reside in the corners.

$$f_D = \frac{V_D(r)}{(2r)^D} = \frac{\pi^{D/2}}{D2^{D-1}\Gamma(D/2)}$$





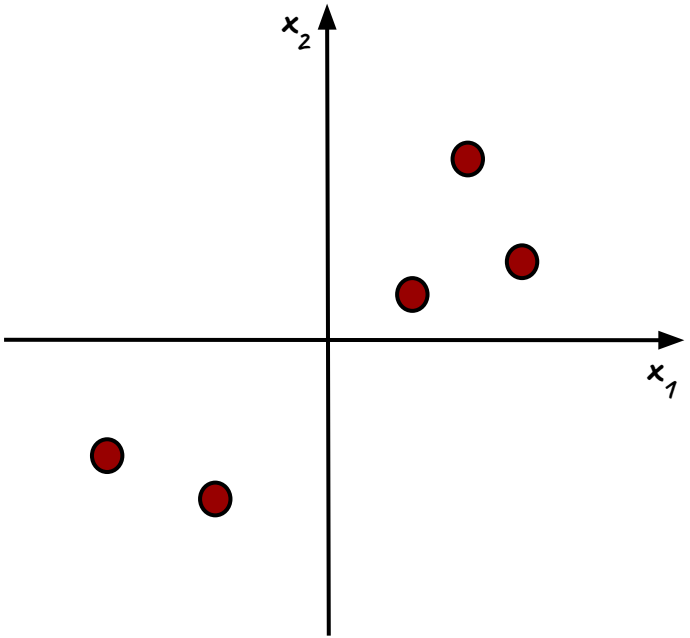
The more dimensions your data span, the more points are needed to uniformly sample the space, up to the point where it becomes mathematically impossible to sample *any* region of the space.

Pay attention that dimensionality isn't just measuring D features for N objects. It could be e.g. a spectrum with D values, or an image with D pixels, etc.

Quantitative example: the SDSS comprises a sample of 357 million sources; each source has 448 measured attributes.
If we normalize the data range to [-1,1], on average, any box of volume one will contain only $2^{448}/(357 \times 10^6) \sim 2$ sources!

Principal Component Analysis (PCA) is the most famous dimensionality reduction algorithm.

Principal Component Analysis (PCA) is the most famous dimensionality reduction algorithm.
We have this dataset composed of two preprocessed-and-scaled features, so they have zero mean and span similar ranges of values.
What would *PCA* do here?

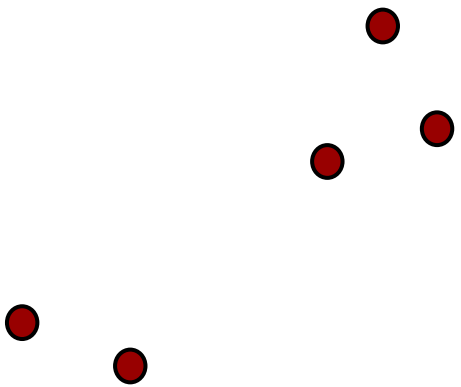


Principal Component Analysis (PCA) is the most famous dimensionality reduction algorithm.

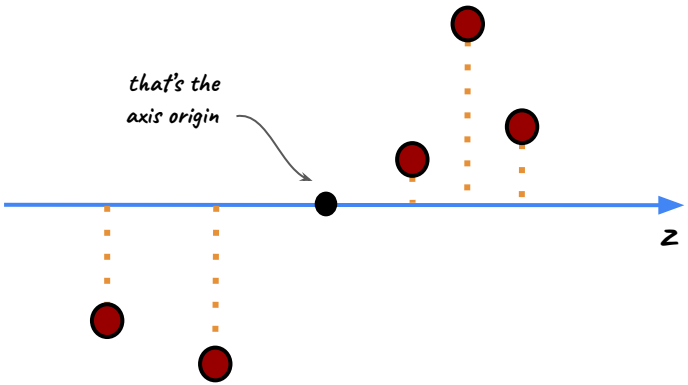
We have this dataset composed of two preprocessed-and-scaled features, so they have zero mean and span similar ranges of values.

What would *PCA* do here?

We remove the two features axis x_1 and x_2, remain only with the examples, and try to choose one single axis that captures what's most important and informative about these examples, by projecting their values on it.

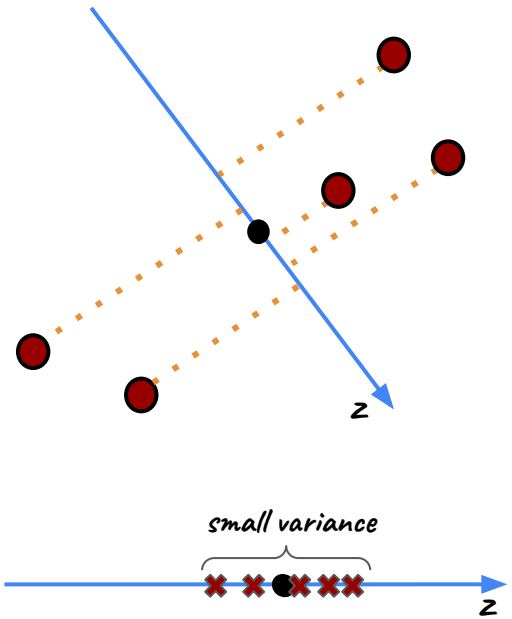


Principal Component Analysis (PCA) is the most famous dimensionality reduction algorithm.
We have this dataset composed of two preprocessed-and-scaled features, so they have zero mean and span similar ranges of values.
What would *PCA* do here?
We remove the two features axis x_1 and x_2, remain only with the examples, and try to choose one single axis that captures what's most important and informative about these examples, by projecting their values on it.



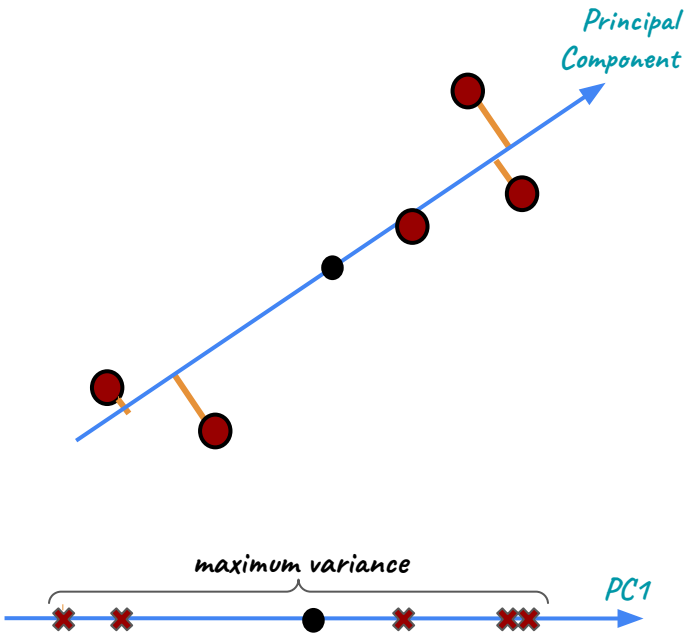
If we take that particular **axis z**, we will *project* the values of the examples on it. In this way we passed from a two dimensional set of features to only one, the value projected on the **z-axis.**
That's actually a good choice for axis to project on, that quite captures the whole spread of these examples dataset, i.e. their **variance** (and thus information).

Principal Component Analysis (PCA) is the most famous dimensionality reduction algorithm.
We have this dataset composed of two preprocessed-and-scaled features, so they have zero mean and span similar ranges of values.
What would *PCA* do here?
We remove the two features axis x_1 and x_2, remain only with the examples, and try to choose one single axis that captures what's most important and informative about these examples, by projecting their values on it.



If we take that particular **axis z**, we will *project* the values of the examples on the **axis**. In this way we passed from a two dimensional set of features to only one, the value projected on the **axis.**
That's actually a good choice for axis to project on, that quite captures the whole spread of these examples dataset, i.e. their ***variance*** (and thus information).
But we could take another choice, a particularly bad choice: the examples are squished into the axis, capturing less of their intrinsic ***variance***.

Principal Component Analysis (PCA) is the most famous dimensionality reduction algorithm.
We have this dataset composed of two preprocessed-and-scaled features, so they have zero mean and span similar ranges of values.
What would *PCA* do here?
We remove the two features axis x_1 and x_2, remain only with the examples, and try to choose one single axis that captures what's most important and informative about these examples, by projecting their values on it.

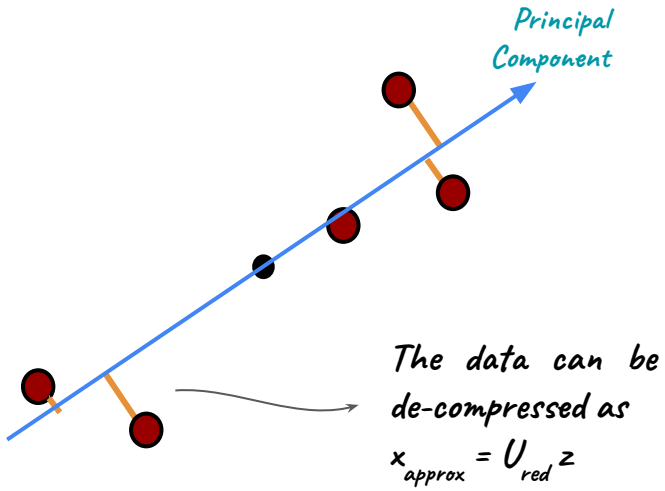


If we take that particular **axis z**, we will *project* the values of the examples on the **axis**. In this way we passed from a two dimensional set of features to only one, the value projected on the **axis.**
That's actually a good choice for axis to project on, that quite captures the whole spread of these examples dataset, i.e. their **variance** (and thus information).
But we could take another choice, a particularly bad choice: the examples are squished into the axis, capturing less of their intrinsic **variance**.
The best choice, as you can imagine, is the axis that maximizes the data **variance**, even though we are now using just one axis to describe the data.
This maximum variance axis is called the *principal component*.

Of course, on a general D dimensional application you could find several $k<D$ *principal components* to project your dataset, all perpendicular.

Notice that *PCA* is not linear regression (more similar to ODR).

Principal Component Analysis (PCA) is the most famous dimensionality reduction algorithm.
We have this dataset composed of two preprocessed-and-scaled features, so they have zero mean and span similar ranges of values.
What would *PCA* do here?
We remove the two features axis x_1 and x_2, remain only with the examples, and try to choose one single axis that captures what's most important and informative about these examples, by projecting their values on it.



The algorithm goes as follows: for k principal components:

- Compute the covariance matrix

$$\Sigma = \frac{1}{m} \sum_{i=1}^{n} (x^{(i)})(x^{(i)})^T$$

- Compute the eigenvectors of Σ (Sigma Value Decomposition)

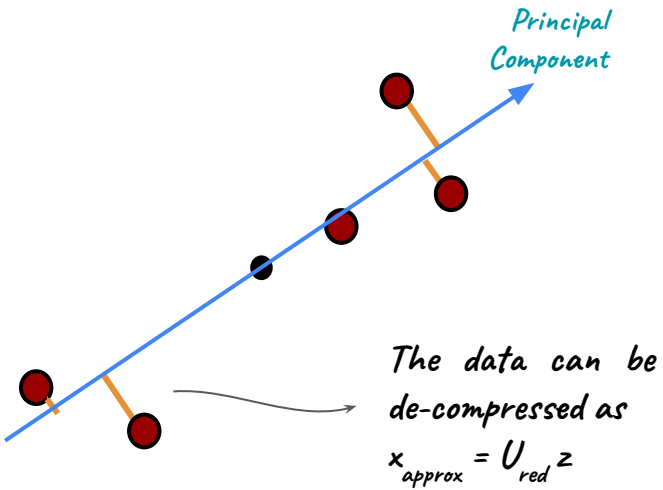
$$U = \begin{bmatrix} \vdots & \vdots & \dots & \vdots \\ u^{(1)} & u^{(2)} & \dots & u^{(m)} \\ \vdots & \vdots & \dots & \vdots \end{bmatrix} \in \mathbb{R}^{n \times n}$$

and take only the first k columns:

$$U_{red} = \begin{bmatrix} \vdots & \vdots & \vdots \\ u^{(1)} & \dots & u^{(k)} \\ \vdots & \vdots & \vdots \end{bmatrix} \in \mathbb{R}^{n \times k}$$

- The (eigen)vectors onto which to project are: $z = U_{red}^T x$

Principal Component Analysis (PCA) is the most famous dimensionality reduction algorithm.
We have this dataset composed of two preprocessed-and-scaled features, so they have zero mean and span similar ranges of values.
What would *PCA* do here?
We remove the two features axis x_1 and x_2, remain only with the examples, and try to choose one single axis that captures what's most important and informative about these examples, by projecting their values on it.



How to choose k?
Two important quantities are the average squared projection error:

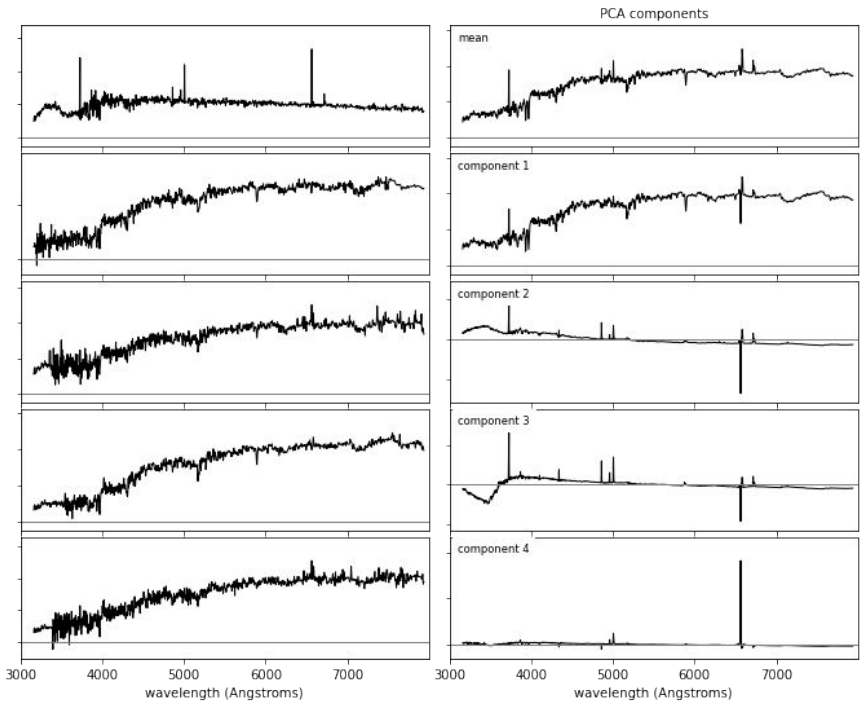
$$\frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - x_{\text{approx}}^{(i)}||^2$$

and the total variation in the data:

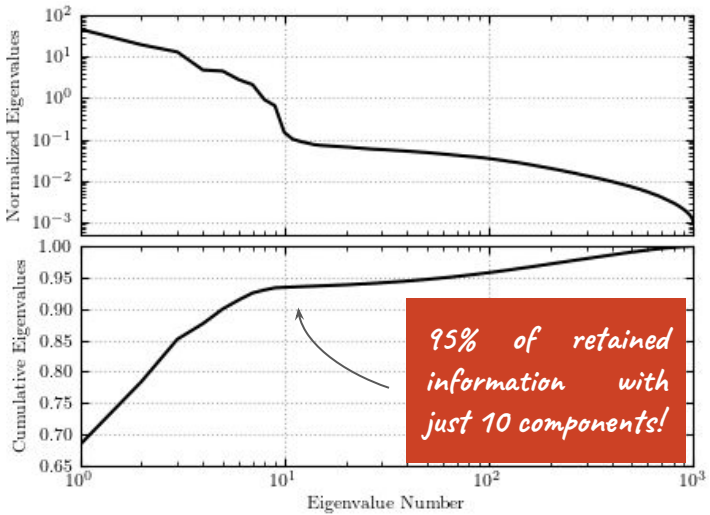
$$\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2$$

k is typically chosen in order to be the smallest value so that the ratio between the two is below 0.01 or 0.05 (or in that order of magnitude), meaning that you are losing only 1% (or 5%) of information, and therefore 99% (or 95%) of the variance in the data is still retained.

What would *PCA* do on a real astrophysical application?

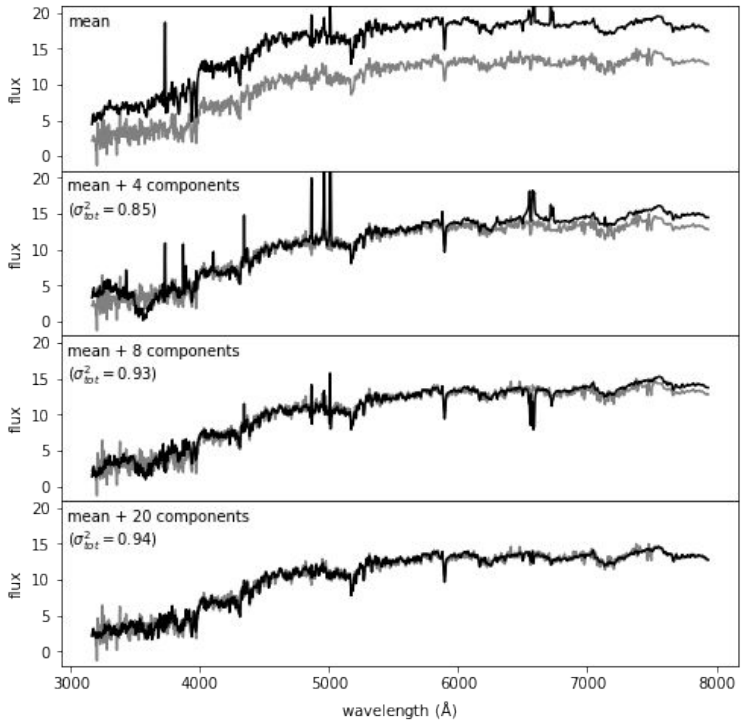


On the left, an example of *PCA* applied to a galaxy spectra from the Sloan Digital Sky Survey (no Cats this time, sorry).
Right panel: the first four *PCA* components; left panel: the reconstructed spectra once summed the previous *PCA* components.

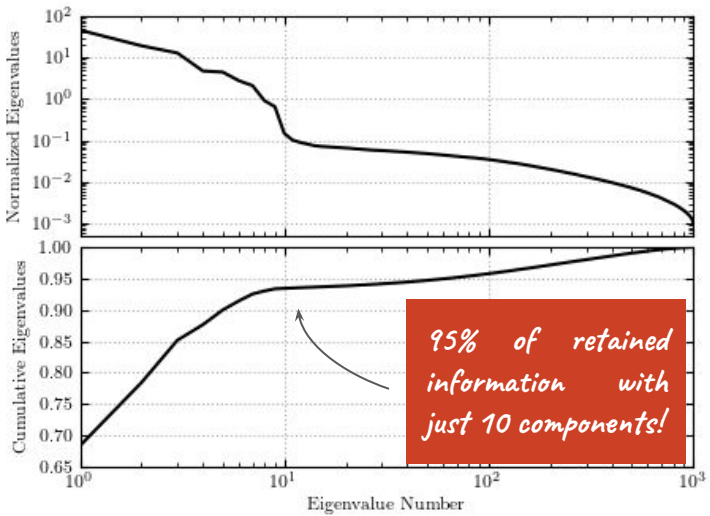


from "Statistics, Data Mining and Machine Learning in Astronomy", Ivezic et al, Princeton University Press

What would *PCA* do on a real astrophysical application?

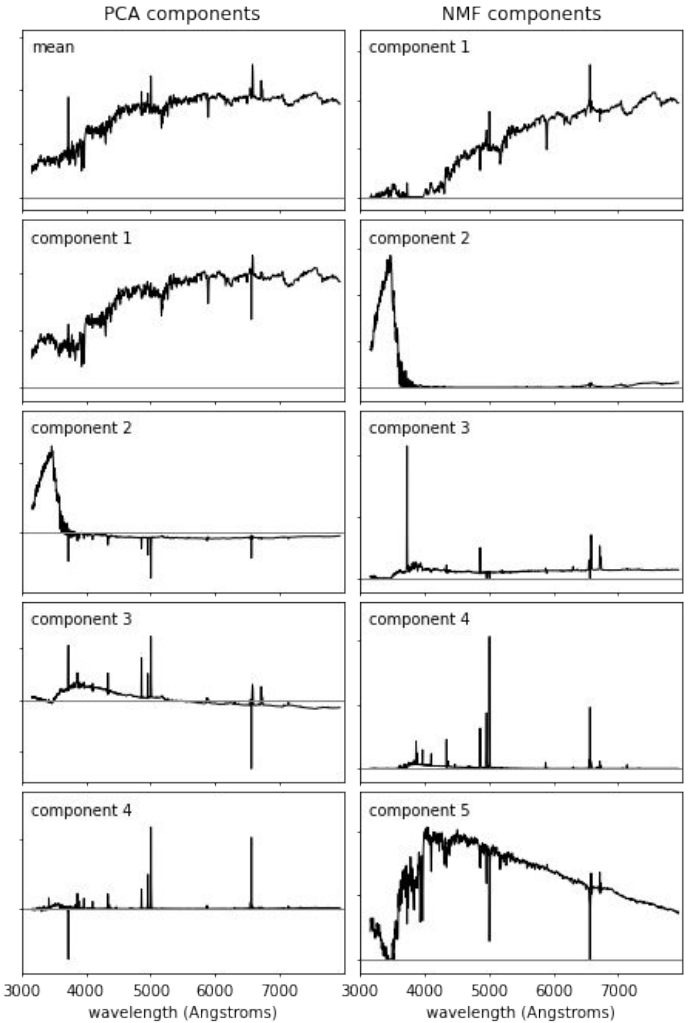


On the left, an example of *PCA* applied to a galaxy spectra from the Sloan Digital Sky Survey (no Cats this time, sorry).
Right panel: the first four *PCA* components; left panel: the reconstructed spectra once summed the previous *PCA* components.



from "Statistics, Data Mining and Machine Learning in Astronomy", Ivezic et al, Princeton University Press

Non-Negative Matrix Factorization



One of the biggest drawbacks of *PCA* is that the principal components can assume positive or negative values. For many physical systems we know a-priori that the data can be represented as a linear sum of ***positive*** components, e.g., spectra.

Non-negative Matrix Factorization (NMF) treats the data as a linear sum of positive-definite components. Its like *PCA*, except the coefficients in the linear combination must be ***non-negative***, thus implying an additive combination of basis parts to reconstruct whole.

Notice that all NMF components are positive, while PCA components could assume negative values.

from "Statistics, Data Mining and Machine Learning in Astronomy", Ivezic et al, Princeton University Press

Sometimes you have data for which the single components are statistically independent (or nearly so). In that case you would be more interested in separating the mixed components. That's what *Independent Component Analysis (ICA)* is for.

Sometimes you have data for which the single components are statistically independent (or nearly so). In that case you would be more interested in separating the mixed components. That's what *Independent Component Analysis (ICA)* is for.
What do we mean for statistically independent data? Let's take the *cocktail party* problem, as illustrated below:

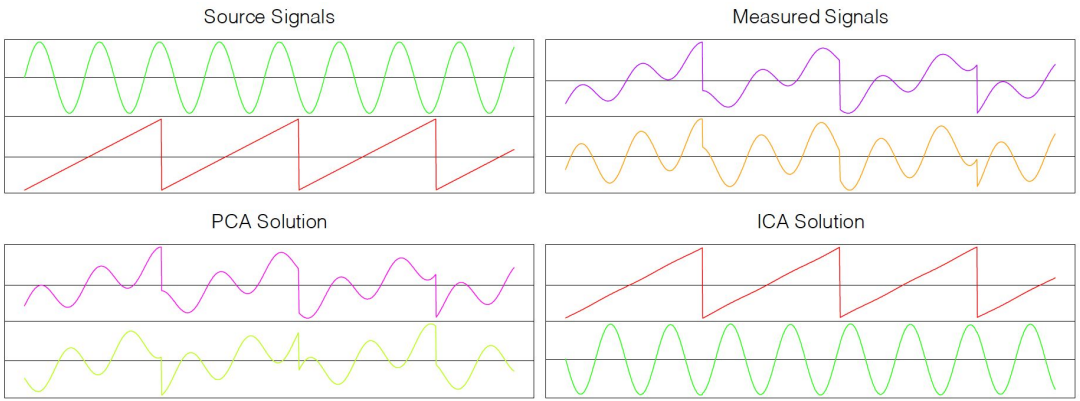


FIGURE 14.37. *Illustration of ICA vs. PCA on artificial time-series data. The upper left panel shows the two source signals, measured at* 1000 *uniformly spaced time points. The upper right panel shows the observed mixed signals. The lower two panels show the principal components and independent component solutions.*

Hastie, Tibshirani, & Friedman, "The Elements of. Statistical Learning", 2001

Sometimes you have data for which the single components are statistically independent (or nearly so). In that case you would be more interested in separating the mixed components. That's what *Independent Component Analysis (ICA)* is for.
What do we mean for statistically independent data? Let's take the *cocktail party* problem, as illustrated below:

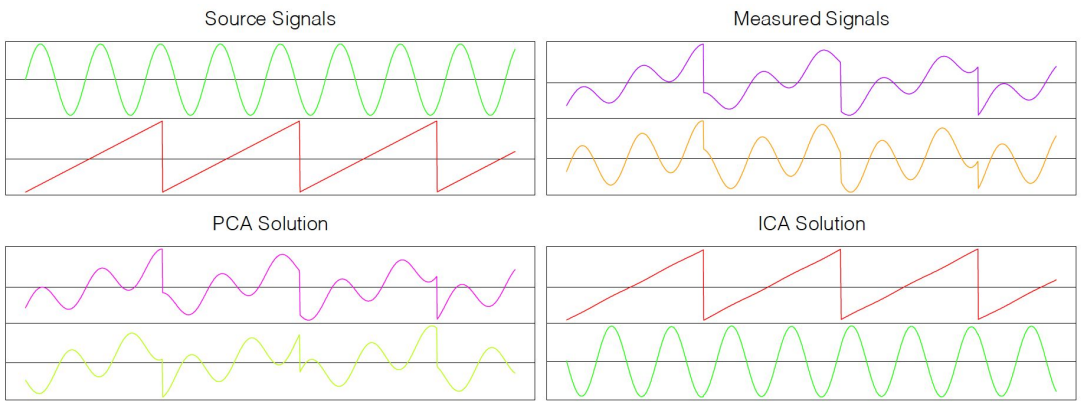


FIGURE 14.37. *Illustration of ICA vs. PCA on artificial time-series data. The upper left panel shows the two source signals, measured at* 1000 *uniformly spaced time points. The upper right panel shows the observed mixed signals. The lower two panels show the principal components and independent component solutions.*

Here the source signals are two voices, at a crowded party, and you are trying to concentrate on just one voice. What you hear is something like the "measured signals" pattern in the plot on the left.

PCA would do an excellent job in reconstructing the signal with reduced dimensionality, but it wouldn't actually isolate the different physical components.
ICA on the other hand can.

Hastie, Tibshirani, & Friedman, "The Elements of. Statistical Learning", 2001

Sometimes you have data for which the single components are statistically independent (or nearly so). In that case you would be more interested in separating the mixed components. That's what *Independent Component Analysis (ICA)* is for.
What do we mean for statistically independent data? Let's take the *cocktail party* problem, as illustrated below:

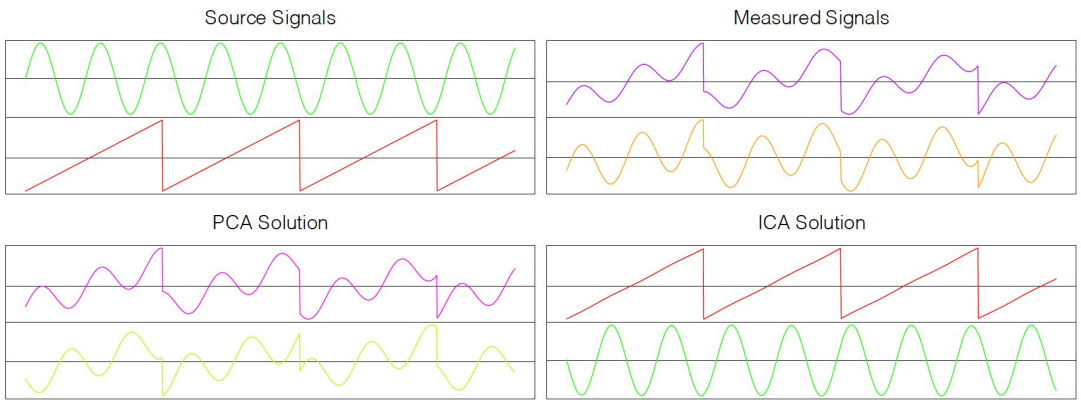


FIGURE 14.37. *Illustration of ICA vs. PCA on artificial time-series data. The upper left panel shows the two source signals, measured at* 1000 *uniformly spaced time points. The upper right panel shows the observed mixed signals. The lower two panels show the principal components and independent component solutions.*

Hastie, Tibshirani, & Friedman, "The Elements of. Statistical Learning", 2001

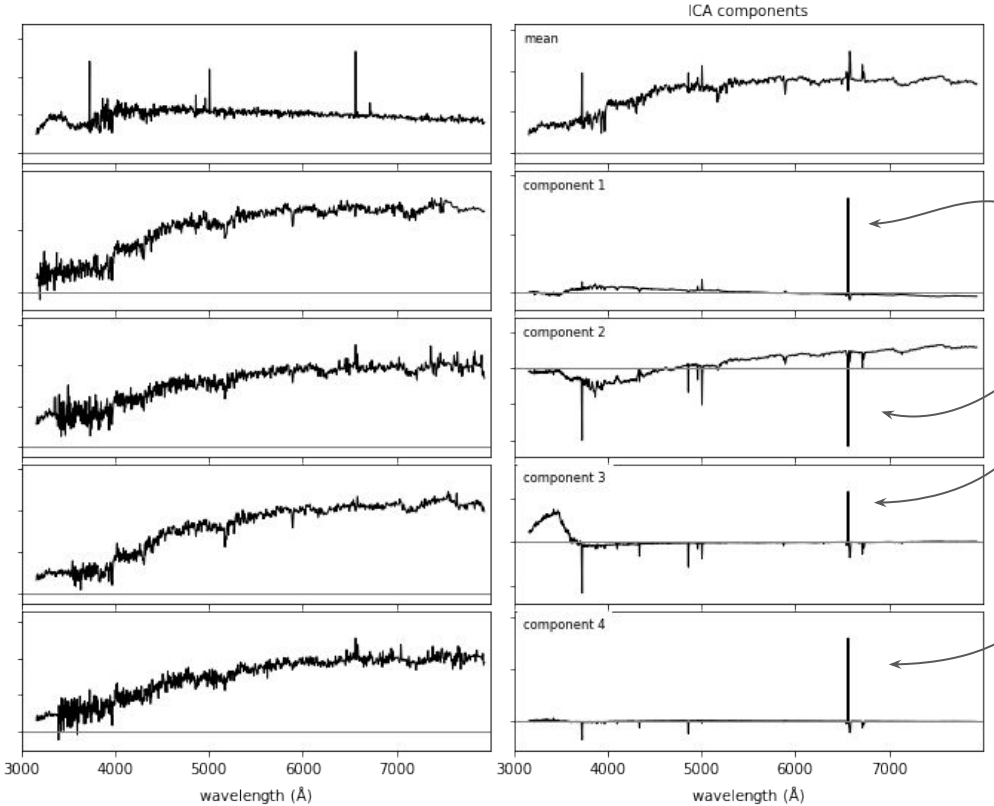
Here the source signals are two voices, at a crowded party, and you are trying to concentrate on just one voice. What you hear is something like the "measured signals" pattern in the plot on the left.

PCA would do an excellent job in reconstructing the signal with reduced dimensionality, but it wouldn't actually isolate the different physical components.
ICA on the other hand can.

ICA is a good choice for a complex system with relatively independent components, e.g. a galaxy is roughly a linear combination of cool stars and hot stars, and a quasar is just a galaxy with others component from an accretion disk and emission line regions. Ideally we want "eigenvectors" that are aligned with those physical traits/regions, as opposed to abstract mathematical constructs.

Independent Component Analysis

Sometimes you have data for which the single components are statistically independent (or nearly so). In that case you would be more interested in separating the mixed components. That's what *Independent Component Analysis (ICA)* is for.
What do we mean for statistically independent data? Let's take the *cocktail party* problem, as illustrated below:

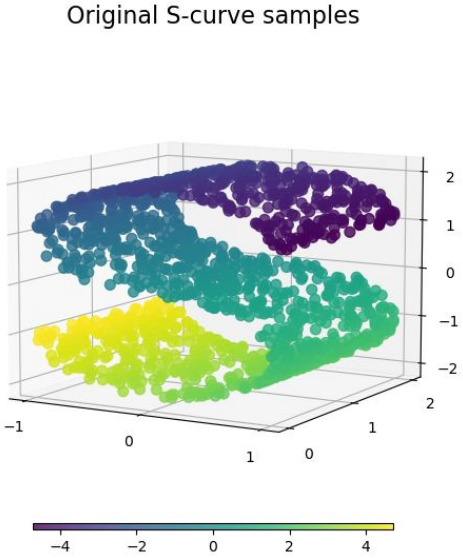


from "Statistics, Data Mining and Machine Learning in Astronomy", Ivezic et al, Princeton University Press

ICA actually looks like it is picking up distinct line regions as the components.

ICA is a good choice for a complex system with relatively independent components, e.g. a galaxy is roughly a linear combination of cool stars and hot stars, and a quasar is just a galaxy with others component from an accretion disk and emission line regions. Ideally we want "eigenvectors" that are aligned with those physical traits/regions, as opposed to abstract mathematical constructs.

PCA, *NNMF* and *ICA* are great, but have a major drawback: these are all ***linear*** dimensionality reduction algorithms. They cannot learn anything from ***non-linear*** datasets. Complex, non-linear structures can be learned by *manifold learning* algorithms.

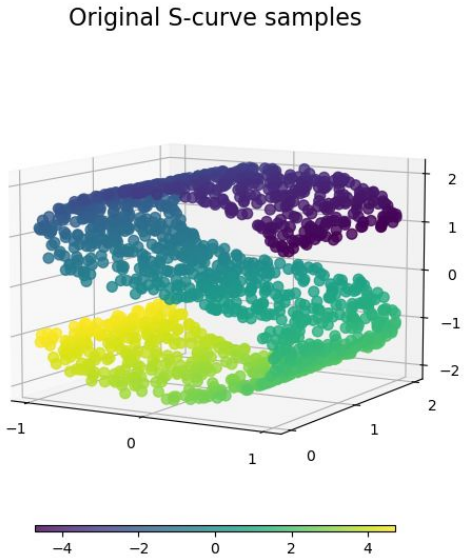


Manifold Learning

PCA, *NNMF* and *ICA* are great, but have a major drawback: these are all ***linear*** dimensionality reduction algorithms. They cannot learn anything from ***non-linear*** datasets. Complex, non-linear structures can be learned by *manifold learning* algorithms.

Manifold learning are based on the idea that the dimensionality of many data sets is only *artificially* high. Although the data points may consist of thousands of features, they may be described as a function of only a few underlying parameters. That is, the data points are actually samples from a low-dimensional *manifold* (*latent space*) that is embedded in a high-dimensional space.

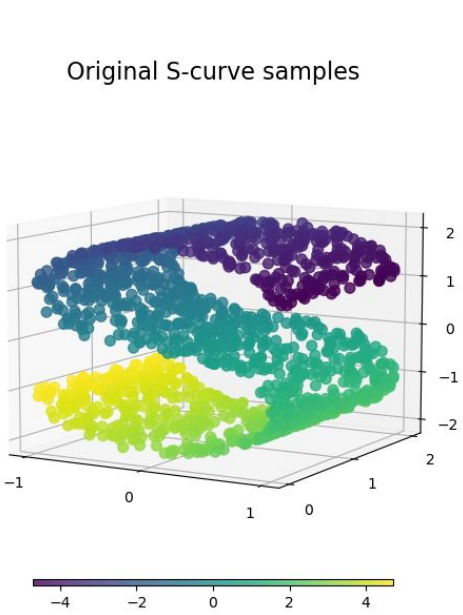
These algorithms attempt to uncover these parameters in order to find a low-dimensional representation of the data.

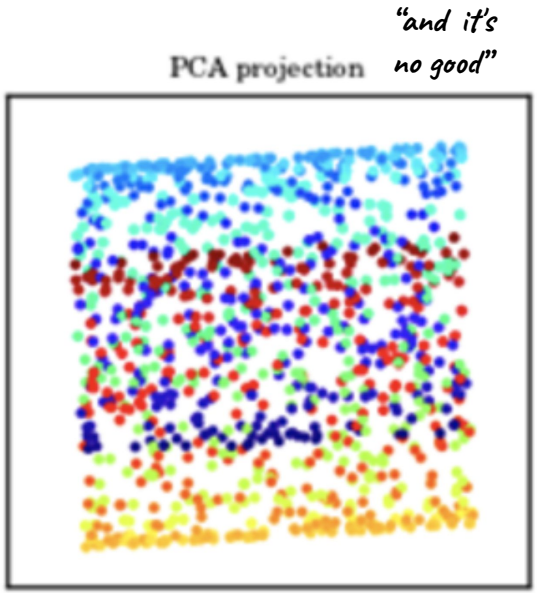


PCA, *NNMF* and *ICA* are great, but have a major drawback: these are all ***linear*** dimensionality reduction algorithms. They cannot learn anything from ***non-linear*** datasets. Complex, non-linear structures can be learned by *manifold learning* algorithms.

Manifold learning are based on the idea that the dimensionality of many data sets is only *artificially* high. Although the data points may consist of thousands of features, they may be described as a function of only a few underlying parameters. That is, the data points are actually samples from a low-dimensional *manifold* (*latent space*) that is embedded in a high-dimensional space.

These algorithms attempt to uncover these parameters in order to find a low-dimensional representation of the data.

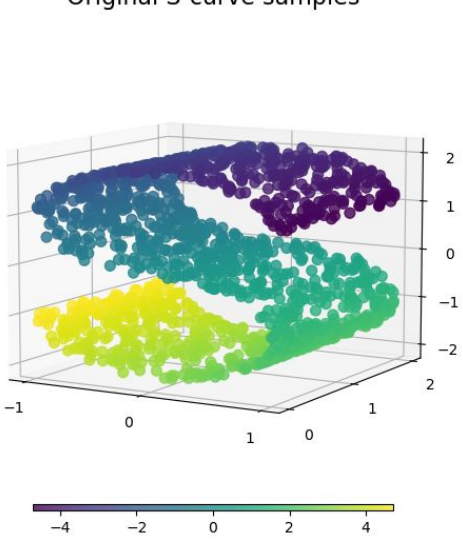


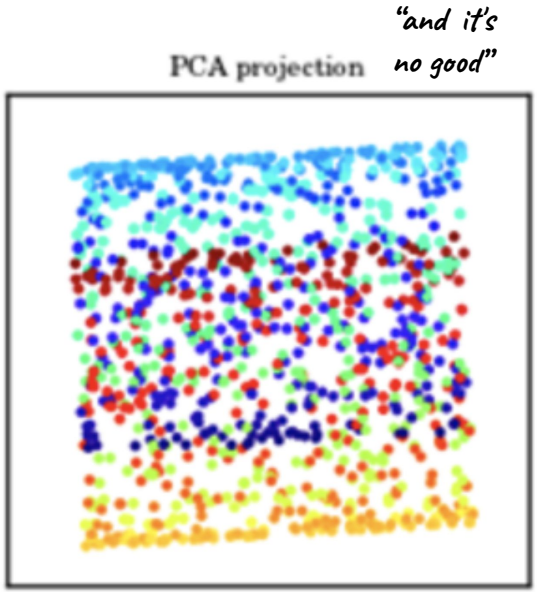


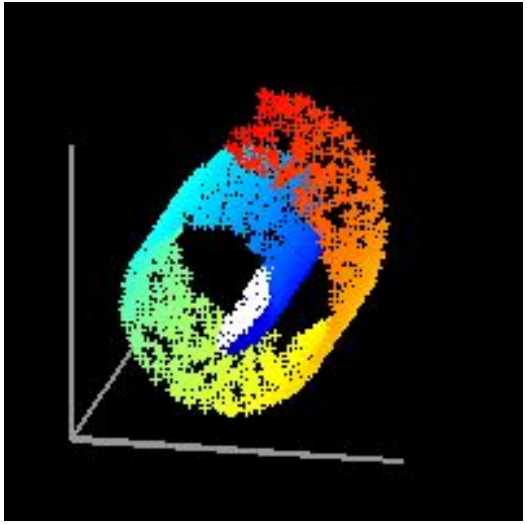
PCA, *NNMF* and *ICA* are great, but have a major drawback: these are all ***linear*** dimensionality reduction algorithms. They cannot learn anything from ***non-linear*** datasets. Complex, non-linear structures can be learned by *manifold learning* algorithms.

Manifold learning are based on the idea that the dimensionality of many data sets is only *artificially* high. Although the data points may consist of thousands of features, they may be described as a function of only a few underlying parameters. That is, the data points are actually samples from a low-dimensional *manifold* (*latent space*) that is embedded in a high-dimensional space.

These algorithms attempt to uncover these parameters in order to find a low-dimensional representation of the data.



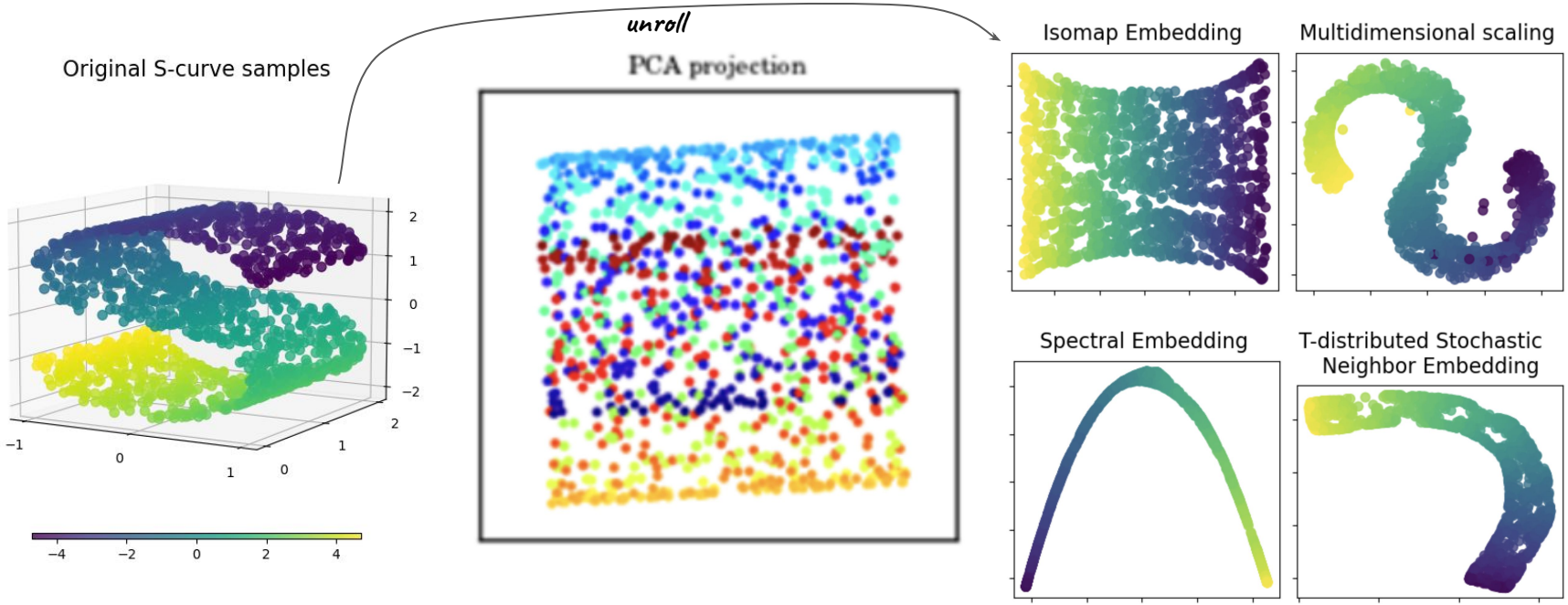




PCA, *NNMF* and *ICA* are great, but have a major drawback: these are all ***linear*** dimensionality reduction algorithms. They cannot learn anything from ***non-linear*** datasets. Complex, non-linear structures can be learned by *manifold learning* algorithms.

Manifold learning are based on the idea that the dimensionality of many data sets is only *artificially* high. Although the data points may consist of thousands of features, they may be described as a function of only a few underlying parameters. That is, the data points are actually samples from a low-dimensional *manifold* (*latent space*) that is embedded in a high-dimensional space.

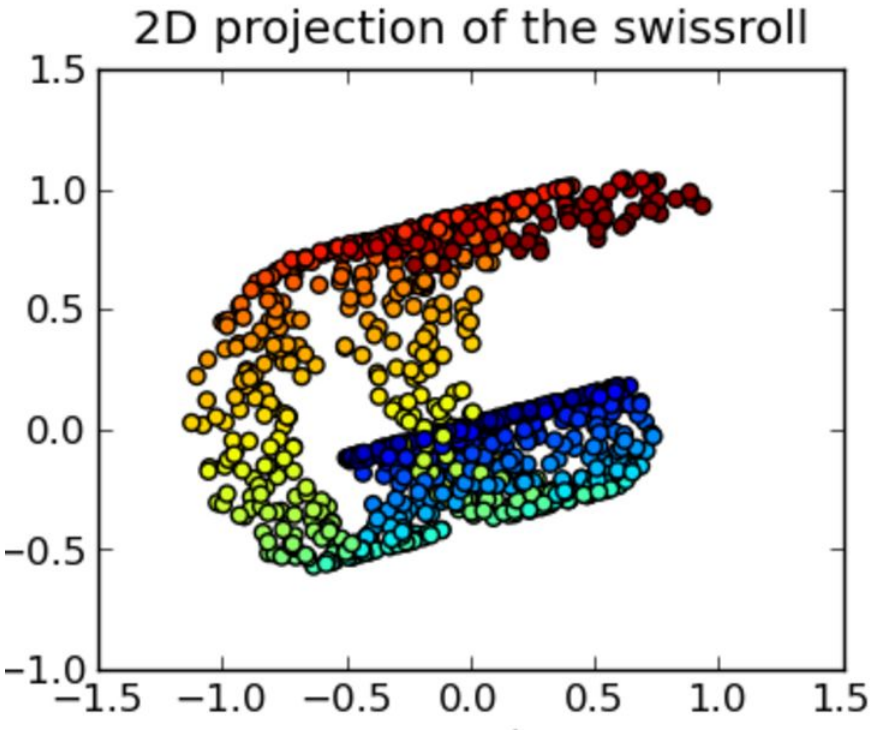
These algorithms attempt to uncover these parameters in order to find a low-dimensional representation of the data.



Multi-Dimensional Scaling (MDS) focuses on creating a mapping that will also preserve the *relative distance* between data

→ if two points are close in the feature space, they should be close in the *latent space*.

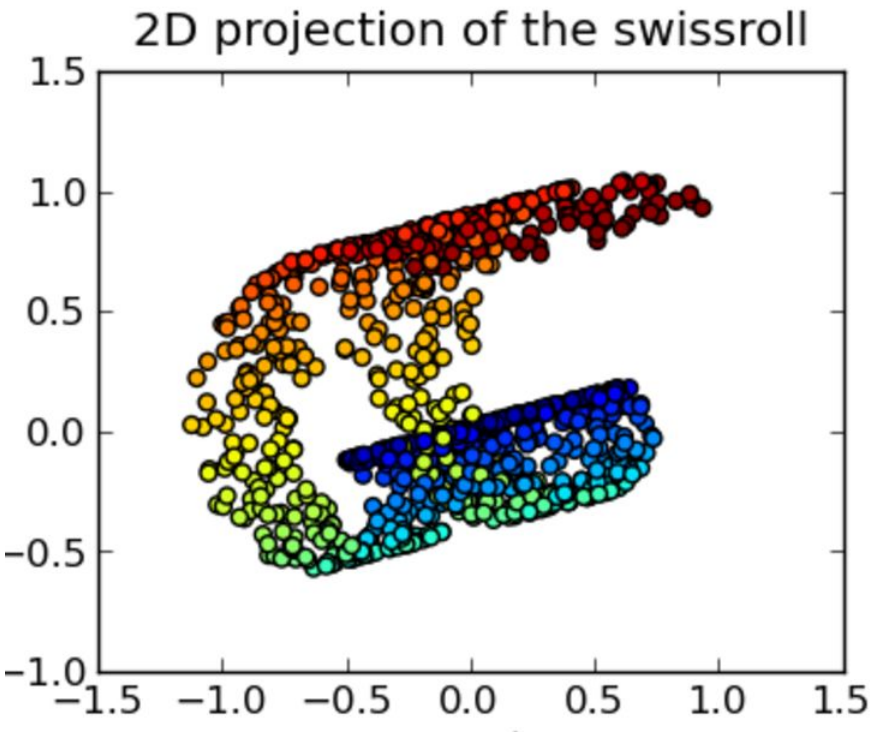
By enforcing such constraint, we can visualize the structure of the data in low dimensions easier.



Multi-Dimensional Scaling (MDS) focuses on creating a mapping that will also preserve the *relative distance* between data

→ if two points are close in the feature space, they should be close in the *latent space*.

By enforcing such constraint, we can visualize the structure of the data in low dimensions easier.



In general, *MDS* is a technique to model similar data as distances in a geometric spaces, and tries to minimize a simple cost function between the distances in the feature space and the distances in *latent space*:

$$J(z) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} (\|z_i - z_j\| - \|x_i - x_j\|)^2$$

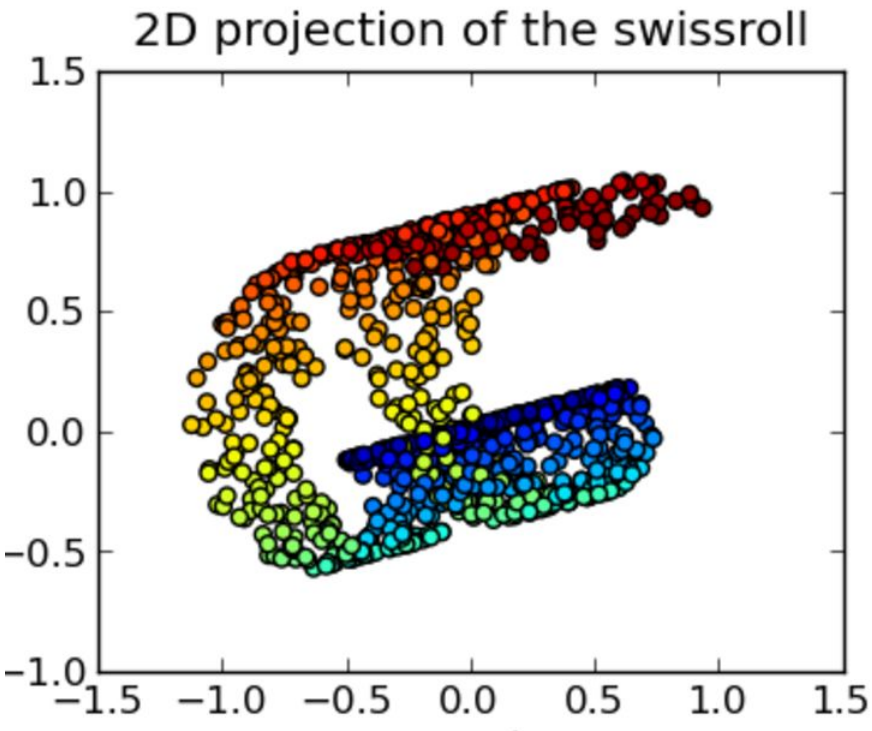
here with the Euclidean distance, but this could be generalized to whatever distance metric you could think of:

$$J(z) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} d3(d2(z_i - z_j), d1(x_i - x_j))$$

Multi-Dimensional Scaling (MDS) focuses on creating a mapping that will also preserve the *relative distance* between data

→ if two points are close in the feature space, they should be close in the *latent space*.

By enforcing such constraint, we can visualize the structure of the data in low dimensions easier.



In general, *MDS* is a technique to model similar data as distances in a geometric spaces, and tries to minimize a simple cost function between the distances in the feature space and the distances in *latent space*:

$$J(z) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} (\|z_i - z_j\| - \|x_i - x_j\|)^2$$

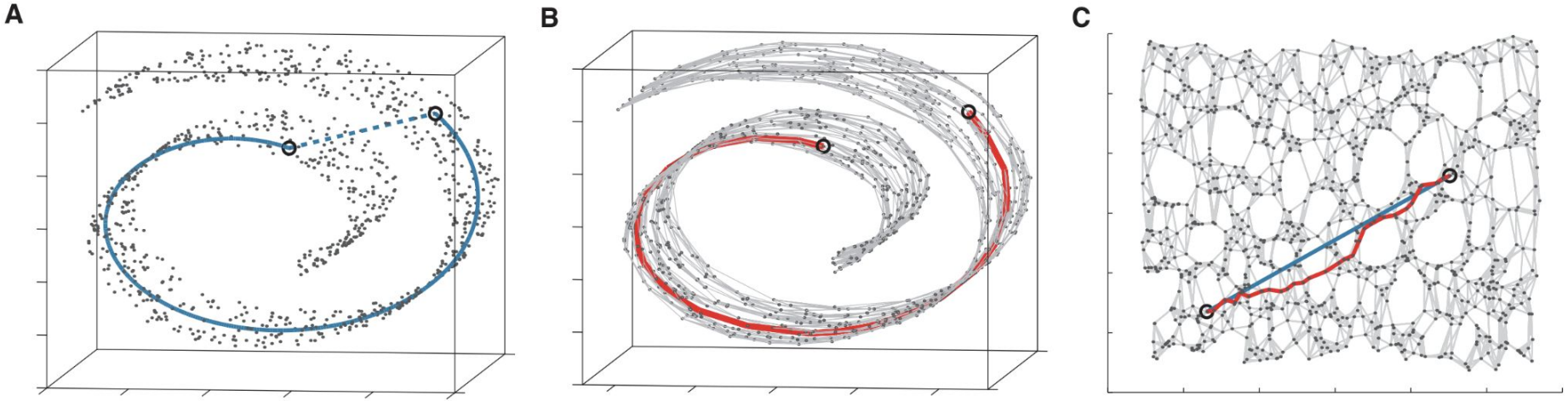
here with the Euclidean distance, but this could be generalized to whatever distance metric you could think of:

$$J(z) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} d3(d2(z_i - z_j), d1(x_i - x_j))$$

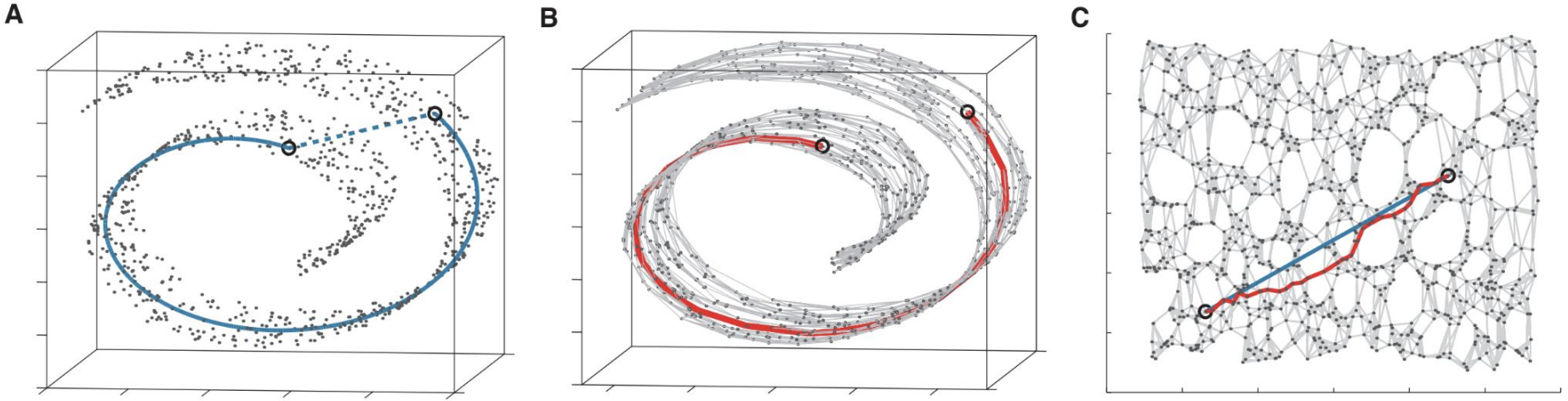
In *Sammon mapping* cost is recalibrated with the distance of the input feature space. Therefore, for smaller distances, we make sure we have a higher precision, so we will not miss the data fine structure:

$$J(z) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \left(\frac{d2(z_i - z_j) - d1(x_i - x_j)}{d1(x_i - x_j)}\right)^2$$

Isometric Mapping (IsoMap) is an extension of *MDS* that captures manifold structure using geodesics to measure distances between all points.



Isometric Mapping (IsoMap) is an extension of *MDS* that captures manifold structure using geodesics to measure distances between all points.



IsoMap algorithm works following these steps:

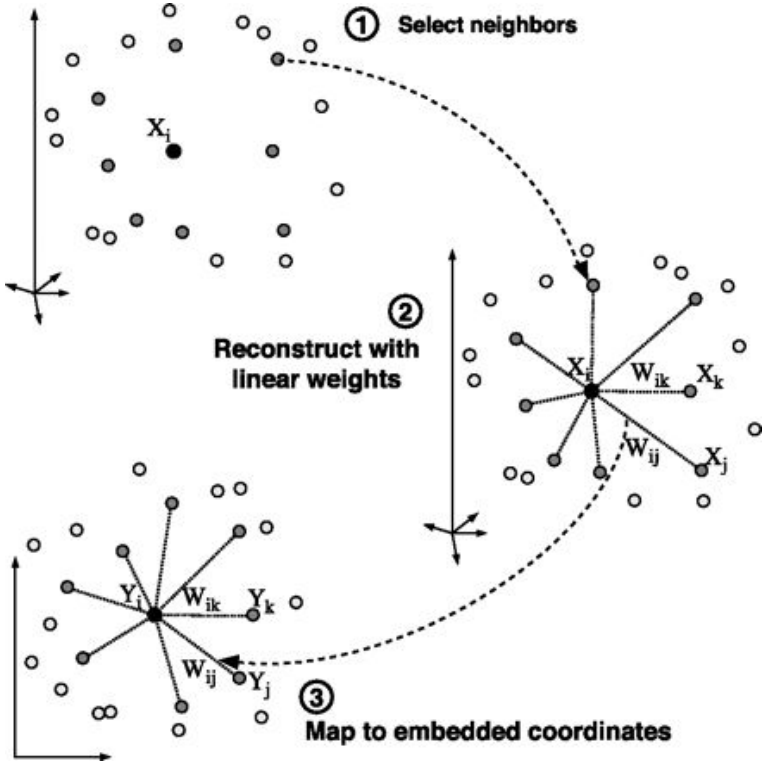
1) ***Nearest neighbor search***, using *BallTree* for efficient neighbor search, as we saw when talking about *Nearest Neighbors*.
2) ***Shortest-path graph search***, building a graph from the ***k*** nodes found in the first step.
3) ***Partial eigenvalue decomposition***. The embedding is encoded in the eigenvectors corresponding to the ***d*** largest eigenvalues of the ***NxN*** isomap kernel.

For the final step of finding the optimal low-dimensional embedding, *MDS* algorithm techniques are used to minimize the reconstruction error:

$$\mathcal{E}_{XY} = |\tau(D_X) - \tau(D_Y)|^2$$

between the original data set X and the low-dimensional embedding Y in *latent space*, where τ is some operator on the distance matrices.

Local Linear Embedding (LLE) attempts to embed high-dimensional data in a lower dimensional space preserving the geometry of the local neighborhoods around each point.
It works in a similar way as *IsoMap*, but without the need to estimate pairwise distances/geodesics between widely separated data points.



Like *IsoMap*, *LLE* finds the ***k-nearest neighbors*** of the points in the first step.
Then it construct a new point approximating each data vector as a weighted linear combination of its *kNNs*. This is achieved minimizing this cost function:

new point ← $J(w) = \sum_i |x_i - \sum_j w_{ij} x_j|^2 \quad \text{with} \sum_j w_{ij} = 1$ → the NNs

Finally, it computes the weights that best reconstruct the vectors from its neighbors, and produce the low-dimensional vectors best reconstructed by these weights.
Now we define the new vector space Y such that we minimize the cost for Y as the new points:

$$J(y) = \sum_i |y_i - \sum_j w_{ij} y_j|^2$$

https://cs.nyu.edu/~roweis/lle/algorithm.html

The *t-distributed Stochastic Neighbor Embedding (t-SNE)* is another manifold learning algorithm.

Probabilities in *latent space* are represented by Student's t-distributions.

The similarity between objects in the high-dimensional space is measured in such a way that similar objects are assigned a higher probability while dissimilar points are assigned a lower probability

The *t-distributed Stochastic Neighbor Embedding (t-SNE)* is another manifold learning algorithm.

Probabilities in *latent space* are represented by Student's t-distributions.

The similarity between objects in the high-dimensional space is measured in such a way that similar objects are assigned a higher probability while dissimilar points are assigned a lower probability

t-SNE works as *MDS* with special functions for the distances:

$$J(z) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} d3(d2(z_i - z_j), d1(x_i - x_j))$$

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|\mathbf{x}_i - \mathbf{x}_k\|^2 / 2\sigma_i^2)}$$

$$D_{KL}(p||q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

$$q_{ij} = \frac{(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2)^{-1}}{\sum_{k \neq m} (1 + \|\mathbf{y}_k - \mathbf{y}_m\|^2)^{-1}}$$

A fundamental hyperparameter of *t-SNE* is the ***"perplexity"*** which says (loosely) how to balance attention between local and global aspects of the data. The parameter is, in a sense, a guess about the number of close neighbors each point has. Getting the most from *t-SNE* may mean analyzing multiple plots with different perplexities.

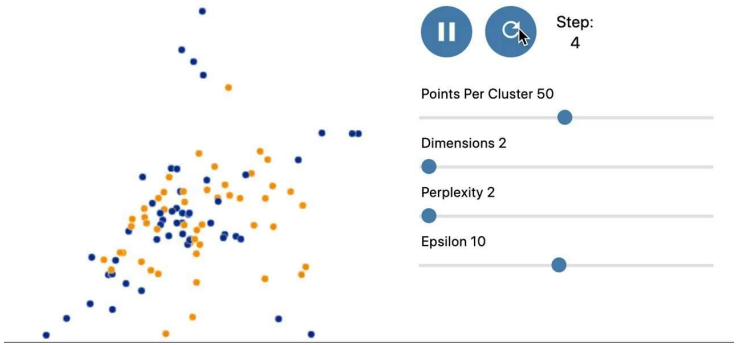
Moreover, *t-SNE* is stochastic: multiple restarts with different seeds can yield different embeddings. However, it is perfectly legitimate to pick the embedding with the least error.

https://distill.pub/2016/misread-tsne/

https://jhui.github.io/2017/01/15/Machine-learning-Multi-dimensional-scaling-and-visualization/

t-SNE is powerful, but it's not a black box, look-at-those-nice blobs algorithm.

1) t-SNE hyperparameters choice greatly influences the final results



Original

Perplexity: 2
Step: 5,000

Perplexity: 5
Step: 5,000

Perplexity: 30
Step: 5,000

Perplexity: 50
Step: 5,000

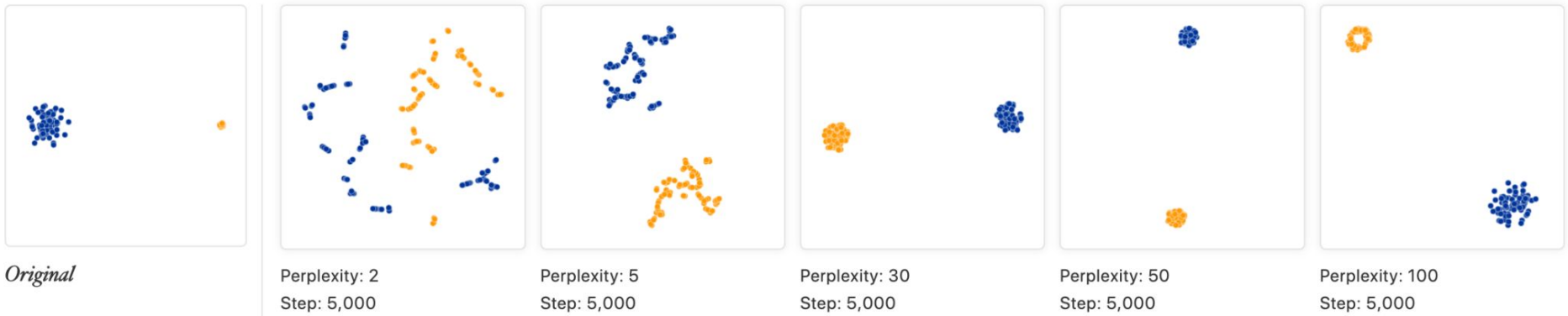
Perplexity: 100
Step: 5,000

https://distill.pub/2016/misread-tsne/

https://jhui.github.io/2017/01/15/Machine-learning-Multi-dimensional-scaling-and-visualization/

t-SNE is powerful, but it's not a black box, look-at-those-nice blobs algorithm.

1) t-SNE hyperparameters choice greatly influences the final results
2) cluster sizes does not mean anything, and does not get reproduced in the final results

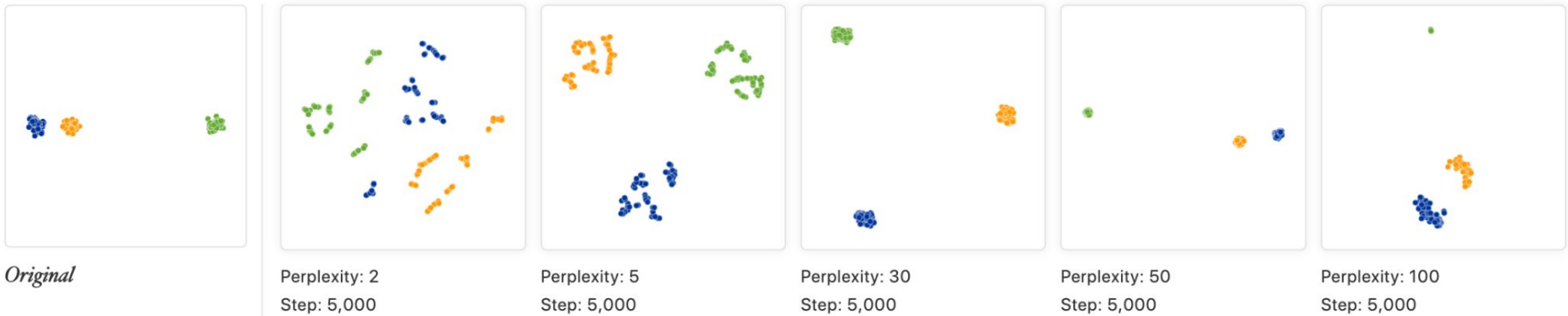


https://distill.pub/2016/misread-tsne/

https://jhui.github.io/2017/01/15/Machine-learning-Multi-dimensional-scaling-and-visualization/

t-SNE is powerful, but it's not a black box, look-at-those-nice blobs algorithm.

1) t-SNE hyperparameters choice greatly influences the final results
2) cluster sizes does not mean anything, and does not get reproduced in the final results
3) similarly, the distances between clusters might not mean anything

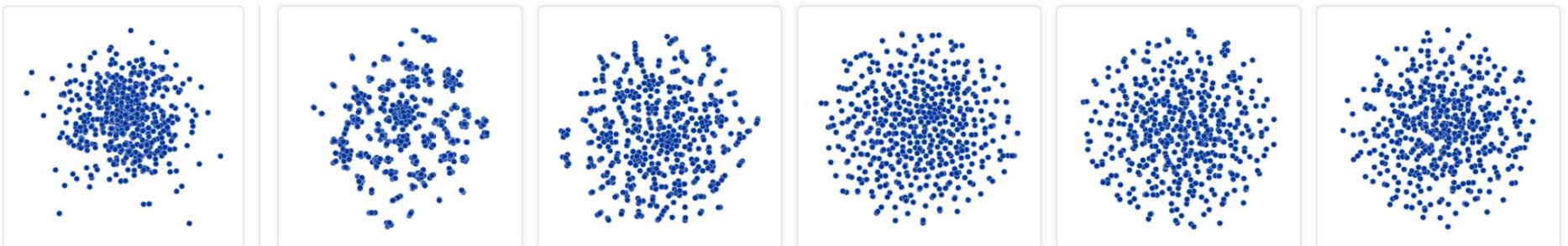


https://distill.pub/2016/misread-tsne/

https://jhui.github.io/2017/01/15/Machine-learning-Multi-dimensional-scaling-and-visualization/

t-SNE is powerful, but it's not a black box, look-at-those-nice blobs algorithm.

1) t-SNE hyperparameters choice greatly influences the final results
2) cluster sizes does not mean anything, and does not get reproduced in the final results
3) similarly, the distances between clusters might not mean anything
4) if you inject random noise into t-SNE, it might give back non-random structures (e.g. with low-perplexity values)



Original

Perplexity: 2
Step: 5,000

Perplexity: 5
Step: 5,000

Perplexity: 30
Step: 5,000

Perplexity: 50
Step: 5,000

Perplexity: 100
Step: 5,000

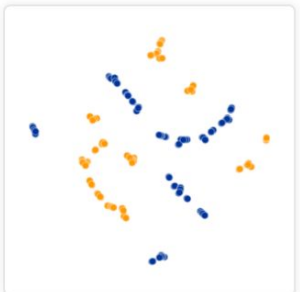
https://distill.pub/2016/misread-tsne/

https://jhui.github.io/2017/01/15/Machine-learning-Multi-dimensional-scaling-and-visualization/

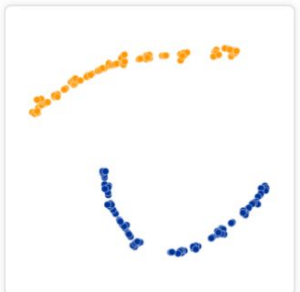
t-SNE is powerful, but it's not a black box, look-at-those-nice blobs algorithm.

1) t-SNE hyperparameters choice greatly influences the final results
2) cluster sizes does not mean anything, and does not get reproduced in the final results
3) similarly, the distances between clusters might not mean anything
4) if you inject random noise into t-SNE, it might give back non-random structures (e.g. with low-perplexity values)
5) don't trust too much the shapes: parallel lines in original space might be non-parallel in a t-SNE plot ("Parallel Convergences")

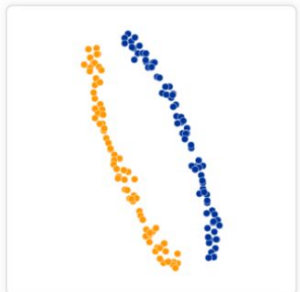
Original



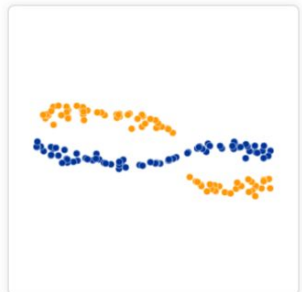
Perplexity: 2
Step: 5,000



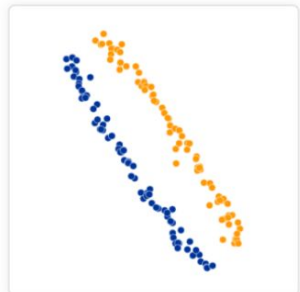
Perplexity: 5
Step: 5,000



Perplexity: 30
Step: 5,000



Perplexity: 50
Step: 5,000



Perplexity: 100
Step: 5,000

https://distill.pub/2016/misread-tsne/

https://jhui.github.io/2017/01/15/Machine-learning-Multi-dimensional-scaling-and-visualization/

t-SNE is powerful, but it's not a black box, look-at-those-nice blobs algorithm.

1) t-SNE hyperparameters choice greatly influences the final results
2) cluster sizes does not mean anything, and does not get reproduced in the final results
3) similarly, the distances between clusters might not mean anything
4) if you inject random noise into t-SNE, it might give back non-random structures (e.g. with low-perplexity values)
5) don't trust too much the shapes: parallel lines in original space might be non-parallel in a t-SNE plot ("Parallel Convergences")
6) *sometimes* you can read topological information off a t-SNE plot, but that typically requires views at multiple perplexities

Original

Perplexity: 2
Step: 5,000

Perplexity: 5
Step: 5,000

Perplexity: 30
Step: 5,000

Perplexity: 50
Step: 5,000

Perplexity: 100
Step: 5,000

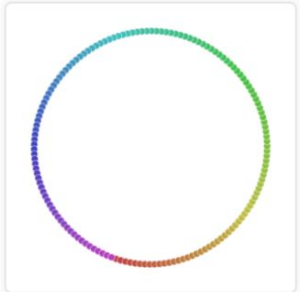
https://distill.pub/2016/misread-tsne/

https://jhui.github.io/2017/01/15/Machine-learning-Multi-dimensional-scaling-and-visualization/

t-SNE is powerful, but it's not a black box, look-at-those-nice blobs algorithm.

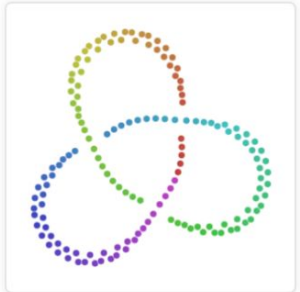
1) t-SNE hyperparameters choice greatly influences the final results
2) cluster sizes does not mean anything, and does not get reproduced in the final results
3) similarly, the distances between clusters might not mean anything
4) if you inject random noise into t-SNE, it might give back non-random structures (e.g. with low-perplexity values)
5) don't trust too much the shapes: parallel lines in original space might be non-parallel in a t-SNE plot ("Parallel Convergences")
6) *sometimes* you can read topological information off a t-SNE plot, but that typically requires views at multiple perplexities

Original



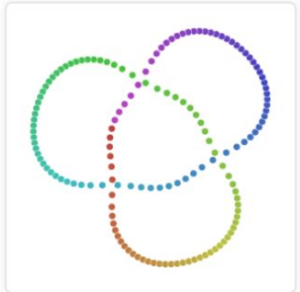
Perplexity: 2
Step: 5,000

Perplexity: 5
Step: 5,000



Perplexity: 30
Step: 5,000

Perplexity: 50
Step: 5,000



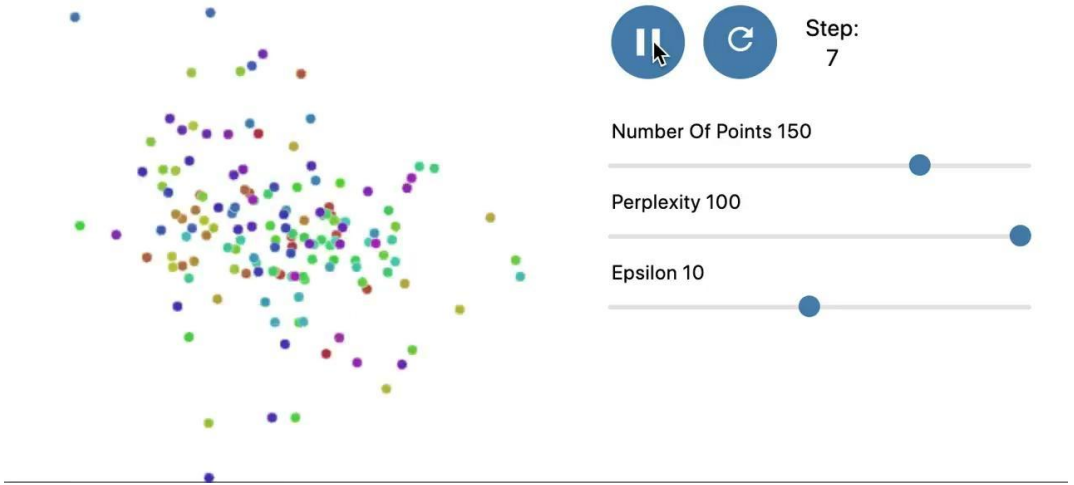
Perplexity: 100
Step: 5,000

https://distill.pub/2016/misread-tsne/

https://jhui.github.io/2017/01/15/Machine-learning-Multi-dimensional-scaling-and-visualization/

t-SNE is powerful, but it's not a black box, look-at-those-nice blobs algorithm.

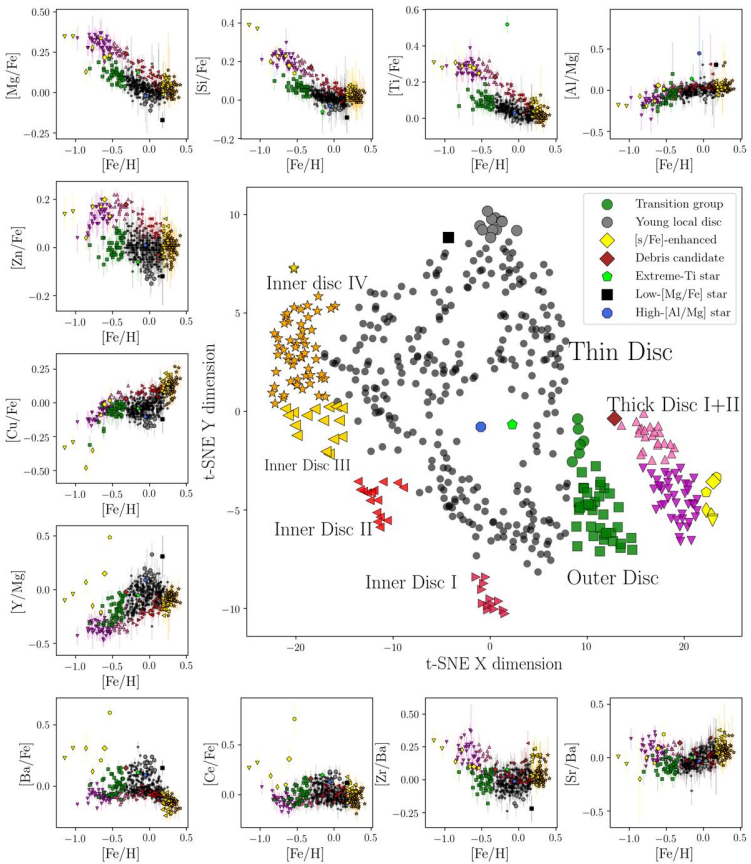
1) t-SNE hyperparameters choice greatly influences the final results
2) cluster sizes does not mean anything, and does not get reproduced in the final results
3) similarly, the distances between clusters might not mean anything
4) if you inject random noise into t-SNE, it might give back non-random structures (e.g. with low-perplexity values)
5) don't trust too much the shapes: parallel lines in original space might be non-parallel in a t-SNE plot ("Parallel Convergences")
6) *sometimes* you can read topological information off a t-SNE plot, but that typically requires views at multiple perplexities



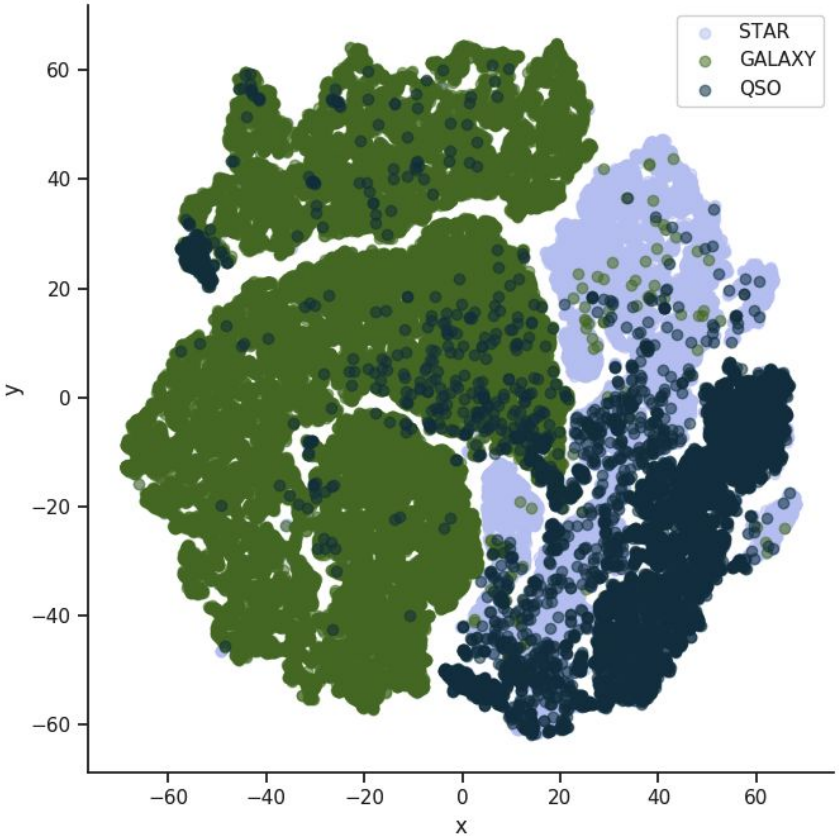
https://distill.pub/2016/misread-tsne/

https://jhui.github.io/2017/01/15/Machine-learning-Multi-dimensional-scaling-and-visualization/

The *t-distributed Stochastic Neighbor Embedding (t-SNE)* is another manifold learning algorithm.



Anders et al., 2019

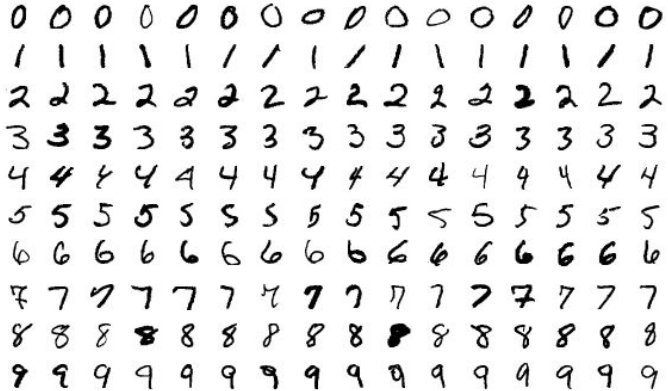


Nakoneczny et al., 2019

Embedding Projector

On https://projector.tensorflow.org/ you can find a web interface showing different dimensionality reduction algorithms (*U-MAP*, *t-SNE* and *PCA*) applied to standard datasets (MNIST digits, Iris, Word2Vec); it also gives the opportunity to upload your own dataset.

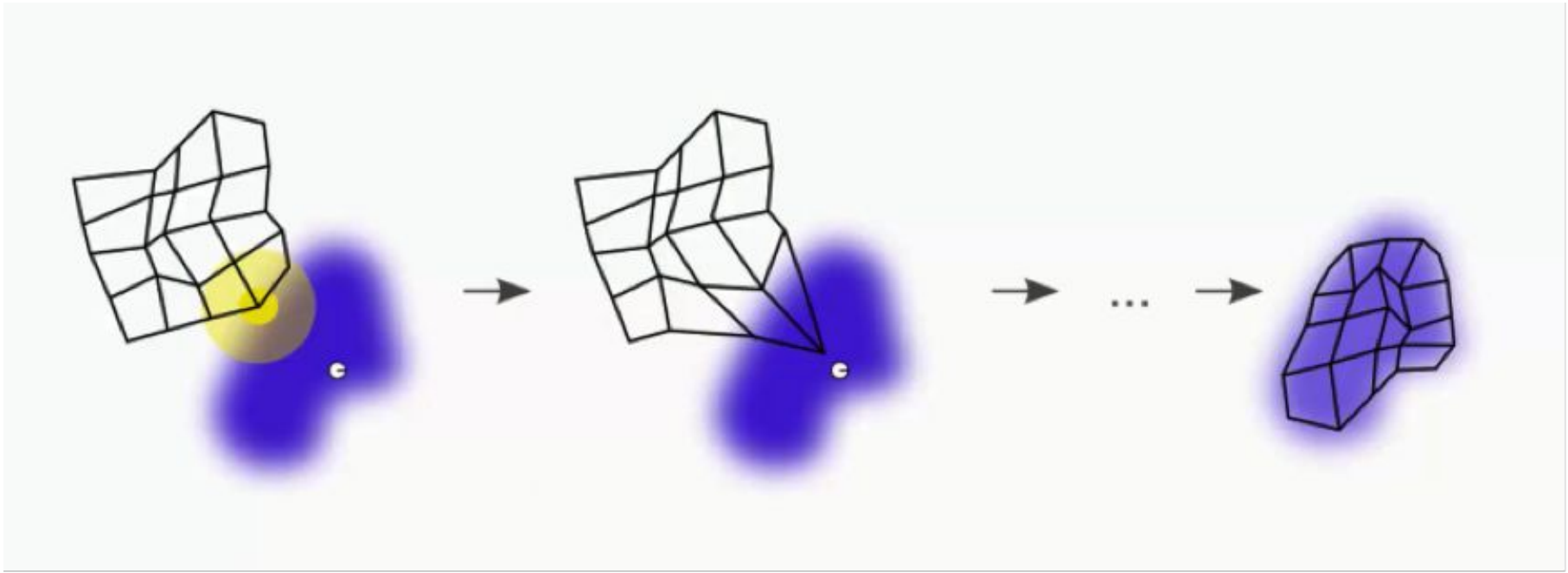
Here you see an example on the MNIST dataset:



which is the golden dataset for, e.g. computer vision, along with cat and dogs.

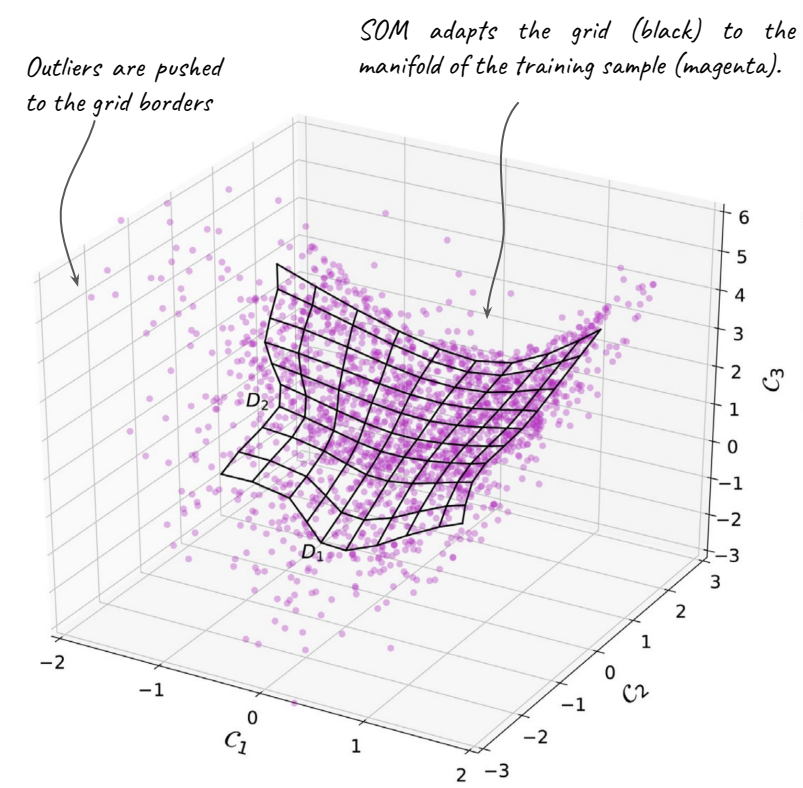
Self-Organizing Maps (or *Kohonen Maps*) are an unsupervised learning, dimensionality reduction algorithm, aiming to project complex manifolds into low-dimensional surfaces, preserving the manifold topology.

A *SOM* identifies data patterns in a *D*-dimensional feature space by sampling it with an adaptive distribution of *weights*. During the training phase, which is unsupervised, the *weights* coordinates in the multi-dimensional space are adjusted to move them close to the input data. The input data set is mapped from the high-dimensional manifold to (usually) a 2D geometry, reproducing not only the *shape* of the input manifold, but also its *density*, i.e., a larger number of test particles is located where data are more clustered.

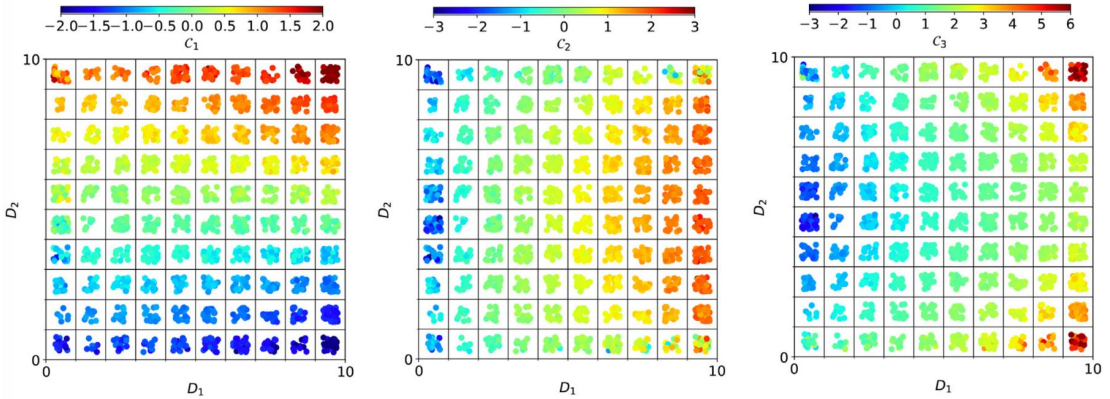


https://www.superdatascience.com/blogs/the-ultimate-guide-to-self-organizing-maps-soms

In a nutshell, SOMs map from high-dimension manifold to 2D grids, built in a way that adapts not only to the manifold shape, but also to the training data density.



Davidzon et al., 2019

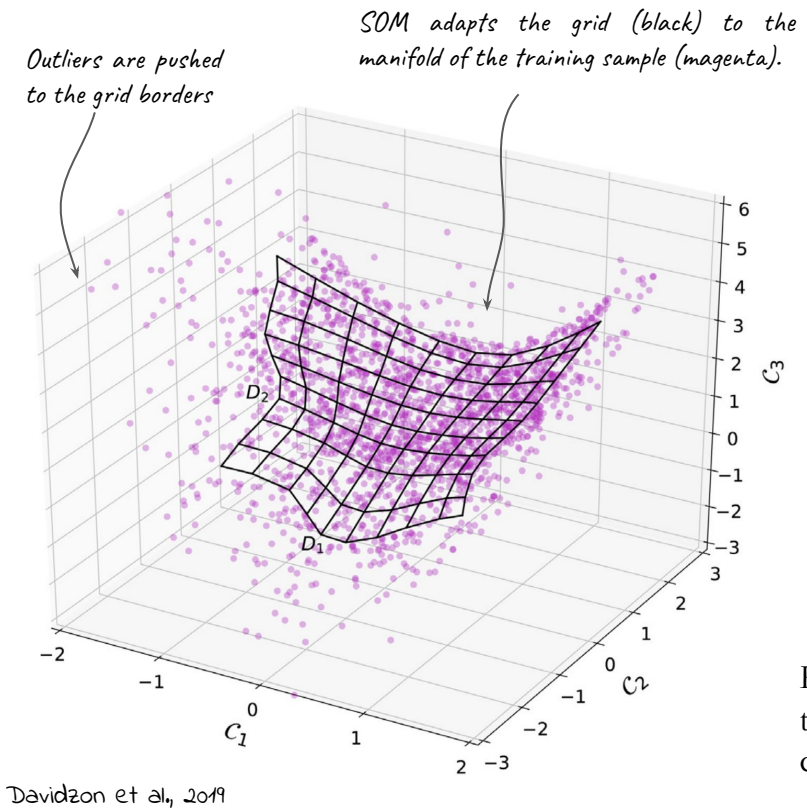


Here you see a simple example with a 10×10 grid and 2000 training objects. Features C_1, C_2, C_3 can be any feature, e.g. galaxy colors, or redshifts, whatever you can think of.

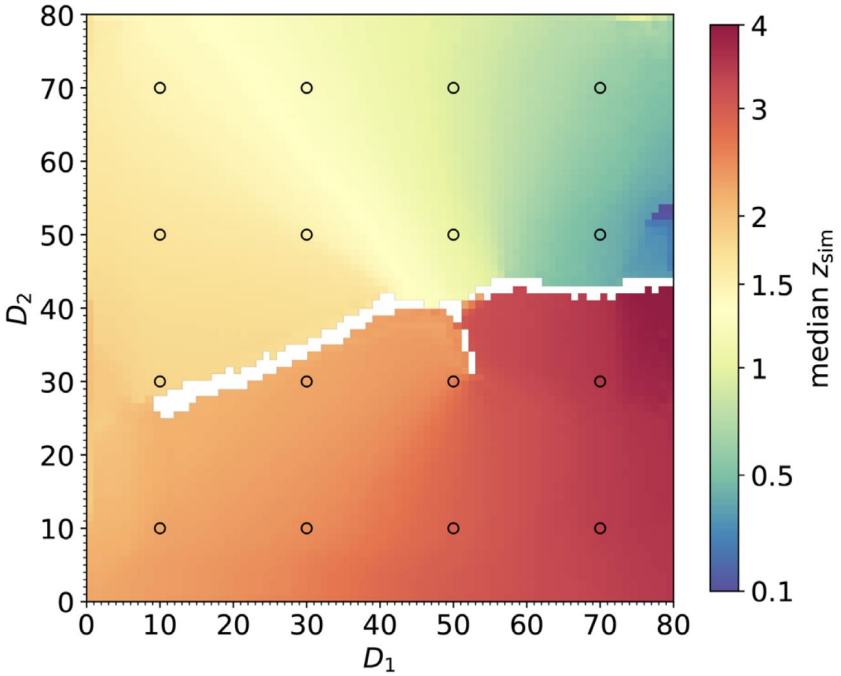
A SOM allocates the 2000 training objects into the 10×10 grid. In each panel, objects are colour coded according to one of their features (C_1, C_2, C_3).

Similar objects are clustered in the same (or nearby) cell and a smooth transition is observed across the grid, while outliers with extreme characteristics (e.g. $C_3 > 4$) are pushed to the grid corners.

In a nutshell, *SOMs* map from high-dimension manifold to 2D grids, built in a way that adapts not only to the manifold shape, but also to the training data density.

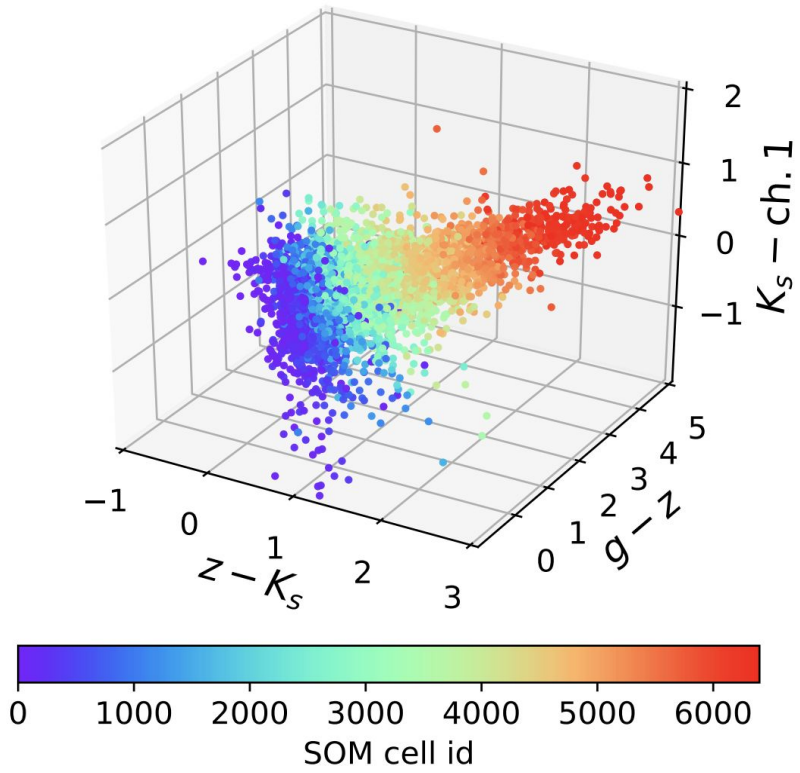


Davidzon et al., 2019

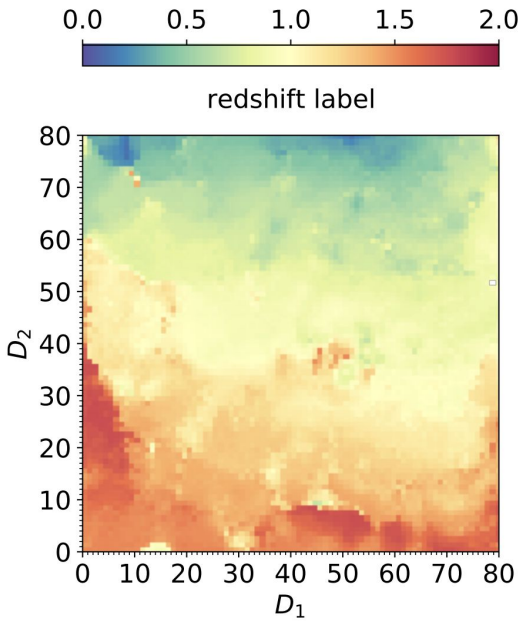


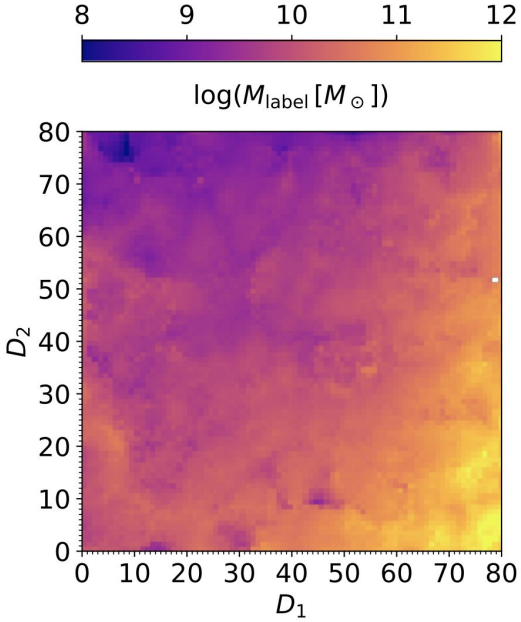
Here the mapped feature is actually a *real* feature, a redshift **z**. One interesting thing to notice, that you already knew if you worked in SED fitting, is the caustic in between: galaxies close in color space can either be at **z < 1** and **z > 3**.

In a nutshell, SOMs map from high-dimension manifold to 2D grids, built in a way that adapts not only to the manifold shape, but also to the training data density.



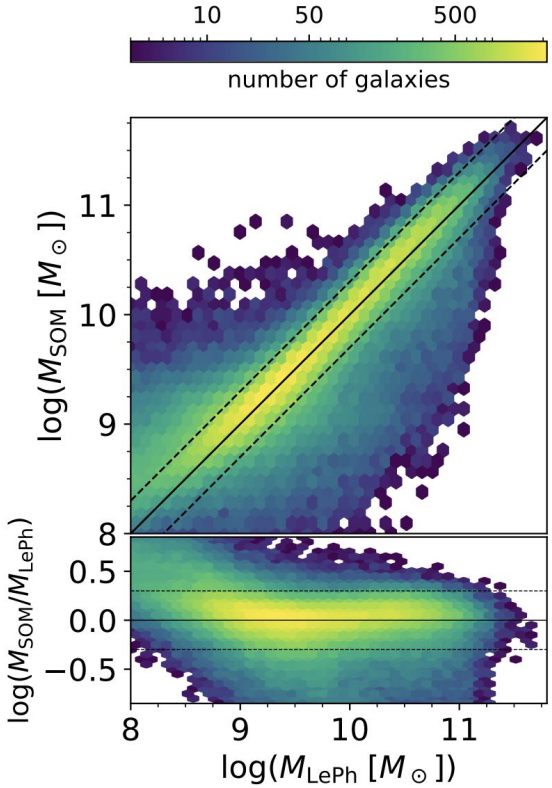
A trained SOM can easily been used as a supervised learning algorithm: if a new example arrives, just look in feature space where it gets mapped on the SOM grid, and assign a value interpolating the training values in the same cell.



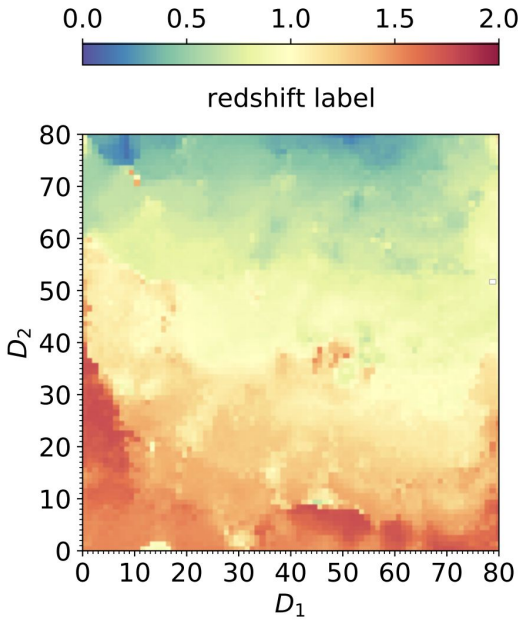


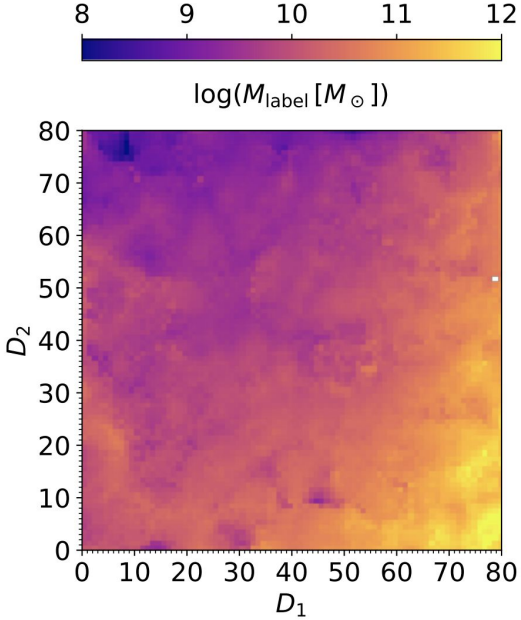
Davidzon et al., 2022

In a nutshell, SOMs map from high-dimension manifold to 2D grids, built in a way that adapts not only to the manifold shape, but also to the training data density.



A trained SOM can easily been used as a supervised learning algorithm: if a new example arrives, just look in feature space where it gets mapped on the SOM grid, and assign a value interpolating the training values in the same cell.





Davidzon et al., 2022